## Contents

1 Getting started guides 3
  1.1 Introduction to Elixir 3
  1.2 Mix - a build tool for Elixir 59
  1.3 ExUnit - a unit test framework 74
  1.4 Meta-programming in Elixir 77

2 Technical guides 89
  2.1 Scoping Rules in Elixir (and Erlang) 89

3 API reference 101
  3.1 v0.13.3 101
  3.2 API Reference (v0.14.0-dev) 101
  3.3 Topical Reference 464

4 Source repo docs 467
  4.1 Readme 467
  4.2 Contributing to Elixir 468
  4.3 Release process 472
  4.4 License 472
  4.5 Changelog 473

Elixir Module Index 503
This is an experiment to organize the different kinds of Elixir documentation in a new way.
1.1 Introduction to Elixir

1.1.1 Interactive Elixir

- 1.1 Installing Erlang
- 1.2 Distributions
- 1.3 Precompiled package
- 1.4 Compiling from source (Unix and MinGW)
- 1.5 Interactive mode

Welcome!

In this tutorial we are going to show you how to get started with Elixir. This chapter will cover installation and how to get started with the Interactive Elixir shell called IEx.

If you find any errors in the tutorial or on the website, please report a bug or send a pull request to our issue tracker. If you suspect it is a language bug, please let us know in the language issue tracker.

Let’s get started!

1.1 Installing Erlang

The only prerequisite for Elixir is Erlang, version 17.0 or later, which can be easily installed with Precompiled packages. In case you want to install it directly from source, it can be found on the Erlang website or by following the excellent tutorial available in the Riak documentation.

For Windows developers, we recommend the precompiled packages. Those on a UNIX platform can probably get Erlang installed via one of the many package management tools.

Note: Although many package management tools provide Erlang, Elixir requires Erlang 17 which has been released just recently. So have that in mind before picking your Erlang installation.

After Erlang is installed, you should be able to open up the command line (or command prompt) and check the Erlang version by typing `erl`. You will see some information as follows:

```
Erlang/OTP 17 (erts-6) [64-bit] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]
```

Notice that depending on how you installed Erlang, Erlang binaries won’t be available in your PATH. Be sure to have Erlang binaries in your PATH, otherwise Elixir won’t work!
After Erlang is up and running, it is time to install Elixir. You can do that via Distributions, Precompiled package or Compiling from Source.

### 1.2 Distributions

This tutorial requires Elixir v0.13 or later and it may be available in some distributions:

- **Homebrew for Mac OS X**
  
  Update your homebrew to latest with `brew update`
  
  Install Elixir: `brew install elixir`

- **Fedora 17+ and Fedora Rawhide**
  
  `sudo yum -y install elixir`

- **Arch Linux (on AUR)**
  
  `yaourt -S elixir`

- **openSUSE (and SLES 11 SP3+)**
  
  Add Erlang devel repo with `zypper ar -f obs://devel:languages:erlang/ erlang`
  
  Install Elixir: `zypper in elixir`

- **Gentoo**
  
  `emerge --ask dev-lang/elixir`

- **Chocolatey for Windows**
  
  `cinst elixir`

If you don’t use any of the distributions above, don’t worry, we also provide a precompiled package!

### 1.3 Precompiled package

Elixir provides a precompiled package for every release. After downloading and unzipping the package, you are ready to run the `elixir` and `iex` commands from the `bin` directory. It is recommended that you also add Elixir’s `bin` path to your `PATH` environment variable to ease development.

### 1.4 Compiling from source (Unix and MinGW)

You can download and compile Elixir in few steps. You can get the latest stable release here, unpack it and then run `make` inside the unpacked directory. After that, you are ready to run the `elixir` and `iex` commands from the `bin` directory. It is recommended that you add Elixir’s `bin` path to your `PATH` environment variable to ease development:

```bash
$ export PATH="$PATH:/path/to/elixir/bin"
```

In case you are feeling a bit more adventurous, you can also compile from master:

```bash
$ git clone https://github.com/elixir-lang/elixir.git
$ cd elixir
$ make clean test
```

If the tests pass, you are ready to go. Otherwise, feel free to open an issue in the issues tracker on Github.
1.5 Interactive mode

When you install Elixir, you will have three new executables: *iex*, *elixir* and *elixirc*. If you compiled Elixir from source or are using a packaged version, you can find these inside the *bin* directory.

For now, let’s start by running *iex* which stands for Interactive Elixir. In interactive mode, we can type any Elixir expression and get its result straight away. Let’s warm up with some basic expressions:

```
Interactive Elixir - press Ctrl+C to exit (type h() ENTER for help)
iex> 40 + 2
42
iex> "hello" <> " world"
"hello world"
```

It seems we are ready to go! We will use the interactive shell quite a lot in the next chapters to get a bit more familiar with the language constructs and basic types, starting in the next chapter.

1.1.2 2 Basic types

- 2.1 Basic arithmetic
- 2.2 Booleans
- 2.3 Atoms
- 2.4 Strings
- 2.5 Anonymous functions
- 2.6 (Linked) Lists
- 2.7 Tuples
- 2.8 Lists or tuples?

In this chapter we will learn more about Elixir basic types: integers, floats, atoms, lists and strings. Some basic types are:

```
iex> 1       # integer
iex> 0x1F    # integer
iex> 1.0     # float
iex> :atom   # atom / symbol
iex> "elixir" # string
iex> [1, 2, 3] # list
iex> {1, 2, 3} # tuple
```

2.1 Basic arithmetic

Open up *iex* and type the following expressions:

```
iex> 1 + 2
3
iex> 5 * 5
25
iex> 10 / 2
5.0
```

Notice that 10 / 2 returned a float 5.0 instead of an integer 5. This is expected. In Elixir, the operator / always returns a float. If you want to do integer division or get the division remainder, you can invoke the *div* and *rem* functions:
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1

Notice that parentheses are not required in order to invoke a function.

Elixir also supports shortcut notations for entering binary, octal and hexadecimal numbers:

iex> 0b1010
10
iex> 0777
511
iex> 0x1F
31

Float numbers require a dot followed by at least one digit and also support e for the exponent number:

iex> 1.0
1.0
iex> 1.0e-10
1.0e-10

Floats in Elixir are 64 bit double precision.

### 2.2 Booleans

Elixir supports `true` and `false` as booleans.

iex> true
true
iex> true == false
false

Elixir provides a bunch of predicate functions to check for a value type. For example, the `is_boolean/1` function can be used to check if a value is a boolean or not:

> Note: Functions in Elixir are identified by name and by number of arguments (i.e. arity). Therefore, `is_boolean/1` identifies a function named `is_boolean` that takes 1 argument. `is_boolean/2` identifies a different (nonexistent) function with the same name but different arity.

iex> is_boolean(true)
true
iex> is_boolean(1)
false

You can also use `is_integer/1`, `is_float/1` or `is_number/1` to check, respectively, if an argument is an integer, a float or either.

> Note: at any moment you can type `h` in the shell to print information on how to use the shell. The `h` helper can also be used to access documentation for any function. For example, typing `h is_integer/1` is going to print the documentation for the `is_integer/1` function. It also works with operators and other constructs (try `h ==/2`).

### 2.3 Atoms

Atoms are constants where their name is their own value. Some other languages call these symbols.
The booleans `true` and `false` are, in fact, atoms:

```iex
iex> true == :true
true
iex> is_atom(false)
true
```

### 2.4 Strings

Strings in Elixir are inserted in between double quotes, and they are encoded in UTF-8:

```iex
iex> "hellö"
"hellö"
```

Elixir also supports string interpolation:

```iex
iex> "hellö #{:world}"
"hellö world"
```

Strings can have line breaks in them or introduce them using escape sequences:

```iex
iex> "hello
...> world"
"hello\nworld"
iex> "hello\nworld"
"hello\nworld"
```

You can print a string using the `IO.puts/1` function from the `IO` module:

```iex
iex> IO.puts "hello\nworld"
hello
world
:ok
```

Notice the `IO.puts/1` function returns the atom `:ok` as result after printing.

Strings in Elixir are represented internally by binaries which are sequences of bytes:

```iex
iex> is_binary("hellö")
true
```

We can also get the number of bytes in a string:

```iex
iex> byte_size("hellö")
6
```

Notice the number of bytes in that string is 6, even though it has 5 characters. That’s because the character “ö” takes 2 bytes to be represented in UTF-8. We can get the actual length of the string, based on the number of characters, by using the `String.length/1` function:

```iex
iex> String.length("hellö")
5
```

The `String` module contains a bunch of functions that operate on strings as defined in the Unicode standard:
Keep in mind single-quoted and double-quoted strings are not equivalent in Elixir as they are represented by different types:

```elixir```
`'hellö'` == "hellö"
false
```

We will talk more about Unicode support and the difference between single and double-quoted strings in the “Binaries, strings and char lists” chapter.

### 2.5 Anonymous functions

Functions are delimited by the keywords `fn` and `end`:

```elixir```
add = fn a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
add_two = fn a -> add.(a, 2) end
#Function<6.71889879/1 in :erl_eval.expr/5>
```

Functions are “first class citizens” in Elixir meaning they can be passed as arguments to other functions just as integers and strings can. In the example, we have passed the function in the variable `add` to the `is_function/1` function which correctly returned `true`. We can also check the arity of the function by calling `is_function/2`.

Note a dot (.) in between the variable and parenthesis is required to invoke an anonymous function.

Anonymous functions are closures, and as such they can access variables that are in scope when the function is defined:

```elixir```
add_two = fn a -> add.(a, 2) end
#Function<6.71889879/1 in :erl_eval.expr/5>
add_two.(2)
4
```

Keep in mind that a variable assigned inside a function does not affect its surrounding environment:

```elixir```
x = 42
42
(fn -> x = 0 end).()
0
x
42
```

### 2.6 (Linked) Lists

Elixir uses square brackets to specify a list of values. Values can be of any type:

```elixir```
[1, 2, true, 3]
[1, 2, true, 3]
length [1, 2, 3]
3
Two lists can be concatenated and subtracted using the `++/2` and `--/2` operators:

```
iex> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex> [1, true, 2, false, 3, true] -- [true, false]
[1, 2, 3, true]
```

Throughout the tutorial, we will talk a lot about the head and tail of a list. The head is the first element of a list and the tail is the remainder of a list. They can be retrieved with the functions `hd/1` and `tl/1`. Let’s assign a list to a variable and retrieve its head and tail:

```
iex> list = [1,2,3]
iex> hd(list)
1
iex> tl(list)
[2, 3]
```

Getting the head or the tail of an empty list is an error:

```
iex> hd []
** (ArgumentError) argument error
```

Oops!

### 2.7 Tuples

Elixir uses curly brackets to define tuples. As lists, tuples can hold any value:

```
iex> {:ok, "hello"}
{:ok, "hello"}
iex> size {:ok, "hello"}
2
```

Tuples store elements contiguously in memory. This means accessing a tuple element per index or getting the tuple size is a fast operation (indexes start from zero):

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
iex> tuple_size(tuple)
2
```

It is also possible to set an element at a particular index in a tuple with `set_elem/3`:

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> set_elem(tuple, 1, "world")
{:ok, "world"}
iex> tuple
{:ok, "hello"}
```

Notice that `set_elem/3` returned a new tuple. The original tuple stored in the `tuple` variable was not modified because Elixir data types are immutable. By being immutable, Elixir code is easier to reason about as you never need to worry if a particular code is mutating your data structure in place.

By being immutable, Elixir also helps eliminate common cases where concurrent code has race conditions because two different entities are trying to change a data structure at the same time.

### 1.1. Introduction to Elixir
2.8 Lists or tuples?

What is the difference between lists and tuples?

Lists are stored in memory as linked lists. This means each element in a list points to the next element, and then to the next element, until it reaches the end of a list. We call each of those pairs in a list a cons cell:

```iex
iex> list = [1|[2|[3|[]]]]
[1, 2, 3]
```

This means accessing the length of a list is a linear operation: we need to traverse the whole list in order to figure out its size. Updating a list is fast as long as we are prepending elements:

```iex
iex> [0] ++ list
[0, 1, 2, 3]
iex> list ++ [4]
[1, 2, 3, 4]
```

The first operation is fast because we are simply adding a new cons that points to the remaining of list. The second one is slow because we need to rebuild the whole list and add a new element to the end.

Tuples, on the other hand, are stored contiguously in memory. This means getting the tuple size or accessing an element by index is fast. However, updating or adding elements to tuples is expensive because it requires copying the whole tuple in memory.

Those performance characteristics dictate the usage of those data structures. One very common use case for tuples is to use them to return extra information from a function. For example, `File.read/1` is a function that can be used to read file contents and it returns tuples:

```iex
iex> File.read("path/to/existing/file")
{:ok, "... contents ..."}
iex> File.read("path/to/unknown/file")
{:error, :enoent}
```

If the path given to `File.read/1` exists, it returns a tuple with the atom :ok as first element and the file contents as second. Otherwise, it returns a tuple with :error and the error reason.

Most of the time, Elixir is going to guide you to do the right thing. For example, there is a `elem/2` function to access a tuple item but there is no built-in equivalent for lists:

```iex
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
```

When “counting” the number of elements in a data structure, Elixir also abides by a simple rule: the function should be named `size` if the operation is in constant time (i.e. the value is pre-calculated) or `length` if the operation requires explicit counting.

For example, we have used 4 counting functions so far: `byte_size/1` (for the number of bytes in a string), `tuple_size/1` (for the tuple size), `length/1` (for the list length) and `String.length/1` (for the number of characters in a string). That said, we use `byte_size` to get the number of bytes in a string, which is cheap, but retrieving the number of uncode characters uses `String.length`, since the whole string needs to be iterated.

Elixir also provides `Port`, `Reference` and `PID` as data types (usually used in process communication), and we will take a quick look at them when talking about processes. For now, let’s take a look at some of the basic operators that go with our basic types.
3 Basic operators

In the previous chapter, we saw Elixir provides +, -, *, / as arithmetic operators, plus the functions div/2 and rem/2 for integer division and remainder.

Elixir also provides ++ and -- to manipulate lists:

```elixir
eix> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
eix> [1,2,3] -- [2]
[1,3]
```

String concatenation is done with <>:

```elixir
eix> "foo" <> "bar"
"foobar"
```

Elixir also provides three boolean operators: or, and and not. These operators are strict in the sense that they expect a boolean (true or false) as their first argument:

```elixir
eix> true and true
true
eix> false or is_atom(:example)
true
```

Providing a non-boolean will raise an exception:

```elixir
eix> 1 and true
++ (ArgumentError) argument error
```

or and and are short-circuit operators. They only execute the right side if the left side is not enough to determine the result:

```elixir
eix> false and error("This error will never be raised")
false
eix> true or error("This error will never be raised")
true
```

Note: If you are an Erlang developer, ‘‘and’’ and ‘‘or’’ in Elixir actually map to the ‘‘andalso’’ and ‘‘orelse’’ operators in Erlang.

Besides these boolean operators, Elixir also provides |, && and ! which accept arguments of any type. For these operators, all values except false and nil will evaluate to true:

```
# or
iex> 1 || true
1
iex> false || 11
11

# and
iex> nil && 13
nil
iex> true && 17
17

# !
iex> !true
false
```
Elixir Documentation, Release

```elixir
iex> !1
false
iex> !nil
true
```

As a rule of thumb, use `and`, `or` and `not` when you are expecting booleans. If any of the arguments are non-boolean, use `&&`, `||` and `!`.

Elixir also provides `==`, `!=`, `===`, `!==`, `<=`, `>=`, `<` and `>` as comparison operators:

```elixir
iex> 1 == 1
true
iex> 1 != 2
true
iex> 1 < 2
true
```

The difference between `==` and `===` is that the latter is more strict when comparing integers and floats:

```elixir
iex> 1 == 1.0
true
iex> 1 === 1.0
false
```

In Elixir, we can compare two different data types:

```elixir
iex> 1 < :atom
true
```

The reason we can compare different data types is pragmatism. Sorting algorithms don’t need to worry about different data types in order to sort. The overall sorting order is defined below:

```plaintext
number < atom < reference < functions < port < pid < tuple < maps < list < bitstring
```

You don’t actually need to memorize this ordering, but it is important just to know an order exists.

Well, that is it for the introduction. In the next chapter, we are going to discuss some basic functions, data type conversions and a bit of control-flow.

### 1.1.4 4 Pattern matching

- 4.1 The match operator
- 4.2 Pattern matching
- 4.3 The pin operator

In this chapter, we will show how the `=` operator in Elixir is actually a match operator and how to use it to pattern match inside data structures. Finally, we will learn about the pin operator `^` used to access previously bound values.

#### 4.1 The match operator

We have used the `=` operator a couple times to assign variables in Elixir.

```elixir
iex> x = 1
1
iex> x
1
```
In Elixir, the `=` operator is actually called the *match operator*. Let’s see why:

```iex
iex> 1 = x
1
iex> 2 = x
** (MatchError) no match of right hand side value: 1
```

Notice that `1 = x` is a valid expression, and it matched because both the left and right side are equal to 1. When the sides do not match, a `MatchError` is raised.

A variable can only be assigned on the left side of `=`:

```iex
iex> 1 = unknown
** (RuntimeError) undefined function: unknown/0
```

Since there is no variable `unknown` previously defined, Elixir imagined you were trying to call a function named `unknown/0`, but there isn’t such function.

### 4.2 Pattern matching

The match operator is not only used to match against simple values, but it is also useful for destructuring more complex data types. For example, we can pattern match on tuples:

```iex
iex> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
iex> a
:hello
iex> b
"world"
```

A pattern match will error in case the sides can’t match. This is, for example, the case when the tuples have different sizes:

```iex
iex> {a, b, c} = {:hello, "world"}
** (MatchError) no match of right hand side value: {:hello, "world"}
```

And also when comparing different types:

```iex
iex> {a, b, c} = [:hello, "world", !]
** (MatchError) no match of right hand side value: [:hello, "world", !]
```

More interestingly, we can match on specific values. The example below asserts that the left side will only match the right side when the right side is a tuple that starts with the atom `:ok`:

```iex
iex> {:ok, result} = {:ok, 13}
{:ok, 13}
iex> result
13
```

```iex
iex> {:ok, result} = {:error, :oops}
** (MatchError) no match of right hand side value: {:error, :oops}
```

We can pattern match on lists:

```iex
iex> [a, b, c] = [1, 2, 3]
[1, 2, 3]
iex> a
1
```

A list also supports matching on its own head and tail:
Similar to the `hd/1` and `tl/1` functions, we can’t match an empty list with a head and tail pattern:

```elixir
defn [h|t] = []
** (MatchError) no match of right hand side value: []
```

The `[head | tail]` format is not only used on pattern matching but also for prepending items to a list:

```elixir
defn list = [1, 2, 3]
[1, 2, 3]
defn [0|list]
[0, 1, 2, 3]
```

Pattern matching allows developers to easily destructure data types such as tuples and lists. As we will see in following chapters, it is one of the foundations of recursion in Elixir and applies to other types as well, like maps and binaries.

### 4.3 The pin operator

Variables in Elixir can be rebound:

```elixir
defn x = 1
1
defn x = 2
2
```

The pin operator `^` can be used when there is no interest in rebinding a variable but rather in matching against its value prior to the match:

```elixir
defn x = 1
1
defn ^x = 2
** (MatchError) no match of right hand side value: 2
defn {x, ^x} = {2, 1}
{2, 1}
defn x
2
```

Notice that if a variable is mentioned more than once in a pattern, all references should bind to the same pattern:

```elixir
defn {x, x} = {1, 1}
1
defn {x, x} = {1, 2}
** (MatchError) no match of right hand side value: {1, 2}
```

In some cases, you don’t care about a particular value in a pattern. It is a common practice to bind those values to the underscore, `_`. For example, if only the head of the list matters to us, we can assign the tail to underscore:

```elixir
defn [h | _] = [1, 2, 3]
[1, 2, 3]
defn h
1
```

The variable `_` is special in that it can never be read from. Trying to read from it gives an unbound variable error:
Although pattern matching allows us to build powerful constructs, its usage is limited. For instance, you cannot make function calls on the left side of a match. The following example is invalid:

```
%> length([1, [2], 3]) = 3
** (ErlangError) erlang error:illegal_pattern
```

This finishes our introduction to pattern matching. As we will see in the next chapter, pattern matching is very common in many language constructs.

## 1.1.5 5 case, cond and if

### 5.1 case

`case` allows us to compare a value against many patterns until we find a matching one:

```
%> case {1, 2, 3} do
...> {4, 5, 6} ->
...>  "This clause won’t match"
...> {1, x, 3} ->
...>  "This clause will match and bind x to 2 in this clause"
...> _ ->
...>  "This clause would match any value"
...> end
```

If you want to pattern match against an existing variable, you need to use the `^` operator:

```
%> x = 1
1
%> case 10 do
...> ^x -> "Won’t match"
...> _ -> "Will match"
...> end
```

Clauses also allow extra conditions to be specified via guards:

```
%> case {1, 2, 3} do
...> {1, x, 3} when x > 0 ->
...>  "Will match"
...> _ ->
...>  "Won’t match"
...> end
```

The first clause above will only match when `x` is positive. The Erlang VM only allows a limited set of expressions in guards:

- comparison operators (`==, !=, ===, !==, >, <, <=, >=`)
• boolean operators (and, or) and negation operators (not, !)
• arithmetic operators (+, -, *, /)
• <> and ++ as long as the left side is a literal
• the in operator
• all the following type check functions:
  – is_atom/1
  – is_binary/1
  – is_bitstring/1
  – is_boolean/1
  – is_float/1
  – is_function/1
  – is_function/2
  – is_integer/1
  – is_list/1
  – is_map/1
  – is_number/1
  – is_pid/1
  – is_port/1
  – is_reference/1
  – is_tuple/1
• plus these functions:
  – abs(number)
  – bit_size(bitstring)
  – byte_size(bitstring)
  – div(integer, integer)
  – elem(tuple, n)
  – hd(list)
  – length(list)
  – map_size(map)
  – node()
  – node(pid | ref | port)
  – rem(integer, integer)
  – round(number)
  – self()
  – size(tuple | bitstring)
  – tl(list)
- trunc(number)
- tuple_size(tuple)

Keep in mind errors in guards do not leak but simply make the guard fail:

```iex> hd(1)
** (ArgumentError) argument error
  :erlang.hd(1)
```

```iex> case 1 do
    ...> x when hd(x) -> "Won’t match"
    ...> x -> "Got: #{x}"
    ...> end
  "Got 1"
```

If none of the clauses match, an error is raised:

```iex> case :ok do
    ...> :error -> "Won’t match"
    ...> end
** (CaseClauseError) no case clause matching: :ok
```

Note anonymous functions can also have multiple clauses and guards:

```iex> f = fn
    ...> x, y when x > 0 -> x + y
    ...> x, y -> x * y
    ...> end
#Function<12.71889879/2 in :erl_eval.expr/5>
```

```iex> f.(1, 3)
4
```

```iex> f.(-1, 3)
-3
```

The number of arguments in each anonymous function clause needs to be the same, otherwise an error is raised.

### 5.2 `cond`

`case` is useful when you need to match against different values. However, in many circumstances, we want to check different conditions and find the first one that evaluates to true. In such cases, one may use `cond`:

```iex> cond do
    ...> 2 + 2 == 5 ->
    "This will not be true"
    ...> 2 * 2 == 3 ->
    "Nor this"
    ...> 1 + 1 == 2 ->
    "But this will"
    ...> end
  "But this will"
```

This is equivalent to `else if` clauses in many imperative languages (although used way less frequently here).

If none of the conditions return true, an error is raised. For this reason, it may be necessary to add a last condition equal to `true`, which will always match:

```iex> cond do
    ...> 2 + 2 == 5 ->
    "This is never true"
    ...> 2 * 2 == 3 ->
```
Finally, note `cond` considers any value besides `nil` and `false` to be true:

```iex
cond do
  hd([1,2,3]) ->
  "1 is considered as true"
  end
"1 is considered as true"
```

### 5.3 if and unless

Besides `case` and `cond`, Elixir also provides the macros `if/2` and `unless/2` which are useful when you need to check for just one condition:

```iex
if true do
  "This works!"
  end
"This works!"
```

```iex
unless true do
  "This will never be seen"
  end
nil
```

If the condition given to `if/2` returns `false` or `nil`, the body given between `do/end` is not executed and it simply returns `nil`. The opposite happens with `unless/2`.

They also support `else` blocks:

```iex
if nil do
  "This won’t be seen"
else
  "This will"
end
"This will"
```

Note: An interesting note regarding `''if/2''` and `''unless/2''` is that they are implemented as macros in the language; they are special language constructs as they would be in many languages. You can check the documentation and the source of `''if/2''` in `''Kernel'` module docs <docs/stable/elixir/Kernel.html>`__. The `''Kernel'` module is also where operators like `''+/2''` and functions like `''is_function/2''` are defined, all automatically imported and available in your code by default.

### 5.4 do blocks

At this point, we have learned four control structures: `case`, `cond`, `if` and `unless`, and they were all wrapped in `do/end` blocks. It happens we could also write `if` as follows:

```iex
if true, do: 1 + 2
3
```

In Elixir, `do/end` blocks are a convenience for passing a group of expressions to `do:`. These are equivalent:
iex> if true do
...>   a = 1 + 2
...>   a + 10
...> end
13
iex> if true, do: (
...>   a = 1 + 2
...>   a + 10
...> )
13

We say the second syntax is using **keyword lists**. We can pass `else` using this syntax:

iex> if false, do: :this, else: :that :that

It is important to keep one small detail in mind when using `do/end` blocks: they always bind to the farthest function call. For example, the following expression:

iex> is_number if true do
...>   1 + 2
...> end

Would be parsed as:

iex> is_number(if true) do
...>   1 + 2
...> end

Which leads to an undefined function error as Elixir attempts to invoke `is_number/2`. Adding explicit parentheses is enough to resolve the ambiguity:

iex> is_number(if true do
...>   1 + 2
...> end)
true

Keyword lists play an important role in the language and are quite common in many functions and macros. We will explore them a bit more in a future chapter. Now it is time to talk about “Binaries, strings and char lists”.

### 1.1.6 6 Binaries, strings and char lists

- 6.1 UTF-8 and Unicode
- 6.2 Binaries (and bitstrings)
- 6.3 Char lists

In “Basic Types”, we learned about strings and used the `is_binary/1` function for checks:

iex> string = "hello"
"hello"
iex> is_binary string
true

In this chapter, we will understand what binaries are, how they associate with strings, and what a single-quoted value, ‘like this’, means in Elixir.
6.1 UTF-8 and Unicode

A string is a UTF-8 encoded binary. In order to understand exactly what we mean by that, we need to understand the difference between bytes and code points.

The Unicode standard assigns code points to many of the characters we know. For example, the letter a has code point 97 while the letter ł has code point 322. When writing the string "hello" to disk, we need to convert this code point to bytes. If we adopted a rule that said one byte represents one code point, we wouldn’t be able to write "hello", because it uses the code point 322 for ł, and one byte can only represent a number from 0 to 255. But of course, given you can actually read "hello" on your screen, it must be represented somehow. That’s where encodings come in.

When representing code points in bytes, we need to encode them somehow. Elixir chose the UTF-8 encoding as its main and default encoding. When we say a string is a UTF-8 encoded binary, we mean a string is a bunch of bytes organized in a way to represent certain code points, as specified by the UTF-8 encoding.

Since we have code points like ł assigned with the number 322, we actually need more than one byte to represent it. That’s why we see a difference when we calculate the byte_size/1 of a string compared to its String.length/1:

```
iex> string = "hello"
"hello"
iex> byte_size string
7
iex> String.length string
5
```

UTF-8 requires one byte to represent the code points h, e and o, but two bytes to represent ł. In Elixir, you can get a code point’s value by using ?:

```
iex> ?a
97
iex> ?ł
322
```

You can also use the functions in the ‘String’ module to split a string in its code points:

```
iex> String.codepoints("hello")
["h", "e", "ł", "ł", "o"]
```

You will see that Elixir has excellent support for working with strings. It also supports many of the Unicode operations. In fact, Elixir passes all the tests showcased in the “The string type is broken” article.

However, strings are just part of the story. If a string is a binary, and we have used the is_binary/1 function, Elixir must have an underlying type empowering strings. And it does. Let’s talk about binaries!

6.2 Binaries (and bitstrings)

In Elixir, you can define a binary using <<>>:

```
iex> <<0, 1, 2, 3>>
<0, 1, 2, 3>
iex> byte_size <<0, 1, 2, 3>>
4
```

A binary is just a sequence of bytes. Of course, those bytes can be organized in any way, even in a sequence that does not make them a valid string:
Elixir Documentation, Release

```
$ iex String.valid?([239, 191, 191])
false
```

The string concatenation operation is actually a binary concatenation operator:

```
$ iex [[0, 1], [2, 3]]
[0, 1, 2, 3]
```

A common trick in Elixir is to concatenate the null byte `[]<>` to a string to see its inner binary representation:

```
$ iex "hello" <> [0]
[104, 101, 197, 130, 197, 130, 111, 0]
```

Each number given to a binary is meant to represent a byte and therefore must go up to 255. Binaries allow modifiers to be given to store numbers bigger than 255 or to convert a code point to its utf8 representation:

```
$ iex [[255], [255]]
[255]
$ iex [[256], [256]]
# truncated
[0]
$ iex [[256, :size(16)], [256, :utf8]]
# use 16 bits (2 bytes) to store the number
[1, 0]
# the number is a code point
""
$ iex [[256, :utf8, 0]]
[196, 128, 0]
```

If a byte has 8 bits, what happens if we pass a size of 1 bit?

```
$ iex [[1, :size(1)], [1, :size(1)]]
[1, 1]
$ iex [[2, :size(1)], [2, :size(1)]]
# truncated
[0]
$ iex is_binary([[1, :size(1)]])
false
$ iex is_bitstring([[1, :size(1)]])
true
$ iex bit_size([[1, :size(1)]])
1
```

The value is no longer a binary, but a bitstring – just a bunch of bits! So a binary is a bitstring where the number of bits is divisible by 8!

We can also pattern match on binaries / bitstrings:

```
$ iex [[0, 1, x]] = [[0, 1, 2]]
[0, 1, 2]
$ iex x
2
$ iex [[0, 1, x]] = [[0, 1, 2, 3]]
** (MatchError): no match of right hand side value: [[0, 1, 2, 3]]
```

Note each entry in the binary is expected to match exactly 8 bits. However, we can match on the rest of the binary modifier:

```
$ iex [[0, 1, x : binary]] = [[0, 1, 2, 3]]
[0, 1, 2, 3]
$ iex x
[2, 3]
```
The pattern above only works if the binary is at the end of `<<>>`. Similar results can be retrieved with the string concatenation operator `<>`:

```elixir
iex> "he" <> rest = "hello"
"hello"
iex> rest
"llo"
```

This finishes our tour of bitstrings, binaries and strings. A string is a UTF-8 encoded binary, and a binary is a bitstring where the number of bits is divisible by 8. Although this shows the flexibility Elixir provides to work with bits and bytes, 99% of the time you will be working with binaries and using the `is_binary/1` and `byte_size/1` functions.

### 6.3 Char lists

A char list is nothing more than a list of characters:

```elixir
iex> 'hełło'
[104, 101, 322, 322, 111]
iex> is_list 'hełło'
true
iex> 'hełło'
'hełło'
```

You can see that, instead of containing bytes, a char list contains the code points of the characters in between single-quotes (note that iex will only output code points if any of the chars is outside the ASCII range). So while double-quotes represent a string (i.e. a binary), single-quotes represents a char list (i.e. a list).

In practice, char lists are used mostly when interfacing with Erlang, in particular old libraries that do not accept binaries as arguments. You can convert a char list to a string and back by using the `to_string/1` and `to_char_list/1` functions:

```elixir
iex> to_char_list "hełło"
[104, 101, 322, 322, 111]
iex> to_string 'hełło'
"hełło"
iex> to_string :hello
"hello"
iex> to_string 1
"1"
```

Note that those functions are polymorphic. They not only convert char lists to strings, but also integers to strings, atoms to strings, and so on.

With binaries, strings, and char lists out of the way, it is time to talk about key-value data structures.

### 1.1.7 7 Keywords, maps and dicts

- 7.1 Keyword lists
- 7.2 Maps
- 7.3 Dicts

So far we haven’t discussed any associative data structures, i.e. data structures that are able to associate a certain value (or multiple values) to a key. Different languages call these different names like dictionaries, hashes, associative arrays, maps, etc.
In Elixir, we have two main associative data structures: keyword lists and maps. It’s time to learn more about them!

7.1 Keyword lists

In many functional programming languages, it is common to use a list of 2-item tuples as the representation of an associative data structure. In Elixir, when we have a list of tuples and the first item of the tuple (i.e. the key) is an atom, we call it a keyword list:

```
iex> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]
iex> list == [a: 1, b: 2]
true
iex> list[:a]
1
```

As you can see above, Elixir supports a special syntax for defining such lists, and underneath they just map to a list of tuples. Since they are simply lists, all operations available to lists, including their performance characteristics, also apply to keyword lists.

For example, we can use `++` to add new values to a keyword list:

```
iex> list ++ [c: 3]
[a: 1, b: 2, c: 3]
iex> [a: 0] ++ list
[a: 0, a: 1, b: 2]
```

Note that values added to the front are the ones fetched on lookup:

```
iex> new_list = [a: 0] ++ list
[a: 0, a: 1, b: 2]
iex> new_list[:a]
0
```

Keyword lists are important because they have two special characteristics:

- They keep the keys ordered as given by the developer.
- They allow a key to be given more than once.

For example, the Ecto library makes use of both features to provide an elegant DSL for writing database queries:

```
query = from w in Weather,
where: w.prcp > 0,
where: w.temp < 20,
select: w
```

Those features are what prompted keyword lists to be the default mechanism for passing options to functions in Elixir. In chapter 5, when we discussed the `if/2` macro, we mentioned the following syntax is supported:

```
iex> if false, do: :this, else: :that
:that
```

The `do:` and `else:` pairs are keyword lists! In fact, the call above is equivalent to:

```
iex> if(false, [do: :this, else: :that])
:that
```

In general, when the keyword list is the last argument of a function, the square brackets are optional.

In order to manipulate keyword lists, Elixir provides the `Keyword` module. Remember though keyword lists are simply lists, and as such they provide the same linear performance characteristics.
as lists. The longer the list, the longer it will take to find a key, to count the number of items, and so on. For this reason, keyword lists are used in Elixir mainly as options. If you need to store many items or guarantee one-key associates with at maximum one-value, you should use maps instead.

Note we can also pattern match on keyword lists:

```iex
iex> [a: a] = [a: 1]
[a: 1]
iex> a
1
iex> [a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1, b: 2]
iex> [b: b, a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1, b: 2]
```

However this is rarely done in practice since pattern matching on lists require the number of items and their order to match.

### 7.2 Maps

Whenever you need a key-value store, maps are the “go to” data structure in Elixir. A map is created using the `%{}` syntax:

```iex
iex> map = %{a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> map[:a]
1
iex> map[2]
:b
```

Compared to keyword lists, we can already see two differences:

- Maps allow any value as key.
- Maps’ keys do not follow any ordering.

If you pass duplicate keys when creating a map, the last one wins:

```iex
iex> %{1 => 1, 1 => 2}
%{1 => 2}
```

When all the keys in a map are atoms, you can use the keyword syntax for convenience:

```iex
iex> map = %{a: 1, b: 2}
%{a: 1, b: 2}
```

In contrast to keyword lists, maps are very useful with pattern matching:

```iex
iex> %{a} = %{a => 1, 2 => :b}
{:a => 1, 2 => :b}
iex> %{a => a} = %{a => 1, 2 => :b}
{:a => 1, 2 => :b}
iex> a
1
iex> %{c => c} = %{a => 1, 2 => :b}
** (MatchError) no match of right hand side value: %{2 => :b, :a => 1}
```

As shown above, a map matches as long as the given keys exist in the given map. Therefore, an empty map matches all maps.

One interesting property about maps is that they provide a particular syntax for updating and accessing atom keys:
Both access and update syntaxes above require the given keys to exist. For example, the last line failed because there is no :c in the map. This is very useful when you are working with maps where you only expect certain keys to exist.

In future chapters, we will also learn about structs, which provide compile-time guarantees and the foundation for polymorphism in Elixir. Structs are built on top of maps where the update guarantees above are proven to be quite useful.

Manipulating maps is done via the ‘Map’ module, it provides a very similar API to the Keyword module. This is because both modules implement the Dict behaviour.

Note: Maps were recently introduced into the Erlang VM with EEP 43. Erlang 17 provides a partial implementation of the EEP, where only “small maps” are supported. This means maps have good performance characteristics only when storing at maximum a couple of dozens keys. To fill in this gap, Elixir also provides the ‘HashDict’ module which uses a hashing algorithm to provide a dictionary that supports hundreds of thousands keys with good performance.

7.3 Dicts

In Elixir, both keyword lists and maps are called dictionaries. In other words, a dictionary is like an interface (we call them behaviours in Elixir) and both keyword lists and maps modules implement this interface.

This interface is defined in the Dict module which also provides an API that delegates to the underlying implementations:

The Dict module allows any developer to implement their own variation of Dict, with specific characteristics, and hook into existing Elixir code. The Dict module also provides functions that are meant to work across dictionaries. For example, Dict.equal?/2 can compare two dictionaries of any kind.

That said, you may be wondering, which of Keyword, Map or Dict modules should you use in your code? The answer is: it depends.

If your code is expecting a keyword as an argument, explicitly use the Keyword module. If you want to manipulate a map, use the Map module. However, if your API is meant to work with any dictionary, use the Dict module (and make sure to write tests that pass different dict implementations as arguments).

This concludes our introduction to associative data structures in Elixir. You will find out that given keyword lists and maps, you will always have the right tool to tackle problems that require associative data structures in Elixir.

1.1.8 8 Modules
In Elixir we group several functions into modules. We’ve already used many different modules in the previous chapters like the ‘“String’ module: 

```elixir
iex> String.length "hello"
5
```

In order to create our own modules in Elixir, we use the `defmodule` macro. We use the `def` macro to define functions in that module:

```elixir
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

```elixir
iex> Math.sum(1, 2)
3
```

In the following sections, our examples are going to get a bit more complex, and it can be tricky to type them all in the shell. It’s about time for us to learn how to compile Elixir code and also how to run Elixir scripts.

### 8.1 Compilation

Most of the time it is convenient to write modules into files so they can be compiled and reused. Let’s assume we have a file named `math.ex` with the following contents:

```elixir
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

This file can be compiled using `elixirc`:

```bash
elixirc math.ex
```

This will generate a file named `Elixir.Math.beam` containing the bytecode for the defined module. If we start `iex` again, our module definition will be available (provided that `iex` is started in the same directory the bytecode file is in):

```elixir
iex> Math.sum(1, 2)
3
```

Elixir projects are usually organized into three directories:

- `ebin` - contains the compiled bytecode
- `lib` - contains elixir code (usually `.ex` files)
- `test` - contains tests (usually `.exs` files)
When working on actual projects, the build tool called mix will be responsible for compiling and setting up the proper paths for you. For learning purposes, Elixir also supports a scripted mode which is more flexible and does not generate any compiled artifacts.

### 8.2 Scripted mode

In addition to the Elixir file extension .ex, Elixir also supports .exs files for scripting. Elixir treats both files exactly the same way, the only difference is in intention. .ex files are meant to be compiled while .exs files are used for scripting, without the need for compilation. For instance, we can create a file called math.exs:

```elixir
defmodule Math do
def sum(a, b)
do_sum(a, b)
end
defp do_sum(a, b)
a + b
end
end
```

```
IO.puts Math.sum(1, 2)
```

And execute it as:

```
elixir math.exs
```

The file will be compiled in memory and executed, printing “3” as the result. No bytecode file will be created. In the following examples, we recommend you write your code into script files and execute them as shown above.

### 8.3 Named functions

Inside a module, we can define functions with def/2 and private functions with defp/2. A function defined with def/2 can be invoked from other modules while a private function can only be invoked locally.

```
defmodule Math do
def sum(a, b) do
do_sum(a, b)
end
defp do_sum(a, b) do
  a + b
end
end
```

```
Math.sum(1, 2) #=> 3
Math.do_sum(1, 2) #=> */* (UndefinedFunctionError)
```

Function declarations also support guards and multiple clauses. If a function has several clauses, Elixir will try each clause until it finds one that matches. Here is an implementation of a function that checks if the given number is zero or not:

```
defmodule Math do
def zero?(0) do
  true
end
def zero?(x) when is_number(x) do
  false
end
end
```
Math.zero?(0) #=> true
Math.zero?(1) #=> false
Math.zero?([1,2,3]) #=> ** (FunctionClauseError)

Giving an argument that does not match any of the clauses raises an error.

8.4 Function capturing

Throughout this tutorial, we have been using the notation name/arity to refer to functions. It happens that this notation can actually be used to retrieve a named function as a function type. Let’s start iex and run the math.exs file defined above:

```elixir
$ iex math.exs
```

```elixir
iex> Math.zero?(0)
ture
iex> fun = &Math.zero!/1
&M Math.zero!/1
iex> is_function fun
ture
iex> fun.(0)
ture
```

Local or imported functions, like is_function/1, can be captured without the module:

```elixir
iex> &is_function/1
&:erlang.is_function/1
iex> (&is_function/1).(fun)
ture
```

Note the capture syntax can also be used as a shortcut for creating functions:

```elixir
iex> fun = &(&1 + 1)
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> fun.(1)
2
```

The &1 represents the first argument passed into the function. &(&1+1) above is exactly the same as fn x -> x + 1 end. The syntax above is useful for short function definitions. You can read more about the capture operator & in the "Kernel.SpecialForms" documentation `<docs/stable/elixir/Kernel.SpecialForms.html'>__.

8.5 Default arguments

Named functions in Elixir also support default arguments:

```elixir
defmodule Concat do
  def join(a, b, sep \ " ") do
    a <> sep <> b
  end
end
```

```elixir
IO.puts Concat.join("Hello", "world") #=> Hello world
IO.puts Concat.join("Hello", "world", ") #=> Hello world
```
Any expression is allowed to serve as a default value, but it won’t be evaluated during the function definition; it will simply be stored for later use. Every time the function is invoked and any of its default values have to be used, the expression for that default value will be evaluated:

```elixir
defmodule DefaultTest do
  def dowork(x \ IO.puts "hello") do
    x
  end
end
```

```iex
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
:ok
```

If a function with default values has multiple clauses, it is recommended to create a function head (without an actual body), just for declaring defaults:

```elixir
defmodule Concat do
  def join(a, b \ nil, sep \ " ")
    def join(a, b, _sep) when nil?(b) do
      a
    end
    def join(a, b, sep) do
      a <> sep <> b
    end
  end
end
```

```iex
IO.puts Concat.join("Hello", "world")     #=> Hello world
IO.puts Concat.join("Hello", "world", ",") #=> Hello_world
IO.puts Concat.join("Hello")              #=> Hello
```

When using default values, one must be careful to avoid overlapping function definitions. Consider the following example:

```elixir
defmodule Concat do
  def join(a, b) do
    IO.puts "***First join"
    a <> b
  end
  def join(a, b, sep \ " ") do
    IO.puts "***Second join"
    a <> sep <> b
  end
end
```

If we save the code above in a file named “concat.ex” and compile it, Elixir will emit the following warning:

```
concat.exs:7: this clause cannot match because a previous clause at line 2 always matches
```

The compiler is telling us that invoking the `join` function with two arguments will always choose the first definition of `join` whereas the second one will only be invoked when three arguments are passed:

```
$ iex concat.exs
```
This finishes our short introduction to modules. In the next chapters, we will learn how to use named functions for recursion, explore Elixir lexical directives that can be used for importing functions from other modules and discuss module attributes.

## 1.1.9 9 Recursion

Due to immutability, loops in Elixir (and in functional programming languages) are written differently from conventional imperative languages. For example, in an imperative language, one would write:

```plaintext
for(i = 0; i < array.length; i++) {
    array[i] = array[i] * 2
}
```

In the example above, we are mutating the array and the helper variable `i`. That’s not possible in Elixir. Instead, functional languages rely on recursion: a function is called recursively until a condition is reached that stops the recursive action from continuing. Consider the example below that prints a string an arbitrary amount of times:

```elixir
defmodule Recursion do
  def print_multiple_times(msg, n) when n <= 1 do
    IO.puts msg
  end

  def print_multiple_times(msg, n) do
    IO.puts msg
    print_multiple_times(msg, n - 1)
  end
end
```

```plaintext
Recursion.print_multiple_times("Hello!", 3)
# Hello!
# Hello!
# Hello!
```

Similar to case, a function may have many clauses. A particular clause is executed when the arguments passed to the function match the clause’s argument patterns and its guard evaluates to `true`.

Above when `print_multiple_times/2` is initially called, the argument `n` is equal to `3`.

The first clause has a guard which says use this definition if and only if `n` is less than or equal to `1`. Since this is not the case, Elixir proceeds to the next clause’s definition.

The second definition matches the pattern and has no guard so it will be executed. It first prints our `msg` and then calls itself passing `n - 1` (2) as the second argument. Our `msg` is printed and `print_multiple_times/2` is called again this time with the second argument set to `1`.

Because `n` is now set to `1`, the guard to our first definition of `print_multiple_times/2` evaluates to `true`, and we execute this particular definition. The `msg` is printed, and there is nothing left to execute.

We defined `print_multiple_times/2` so that no matter what number is passed as the second argument it either triggers our first definition (known as a “base case”) or it triggers our second definition which will ensure that we get exactly 1 step closer to our base case.
Let’s now see how we can use the power of recursion to sum a list of numbers.

```elixir
defmodule Math do
def sum_list([head|tail], accumulator) do
  sum_list(tail, head + accumulator)
end

def sum_list([], accumulator) do
  accumulator
end
end

Math.sum_list([1, 2, 3], 0) #=> 6
```

We invoke `sum_list` with a list `[1,2,3]` and the initial value 0 as arguments. We will try each clause until we find one that matches according to the pattern matching rules. In this case, the list `[1,2,3]` matches against `[head|tail]` which assigns `head = 1` and `tail = [2,3]` while `accumulator` is set to 0.

Then, we add the head of the list to the accumulator `head + accumulator` and call `sum_list` again, recursively, passing the tail of the list as its first argument. The tail will once again match `[head|tail]` until the list is empty, as seen below:

```
sum_list [1, 2, 3], 0
sum_list [2, 3], 1
sum_list [3], 3
sum_list [], 6
```

When the list is empty, it will match the final clause which returns the final result of 6.

The process of taking a list and “reducing” it down to one value is known as a “reduce” algorithm and is central to functional programming.

What if we instead want to double all of the values in our list?

```elixir
defmodule Math do
def double_each([head|tail]) do
  [head * 2| double_each(tail)]
end

def double_each([]) do
  []
end
end
```

Here we have used recursion to traverse a list doubling each element and returning a new list. The process of taking a list and “mapping” over it is known as a “map” algorithm.

Recursion and tail call optimization are an important part of Elixir and are commonly used to create loops. However, when programming Elixir you will rarely use recursion as above to manipulate lists.

The ‘`Enum` module </docs/stable/elixir/Enum.html>`__, which we are going to study in the next chapter, already provides many conveniences for working with lists. For instance, the examples above could be written as:

```
iex> Enum.reduce([1, 2, 3], 0, fn(x, acc) -> x + acc end) 6
iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end) [2, 4, 6]
```

Or, using the capture syntax:
So let’s take a deeper look at Enumerables and Streams.

1.1.10 10 Enumerables and Streams

- 10.1 Enumerables
- 10.2 Eager vs Lazy
- 10.3 Streams

10.1 Enumerables

Elixir provides the concept of enumerables and the `Enum` module to work with them. We have already learned two enumerables: lists and maps.

```iex
iex> Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]
```

The `Enum` module provides a huge range of functions to transform, sort, group, filter and retrieve items from enumerables. It is one of the modules developers use frequently in their Elixir code.

Elixir also provides ranges:

```iex
iex> Enum.map(1..3, fn x -> x * 2 end)
[2, 4, 6]
```

Since the `Enum` module was designed to work across different data types, its API is limited to functions that are useful across many data types. For specific operations, you may need to reach to modules specific to the data types. For example, if you want to insert an element at a given position in a list, you should use the `List.insert_at/3` function from the `List` module, as it would make little sense to insert a value into, for example, a range.

We say the functions in the `Enum` module are polymorphic because they can work with diverse data types. In particular, the functions in the `Enum` module can work with any data type that implements the `Enumerable` protocol. We are going to discuss Protocols in a later chapter, for now we are going to move on to a specific kind of enumerable called streams.

10.2 Eager vs Lazy

All the functions in the `Enum` module are eager. Many functions expect an enumerable and return a list back:

```iex
iex> odd? = &((rem(&1, 2) != 0)
#Function<6.80484245/1 in :erl_eval.expr/5>
```

```iex
iex> Enum.filter(1..3, odd?)
[1, 3]
```

Elixir Documentation, Release
This means that when performing multiple operations with `Enum`, each operation is going to generate an intermediate list until we reach the result:

```iex
1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
7500000000
```

The example above has a pipeline of operations. We start with a range and then multiply each element in the range by 3. This first operation will now create and return a list with 100_000 items. Then we keep all odd elements from the list, generating a new list, now with 50_000 items, and then we sum all entries.

As an alternative, Elixir provides the `Stream` module which supports lazy operations:

```iex
1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
7500000000
```

Instead of generating intermediate lists, streams create a series of computations that are invoked only when we pass it to the `Enum` module. Streams are useful when working with large, possibly infinite, collections.

### 10.3 Streams

Streams are lazy, composable enumerables.

They are lazy because, as shown in the example above, `1..100_000 |> Stream.map(&(&1 * 3))` returns a data type, an actual stream, that represents the map computation over the range `1..100_000`:

```iex
1..100_000 |> Stream.map(&(&1 * 3))
#Stream<1..100_000, funs: [Function<34.16982430/1 in Stream.map/2>]>]
```

Furthermore, they are composable because we can pipe many stream operations:

```iex
1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?)
#Stream<1..100_000, funs: [...]>]
```

Many functions in the `Stream` module accept any enumerable as argument and return a stream as result. It also provides functions for creating streams, possibly infinite. For example, `Stream.cycle/1` can be used to create a stream that cycles a given enumerable infinitely. Be careful to not call a function like `Enum.map/2` on such streams, as they would cycle forever:

```iex
stream = Stream.cycle([1, 2, 3])
#Function<15.16982430/2 in Stream.cycle/1>]
```

```iex
Enum.take(stream, 10)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

On the other hand, `Stream.unfold/2` can be used to generate values from a given initial value:

```iex
stream = Stream.unfold("hełło", &String.next_codepoint/1)
#Function<15.16982430/2 in Stream.unfold/1>]
```

```iex
Enum.take(stream, 3)
["h", "e", "ł"]
```

Another interesting function is `Stream.resource/3` which can be used to wrap around resources, guaranteeing they are opened right before enumeration and closed afterwards, even in case of failures. For example, we can use it to stream a file:

```iex
stream = File.stream!("path/to/file")
#Function<18.16982430/2 in Stream.resource/3>]
```

```iex
Enum.take(stream, 10)
```
The example above will fetch the first 10 lines of the file you have selected. This means streams can be very useful for handling large files or even slow resources like network resources.

The amount of functions and functionality in `Enum <docs/stable/elixir/Enum.html>` and `Stream <docs/stable/elixir/Stream.html>` modules can be daunting at first but you will get familiar with them case by case. In particular, focus on the Enum module first and only move to Stream for the particular scenarios where laziness is required to either deal with slow resources or large, possibly infinite, collections.

Next we’ll look at a feature central to Elixir, Processes, which allows us to write concurrent, parallel and distributed programs in an easy and understandable way.

### 1.1.11 11 Processes

- 11.1 spawn
- 11.2 send and receive
- 11.3 Links

In Elixir, all code runs inside processes. Processes are isolated from each other, run concurrent to one another and communicate via message passing. Processes are not only the basis for concurrency in Elixir, but they also provide the means for building distributed and fault-tolerant programs.

Elixir’s processes should not be confused with operating system processes. Processes in Elixir are extremely lightweight in terms of memory and CPU (unlike threads in many other programming languages). Because of this, it is not uncommon to have dozens of thousands of processes running simultaneously.

In this chapter, we will learn about the basic constructs for spawning new processes, as well as sending and receiving messages between different processes.

#### 11.1 spawn

The basic mechanism for spawning new processes is with the auto-imported `spawn/1` function:

```iex
iex> spawn fn -> 1 + 2 end
#PID<0.43.0>
```

`spawn/1` takes a function which it will execute in another process.

Notice `spawn/1` returns a PID (process identifier). At this point, the process you spawned is very likely dead. The spawned process will execute the given function and exit after the function is done:

```iex
iex> pid = spawn fn -> 1 + 2 end
#PID<0.44.0>
iex> Process.alive?(pid)
false
```

**Note:** you will likely get different process identifiers than the ones we are getting in this guide.

We can retrieve the PID of the current process by calling `self/0`:

```iex
iex> self()
#PID<0.41.0>
iex> Process.alive?(self())
true
```

Processes get much more interesting when we are able to send and receive messages.
11.2 send and receive

We can send messages to a process with `send/2` and receive them with `receive/1`:

```iex
iex> send self(), {:hello, "world"}
{:hello, "world"}
iex>
```

```iex
receive do
  ...> {:hello, msg} -> msg
  ...> {:world, msg} -> "won’t match"
  ...> end
"world"
```

When a message is sent to a process, the message is stored in the process mailbox. The `receive/2` block goes through the current process mailbox searching for a message that matches any of the given patterns. `receive/1` supports many clauses, like `case/2`, as well as guards in the clauses.

If there is no message in the mailbox matching any of the patterns, the current process will wait until a matching message arrives. A timeout can also be specified:

```iex
receive do
  ...> {:hello, msg} -> msg
  ...> after 1_000 -> "nothing after 1s"
  ...> end
"nothing after 1s"
```

A timeout of 0 can be given when you already expect the message to be in the mailbox.

Let’s put all together and send messages in between processes:

```iex
parent = self()
#PID<0.41.0>
iex> spawn fn -> send(parent, {:hello, self()}) end
#PID<0.48.0>
iex> receive do
  ...> {:hello, pid} -> "Got hello from #{inspect pid}"
  ...> end
"Got hello from #PID<0.48.0>"
```

While in the shell, you may find the helper `flush/0` quite useful. It flushes and prints all the messages in the mailbox.

```iex
send self(), :hello
:hello
flush()
:hello
:ok
```

Before we finish this chapter, let’s talk about process links.

11.3 Links

The most common form of spawning in Elixir is actually via `spawn_link/1`. Before we show an example with `spawn_link/1`, let’s try to see what happens when a process fails:

```iex
spawn fn -> raise "oops" end
#PID<0.58.0>
```

Well... nothing happened. That’s because processes are isolated. If we want the failure in one process to propagate to another one, we should link them. This can be done with `spawn_link/1`:

1.1. Introduction to Elixir
When a failure happens in the shell, the shell automatically traps the failure and shows it nicely formatted. In order to understand what would really happen in our code, let’s use `spawn_link/1` inside a file and run it:

```
# spawn.exs
spawn_link fn -> raise "oops" end
receive do
  :hello -> "let’s wait until the process fails"
end
```

This time the process failed and brought the parent process down as they are linked. Linking can also be done manually by calling `Process.link/2`. We recommend you to take a look at the ‘‘Process’’ module `/docs/stable/elixir/Process.html` for other functionality provided by processes.

Process and links play an important role when building fault-tolerant systems. In Elixir applications we often link our processes to supervisors which will detect when a process die and start a new process in its place. This is only possible because processes don’t share anything by default so there is no way the crash of a process can corrupt the state of another process!

While other languages would require us to catch/handle exceptions, in Elixir we are actually fine with letting processes fail because we expect supervisors to properly restart our systems. “Failing fast” is a common philosophy when writing Elixir software!

Next we’ll explore the world of IO.

### 1.1.12 12 IO

This chapter is a quick introduction to input/output mechanisms in Elixir and related modules, like ‘‘IO’’ `/docs/stable/elixir/IO.html`‘, ‘‘File’’ `/docs/stable/elixir/File.html`‘ and ‘‘Path’’ `/docs/stable/elixir/Path.html`‘.

We had originally sketched this chapter to come much earlier in the getting started guide. However, we noticed the IO system provides a great opportunity to shed some light on some philosophies and curiosities of Elixir and the VM.

#### 12.1 The IO module

The IO module in Elixir is the main mechanism for reading and writing to the standard io (:stdio), standard error (:stderr), files and other IO devices. Usage of the module is pretty straight-forward:

```
iex> IO.puts "hello world"
"hello world"
:ok
iex> IO.gets "yes or no? 
yes or no? yes
"yes\n"
```
By default, the functions in the IO module use the standard input and output. We can pass the :stderr as argument to write to the standard error device:

```iex
IO.puts :stderr, "hello world"
"hello world"
:ok
```

### 12.2 The File module

The `File` module contains functions that allows us to open files as IO devices. By default, files are opened in binary mode, which requires developers to use the specific `IO.binread/2` and `IO.binwrite/2` functions from the IO module:

```iex
{:ok, file} = File.open "hello", [:write]
{:ok, #PID<0.47.0>}
IO.binwrite file, "world"
:ok
File.close file
:ok
File.read "hello"
{:ok, "world"
```

A file can also be opened with :utf8 encoding which allows the remaining functions in the IO module to be used:

```iex
{:ok, file} = File.open "another", [:write, :utf8]
{:ok, #PID<0.48.0>}
```

Besides functions for opening, reading and writing files, the File module has many functions that work on the file system. Those functions are named after their UNIX equivalents. For example, `File.rm/1` can be used to remove files, `File.mkdir/1` to create directories, `File.mkdir_p/1` creates directories guaranteeing their parents exists and there is even `File.cp_r/2` and `File.rm_rf/2` which copy and remove files and directories recursively.

You will also notice that functions in the File module have two variants, one with ! (bang) in its name and others without. For example, when we read the “hello” file above, we have used the one without !. Let’s try some new examples:

```iex
File.read "hello"
{:ok, "world"
File.read! "hello"
"world"
File.read "unknown"
{:error, noent}
File.read! "unknown"
** (File.Error) could not read file unknown: no such file or directory
```

Notice that when the file does not exist, the version with ! raises an error. That said, the version without ! is preferred when you want to handle different outcomes with pattern matching. However, if you expect the file to be there, the bang variation is more useful as it raises a meaningful error message. That said, never write:

```iex
{:ok, body} = File.read(file)
```

Instead write:

```iex
case File.read(file) do
  {:ok, body} -> # handle ok
  {:error, r} -> # handle error
end
```

or
File.read!(file)

12.3 The Path module

The majority of the functions in the File module expects paths as arguments. Most commonly, those paths will be binaries and they can be manipulated with the ‘Path <docs/stable/elixir/Path.html>‘ module:

```iex
iex> Path.join("foo", "bar")
"foo/bar"
iex> Path.expand("~/hello")
"/Users/jose/hello"
```

With this we have covered the main modules for doing IO and interacting with the file system. Next we will discuss some curiosities and advanced topics regarding IO. Those sections are not necessary to write Elixir code, so feel free to skip them, but they do provide an overview of how the IO system is implemented in the VM and other curiosities.

12.4 Processes and group leaders

You may have noticed that File.open/2 returned a tuple containing a PID:

```iex
iex> {:ok, file} = File.open "hello", [:write]
{:ok, #PID<0.47.0>}
```

That’s because the IO module actually works with processes. When you say IO.write(pid, binary), the IO module will send a message to the process with the desired operation. Let’s see what happens if we use our own process:

```iex
iex> pid = spawn fn -> ...
...> receive do: (msg -> IO.inspect msg)
...> end
#PID<0.57.0>
iex> IO.write(pid, "hello")
{:io_request, #PID<0.41.0>, #PID<0.57.0>, {:put_chars, :unicode, "hello")}
** (ErlangError) erlang error: :terminated
```

After IO.write/2, we can see the request sent by the IO module printed, which then fails since the IO module expected some kind of result, which we did not supply.

The ‘StringIO <docs/stable/elixir/StringIO.html>‘ module provides an implementation of the IO device messages on top of a string:

```iex
iex> {:ok, pid} = StringIO.open("hello")
{:ok, #PID<0.43.0>}
iex> IO.read(pid, 2)
"he"
```

By modelling IO devices with processes, the Erlang VM allows different nodes in the same network to exchange file processes and read/write files in between nodes. Of all IO devices, there is one that is special to each process, called group leader.

When you write to :stdio, you are actually sending a message to the group leader, which writes to :stdio:

```iex
iex> IO.puts :stdio, "hello"
hello
:ok
iex> IO.puts Process.group_leader, "hello"
```
hello
:ok

The group leader can be configured per process and is used in different situations, for example, when executing code in remote nodes, it guarantees messages printed in the remote node are redirected and printed in the “executor” one.

12.5 iodata and chardata

In all examples above, we have used binaries/strings when writing to files. In the chapter “Binaries, strings and char lists”, we mentioned how strings are simply bytes while char lists are lists with code points.

The functions in IO and File also allow lists to be given as arguments. Not only that, they also allow a mixed list of lists, integers and binaries to be given:

iex> IO.puts 'hello world'
hello world
:ok
iex> IO.puts ['hello', ?, \s, "world"]
hello world
:ok

However, this requires some attention. A list may represent either a bunch of bytes or a bunch of characters and which one to use depends on the encoding of the IO device. If the file is opened without encoding, the file is expected to be in raw mode, and the functions in the IO module starting with bin* must be used. Those functions expect an iodata as argument, i.e. it expects a list of integers representing bytes and binaries to be given.

On the other hand, :stdio and files opened with :utf8 encoding work with the remaining functions in the IO module and those expect a char_data as argument, i.e. they expect a list of characters or strings to be given.

Although this is a subtle difference, you only need to worry about those details if you intend to pass lists to those functions. Binaries are already represented by the underlying bytes and as such their representation is always raw.

This finishes our tour on IO devices and IO related functionality. We have learned about four Elixir modules, `IO <docs/stable/elixir/IO.html>`__, `File <docs/stable/elixir/File.html>`__, `Path <docs/stable/elixir/Path.html>`__ and `StringIO <docs/stable/elixir/StringIO.html>`__, as well as how the VM uses processes for the underlying IO mechanisms and how to use (char and io) data for IO operations.

1.1.13 13 alias, require and import

In order to facilitate software reuse, Elixir provides three directives. As we are going to see below, they are called directives because they have lexical scope.

13.1 alias

alias allows you to set up aliases for any given module name. Imagine our Math module uses a special list implementation for doing math specific operations:
defmodule Math do
  alias Math.List, as: List
end

From now on, any reference to List will automatically expand to Math.List. In case one wants to access the original List, it can be done by accessing the module via Elixir:

List.flatten #=> uses Math.List.flatten
Elixir.List.flatten #=> uses List.flatten
Elixir.Math.List.flatten #=> uses Math.List.flatten

Note: All modules defined in Elixir are defined inside a main Elixir namespace. However, for convenience, you can omit "Elixir." when referencing them.

Aliases are frequently used to define shortcuts. In fact, calling alias without an as option sets the alias automatically to the last part of the module name, for example:

alias Math.List

Is the same as:

alias Math.List, as: List

Note that alias is lexically scoped, which allows you to set aliases inside specific functions:

defmodule Math do
  def plus(a, b) do
    alias Math.List
    # ...
  end

  def minus(a, b) do
    # ...
  end
end

In the example above, since we are invoking alias inside the function plus/2, the alias will just be valid inside the function plus/2. minus/2 won’t be affected at all.

13.2 require

Elixir provides macros as a mechanism for meta-programming (writing code that generates code).

Macros are chunks of code that are executed and expanded at compilation time. This means, in order to use a macro, we need to guarantee its module and implementation are available during compilation. This is done with the require directive:

iex> Integer.odd?(3)
** (CompileError) iex:1: you must require Integer before invoking the macro Integer.odd?/1
iex> require Integer
nil
iex> Integer.odd?(3)
true

In Elixir, Integer.odd?/1 is defined as a macro so it can be used as guards. This means that, in order to invoke Integer.odd?/1, we need to first require the Integer module.
In general a module does not need to be required before usage, except if we want to use the macros available in that module. An attempt to call a macro that was not loaded will raise an error. Note that like the alias directive, require is also lexically scoped. We will talk more about macros in a later chapter.

13.3 import

We use import whenever we want to easily access functions or macros from other modules without using the qualified name. For instance, if we want to use the duplicate function from List several times, we can simply import it:

```
import List, only: [duplicate: 2]
```

```
il
duplicate :ok, 3
[:ok, :ok, :ok]
```

In this case, we are importing only the function duplicate (with arity 2) from List. Although only: is optional, its usage is recommended. except could also be given as an option.

import also supports :macros and :functions to be given to :only. For example, to import all macros, one could write:

```
import Integer, only: :macros
```

Or to import all functions, you could write:

```
import Integer, only: :functions
```

Note that import is also lexically scoped, this means we can import specific macros inside specific functions:

```
defmodule Math do
  def some_function do
    import List, only: [duplicate: 2]
    # call duplicate
  end
end
```

In the example above, the imported List.duplicate/2 is only visible within that specific function. duplicate/2 won’t be available in any other function in that module (or any other module for that matter).

Note that importing a module automatically requires it.

13.4 Aliases

At this point, you may be wondering, what exactly an Elixir alias is and how it is represented?

An alias in Elixir is a capitalized identifier (like String, Keyword, etc) which is converted to an atom during compilation. For instance, the String alias translates by default to the atom :"Elixir.String":

```
import String
true
```

```
EXPORT
"Elixir.String"
```

```
String
```

By using the alias/2 directive, we are simply changing what an alias translates to.

Aliases work as described because in the Erlang VM (and consequently Elixir), modules are represented by atoms. For example, that’s the mechanism we use to call Erlang modules:
This is also the mechanism that allows us to dynamically call a given function in a module:

```iex
mod = :lists
mod.flatten([1, [2], 3])
```

In other words, we are simply calling the function `flatten` on the atom `:lists`.

### 13.5 Nesting

Now that we have talked about aliases, we can talk about nesting and how it works in Elixir. Consider the following example:

```elixir
defmodule Foo do
  defmodule Bar do
  end
end
```

The example above will define two modules `Foo` and `Foo.Bar`. The second can be accessed as `Bar` inside `Foo` as long as they are in the same lexical scope. If later the developer decides to move `Bar` to another file, it will need to be referenced by its full name (`Foo.Bar`) or an alias needs to be set using the `alias` directive discussed above.

In other words, the code above is exactly the same as:

```elixir
defmodule Elixir.Foo do
  defmodule Elixir.Foo.Bar do
  end
  alias Elixir.Foo.Bar, as: Bar
end
```

As we will see in later chapters, aliases also play a crucial role in macros, to guarantee they are hygienic. With this we are almost finishing our tour about Elixir modules, the last topic to cover is module attributes.

### 1.1.14 14 Module attributes

- 14.1 As annotations
- 14.2 As constants
- 14.3 As temporary storage

Module attributes in Elixir serve three purposes:

1. They serve to annotate the module, often with information to be used by the user or the VM.
2. They work as constants.
3. They work as a temporary module storage to be used during compilation.

Let’s check each case, one by one.
14.1 As annotations

Elixir brings the concept of module attributes from Erlang. For example:

```elixir
defmodule MyServer do
  @vsn 2
end
```

In the example above, we are explicitly setting the version attribute for that module. `@vsn` is used by the code reloading mechanism in the Erlang VM to check if a module has been updated or not. If no version is specified, the version is set to the MD5 checksum of the module functions.

Elixir has a handful of reserved attributes. Here are just a few of them, the most commonly used ones:

- `@moduledoc` - provides documentation for the current module.
- `@doc` - provides documentation for the function or macro that follows the attribute.
- `@behaviour` - (notice the British spelling) used for specifying an OTP or user-defined behaviour.
- `@before_compile` - provides a hook that will be invoked before the module is compiled. This makes it possible to inject functions inside the module exactly before compilation.

`@moduledoc` and `@doc` are by far the most used attributes, and we expect you to use them a lot. Elixir treats documentation as first-class and provides many functions to access documentation:

```elixir
iex> defmodule MyModule do ...
  ...> @moduledoc "It does **x**"
  ...> @doc "Returns the version"
  ...> def version, do: 1
  ...> end
  {}
```

Elixir promotes the use of markdown with heredocs to write readable documentation. Heredocs are multiline strings, they start and end with triple quotes, keeping the formatting of the inner text:

```elixir
defmodule Math do
  @moduledoc "This module provides mathematical functions as sin, cos and constants like pi."
  
  # Examples
  
  Math.pi
  #> 3.1415...

end
```

1.1. Introduction to Elixir
We also provide a tool called ExDoc which is used to generate HTML pages from the documentation.

You can take a look at the docs for Module for a complete list of supported attributes. Elixir also uses attributes to define typespecs, via:

- `@spec` - provides a specification for a function.
- `@callback` - provides a specification for the behavior callback.
- `@type` - defines a type to be used in `@spec`.
- `@typep` - defines a private type to be used in `@spec`.
- `@opaque` - defines an opaque type to be used in `@spec`.

This section covers built-in attributes. However, attributes can also be used by developers or extended by libraries to support custom behaviour.

### 14.2 As constants

Elixir developers will often use module attributes to be used as constants:

```elixir
defmodule MyServer do
  @initial_state %{host: "147.0.0.1", port: 3456}
  IO.inspect @initial_state
end
```

Note: Unlike Erlang, user defined attributes are not stored in the module by default. The value exists only during compilation time. A developer can configure an attribute to behave closer to Erlang by calling ```Module.register_attribute/3``` [docs/stable/elixir/Module.html#register_attribute/3].

Trying to access an attribute that was not defined will print a warning:

```elixir
defmodule MyServer do
  @unknown
end
```

warning: undefined module attribute @unknown, please remove access to @unknown or explicitly set it to nil before access

Finally, attributes can also be read inside functions:

```elixir
defmodule MyServer do
  @my_data 14
  def first_data, do: @my_data
  @my_data 13
  def second_data, do: @my_data
end
```

```
MyServer.first_data #=> 14
MyServer.second_data #=> 13
```

Notice that reading an attribute inside a function takes a snapshot of its current value. In other words, the value is read at compilation time and not at runtime. As we are going to see, this makes attributes useful to be used as storage during module compilation.

### 14.3 As temporary storage

One of the projects in the Elixir organization is the ‘‘Plug’ project [https://github.com/elixir-lang/plug]’, which is meant to be a common foundation for building web libraries and frameworks in Elixir.
The Plug library also allows developers to define their own plugs which can be run in a web server:

```elixir
defmodule MyPlug do
  use Plug.Builder

  plug :set_header
  plug :send_ok

  def set_header(conn, _opts) do
    put_resp_header(conn, "x-header", "set")
  end

  def send_ok(conn, _opts) do
    send(conn, 200, "ok")
  end
end
```

```erlang
IO.puts "Running MyPlug with Cowboy on http://localhost:4000"
```

In the example above, we have used the `plug/1` macro to connect functions that will be invoked when there is a web request. Internally, every time you call `plug/1`, the Plug library stores the given argument in a `@plugs` attribute. Just before the module is compiled, Plug runs a callback that defines a method (call/2) which handles http requests. This method will run all plugs inside `@plugs` in order.

In order to understand the underlying code, we’d need macros, so we will revisit this pattern in the meta-programming guide. However the focus here is exactly on how using module attributes as storage allow developers to create DSLs.

Another example comes from the ExUnit framework which uses module attributes as annotation and storage:

```elixir
defmodule MyTest do
  use ExUnit.Case

  @tag :external
  test "contacts external service" do
    # ...
  end
end
```

Tags in ExUnit are used to annotate tests. Tags can be later used to filter tests. For example, you can avoid running external tests on your machine because they are slow and dependent on other services, while they can still be enabled in your build system.

We hope this section shines some light on how Elixir supports meta-programming and how module attributes play an important role when doing so.

In the next chapters we’ll explore structs and protocols before moving to exception handling and other constructs like sigils and comprehensions.

### 1.1.15 15 Structs

In early chapters, we have learned about maps:

```iex
  map = %{a: 1, b: 2}
  map
  map[:a]
  map[a]
```

```iex
  map = %{a: 1, b: 2}
  map[:a]
  map[a]
```

```iex
  map = %{a: 1, b: 2}
  map[a]
```

```iex
  map = %{a: 3, b: 2}
  map[a]
```
Structs are extensions on top of maps that bring default values, compile-time guarantees and polymorphism into Elixir.

To define a struct, we just need to call `defstruct/1` inside a module:

```elixir
defmodule User do  
  ...
  defstruct name: "jose", age: 27
  ...
end
```

We can now create “instances” of this struct by using the `%User{}` syntax:

```elixir
iex> %User{}
%User{name: "jose", age: 27}
iex> %User{name: "eric"}
%User{name: "eric", age: 27}
iex> is_map(%User{})
true
```

Structs give compile-time guarantees that the provided fields exist in the struct:

```elixir
iex> %User{oops: :field}
** (CompileError) iex:3: unknown key :oops for struct User
```

When discussing maps, we demonstrated how we can access and update existing fields of a map. The same applies to structs:

```elixir
iex> jose = %User{}
%User{name: "jose", age: 27}
iex> jose.name
"jose"
iex> eric = %{jose | name: "eric"}
%User{name: "eric", age: 27}
iex> %{eric | oops: :field}
** (ArgumentError) argument error
```

By using the update syntax, the VM is aware no new keys will be added to the map/struct, allowing the maps to share their structure in memory. In the example above, both `jose` and `eric` share the same key structure in memory.

Structs can also be used in pattern matching and they guarantee the structs are of the same type:

```elixir
iex> %User{name: name} = jose
%User{name: "jose", age: 27}
iex> name
"jose"
iex> %User{} = {}  
** (MatchError) no match of right hand side value: {}
```

Matching works because structs store a field named `__struct__` inside the map:

```elixir
iex> jose.__struct__
User
```

Overall, a struct is just a bare map with default fields. Notice we say it is a bare map because none of the protocols implemented for maps are available for structs. For example, you can’t enumerate nor access a struct:

```elixir
iex> user = %User{}
%User{name: "jose", age: 27}
iex> user[:name]
** (Protocol.UndefinedError) protocol Access not implemented for %User{age: 27, name: "jose"}
```
A struct also is not a dictionary and therefore can’t be used with the `Dict` module:

```
iex> Dict.get(%User{}, :name)
** (ArgumentError) unsupported dict: %User{name: "jose", age: 27}
```

We will cover how structs interacts with protocols in the next chapter.

### 1.1.16 16 Protocols

- 16.1 Protocols and structs
- 16.2 Falling back to Any
- 16.3 Built-in protocols

Protocols are a mechanism to achieve polymorphism in Elixir. Dispatching on a protocol is available to any data type as long as it implements the protocol. Let’s see an example.

In Elixir, only `false` and `nil` are treated as false. Everything else evaluates to true. Depending on the application, it may be important to specify a `blank?` protocol that returns a boolean for other data types that should be considered blank. For instance, an empty list or an empty binary could be considered blanks.

We could define this protocol as follows:

```elixir
defprotocol Blank do
  @doc "Returns true if data is considered blank/empty"
  def blank?(data)
end
```

The protocol expects a function called `blank?` that receives one argument to be implemented. We can implement this protocol for different Elixir data types as follows:

```elixir
# Integers are never blank
defimpl Blank, for: Integer do
  def blank?(_), do: false
end

# Just empty list is blank
defimpl Blank, for: List do
  def blank?([],), do: true
  def blank?(_), do: false
end

# Just empty map is blank
defimpl Blank, for: Map do
  def blank?(map), do: map_size(map) == 0
end

# Just the atoms false and nil are blank
defimpl Blank, for: Atom do
  def blank?(false), do: true
  def blank?(nil), do: true
  def blank?(_), do: false
end
```

And we would do so for all native data types. The types available are:

### 1.1. Introduction to Elixir
Elixir Documentation, Release

• Atom
• BitString
• Float
• Function
• Integer
• List
• Map
• PID
• Port
• Reference
• Tuple
Now with the protocol defined and implementations in hand, we can invoke it:
iex> Blank.blank?(0)
false
iex> Blank.blank?([])
true
iex> Blank.blank?([1, 2, 3])
false

Passing a data type that does not implement the protocol raises an error:
iex> Blank.blank?("hello")
** (Protocol.UndefinedError) protocol Blank not implemented for "hello"

16.1 Protocols and structs
The power of Elixir’s extensibility comes when protocols and structs are used together.
In the previous chapter, we have learned that although structs are maps, they do not share protocol implementations
with maps. Let’s define a User struct as in the previous chapter:
iex> defmodule User do
...>
defstruct name: "jose", age: 27
...> end
{:module, User,
<<70, 79, 82, ...>>, {:__struct__, 0}}

And then check:
iex> Blank.blank?(%{})
true
iex> Blank.blank?(%User{})
** (Protocol.UndefinedError) protocol Blank not implemented for %User{age: 27, name: "jose"}

Instead of sharing protocol implementation with maps, structs require their own protocol implementation:
defimpl Blank, for: User do
def blank?(_), do: false
end

48

Chapter 1. Getting started guides


If desired you could come up with your own semantics for a user being blank. Not only that, you could use structs to build more robust data types, like queues, and implemented all relevant protocols, like Enumerable and possibly Blank for such data type.

In many cases though, developers may want to provide a default implementation for structs, as explicitly implementing the protocol for all structs can be tedious. That’s when falling back to Any comes in handy.

### 16.2 Falling back to Any

It may be convenient to provide a default implementation for all types. This can be achieved by setting @fallback_to_any to true in the protocol definition:

```elixir
defprotocol Blank do
  @fallback_to_any true
  def blank?(data)
end
```

Which can now be implemented as:

```elixir
defimpl Blank, for: Any do
  def blank?(_), do: false
end
```

Now all data types (including structs) that we have not implemented the Blank protocol for will be considered non-blank.

### 16.3 Built-in protocols

Elixir ships with some built-in protocols. In previous chapters, we have discussed the Enum module which provides many functions that work with any data structure that implements the Enumerable protocol:

```elixir
iex> Enum.map [1, 2, 3], fn(x) -> x * 2 end
[2,4,6]
```

```elixir
iex> Enum.reduce 1..3, 0, fn(x, acc) -> x + acc end
6
```

Another useful example is the String.Chars protocol, which specifies how to convert a data structure with characters to a string. It’s exposed via the to_string function:

```elixir
iex> to_string :hello
"hello"
```

Notice that string interpolation in Elixir calls the to_string function:

```elixir
iex> "age: #{25}"
"age: 25"
```

The snippet above only works because numbers implement the String.Chars protocol. Passing a tuple, for example, will lead to an error:

```elixir
iex> tuple = {1, 2, 3}
{i, 2, 3}
iex> "tuple: #{tuple}"
** (Protocol.UndefinedError) protocol String.Chars not implemented for {1, 2, 3}
```

When there is a need to “print” a more complex data structure, one can simply use the inspect function, based on the Inspect protocol:
The `Inspect` protocol is the protocol used to transform any data structure into a readable textual representation. This is what tools like IEx use to print results:

```
%iex> {1, 2, 3}
{1,2,3}
%iex> %User{}
%User{name: "jose", age: 27}
```

Keep in mind that, by convention, whenever the inspected value starts with `#`, it is representing a data structure in non-valid Elixir syntax. This means the inspect protocol is not reversible as information may be lost along the way:

```
%iex> inspect &(&1+2)
"#Function<6.71889879/1 in :erl_eval.expr/5>"
```

There are other protocols in Elixir but this covers the most common ones. In the next chapter we will learn a bit more about error handling and exceptions in Elixir.

### 1.1.17 try, catch and rescue

Elixir has three error mechanisms: errors, throws and exits. In this chapter we will explore each of them and include remarks about when each should be used.

#### 17.1 Errors

A sample error can be retrieved by trying to add a number into an atom:

```
%iex> :foo + 1
** (ArithmeticError) bad argument in arithmetic expression :erlang.+{:foo, 1}
```

A runtime error can be raised any time by using the `raise/1` macro:

```
%iex> raise "oops"
** (RuntimeError) oops
```

Other errors can be raised with `raise/2` passing the error name and a list of keyword arguments:

```
%iex> raise ArgumentError, message: "invalid argument foo"
** (ArgumentError) invalid argument foo
```

You can also define your own errors using the `defexception/2` macro. The most common case is to define an exception with a message field:

```
%iex> defexception MyError, message: "default message"
%iex> raise MyError
** (MyError) default message
```
Elixir Documentation, Release

```
iex> raise MyError, message: "custom message"
** (MyError) custom message

Exceptions can be rescued by using the try/rescue construct:

```
iex> try do
...>   raise "oops"
...>   rescue
...>     e in RuntimeError -> e
...> end
RuntimeError[message: "oops"]
```

The example above rescues the runtime error and returns the error itself which is then printed in the iex session. In practice Elixir developers rarely use the try/rescue construct though. For example, many languages would force you to rescue an error when a file cannot open successfully. Elixir instead provides a File.read/1 function which returns a tuple containing information if the file was opened with success or not:

```
iex> File.read "hello"
{:error, :enoent}
iex> File.write "hello", "world"
:ok
iex> File.read "hello"
{:ok, "world"}
```

There is no try/rescue here. In case you want to handle multiple outcomes of opening a file, you can simply use pattern matching with case:

```
iex> case File.read "hello" do
...>   {:ok, body} -> IO.puts "got ok"
...>   {:error, body} -> IO.puts "got error"
...> end
```

At the end of the day, it is up to your application to decide if an error while opening a file is exceptional or not. That’s why Elixir doesn’t impose exceptions on File.read/1 and many other functions. Instead we leave it up to the developer to choose the best way to proceed.

For the cases where you do expect a file to exist (and the lack of a file is truly an error) you can simply use File.read!/1:

```
iex> File.read! "unknown"
** (File.Error) could not read file unknown: no such file or directory
   (elixir) lib/file.ex:305: File.read!/1
```

In other words, we avoid using try/rescue because we don’t use errors for control flow. In Elixir, we take errors literally: they are reserved to unexpected and/or exceptional situations. In case you actually need flow control constructs, throws must be used. That’s what we are going to see next.

### 17.2 Throws

In Elixir, one can throw a value to be caught later. throw and catch are reserved for situations where it is not possible to retrieve a value unless by using throw and catch.

Those situations are quite uncommon in practice unless when interfacing with a library that does not provide the proper APIs. For example, let’s imagine the Enum module did not provide any API for finding a value and we need to find the first number that is a multiple of 13:

```
iex> try do
...>   Enum.each -50..50, fn(x) ->
```
if rem(x, 13) == 0, do: throw(x)
end
"Got nothing"
catch
x -> "Got #{x}"
end
"Got -39"

However, in practice one can simply use `Enum.find/2`:

```iex
Enum.find -50..50, &({rem(&1, 13) == 0})
```

17.3 Exits

Every Elixir code runs inside processes that communicates with each other. When a process dies, it sends an exit signal. A process can also die by explicitly sending an exit signal:

```iex
spawn_link fn -> exit(1) end
#PID<0.56.0>
** (EXIT from #PID<0.56.0>) 1
```

In the example above, the linked process died by sending an exit signal with value of 1. The Elixir shell automatically handles those messages and prints them to the terminal.

exit can also be “caught” using try/catch:

```iex
try do
exit "I am exiting"
catch :exit, _ -> "not really"
end
"not really"
```

Using try/catch is already unCommon and using it to catch exits is even more rare.

exit signals are an important part of the fault tolerant system provided by the Erlang VM. Processes usually run under supervision trees which are themselves processes that just wait for exit signals of the supervised processes. Once an exit signal is received, the supervision strategy kicks in and the supervised process is restarted.

It is exactly this supervision system that makes constructs like try/catch and try/rescue so uncommon in Elixir. Instead of rescuing a certain error, we’d rather “fail fast” since the supervision tree will guarantee our application will go back to a known initial state after the error.

17.4 After

Sometimes it is necessary to use try/after to guarantee a resource is cleaned up after some particular action. For example, we can open a file and guarantee it is closed with try/after block:

```iex
{:ok, file} = File.open "sample", [:utf8, :write]
try do
IO.write file, "José"
raise "oops, something went wrong"
after
File.close(file)
end
** (RuntimeError) oops, something went wrong
```
17.5 Variables scope

It is important to bear in mind that variables defined inside `try/catch/rescue/after` blocks do not leak to the outer context. This is because the `try` block may fail and as such the variables may never be bound in the first place. In other words, this code is invalid:

```iex
iex> try do
...> from_try = true
...> after
...> from_after = true
...> end
iex> from_try
** (RuntimeError) undefined function: from_try/0
iex> from_after
** (RuntimeError) undefined function: from_after/0
```

This finishes our introduction to `try`, `catch` and `rescue`. You will find they are used less frequently in Elixir than in other languages although they may be handy in some situations where a library or some particular code is not playing "by the rules".

It is time to talk about some Elixir constructs like comprehensions and sigils.

1.1.18 18 Comprehensions

```
• 18.1 Generators and filters
• 18.2 Bitstring generators
• 18.3 Into
```

In Elixir, it is common to loop over Enumerables, often filtering some results, and mapping to another list of values. Comprehensions are syntax sugar for such constructs, grouping those common tasks into the `for` special form.

For example, we can get all the square values of elements in a list as follows:

```iex
iex> for n <- [1, 2, 3, 4], do: n * n
[1, 4, 9, 16]
```

A comprehension is made of three parts: generators, filters and collectables.

18.1 Generators and filters

In the expression above, `n <- [1, 2, 3, 4]` is the generator. It is literally generating values to be used in the comprehensions. Any enumerable can be passed in the right-hand side the generator expression:

```iex
iex> for n <- 1..4, do: n * n
[1, 4, 9, 16]
```

Generator expressions also support pattern matching, ignoring all non-matching patterns. Imagine that instead of a range, we have a keyword list where the key is the atom `:good` or `:bad` and we only want to calculate the square of the good values:

```iex
iex> values = [good: 1, good: 2, bad: 3, good: 4]
iex> for {:good, n} <- values, do: n * n
[1, 4, 16]
```
Alternatively, filters can be used to filter some particular elements out. For example, we can get the square of only odd numbers:

```elixir
require Integer
iex> for n <- 1..4, Integer.odd?(n), do: n * n
[1, 9]
```

A filter will keep all values except `nil` or `false`.

Comprehensions in general provide a much more concise representation than using the equivalent functions from the `Enum` and `Stream` modules. Furthermore, comprehensions also allow multiple generators and filters to be given. Here is an example that receives a list of directories and deletes all files in those directories:

```elixir
for dir <- dirs,
   file <- File.ls!(dir),
   path = Path.join(dir, file),
   File.regular?(path) do
   File.rm!(path)
end
```

Keep in mind that variable assignments inside the comprehension, be it in generators, filters or inside the block, are not reflected outside of the comprehension.

### 18.2 Bitstring generators

Bitstring generators are also supported and are very useful when you need to organize bitstring streams. The example below receives a list of pixels from a binary with their respective red, green and blue values and convert them into triplets:

```elixir
iex> pixels = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234, 32, 15>>
iex> for <<r::8, g::8, b::8 <- pixels>>, do: {r, g, b}
{(213, 45, 132), (64, 76, 32), (76, 0, 0), (234, 32, 15)}
```

A bitstring generator can be mixed with the “regular” enumerable generators and provide filters as well.

### 18.3 Into

In the examples above, the comprehension returned a list as a result. However, the result of a comprehension can be inserted into different data structures by passing the :into option. For example, we can use bitstring generators with the :into option to easily remove all spaces in a string:

```elixir
iex> for <<c <- " hello world ">>, c != ?\s, into: ", do: <<c>>
"helloworld"
```

Sets, maps and other dictionaries can also be given with the :into option. In general, the :into accepts any structure as long as it implements the `Collectable` protocol.

For example, the `IO` module provides streams, that are both `Enumerable` and `Collectable`. You can implement an echo terminal that returns whatever is typed, but in upcase, using comprehensions:

```elixir
iex> stream = IO.stream(:stdio, :line)
iex> for line <- stream, into: stream do
...>   String.upcase(line) <> "\n"
...> end
```

Now type any string into the terminal and you will see the same value will be printed in upcase. Unfortunately, this example also got your shell stuck in the comprehension, so you will need to hit `Ctrl+C` twice to get out of it. :)}
1.1.19 19 Sigils

- 19.1 Regular expressions
- 19.2 Strings, char lists and words sigils
- 19.3 Custom sigils

We have already learned Elixir provides double-quoted strings and single-quoted char lists. However, this only covers the surface of structures that have textual representation in the language. Atoms are, for example, another structure which are mostly created via the `:atom` representation.

One of Elixir’s goals is extensibility: developers should be able to extend the language to particular domains. Computer science has become such a wide field that it is impossible for a language to tackle many fields as part of its core. Our best bet is to rather make the language extensible, so developers, companies and communities can extend the language to their relevant domains.

In the chapter, we are going to explore sigils, which are one of the mechanisms provided by the language for working with textual representations.

### 19.1 Regular expressions

Sigils start with the tilde (~) character which is followed by a letter and then a separator. The most common sigil in Elixir is `~r` for regular expressions:

```elixir
# A regular expression that returns true if the text has foo or bar
iex> regex = ~r/foo|bar/  # ~r/foo|bar/
true
iex> "foo" =~ regex  # ~r/foo/  # ~r/foo|bar/
true
iex> "bat" =~ regex  # ~r/|bar/  # ~r/foo|bar/
false
```

Elixir provides Perl-compatible regular expressions (regexes), as implemented by the PCRE library. Regexes also support modifiers. For example, the `i` modifier makes a regular expression case insensitive:

```elixir
iex> "HELLO" =~ ~r/hello/ # ~r/HELLO/  # ~r/hello/  # ~r/HELLO|hello/  # ~r/HELLO/i
false
iex> "HELLO" =~ ~r/hello/i  # ~r/HELLO/  # ~r/hello/  # ~r/HELLO/i
true
```

Check out the 'Regex module <https://docs.stable.elixir-lang.org/elixir/Regex.html>' for more information on other modifiers and the supported operations with regular expressions.

So far, all examples have used `/` to delimit a regular expression. However sigils support 8 different separators:

```elixir
~r/hello/  # ~r/hello/  # ~r/hello/  # ~r/hello/  # ~r/hello/  # ~r/hello/  # ~r/hello/  # ~r/hello/
```

The reasoning in supporting different operators is that different separators can be more convenient to different sigils. For example, using parentheses for regular expressions may be a confusing choice as they can get mixed with the parentheses inside the regex. However, parentheses can be handy for other sigils, as we will see in the next section.
### 19.2 Strings, char lists and words sigils

Besides regular expressions, Elixir ships with three other sigils.

The `~s` sigil is used to generate strings, similar to double quotes:

```iex
eo~s(this is a string with "quotes")
"this is a string with \"quotes\"
```

While `~c` is used to generate char lists:

```iex
~c(this is a string with "quotes")
\'this is a string with "quotes"\'
```

The `~w` sigil is used to generate a list of words separated by white space:

```iex
~w(foo bar bat)
["foo", "bar", "bat"]
```

The `~w` sigil also accepts the `c`, `s` and `a` modifiers to choose the format of the result:

```iex
~w(foo bar bat)a
[:foo, :bar, :bat]
```

Besides lowercase sigils, Elixir supports uppercase sigils. While both `~s` and `~S` will return strings, the first one allows escape codes and interpolation while the second does not:

```iex
~s(String with escape codes \x26 interpolation)
"String with escape codes & interpolation"
~S(String without escape codes and without #{interpolation})
"String without escape codes and without \#{interpolation}"
```

The following escape codes can be used in strings and char lists:

- \" – double quote
- \’ – single quote
- \\ – single backslash
- \a – bell/alert
- \b – backspace
- \d – delete
- \e – escape
- \f – form feed
- \n – newline
- \r – carriage return
- \s – space
- \t – tab
- \v – vertical tab
- \DDD, \DD, \D – character with octal representation DDD, DD or D (example: \377)
- \xDD – character with hexadecimal representation DD (example: \x13)
- \x{D\ldots} – character with hexadecimal representation with one or more hexadecimal digits (example: \x{abc13})
Sigils also support heredocs which is when triple double- or single-quotes are used as separators:

```elixir
iex> ~s
...> this is
...> a heredoc string
...> 
```

The most common case for heredoc sigils is when writing documentation. For example, if you need to write escape characters in your documentation, it can become error prone as we would need to double-escape some characters:

```elixir
@doc ""
Converts double-quotes to single-quotes.

## Examples

iex> convert("\\"foo\\\"")
'foo'

"
```

By using using ~S, we can avoid this problem altogether:

```elixir
@doc ~S
Converts double-quotes to single-quotes.

## Examples

iex> convert(""foo"")
'foo'

"
```

### 19.3 Custom sigils

As hinted at the beginning of this chapter, sigils in Elixir are extensible. In fact, the sigil ~r/foo/i is equivalent to calling the `sigil_r` function with two arguments:

```elixir
iex> sigil_r(<<"foo">>, 'i')
~r"foo"i
```

That said, we can access the documentation for the ~r sigil via the `sigil_r` function:

```elixir
iex> h sigil_r
...
```

We can also provide our own sigils by simply implementing the proper function. For example, let’s implement the ~i(13) sigil that returns an integer:

```elixir
iex> defmodule MySigils do
...>   def sigil_i(binary, []), do: binary_to_integer(binary)
...> end
iex> import MySigils
iex> ~i(13)
13
```

Sigils can also be used to do compile-time work with the help of macros. For example, regular expressions in Elixir are compiled into efficient representation during compilation of the source code, therefore skipping this step at runtime.
If you have interest in the subject, we recommend you to learn more about macros and check how those sigils are implemented in the `Kernel` module.

### 1.1.20 20 Where to go next

- 20.1 Build your first Elixir project
- 20.2 Community and other resources
- 20.3 A Byte of Erlang

Eager to learn more? Keep on reading!

#### 20.1 Build your first Elixir project

In order to get your first project started, Elixir ships with a build tool called `mix`.

You can get your new project started by simply running:

```bash
mix new path/to/new/project
```

You can learn more about Mix and other applications that ship with Elixir in the links below:

- Mix - a build tool for Elixir
- ExUnit - a unit test framework

#### 20.2 Community and other resources

On the sidebar, you can find the link to some Elixir books and screencasts. They are plenty of Elixir resources out there, like conference talks, open source projects, and other learning material produced by the community.

Remember that in case of any difficulties, you can always visit the `#elixir-lang` channel on `irc.freenode.net` or send a message to the mailing list. You can be sure that there will be someone willing to help. And to keep posted on the latest news and announcements, follow the blog and follow language developments on the elixir-core mailing list.

Don’t forget you can also check the source code of Elixir itself, which is mostly written in Elixir (mainly the `lib` directory), or explore Elixir’s documentation.

#### 20.3 A Byte of Erlang

As the main page of this site puts it:

> Elixir is a programming language built on top of the Erlang VM.

Sooner than later, an Elixir developer will want to interface with existing Erlang libraries. Here’s a list of online resources that cover Erlang’s fundamentals and its more advanced features:

- This [Erlang Syntax: A Crash Course](https://example.com/erlang-syntax) provides a concise intro to Erlang’s syntax. Each code snippet is accompanied by equivalent code in Elixir. This is an opportunity for you to not only get some exposure to the Erlang’s syntax but also review some of the things you have learned in the present guide.

- Erlang’s official website has a short [tutorial](https://example.com/erlang-tutorial) with pictures that briefly describe Erlang’s primitives for concurrent programming.
Learn You Some Erlang for Great Good! is an excellent introduction to Erlang, its design principles, standard library, best practices and much more. If you are serious about Elixir, you’ll want to get a solid understanding of Erlang principles. Once you have read through the crash course mentioned above, you’ll be able to safely skip the first couple of chapters in the book that mostly deal with the syntax. When you reach The Hitchhiker’s Guide to Concurrency chapter, that’s where the real fun starts.

1.2 Mix - a build tool for Elixir

1.2.1 1 Introduction to Mix

- 1.1 Bootstrapping
  - 1.1.1 mix.exs
  - 1.2 lib/my_project.ex
  - 1.3 test/my_project_test.exs
  - 1.4 test/test_helper.exs

- 1.2 Exploring

- 1.3 Compilation

- 1.4 Dependencies
  - 1.4.1 Source Code Management (SCM)
  - 1.4.2 Compiling dependencies
  - 1.4.3 Repeatability
  - 1.4.4 Dependencies tasks
  - 1.4.5 Dependencies of dependencies

- 1.5 Umbrella projects

- 1.6 Environments

Mix is a build tool that provides tasks for creating, compiling, testing (and soon releasing) Elixir projects. Mix is inspired by the Leiningen build tool for Clojure and was written by one of its contributors.

In this chapter, you will learn how to create projects using mix and install dependencies. In the following sections, we will also learn how to create OTP applications and create custom tasks with Mix.

1.1 Bootstrapping

In order to start your first project, simply use the mix new command passing the path to your project. For now, we will create an project called my_project in the current directory:

```
$ mix new my_project --bare
```

Mix will create a directory named my_project with few files in it:

```
.gitignore
README.md
mix.exs
lib/my_project.ex
test/test_helper.exs
test/my_project_test.exs
```

Let’s take a brief look at some of these.
Note: Mix is an Elixir executable. This means that in order to run mix, you need to have elixir’s executable in your PATH. If not, you can run it by passing the script as argument to elixir:

```bash
$ bin/elixir bin/mix new ./my_project
```

Note that you can also execute any script in your PATH from Elixir via the -S option:

```bash
$ bin/elixir -S mix new ./my_project
```

When using -S, elixir finds the script wherever it is in your PATH and executes it.

### 1.1.1 mix.exs

This is the file with your projects configuration. It looks like this:

```elixir
defmodule MyProject.Mixfile do
  use Mix.Project

  def project do
    [app: :my_project,
     version: "0.0.1",
     deps: deps]
  end

  # Configuration for the OTP application
  def application do
    []
  end

  # Returns the list of dependencies in the format:
  # {:foobar, git: "https://github.com/elixir-lang/foobar.git", tag: "0.1"}
  #
  # To specify particular versions, regardless of the tag, do:
  # {:barbat, "~> 0.1", github: "elixir-lang/barbat"}
  defp deps do
    []
  end
end
```

Our `mix.exs` defines two functions: `project`, which returns project configuration like the project name and version, and `application`, which is used to generate an Erlang application that is managed by the Erlang Runtime. In this chapter, we will talk about the `project` function. We will go into detail about what goes in the `application` function in the next chapter.

### 1.1.2 lib/my_project.ex

This file contains a simple module definition to lay out our code:

```elixir
defmodule MyProject do
  end
```

### 1.1.3 test/my_project_test.exs

This file contains a stub test case for our project:
defmodule MyProjectTest do
  use ExUnit.Case

  test "the truth" do
    assert true
  end
end

It is important to note a couple things:

1. Notice the file is an Elixir script file (.exs). This is convenient because we don’t need to compile test files before running them;

2. We define a test module named MyProjectTest, using ExUnit.Case to inject default behavior and define a simple test. You can learn more about the test framework in the ExUnit chapter;

1.1.4 test/test_helper.exs

The last file we are going to check is the test_helper.exs, which simply sets up the test framework:

ExUnit.start

This file will be automatically required by Mix every time before we run our tests. And that is it, our project is created. We are ready to move on!

1.2 Exploring

Now that we created our new project, what can we do with it? In order to check the commands available to us, just run the help task:

$ mix help

It will print all the available tasks. You can get further information by invoking mix help TASK.

Play around with the available tasks, like mix compile and mix test, and execute them in your project to check how they work.

1.3 Compilation

Mix can compile our project for us. The default configurations uses lib/ for source files and ebin/ for compiled beam files. You don’t even have to provide any compilation-specific setup but if you must, some options are available. For instance, if you want to put your compiled files in another directory besides ebin, simply set in :compile_path in your mix.exs file:

def project do
  [compile_path: "ebin"]
end

In general, Mix tries to be smart and compiles only when necessary.

Note that after you compile for the first time, Mix generates a my_project.app file inside your ebin directory. This file defines an Erlang application based on the contents of the application function in your Mix project.

The .app file holds information about the application, what are its dependencies, which modules it defines and so forth. The application is automatically started by Mix every time you run some commands and we will learn how to configure it in the next chapter.
1.4 Dependencies

Mix is also able to manage dependencies. Dependencies should be listed in the project settings, as follows:

```elixir
def project do
    [app: :my_project,
     version: "0.0.1",
     deps: deps]
end
defp deps do
    [{:some_project, ">= 0.3.0"},
     {:another_project, git: "https://example.com/another/repo.git", tag: "v1.0.2"}]
end
```

**Note:** Although not required, it is common to split dependencies into their own function.

The dependency is represented by an atom, followed by a requirement and some options. By default, Mix uses hex.pm to fetch dependencies but it can also fetch them from a git repository or directly from the file system.

When using Hex, you must use requirements to specify which versions of a given dependency you accept. It supports common operators like `>=`, `<=`, `>`, `-=` as follows:

- `== 2.0.0`
- `>= 2.0.0`
- `> 2.0.0`

Requirements also support **and** and **or** for complex conditions:

- `>= 2.0.0 and < 2.1.0`

Since the example above is such a common requirement, it can be expressed as:

"~> 2.0.0"

Note that setting the version requirement for git repositories does not affect the branch or tag that is checked out, so while a definition like the following is possible:

```elixir
{:some_project, "~> 0.5.0", github: "some_project/other", tag: "v0.3.0"}
```

It will lead to a dependency that will never be satisfied, because the tag being checked out does not match the version requirement.

### 1.4.1 Source Code Management (SCM)

Mix is designed in a way it can support multiple SCM tools, Hex packages being the default, while :git and :path are supported as options. The most common options are:

- :git - the dependency is a git repository that is retrieved and updated by Mix;
- :path - the dependency is simply a path in the file system;
- :compile - how to compile the dependency;
- :app - the path of the application expected to be defined by the dependency;
- :env - the environment to use from the dependency (more info below), defaults to :prod;

Each SCM may support custom options. :git, for example, supports the following:
Elixir Documentation, Release

- :ref - an optional reference (a commit) to checkout the git repository;
- :tag - an optional tag to checkout the git repository;
- :branch - an optional branch to checkout the git repository;
- :submodules - when true, initializes submodules recursively in the dependency;

1.4.2 Compiling dependencies

In order to compile a dependency, Mix looks into the repository for the best way to proceed. If the dependency contains one of the files below, it will proceed as follows:

1. mix.exs - compiles the dependency directly with Mix by invoking the compile task;
2. rebar.config or rebar.config.script - compiles using rebar compile deps_dir=DEPS, where DEPS is the directory where Mix will install the project dependencies by default;
3. Makefile - simply invokes make;

If the dependency does not contain any of the above, you can specify a command directly with the :compile option:

{ :some_dep, git: "...", compile: "./configure && make" }

If :compile is set to false, nothing is done.

1.4.3 Repeatability

An important feature in any dependency management tool is repeatability. For this reason when you first get your dependencies, Mix will create a file called mix.lock that contains checked out references for each dependency.

When another developer gets a copy of the same project, Mix will checkout exactly the same references, ensuring other developers can “repeat” the same setup.

Locks are automatically updated when deps.update is called and can be removed with deps.unlock.

1.4.4 Dependencies tasks

Elixir has many tasks to manage the project dependencies:

- mix deps - List all dependencies and their status;
- mix deps.get - Get all unavailable dependencies;
- mix deps.compile - Compile dependencies;
- mix deps.update - Update dependencies;
- mix deps.clean - Remove dependencies files;
- mix deps.unlock - Unlock the given dependencies;

Use mix help to get more information.
1.4.5 Dependencies of dependencies

If your dependency is another Mix or rebar project, Mix does the right thing: it will automatically fetch and handle all dependencies of your dependencies. However, if your project has two dependencies that share the same dependency and the SCM information for the shared dependency doesn’t match between the parent dependencies, Mix will mark that dependency as diverged and emit a warning. To solve this issue you can declare the shared dependency in your project with the option `override: true` and Mix will use that SCM information to fetch the dependency.

1.5 Umbrella projects

It can be convenient to bundle multiple Mix projects together and run Mix tasks for them at the same time. They can be bundled and used together in what is called an umbrella project. An umbrella project can be created with the following command:

```
$ mix new project --umbrella
```

This will create a `mix.exs` file with the following contents:

```
defmodule Project.Mixfile do
  use Mix.Project

  def project do
    [apps_path: "apps"]
  end
end
```

The `apps_path` option specifies the directory where subprojects will reside. Mix tasks that run in the umbrella project will run for every project in the `apps_path` directory. For example `mix compile` or `mix test` will compile or test every project in the directory. It’s important to note that an umbrella project is neither a regular Mix project, nor is it an OTP application nor can code source files be added.

If there are interdependencies between subprojects these have to be specified so that Mix can compile the projects in the correct order. If Project A depends on Project B, the dependency has to be specified in Project A’s `mix.exs` file; modify the `mix.exs` file to specify the dependency:

```
defmodule A.Mixfile do
  use Mix.Project

  def project do
    [app: :a,
     deps_path: "./..//deps",
     lockfile: "./..//mix.lock",
     deps: deps]
  end
end
```

Note the `deps_path` and `lockfile` options in the subproject above. If you have these options in all the subprojects in the umbrella they will share their dependencies. `mix new` inside the apps directory will automatically create a project with these options pre-set.
1.6 Environments

Mix has the concept of environments that allows a developer to customize compilation and other options based on an external setting. By default, Mix understands three environments:

- **dev** - the one in which mix tasks are run by default;
- **test** - used by `mix test`;
- **prod** - the environment in which dependencies are loaded and compiled;

By default, these environments behave the same and all configuration we have seen so far will affect all three environments. Customization per environment can be done by accessing `Mix.env`:

```elixir
def project do
  [deps_path: deps_path(Mix.env)]
end
```

defp deps_path(:prod), do: "prod_deps"
defp deps_path(_), do: "deps"

Mix will default to the **dev** environment (except for tests). The environment can be changed via the `MIX_ENV` environment variable:

```
$ MIX_ENV=prod mix compile
```

In the next chapters, we will learn more about building OTP applications with Mix and how to create your own tasks.

1.2.2 2 Building OTP apps with Mix

- 2.1 The Stacker server
  - 2.1.1 Learning more about callbacks
  - 2.1.2 Crashing a server
- 2.2 Supervising our servers
- 2.3 Who supervises the supervisor?
- 2.4 Starting applications
- 2.5 Configuring applications

Where do we keep state in Elixir?

Our software needs to keep state, configuration values, data about the running system, etc. We have learned in previous sections how we can use processes/actors to keep state, receiving and responding to messages in a loop but this approach seems to be brittle. What happens if there is an error in our actor and it crashes? Even more, is it really required to create a new process when all we want to do is to keep simple configuration values?

In this chapter, we will answer those questions by building an OTP application. In practice, we don’t need Mix in order to build such applications, however Mix provides some conveniences that we are going to explore throughout this chapter.

2.1 The Stacker server

Our application is going to be a simple stack that allow us push and pop items as we wish. Let’s call it stacker:

```
$ mix new stacker --bare
```
Elixir Documentation, Release

Our application is going to have one stack which may be accessed by many processes at the same time. To achieve that, we will create a server that is responsible to manage the stack. Clients will send messages to the server whenever they want to push or pop something from the stack.

Since creating such servers is a common pattern when building Erlang and Elixir applications, we have a behavior in OTP that encapsulates common server functionality called **GenServer**. Let’s create a file named `lib/stacker/server.ex` with our first server:

```elixir
defmodule Stacker.Server do
  use GenServer.Behaviour

  def init(stack) do
    {:ok, stack }
  end

  def handle_call(:pop, _from, [h|stack]) do
    {:reply, h, stack }
  end

  def handle_cast({ :push, new }, stack) do
    {:noreply, [new|stack] }
  end
end
```

Our server defines three callbacks: `init/1`, `handle_call/3` and `handle_cast/2`. We never call those functions directly, they are called by OTP whenever we interact with the server. We will go into details about these soon, let’s just ensure it works as expected. To do so, run `iex -S mix` on your command line to start iex with mix and type the following:

```
# Let’s start the server using Erlang’s :gen_server module.
# It expects 3 arguments: the server module, the initial
# stack and some options (if desired):
iex> { :ok, pid } = :gen_server.start_link(Stacker.Server, [], [])
{ :ok, <...> }

# Now let’s push something onto the stack
iex> :gen_server.cast(pid, { :push, 13 })
:ok

# Now let’s get it out from the stack
# Notice we are using *call* instead of *cast*
iex> :gen_server.call(pid, :pop)
13
```

Excellent, our server works as expected! There are many things happening behind the scenes, so let’s discuss them one by one.

First, we started the server using the `:gen_server` module from OTP <http://www.erlang.org/doc/man/gen_server.html>. Notice we have used `start_link`, which starts the server and links our current process to the server. In this scenario, if the server dies, it will send an exit message to our process, making it crash too. We will see this in action later. The `start_link` function returns the process identifier (`pid`) of the newly spawned server.

Later, we have sent a `cast` message to the `pid`. The message was `{ :push, 13 }`, written in the same format as we specified in the `handle_cast/2` callback in `Stacker.Server`. Whenever we send a `cast` message, the `handle_cast/2` callback will be invoked to handle the message.

Then we finally read what was on the stack by sending a `call` message, which will dispatch to the `handle_call/3` callback. So, what is the difference between `cast` and `call` after all?
cast messages are asynchronous: we simply send a message to the server and don’t expect a reply back. That’s why our handle_cast/2 callback returns `{ :noreply, [new|stack] }`. The first item of the tuple states nothing should be replied and the second contains our updated stack with the new item.

On the other hand, call messages are synchronous. When we send a call message, the client expects a response back. In this case, the handle_call/3 callback returns `{ :reply, h, stack }`, where the second item is the term to be returned and the third is our new stack without its head. Since calls are able to send messages back to the client, it also receives the client information as argument (_from).

### 2.1.1 Learning more about callbacks

In the GenServer’s case, there are 8 different values a callback such as handle_call or handle_cast can return:

- `{ :reply, reply, new_state }`
- `{ :reply, reply, new_state, timeout }`
- `{ :reply, reply, new_state, :hibernate }`
- `{ :noreply, new_state }`
- `{ :noreply, new_state, timeout }`
- `{ :noreply, new_state, :hibernate }`
- `{ :stop, reason, new_state }`
- `{ :stop, reason, reply, new_state }

There are 6 callbacks required to be implemented in a GenServer. The GenServer.Behaviour module defines all of them automatically but allows us to customize the ones we need. The list of callbacks are:

- `init(args)` - invoked when the server is started;
- `handle_call(msg, from, state)` - invoked to handle call messages;
- `handle_cast(msg, state)` - invoked to handle cast messages;
- `handle_info(msg, state)` - handle all other messages which are normally received by processes;
- `terminate(reason, state)` - called when the server is about to terminate, useful for cleaning up;
- `code_change(old_vsn, state, extra)` - called when the application code is being upgraded live (hot code swap);

### 2.1.2 Crashing a server

Of what use is a server if we cannot crash it?

It is actually quite easy to crash our server. Our handle_call/3 callback only works if there is something on the stack (remember `[h|t]` won’t match an empty list). So let’s simply send a message when the stack is empty:

```elixir
# Start another server, but with an initial :hello item
iex> { :ok, pid } = :gen_server.start_link(Stacker.Server, [:hello], [])
{:ok, <...>}

# Let’s get our initial item:
iex> :gen_server.call(pid, :pop)
:hello

# And now let’s call pop again
iex> :gen_server.call(pid, :pop)
=ERROR REPORT==== 6-Dec-2012::19:15:33 === ...
```

1.2. Mix - a build tool for Elixir
You can see there are two error reports. The first one is generated by server, due to the crash. Since the server is linked to our process, it also sent an exit message which was printed by IEx as `** (exit) ...`

Since our servers may eventually crash, it is common to supervise them, and that’s what we are going to next. There is a bit more to GenServer than what we have seen here. For more information, check ‘GenServer.Behaviour’s documentation’.

### 2.2 Supervising our servers

When building applications in Erlang/Elixir, a common philosophy is to fail fast. Resources are going to become unavailable, timeout in between services are going to happen and other possible failures exist. That’s why it is important to recover and react to such failures. With this in mind, we are going to write a supervisor for our server.

Create a file at `lib/stacker/supervisor.ex` with the following:

```elixir
defmodule Stacker.Supervisor do
  use Supervisor.Behaviour

  # A convenience to start the supervisor
  def start_link(stack) do
    :supervisor.start_link(__MODULE__, stack)
  end

  # The callback invoked when the supervisor starts
  def init(stack) do
    children = [ worker(Stacker.Server, [stack]) ]
    supervise children, strategy: :one_for_one
  end
end
```

In case of supervisors, the only callback that needs to be implemented is `init(args)`. This callback needs to return a supervisor specification, in this case returned by the helper function `supervisor/2`.

Our supervisor is very simple: it has to supervise one worker, in this case, Stacker.Server and the worker will be started by receiving one argument, which is the default stack. The defined worker is then going to be supervised using the `:one_for_one` strategy, which restarts each worker after it dies.

Given that our worker is specified by the Stacker.Server module passing the `stack` as argument, the supervisor will by default invoke the `Stacker.Server.start_link(stack)` function to start the worker, so let’s implement it:

```elixir
defmodule Stacker.Server do
  use GenServer.Behaviour

  def start_link(stack) do
    :gen_server.start_link({ :local, :stacker }, __MODULE__, stack, [])
  end

  def init(stack) do
    { :ok, stack }
  end

  def handle_call(:pop, _from, [h|stack]) do
    { :reply, h, stack }
  end
end
```
The `start_link` function is quite similar to how we were starting our server previously, except that now we passed one extra argument: `{:local, :stacker}`. This argument registers the server on our local nodes, allowing it to be invoked by the given name (in this case, `:stacker`), instead of directly using the `pid`.

With our supervisor in hand, let’s start the console by running `iex -S mix` once again, which will recompile our files too:

```elixir
# Now we will start the supervisor with a
# default stack containing :hello
iex> Stacker.Supervisor.start_link([{:hello}])
{:ok,<...>}

# And we will access the server by name since
# we registered it
iex> :gen_server.call(:stacker, :pop)
:hello
```

Notice the supervisor started the server for us and we were able to send messages to it via the name `:stacker`. What happens if we crash our server again?

```elixir
iex> :gen_server.call(:stacker, :pop)
=ERROR REPORT==== 6-Dec-2012::19:15:33 ===
... 5 times ...
```
2.3 Who supervises the supervisor?

We have built our supervisor but a pertinent question is: who supervises the supervisor? To answer this question, OTP contains the concept of applications. Applications can be started and stopped as a unit and, when doing so, they are often linked to a supervisor.

In the previous chapter, we have learned how Mix automatically generates an .app file every time we compile our project based on the information contained on the application function in our mix.exs file.

The .app file is called application specification and it must contain our application dependencies, the modules it defines, registered names and many others. Some of this information is filled in automatically by Mix but other data needs to be added manually.

In this particular case, our application has a supervisor and, furthermore, it registers a server with name :stacker. That said, it is useful to add to the application specification all registered names in order to avoid conflicts. If it happens that two applications register the same name, we will be able to find about this conflict sooner. So, let’s open the mix.exs file and edit the application function to the following:

```elixir
def application do
  [ registered: [:stacker],
    mod: { Stacker, [:hello] } ]
end
```

In the :registered key we specify all names registered by our application. The :mod key specifies that, as soon as the application is started, it must invoke the application module callback. In this case, the application module callback will be the Stacker module and it will receive the default stack [:hello] as argument. The callback must return the pid of the supervisor which is associated to this application.

With this in mind, let’s open up the lib/stacker.ex file and add the following:

```elixir
defmodule Stacker do
  use Application.Behaviour
  
def start(_type, stack) do
    Stacker.Supervisor.start_link(stack)
  end
end
```

The Application.Behaviour expects two callbacks, start(type, args) and stop(state). We are required to implement start/2 though we have decided to not bother about stop(state) for now.

After adding the application behavior above, all you need to do is to start iex -S mix once again. Our files are going to be recompiled and the supervisor (and consequently our server) will be automatically started:

```elixir
iex> :gen_server.call(:stacker, :pop)
:hello
```

Amazing, it works! As you may have noticed, the application start/2 callback receives a type argument, which we have ignored. The type controls how the VM should behave when the supervisor, and consequently our application, crashes. You can learn more about it by reading the documentation for ‘Application.Behaviour’<docs/stable/elixir/Application.html'>.

Finally, notice that mix new supports a --sup option, which tells Mix to generate a supervisor with an application module callback, automating some of the work we have done here. Try it!

2.4 Starting applications

We did not have to, at any point, start the application we have just defined. That’s because Mix starts all applications, and all application dependencies, by default. We can start any application manually by calling functions from the
:application module provided by OTP:

```iex
:application.start(:stacker)
{ :error, { :already_started, :stacker } }
```

In this case, since the application was previously started, it returns so as an error message.

Mix not only starts your application but all of your application dependencies. Notice there is a difference between your project dependencies (the ones defined under the `deps` key we have discussed in the previous chapter) and the application dependencies.

The project dependencies may contain your test framework or a compile-time only dependency. The application dependency is everything you depend on at runtime. Any application dependency needs to be explicitly added to the `application` function too:

```elixir
def application do
  [ registered: [:stacker],
    applications: [:some_dep],
    mod: { Stacker, [:hello] } ]
end
```

When running tasks on Mix, it will ensure the application and all application dependencies are started.

### 2.5 Configuring applications

Besides the `:registered`, `:applications` and `:mod` keys we have seen above, applications also support configuration values that can be get and set explicitly.

Still in the command line, try:

```iex
:application.get_env(:stacker, :foo)
:undefined
iex> :application.set_env(:stacker, :foo, :bar)
:ok
iex> :application.get_env(:stacker, :foo)
{ :ok, :bar }
```

This is a very useful mechanism for providing configuration values in your applications without a need to create the whole supervise chain. Default values for the application configuration can be defined in the `mix.exs` file as follows:

```elixir
def application do
  [ registered: [:stacker],
    mod: { Stacker, [:hello] },
    env: [foo: :bar] ]
end
```

Now, leave the current shell and restart it with `iex -S mix`:

```iex
:application.get_env(:stacker, :foo)
{ :ok, :bar }
```

For example, IEx and ExUnit are two applications that ship with Elixir that relies on such configuration values, as seen in their `mix.exs` files: IEx and ExUnit. Such applications then provide wrappers for reading and setting such values.

With this note, we finalize this chapter. We have learned how to create servers, supervise them, hook them into our application life cycle and provide simple configuration options. In the next chapter, we will learn how to create custom tasks in Mix.
3 Creating custom Mix tasks

In Mix, a task is simply an Elixir module inside the Mix.Tasks namespace containing a run/1 function. For example, the compile task is a module named Mix.Tasks.Compile.

Let's create a simple task:

```elixir
defmodule Mix.Tasks.Hello do
  use Mix.Task

  @shortdoc "This is short documentation, see"
  @moduledoc "A test task."
  def run(_) do
    IO.puts "Hello, World!"
  end
end
```

Save this module to a file named hello.ex then compile and run it as follows:

```
$ elixirc hello.ex
$ mix hello
Hello, World!
```

The module above defines a task named hello. The function run/1 takes a single argument that will be a list of binary strings which are the arguments that were passed to the task on the command line.

When you invoke mix hello, this task will run and print Hello, World!. Mix uses its first argument (hello in this case) to lookup the task module and execute its run function.

You're probably wondering why we have a @moduledoc and @shortdoc. Both are used by the help task for listing tasks and providing documentation. The former is used when mix help TASK is invoked, the latter in the general listing with mix help.

Besides those two, there is also @hidden attribute that, when set to true, marks the task as hidden so it does not show up on mix help. Any task without @shortdoc also won’t show up.

3.1 Common API

When writing tasks, there are some common mix functionality we would like to access. There is a gist:

- Mix.Project.config - Returns the project configuration under the function project: Notice this function returns an empty configuration if no mix.exs file exists in the current directory, allowing many Mix functions to work even if a mix.exs project is not defined;
- Mix.Project.get! - Access the module for the current project, useful in case you want to access special functions in the project. It raises an exception if no project is defined;
• **Mix.shell** - The shell is a simple abstraction for doing IO in Mix. Such abstractions make it easy to test existing mix tasks. In the future, the shell will provide conveniences for colored output and getting user input;

• **Mix.Task.run(task, args)** - This is how you invoke a task from another task in Mix; Notice that if the task was already invoked, it works as no-op;

There is more to the Mix API, so feel free to check the documentation, with special attention to `Mix.Task </docs/stable/mix/Mix.Task.html>` and `Mix.Project </docs/stable/mix/Mix.Project.html>`.

### 3.2 Namespaced Tasks

While tasks are simple, they can be used to accomplish complex things. Since they are just Elixir code, anything you can do in normal Elixir you can do in Mix tasks. You can distribute tasks however you want just like normal libraries and thus they can be reused in many projects.

So, what do you do when you have a whole bunch of related tasks? If you name them all like `foo`, `bar`, `baz`, etc, eventually you’ll end up with conflicts with other people’s tasks. To prevent this, Mix allows you to namespace tasks.

Let’s assume you have a bunch of tasks for working with Riak.

```elixir
defmodule Mix.Tasks.Riak do
  defmodule Dostuff do
    ...
  end

  defmodule Dootherstuff do
    ...
  end
end
```

Now you’ll have two different tasks under the modules `Mix.Tasks.Riak.Dostuff` and `Mix.Tasks.Riak.Dootherstuff` respectively. You can invoke these tasks like so: `mix riak.dostuff` and `mix riak.dootherstuff`. Pretty cool, huh?

You should use this feature when you have a bunch of related tasks that would be unwieldy if named completely independently of each other. If you have a few unrelated tasks, go ahead and name them however you like.

### 3.3 OptionParser

Although not a Mix feature, Elixir ships with an OptionParser which is quite useful when creating mix tasks that accepts options. The OptionParser receives a list of arguments and returns a tuple with parsed options and the remaining arguments:

```elixir
OptionParser.parse(["--debug"])
#=> { [debug: true], [] }

OptionParser.parse(["--source", "lib"])
#=> { [source: "lib"], [] }

OptionParser.parse(["--source", "lib", "test/enum_test.exs", "--verbose"])
#=> { [source: "lib", verbose: true], ["test/enum_test.exs"] }
```

Check `OptionParser </docs/stable/elixir/OptionParser.html>` documentation for more information.
3.4 Sharing tasks

After you create your own tasks, you may want to share them with other developers or re-use them inside existing projects. In this section, we will see different ways to share tasks in Mix.

3.4.1 As a dependency

Imagine you’ve created a Mix project called my_tasks which provides many tasks. By adding the my_tasks project as a dependency to any other project, all the tasks in my_tasks will be available in the parent project. It just works!

3.4.2 As an archive

Mix tasks are useful not only inside projects, but also to create new projects, automate complex tasks and to avoid repetitive work. For such cases, you want a task always available in your workflow, regardless if you are inside a project or not.

For such cases, Mix allows developers to install and uninstall archives locally. To generate an archive for the current project and install it locally, run:

$ mix do archive, local.install

Archives can be installed from a path or any URL:

$ mix local.install http://example.org/path/to/sample/archive.ez

After installing an archive, you can run all tasks contained in the archive, list them via mix local or uninstall the package via mix local.uninstall archive.ez.

3.4.3 MIX_PATH

The last mechanism for sharing tasks is MIX_PATH. By setting up your MIX_PATH, any task available in the MIX_PATH will be automatically visible to Mix. Here is an example:

$ export MIX_PATH="/full/path/to/my/project/ebin"

This is useful for complex projects that must be installed at /usr or /opt but still hook into Mix facilities.

With all those options in mind, you are ready to go out, create and install your own tasks! Enjoy!

1.3 ExUnit - a unit test framework

1.3.1 1 Introduction to ExUnit
ExUnit is a unit test framework that ships with Elixir.

Using ExUnit is quite easy, here is a file with the minimum required:

```
ExUnit.start

defmodule MyTest do
  use ExUnit.Case

  test "the truth" do
    assert true
  end
end
```

In general, we just need to invoke `ExUnit.start`, define a test case using `ExUnit.Case` and our batch of tests. Most tests will be invoked as part of a Mix project.

```
mix test
```

Otherwise, assuming we saved this file as `assertion_test.exs`, we can run it directly:

```
bin/elixir assertion_test.exs
```

In this chapter, we will discuss the most common features available in ExUnit and how to customize it further.

### 1.1 Starting ExUnit

ExUnit is usually started via `ExUnit.start`. This function accepts a couple options, so check its documentation for more details. For now, we will just detail the most common ones:

- **:formatter** - When you run tests with ExUnit, all the IO is done by the formatter. Developers can define their own formatters and this is the configuration that tells ExUnit to use a custom formatter;
- **:max_cases** - As we are going to see soon, ExUnit allows you to easily run tests concurrently. This is very useful to speed up your tests that have no side affects. This option allows us to configure the maximum number of cases ExUnit runs concurrently.

### 1.2 Defining a test case

After ExUnit is started, we can define our own test cases. This is done by using `ExUnit.Case` in our module:

```
use ExUnit.Case
```

`ExUnit.Case` provides some features, so let’s take a look at them.
### 1.2.1 The test macro

`ExUnit.Case` runs all functions whose name start with `test` and expects one argument:

```elixir
def test_the_truth(_) do
  assert true
end
```

As a convenience to define such functions, `ExUnit.Case` provides a `test` macro, which allows one to write:

```elixir
test "the truth" do
  assert true
end
```

This construct is considered more readable. The `test` macro accepts either a binary or an atom as name.

### 1.2.2 Assertions

Another convenience provided by `ExUnit.Case` is to automatically import a set of assertion macros and functions, available in `ExUnit.Assertions` available in `ExUnit.Assertions` available in `ExUnit.Assertions` available in `ExUnit.Assertions` available in `ExUnitAssertions.html>` __.

In the majority of tests, the only assertion macros you will need to use are `assert` and `refute`:

```elixir
assert 1 + 1 == 2
refute 1 + 3 == 3
```

`ExUnit` automatically breaks those expressions apart and attempt to provide detailed information in case the assertion fails. For example, the failing assertion:

```elixir
assert 1 + 1 == 3
```

Will fail as:

```
Expected 2 to be equal to (==) 3
```

However, some extra assertions are convenient to make testing easier for some specific cases. A good example is the `assert_raise` macro:

```elixir
assert_raise ArithmeticError, "bad argument in arithmetic expression", fn ->
  1 + "test"
end
```

So don’t forget to check `ExUnit.Assertions` documentation for more examples.

### 1.2.3 Callbacks

`ExUnit.Case` defines four callbacks: `setup`, `teardown`, `setup_all` and `teardown_all`:

```elixir
defmodule CallbacksTest do
  use ExUnit.Case, async: true

  setup do
    IO.puts "This is a setup callback"
    :ok
  end

  test "the truth" do
```
In the example above, the setup callback will be run before each test. In case a setup_all callback is defined, it would run once before all tests in that module.

A callback must return :ok or {:ok, data}. When the latter is returned, the data argument must be a keywords list containing metadata about the test. This metadata can be accessed in any other callback or in the test itself:

```elixir
defmodule CallbacksTest do
  use ExUnit.Case, async: true

  setup do
    IO.puts "This is a setup callback"
    {:ok, from_setup: :hello}
  end

  test "the truth", meta do
    assert meta[:from_setup] == :hello
  end

  teardown meta do
    assert meta[:from_setup] == :hello
    :ok
  end
end
```

Metadata is used when state need to be explicitly passed to tests.

### 1.2.4 Async

Finally, ExUnit also allows test cases to run concurrently. All you need to do is pass the `async` option set to true:

```elixir
use ExUnit.Case, async: true
```

This will run this test case concurrently with other test cases which are async too. The tests inside a particular case are still run sequentially.

### 2 Lots To Do

ExUnit is still a work in progress. Feel free to visit our issues tracker to add issues for anything you’d like to see in ExUnit and feel free to contribute.

### 1.4 Meta-programming in Elixir

#### 1.4.1 1 Quote and unquote
An Elixir program can be represented by its own data structures. In this chapter, we will learn how those structures look like and how to compose them. The concepts learned in this chapter are the building blocks for macros, which we are going to take a deeper look at the next chapter.

1.1 Quoting

The building block of an Elixir program is a tuple with three elements. For example, the function call \( \text{sum}(1, 2, 3) \) is represented internally as:

\[
\{\text{sum}, [], [1, 2, 3]\}
\]

You can get the representation of any expression by using the `quote` macro:

```elixir
iex> quote do: \(\text{sum}(1, 2, 3)\)
\{
  \text{sum}, [], [1, 2, 3]\}
```

The first element is the function name, the second is a keyword list containing metadata and the third is the arguments list.

Operators are also represented as such tuples:

```elixir
iex> quote do: \(1 + 2\)
\{
  \text{+}, \{\text{context: Elixir, import: Kernel}\}, [1, 2]\}
```

Even a map is represented as a call to `%{}`:

```elixir
iex> quote do: %{1 => 2}
\{
  %{}, [], [{1, 2}]\}
```

Variables are also represented using such triplets, except the last element is an atom, instead of a list:

```elixir
iex> quote do: \(x\)
\{
  \text{x}, [], \text{Elixir}\}
```

When quoting more complex expressions, we can see the code is represented in such tuples, often nested inside each other resembling a tree. Many languages would call those representations the Abstract Syntax Tree (AST). Elixir calls them quoted expressions:

```elixir
iex> quote do: \(\text{sum}(1, 2 + 3, 4)\)
\{
  \text{sum}, [], [1, \{\text{+}, \{\text{context: Elixir, import: Kernel}\}, [2, 3]\}, 4]\}
```

Sometimes when working with quoted expressions, it may be useful to get the textual code representation back. This can be done with `Macro.to_string/1`:

```elixir
iex> Macro.to_string(quote do: \(\text{sum}(1, 2 + 3, 4)\))
"\(\text{sum}(1, 2 + 3, 4)\)"
```

In general, the tuples above follow the following format:

\[
\{\text{tuple} | \text{atom, list, list | atom}\}
\]

- The first element is an atom or another tuple in the same representation;
- The second element is a keyword list containing information like metadata like numbers and contexts;
- The third element is either a list of arguments for the function call or an atom. When an atom, it means the tuple represents a variable.

Besides the tuple defined above, there are five Elixir literals that when quoted return themselves (and not a tuple). They are:
Most of Elixir code has straightforward translation to its underlying quoted expression. We recommend you to try out different code samples and see what gets out of it. For example, what `String.upcase("foo")` expands to? We have also learned that `if(true, do: this, else: that)` is the same as `if true do this else that end`. How does this affirmation hold with quoted expressions?

### 1.2 Unquoting

Quote is about retrieving the inner representation of some particular chunk of code. However, sometimes it may be necessary to inject some other particular chunk of code inside the representation we want to retrieve.

For example, imagine you have a variable `number` which contains the number you want to inject inside a quoted expression. The number can be injected into the quoted representation by using `unquote`:

```
iex> number = 13
iex> Macro.to_string(quote do
       11 + unquote(number))
"11 + 13"
```

`unquote` can even be used to inject function names:

```
iex> fun = :hello
iex> Macro.to_string(quote do
       unquote(fun)(:world))
"hello(:world)"
```

In some cases, it may be necessary to inject many values inside a list. For example, imagine you have a list containing `[1, 2, 6]` and we want to inject `[3, 4, 5]` into it. Using `unquote` won’t yield the desired result:

```
iex> inner = [3, 4, 5]
iex> Macro.to_string(quote do
       [1, 2, unquote(inner), 6])
"[1, 2, [3, 4, 5], 6]"
```

That’s when `unquote_splicing` becomes handy:

```
iex> inner = [3, 4, 5]
iex> Macro.to_string(quote do
       [1, 2, unquote_splicing(inner), 6])
"[1, 2, 3, 4, 5, 6]"
```

Unquoting is very useful when working with macros. When writing macros, developers are able to receive code chunks and inject them inside other code chunks, being able to transform code or write code that generates code during compilation.

### 1.3 Escaping

As we have seen at the beginning of this chapter, only some values are valid quoted expressions. For example, a map is not a valid quoted expression. Nor a tuple with three elements. However, we have seen before those values can be expressed as a quoted expression:

```
iex> quote do: %{1 => 2}
{:%{}, [], [{1, 2}]}%
```

In some cases, you may need to inject such `values` into quoted `expressions`. To do that, we need to first escape those values into quoted expressions with the help of `Macro.escape/1`:

```
iex> quote do: Macro.escape(%{1 => 2})
{:%{}, [], [{1, 2}]}%
```
iex> map = %{hello: :world}
iex> Macro.escape(map)
{:%, [], [hello: :world]}

In other words, it is important to make a distinction in between a regular Elixir value (like a list, a map, a process, a reference, etc) and a quoted expression. Some values like integers, atoms and strings have a quoted expression equal to the value itself. Other values like maps needs to be explicitly converted. Finally, some values like functions and references cannot be converted to a quoted expression at all.

Macros receive quoted expressions and must return quoted expressions. However, sometimes during the function of a macro, you may need to work with values and making a distinction in between them may be required.

You can read more about `quote` and `unquote` in the *Kernel.SpecialForms* module [here](https://docs.stable.elixir/Kernel.SpecialForms.html). Documentation for `Macro.escape/1` and other quoted expressions related functions can be found in the *Macro* module [here](https://docs.stable.elixir/Macro.html).

With this introduction we have laid the ground to finally write out first macro, so let’s move to the next chapter.

### 1.4.2 2 Macros

- 2.1 Our first macro
- 2.2 Macros hygiene
- 2.3 The environment
- 2.4 Private macros
- 2.5 Write macros responsibly

Macros can be defined in Elixir using `defmacro/2`.

For this chapter, we will be using files instead of running code samples in IEx. That’s because the code samples will span multiple lines of code and typing them all in IEx can be counter-productive. You should be able to run the code samples by saving them into a `macros.exs` file and running it with `elixir macros.exs` or `iex macros.exs`.

#### 2.1 Our first macro

In order to better understand how macros work, let’s create a new module where we are going to implement `unless`, which does the opposite of `if`, as a macro and as a function:

```elixir
defmodule Unless do
  def fun_unless(clause, expression) do
    if (!clause, do: expression)
  end

  defmacro macro_unless(clause, expression) do
    quote do
      if (!unquote(clause), do: unquote(expression))
    end
  end
end
```

The function receives the arguments and pass them to `if`. However, the macro will receive quoted expressions, as we have learned in the previous chapter, inject them into the quote, finally returning another quoted expression.

Let’s start `iex` with the module above:
$ iex macros.exs

And play with those definitions:

```iex
require Unless
```

```iex
iex> Unless.macro_unless true, IO.puts "this should never be printed"
nil
```

```iex
iex> Unless.fun_unless true, IO.puts "this should never be printed"
"this should never be printed"
nil
```

Note that in our macro implementation, the sentence was not printed, although it was printed in our function implementation. That’s because the arguments to a function call are evaluated before calling the function. However macros do not evaluate the arguments, instead they receive the arguments as quoted expressions which are then transformed into other quoted expressions. In this case, we have rewritten our `unless` macro to become an `if` behind the scenes.

In other words, when invoked as:

```iex
Unless.macro_unless true, IO.puts "this should never be printed"
```

Our `macro_unless` macro received the following:

```
macro_unless(true, \{\:, [], [IO, :puts], [], ["this should never be printed"]\})
```

And it then return a quoted expression as follows:

```
\{\:, [], [
  \{\:, [], [true],
  \{\:, [], [IO, :puts], [], ["this should never be printed"]\}\}
```

We can actually verify this is the case by using `Macro.expand_once/2`:

```iex
iex> expr = quote do: Unless.macro_unless(true, IO.puts "this should never be printed")
iex> res = Macro.expand_once(expr, __ENV__)
iex> IO.puts Macro.to_string(res)
if(!true) do
  IO.puts("this should never be printed")
end
:ok
```

`Macro.expand_once/2` receives a quoted expression and expands it according to the current environment. In this case, it expanded/invoked the `Unless.macro_unless/2` macro and returned its result. We then proceeded to convert the returned quoted expression to a string and print it (we will talk about `__ENV__` still in this chapter).

That’s what macros are all about. They are about receiving quoted expressions and transforming them into something else. In fact, `unless/2` in Elixir is implemented as a macro:

```iex
defmacro unless(clause, options) do
  quote do
    if(!unquote(clause), do: unquote(options))
  end
end
```

Not only `unless/2`, `defmacro/2`, `def/2`, `defprotocol/2` and many constructs used throughout this getting started guide are implemented in pure Elixir, often as a macros. This means that the constructs being used to build the language, can be used by developers to extend the language to the domains they are working on.

We can define any function and macro we want, including ones that override the built-in definitions provided by Elixir. The only exceptions are Elixir special forms which are not implemented in Elixir and therefore cannot be overridden, the full list of special forms is available in "`Kernel.SpecialForms`" <docs/stable/elixir/Kernel.SpecialForms.html>`.  

1.4. Meta-programming in Elixir
2.2 Macros hygiene

Elixir macros have late resolution. This guarantees that a variable defined inside a quote won’t conflict with a variable defined in the context where that macro is expanded. For example:

```elixir
defmodule Hygiene do
  defmacro no_interference do
    quote do: a = 1
  end
end
defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.no_interference
    a
  end
end

HygieneTest.go
# => 13
```

In the example above, even if the macro injects `a = 1`, it does not affect the variable `a` defined by the `go` function. In case the macro wants to explicitly affect the context, it can use `var!`:

```elixir
defmodule Hygiene do
  defmacro interference do
    quote do: var!(a) = 1
  end
end
defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.interference
    a
  end
end

HygieneTest.go
# => 1
```

Variables hygiene only works because Elixir annotates variables with their context. For example, a variable `x` defined at the line 3 of a module, would be represented as:

```
{:x, [line: 3], nil}
```

However, a quoted variable is represented as:

```elixir
defmodule Sample do
  def quoted do
    quote do: x
  end
end

Sample.quoted #=> {:x, [line: 3], Sample}
```

Notice that the third element in the quoted variable is the atom `Sample`, instead of `nil`, which marks the variable as
coming from the Sample module. Therefore, Elixir considers those two variables come from different contexts and handle them accordingly.

Elixir provides similar mechanisms for imports and aliases too. This guarantees macros will behave as specified by its source module rather than conflicting with the target module where the macro is expanded. Hygiene can be passed under specific situations by using macros like \texttt{var!/2} and \texttt{alias!/2}, although one must be careful when using those as they directly change the user environment.

### 2.3 The environment

When using \texttt{Macro.expand_once/2} earlier in this chapter, we have used the special form \texttt{__ENV__}.

\texttt{__ENV__} returns an instance of \texttt{Macro.Env} which contains useful information about the compilation environment. It contains useful information like the current module, file and line, all variables defined in the current scope, as well as imports, requires and so on.

Let’s give it a try:

```iex
iex> __ENV__.module
nil
iex> __ENV__.file
"iex"
iex> __ENV__.requires
[IEx.Helpers, Kernel, Kernel.Typespec]
iex> require Integer
nil
iex> __ENV__.requires
[IEx.Helpers, Integer, Kernel, Kernel.Typespec]
```

Many of the functions in the \texttt{Macro} module expect an environment. You can read more about them in \emph{the docs for the ‘Macro’ module} \url{<docs/stable/elixir/Macro.html>} and learn more about the compilation environment with ‘Macro.Env \url{<docs/stable/elixir/Macro.Env.html>}’.

### 2.4 Private macros

Elixir also supports private macros via \texttt{defmacrop}. As private functions, these macros are only available inside the module that defines them, and only at compilation time.

It is important that a macro is defined before its usage. Failing to define a macro before its invocation will raise an error at runtime, since the macro won’t be expanded and will be translated to a function call:

```iex
defmodule Sample do
  ...> def four, do: two + two
  ...> defmacrop two, do: 2
  ...> end
** (CompileError) iex:2: function two/0 undefined
```

### 2.5 Write macros responsibly

Macros are a powerful construct and Elixir provides many mechanisms to ensure they are used responsibly:

- Macros are hygienic: by default, variables defined inside the macro are not going to affect the user code. Furthermore, functions calls and aliases available in the macro context are not going to leak into the user context;
- Macros are lexical: it is impossible to inject code or macros globally. Before using a macro, you need to explicitly require or import the module that defines the macro;
Macros are explicit: it is impossible to run a macro without explicitly invoking it. For example, some languages allow developers to completely rewrite functions behind the scenes, often via parse transforms or via some reflection mechanisms. In Elixir, a macro must be explicitly invoked in the caller;

Macros’ language is clear: many languages provide syntax shortcuts for `quote` and `unquote`. In Elixir, we preferred to have those explicitly spelled out, in order to clearly delimit the boundaries of a macro definition;

Even if Elixir attempts its best to provide a safe environment, the major responsibility still falls on the developers. That’s why the first rule of the macro club is **write macros responsibly**. Macros are harder to write than ordinary Elixir functions and it’s considered to be bad style to use them when they’re not necessary. Elixir already provides elegant mechanisms to write your every day code and macros should be saved as last resort.

With those lessons, we finish our introduction to macros. The next chapter is a brief discussion on DSLs, showing how we can mix macros and module attributes to annotate and extend modules and functions.

### 1.4.3 3 Domain Specific Languages

- 3.1 The `test` macro
- 3.2 Storing information with attributes

**Domain Specific Languages** allows developers to specialize their application to a particular domain. There are many language features that come together to aid developers to write Domain Specific Languages and in this chapter we will focus on just one of them.

In particular, we will focus on how macros and module attributes can be used together to create domain specific modules, that are focused on solving one particular example. To showcase our implementation, we will write a very simple test case implementation.

The goal is to build a module named `TestCase` that allows us to write the following:

```elixir
defmodule MyTest do
  use TestCase

  test "arithmetic operations" do
    4 = 2 + 2
  end

  test "list operations" do
    [1, 2, 3] = [1, 2] ++ [3]
  end
end
```

MyTest.run

In the example above, by using `TestCase`, we can define tests using the `test` macro and it automatically defines a function named `run` that will automatically run all tests for us. Our prototype will also simply rely on the match operator (\(=\)) as a mechanism to do assertions.

### 3.1 The `test` macro

Let’s start by defining a module that simply defines and imports the `test` macro when used:

```elixir
defmodule TestCase do
  # Callback invoked when by `use`.
  #
end
```

84 Chapter 1. Getting started guides
# For now it simply returns a quoted expression that
# imports the module itself into the user code.
@doc false
defmacro __using__(_opts) do
  quote do
    import TestCase
  end
end

@doc ""
Defines a test case with the given description.
"
## Examples

test "arithmetic operations" do
  4 = 2 + 2
end
"

defmacro test(description, do: block) do
  function_name = binary_to_atom("test " <> description)
  quote do
    def unquote(function_name)(), do: unquote(block)
  end
end

Assuming we defined TestCase in a file named tests.exs, we can open it up iex tests.exs and define our
first tests:

defmodule MyTest do
  ...> use TestCase
  ...> test "hello" do
  ...>   "hello" = "world"
  ...> end
  ...> end

For now we don’t have a mechanism to run tests, but we know that a function named “test hello” was defined behind
the scenes, so we can invoke it and it should fail:

defmodule MyTest do
  ...> test "hello"()
** (MatchError) no match of right hand side value: "world"

## Storing information with attributes

In order to finish our TestCase implementation, we need to be able to retrieve all defined test cases. One way of
doing such is by retrieving the tests at runtime via __MODULE__.____info__(:functions) which returns a list
of all functions in a given module. However, considering that we may want to store more information with each tests
beyond the test name, are more flexible approach is required.

When discussing module attributes in early chapters, we have discussed how they can be used as temporary storage
and that’s exactly what we will do in this section.

In the __using__/1 implementation, we will initialize a module attribute named @tests to an empty list, then
store each defined test in this attribute until we compile into into a run function.

Here is the updated code for the TestCase module:
defmodule TestCase do
  @doc false
  defmacro __using__(_opts) do
    quote do
      # Initialize @tests to an empty list
      @tests []
      # Invoke TestCase.__before_compile__/1 before the module is compiled
      @before_compile TestCase
    end
  end

  @doc ""
  Defines a test case with the given description.
  
  ## Examples
  
  test "arithmetic operations" do
    4 = 2 + 2
  end
  ""
  defmacro test(description, do: block) do
    function_name = binary_to_atom("test " <> description)
    quote do
      # Prepend the newly defined test to the list of tests
      @tests [unquote(function_name)|@tests]
      def unquote(function_name)(), do: unquote(block)
    end
  end

  # This will be invoked right before the target module is compiled
  # giving us the perfect opportunity to inject the 'run/0' function
  @doc false
  defmacro __before_compile__/1(env) do
    quote do
      def run do
        Enum.each @tests, fn name ->
          IO.puts "Running #{name}"
          apply(__MODULE__, name, [])
        end
      end
      end
    end
  end
end

By starting a new IEx session, we can now define our tests and run them:

iex> defmodule MyTest do
    ...
    use TestCase
    ...
    test "hello" do
      "hello" = "world"
    end
    ...
  end
iex> MyTest.run
Running test hello
Although we have glanced over some details, the bulk of how we can create domain specific modules in Elixir is here. Macros allows us to return quoted expressions that are executed in the caller, which in turn we use to transform code and to store relevant information in the target module via module attributes. Finally, with callbacks such as @before_compile, we have the perfect opportunity to inject code into the module when its definition is complete.

Besides @before_compile, there are other useful module attributes, like @on_definition and @after_compile and you can more about read them in the docs for the ‘‘Module’’ module </docs/stable/elixir/Module.html>‘‘. You can also find useful documentation about macros in the ‘‘Macro module </docs/stable/elixir/Macro.html>‘‘ and about the compilation environment in ‘‘Macro.Env </docs/stable/elixir/Macro.Env.html>‘‘.
2.1 Scoping Rules in Elixir (and Erlang)

• Types of Scope
• Elixir Scopes Are Lexical
• Scope Nesting and Shadowing
• The Top Level Scope
• Function Clause Scope
• Named Functions And Modules
• Case-like Clauses
• Try Blocks
• Comprehensions
• require, import, and alias
• Differences from Erlang

For everyday use it is sufficient to understand the basics of scoping rules in Elixir: that there’s the top level scope and function clause scope, and that named functions have their own peculiar differences from the more conventional anonymous functions.

But there are, in fact, quite a few rules you need to know to get a complete picture of the way scopes work in Elixir. In this technical article we will take a close look at all of the scoping rules and learn in what ways they differ from Erlang.

2.1.1 Types of Scope

In Elixir there are two types of scope:
• the top level scope
• function clause scope

There are a number of constructs that create new scope:
• modules and module-like structures: defmodule, defprotocol, defimpl
• functions: fn, def, defp
• comprehensions: for
• try block bodies
Most of the time user code in Elixir is structured in the following way. At the top level we define modules. Each module contains a number of attributes and function clauses. Inside a function clause there can be arbitrary number of expressions including control flow constructs like `case`, `if`, or `try`:

```
abc = "abc"            # T: top level scope

defmodule M do
  # M: module’s scope
  @doc "factorial"
  @limit 13
  def foo(n) do
    # F: function clause scope
    x = case n do
      0 -> 1
      i when i > 0 -> n * foo(n - 1)
      _ -> :undef
    end
    for x <- [1,2,3] do
      -x
    end
  end
end
```

Another way to visualise that structure, schematically:

```
# Figure 1
```

```
+-----------------------+
| Top level |
|            |            |
| +------------------+ +------------------+ |
| | Module | | Module |
| | +-------+ | +-------+ |
| | | Function clause | | Function clause |
| | +---------+ | +---------+ |
| | | Comprehension | | Comprehension |
| | +---------+ | +---------+ |
| | | Anon. function | | Anon. function |
| | +---------+ | +---------+ |
| | | Try block | | Try block |
| | +---------+ | +---------+ |
| | +-----------+ | +-----------+ |
| +---------------------------------+ |
```

When working in the interactive shell, the scope hierarchy is usually flat (“function clause” in the graphic below now refers to anonymous functions instead of named functions):

```
# Figure 2
```

```
+-----------------------+
| Top level |
+-----------+
```

---

**Chapter 2. Technical guides**
Those are the two most commonly seen structures for code organisation in Elixir.

In the general case, however, all scopes are arbitrarily nestable: we could imagine a case expression inside a comprehension or a top-level if expression defining different modules depending on some condition. For example:

```elixir
defmodule M do
  def say do
    "one"
  end
end

defmodule N do
  def say do
    "two"
  end
end
```

```bash
M.say #=> undefined function: M.say/0
N.say #=> undefined function: N.say/0
```

# define M
f.(1)
M.say #=> "one"
N.say #=> undefined function: N.say/0

# define N
f.(2)
M.say #=> "one"
N.say #=> "two"
```

In order to understand how the example above works, you should be aware of the fact the a module definition creates the module as its side-effect, so the module itself will be available globally. Only the name of the module is affected by the nesting of the defmodule call as we’ll see later in this article.

### 2.1. Scoping Rules in Elixir (and Erlang)
2.1.2 Elixir Scopes Are Lexical

This means that it is possible to determine the scope of every identifier only by looking at the source code. All variable bindings introduced in a scope are available until the end of that scope. Elixir has a few special forms that treat scopes a little differently (namely require, import, and alias). We will examine them at the end of this article.

2.1.3 Scope Nesting and Shadowing

According to the rules of lexical scope, any variables defined in the surrounding scope are accessible in all other scopes it contains.

In Figure 1 above, any variable defined in the top level scope will be accessible in the module’s scope and any scope nested inside it, and so on.

There is an exception to this rule which applies only to named functions: any variable coming from the surrounding scope has to be unquoted inside a function clause body.

Any variable in a nested scope whose name coincides with a variable from the surrounding scope will shadow that outer variable. In other words, the variable inside the nested scope temporarily hides the variable from the surrounding scope, but does not affect it in any way.

2.1.4 The Top Level Scope

The top level scope includes every variable and identifier defined outside of any other scope.

```elixir
x #=> undefined function: x/0
x = 1
x #=> 1
f = fn -> x end
f.() #=> 1
```

Named functions cannot be defined at the top level because a named function always belongs within a module. However, named functions can be imported into any lexical scope (including the top level scope) like this:

```elixir
import String, only: [reverse: 1]
reverse "Hello" #=> "olleH"
```

In fact, all functions and macros from the Kernel module are autoimported in the top level scope by the compiler.

2.1.5 Function Clause Scope

Each function clause defines a new lexical scope: any new variable bound inside it will not be available outside of that clause:

```elixir
defmodule M do
  def foo(x), do: -x

  # this ’x’ is completely independent from the one in ’foo/1’
  def bar(x), do: 2*x

  x = 1
```

shadowing in action: the 'x' in the argument list creates a variable local to the function clause’s body and has nothing to do with the previously defined ‘x’

```elixir
defmodule M
do
  def foo, do: "hi"
  foo() # will cause CompileError: undefined function foo/0
end
```

Apart from named functions, a new function clause scope is created for each module-like block, anonymous function, try block body, or comprehension body (see below).

```elixir
f = fn(x) ->
  a = x - 1
end

a #=> undefined function: a/0

f #=> (still the anonymous function defined above)
g #=> (the anonymous function we’ve just defined)
```

### 2.1.6 Named Functions And Modules

As mentioned before, named function have a couple of peculiarities.

First, defining a named function does not introduce a new binding into the current scope:

```elixir
defmodule M do
  def foo, do: "hi"
  foo() # will cause CompileError: undefined function foo/0
end
```

Second, named functions cannot directly access surrounding scope, one has to use `unquote` to achieve that:

```elixir
defmodule M do
  a = 1
end
```
Elixir Documentation, Release

# 'a' inside unquote() unambiguously refers to 'a' defined
# in the module's scope
def a, do: unquote(a)

# 'a' inside the body unambiguously refers to the function 'a/0'
def a(b), do: a + b
end

M.a #=> 1
M.a 3 #=> 4

Module scope works just like function clause scope: any variables defined between defmodule (or defprotocol, etc.) and its corresponding end will not be accessible outside of the module, but they will be available in the nested scopes of that module as per usual (modulo the unquoting caveat of named functions mentioned above).

It is important to understand that a module’s scope exists as long as it is being compiled. In other words, variables are not “compiled into” the module. The Module.function syntax is only applicable to named functions and that’s another thing that makes such functions special:

```elixir
defmodule M do
  x = "hello"
  def hi, do: unquote(x)
end
```

M.hi #=> "hello"
M.x #=> undefined function: x/0

You may be wondering how local function calls work when named functions don’t produce name bindings and don’t have direct access to the surrounding scope. The answer to this lies in the following rule followed by Elixir when trying to resolve an identifier to its value:

Any unbound identifier is treated as a local function call.

Let’s see how this works in code:

```elixir
defmodule P do
def f, do: "I am P’s f"
def g, do: f
end
defmodule Q do
def f, do: "I am Q’s f"
def g, do: f
end
```

# both P’s 'g' and Q’s 'g' refer to their local buddy named 'f'
P.g #=> "I am P’s f"
Q.g #=> "I am Q’s f"

# let’s make ‘f’ local in the top level scope
f #=> undefined function: f/0
import P
f #=> "I am P’s f"
```

One more note about module naming and nested modules: modules are always defined at the top level, no matter in what scope the actual call to defmodule is located. This means that as long the VM can find the .beam file with the module’s code at run time, it does not matter in which scope you reference that module’s name.

What the scoping does affect is the name the module will get:
defmodule P do
  # The actual module name will be P.Q, but it is implicitly aliased to Q
  # in P’s scope
  defmodule Q do
    def q(false), do: "sorry"
    def q(true) do
      # The actual module name will be P.Q.M
      defmodule M do
        def say, do: "hi"
      end
    end
  end
  # Q is resolved to P.Q
  def foo do
    Q.q false
  end
  # At run time, this has the same exact implementation as foo
  def bar do
    P.Q.q false
  end
end
P.foo #=> "sorry"
P.bar #=> "sorry"
P.Q.q false #=> "sorry"
# the module hasn’t been defined yet
P.Q.M.say #=> undefined function: P.Q.M.say/0
# after this call the P.Q.M module will become available
P.Q.q true
P.Q.M.say #=> "hi"

2.1.7 Case-like Clauses

Control flow constructs case, receive, and cond share a common trait:

- any variable introduced in a clause pattern/condition will be accessible only within that clause’s body
- any variable introduced inside some (but not all) clause bodies will become available in the surrounding scope (possibly with the default nil value)

Here are some examples of those rules in action:

case x do
  # both ’result’ and ’a’ are visible only within this clause’s body
  {:ok, result}=a -> IO.inspect(result); a

  # ’error’ is actually bound in the surrounding scope; its value will be nil
  # if ’x’ does not match :error
  :error -> error = true

  # ordinary shadowing: this ’x’ is visible only within the clause’s body and
  # it doesn’t affect the ’x’ from the surrounding scope
  [x] -> IO.inspect(x)
end
result  #=> undefined function: result/0
a       #=> undefined function: a/0

error  #=> true if x == :error, otherwise nil

Note: due to a bug in the 0.12.x series, cond’s conditions actually leak bindings to the surrounding scope. This should be fixed in 0.13.1.

```elixir
cond do
  a0 = false -> a = a0
  b = 1    -> b
  c = 2    -> c = 2
  true     -> d = 3
end
```

```elixir
a       #=> false (bound to false inside the 1st condition’s body)
b       #=> undefined function: b/0
c       #=> nil (the 2nd condition is truthy, so 'c = 2' was not evaluated)
d       #=> nil (the body with 'd = 3' was not evaluated,
              # so 'd' also leaks with the default value)
```

```elixir
if x = 3 do
  case y = :ok do
    :ok   -> :ok
    :error -> a = "it’s an error"
  end
else
  z = 11
end
```

```elixir
x       #=> 3
y       #=> :ok
a       #=> nil
z       #=> nil
```

### 2.1.8 Try Blocks

The `try` block works similar to `case` and `receive`, but it creates new scope, so it never leaks variable bindings to the surrounding scope.

```elixir
try do
  # all of the variables defined here are local to this block
  # (like in a function clause scope)
  a = 1
  b = a + 1
  c = d
rescue
  # these work like bindings in 'case' patterns
  x in [RuntimeError] -> y = x
  x                    -> z = x
end
```

```elixir
# none of the variables have leaked
a       #=> undefined function: a/0
b       #=> undefined function: b/0
c       #=> undefined function: c/0
d       #=> undefined function: d/0
```
2.1.9 Comprehensions

Comprehensions consist of two parts: the generator and the body.

Variables introduced in the generator part will only be visible within the body.

\[
\text{for } a = x \leftarrow [1, 2, 3, 4], \text{ do: } b = \{a, x\}
\]

\[
\text{#=> } \{\{1, 1\}, \{2, 2\}, \{3, 3\}, \{4, 4\}\}
\]

The comprehension body itself works like function clause scope:

\[
\text{for } x \leftarrow ["abc", "def"] \text{ do}
\]

\[
\text{import String, only: [reverse: 1]}
\]

\[
b = \text{reverse } x
\]

\[
\text{end}
\]

\[
\text{#=> } ["cba", "fed"]
\]

\[
b
\]

\[
\text{reverse } "hello"
\]

\[
\text{#=> undefined function: reverse/1}
\]

2.1.10 require, import, and alias

All of the rules described so far apply to variable bindings. When it comes to one of these three special forms, their effect persists until the end of the do block they are called in. Effectively, those forms see a slightly different scope division in which control flow constructs create a new lexical scope:

\[
\text{# top level scope}
\]

\[
defmodule M \text{ do}
\]

\[
\text{import String, only: [reverse: 1]}
\]

\[
def \text{foo}
\]

\[
\text{import String, only: [strip: 1]}
\]

\[
\text{IO.puts reverse("abc")} \text{ # ok: inherited from the surrounding scope}
\]

\[
\text{if true do}
\]

\[
\text{import String, only: [downcase: 1]}
\]

\[
\text{else}
\]

\[
\text{import String, only: [upcase: 1]}
\]

\[
\text{end}
\]

97
"hello"
>|> strip # ok: made local in the current scope with ‘import’
>|> downcase # error: no local function downcase/1
>|> upcase # ditto

def bar do
  # new scope
  IO.puts reverse("abc") # ok: inherited from the surrounding scope
  strip("hello") # error: no local function strip/1
end
def end

2.1.11 Differences from Erlang

Most of the scoping rules described here have been inherited from Erlang.

One notable difference is that modules simply contain forms and function clauses, they don’t have scope nor allow arbitrary expressions like modules in Elixir do.

There are two differences in the way case clause scope works in Erlang:

1. both bindings introduced in the pattern and in the body of a clause modify the surrounding scope
2. those variables that are bound in some (but not all) of the clauses will remain unbound in the surrounding scope (instead of getting the nil value like they do in Elixir); they are also called unsafe variables

```erlang
case 1 of
  1=A -> B = A;
  _   -> C = 1
end.
```

A. %=> 1  B. %=> 1  C. %=> variable ‘C’ is unbound

There is an if construct in Erlang that looks similar to cond, but works differently. It only allows guard expressions as conditions and those do not let you introduce variable bindings. Variables bound in clause bodies leak to the surrounding scope the same way they do in case.

```erlang
X = 1,
if
  X -> A = X;
  true -> B = X
end.
```

A. %=> variable ‘A’ is unbound  B. %=> 1

```erlang
Y = true,
if
  Y -> P = Y;
  true -> Q = Y
end.
```
P. % => true
Q. % => variable 'Q' is unbound

Refer to this page for more information about Erlang control flow constructs.

An assorted list of resources that describe various aspects of Erlang’s scoping rules:

- Matching, Guards and Scope of Variables from Erlang’s Getting Started guide.
- Scope of variables in the Erlang course.
- Static rules of variable scoping in Erlang paper
- case expression scope question on Erlang’s mailing list
3.1 v0.13.3

3.2 API Reference (v0.14.0-dev)

The API reference is broken down into separate apps that are all distributed with Elixir core. Each one contains its own set of modules.

3.2.1 Elixir v0.14.0-dev

- Modules
- Exceptions
- Protocols

**Modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>Agents are a simple abstraction around state</td>
</tr>
<tr>
<td>Application</td>
<td>A module for working with applications and defining application callbacks</td>
</tr>
<tr>
<td>Atom</td>
<td>This module provides data encoding and decoding functions according to RFC 4648</td>
</tr>
<tr>
<td>Base</td>
<td>Utilities for defining behaviour interfaces</td>
</tr>
<tr>
<td>Behaviour</td>
<td>This module provides macros and operators for bitwise operators. These macros can be used in guards.</td>
</tr>
<tr>
<td>Bitwise</td>
<td>Utilities for managing code compilation, code evaluation and code loading</td>
</tr>
<tr>
<td>Code</td>
<td>This module specifies the Dict API expected to be implemented by different dictionaries. It also provides a set of algorithms that enumerate over collections according to the Enumerable protocol.</td>
</tr>
<tr>
<td>Dict</td>
<td>Functions to format throw/catch/exit and exceptions</td>
</tr>
<tr>
<td>Enum</td>
<td>This module contains functions to manipulate files</td>
</tr>
<tr>
<td>Exception</td>
<td>A struct responsible to hold file information</td>
</tr>
<tr>
<td>File</td>
<td>Defines a File.Stream struct returned by File.stream!/2</td>
</tr>
<tr>
<td>File.Stat</td>
<td>Functions for working with floating point numbers</td>
</tr>
<tr>
<td>File.Stream</td>
<td>A behaviour module for implementing event handling functionality</td>
</tr>
<tr>
<td>Float</td>
<td>A behaviour module for implementing the server of a client-server relation</td>
</tr>
<tr>
<td>GenEvent</td>
<td>A key-value store</td>
</tr>
<tr>
<td>GenServer</td>
<td>A set store</td>
</tr>
<tr>
<td>HashDict</td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3.1 – continued from previous page

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>Functions handling IO</td>
</tr>
<tr>
<td>IO.Stream</td>
<td>Defines a IO.Stream struct returned by IO.stream/2 and IO.binstream/2</td>
</tr>
<tr>
<td>Inspect.Algebra</td>
<td>A set of functions for creating and manipulating algebra documents, as described in <a href="http://example.com">&quot;Strictly Pretty&quot; (2000) by Christian Lindig</a></td>
</tr>
<tr>
<td>Inspect.Opts</td>
<td>Defines the Inspect.Opts used by the Inspect protocol</td>
</tr>
<tr>
<td>Integer</td>
<td>Functions for working with integers</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel provides the default macros and functions Elixir imports into your environment. These can be skipped or cherry-picked via the <code>import</code> macro.</td>
</tr>
<tr>
<td>Kernel.ParallelCompiler</td>
<td>A module responsible for compiling files in parallel</td>
</tr>
<tr>
<td>Kernel.ParallelRequire</td>
<td>A module responsible for requiring files in parallel</td>
</tr>
<tr>
<td>Kernel.SpecialForms</td>
<td>In this module we define Elixir special forms. Special forms cannot be overridden by the developer.</td>
</tr>
<tr>
<td>Kernel.Typespec</td>
<td>Provides macros and functions for working with typespecs</td>
</tr>
<tr>
<td>Keyword</td>
<td>A keyword is a list of tuples where the first element of the tuple is an atom and the second element can be any value</td>
</tr>
<tr>
<td>List</td>
<td>Implements functions that only make sense for lists and cannot be part of the Enum protocol. In general, favor using the Enum API instead of List.</td>
</tr>
<tr>
<td>Macro</td>
<td>Conveniences for working with macros</td>
</tr>
<tr>
<td>Macro.Env</td>
<td>A struct that holds compile time environment information</td>
</tr>
<tr>
<td>Map</td>
<td>A Dict implementation that works on maps</td>
</tr>
<tr>
<td>Module</td>
<td>This module provides many functions to deal with modules during compilation time. It allows a developer to dynamically attach documentation, add, delete and register attributes, among others.</td>
</tr>
<tr>
<td>Node</td>
<td>Functions related to VM nodes</td>
</tr>
<tr>
<td>OptionParser</td>
<td>This module contains functions to parse command line arguments</td>
</tr>
<tr>
<td>Path</td>
<td>This module provides conveniences for manipulating or retrieving file system paths</td>
</tr>
<tr>
<td>Port</td>
<td>Functions related to Erlang ports</td>
</tr>
<tr>
<td>Process</td>
<td>Conveniences for working with processes and the process dictionary</td>
</tr>
<tr>
<td>Protocol</td>
<td>Functions for working with protocols</td>
</tr>
<tr>
<td>Range</td>
<td>Defines a Range</td>
</tr>
<tr>
<td>Record</td>
<td>Module to work, define and import records</td>
</tr>
<tr>
<td>Regex</td>
<td>Regular expressions for Elixir built on top of Erlang’s <code>re</code> module</td>
</tr>
<tr>
<td>Set</td>
<td>This module specifies the Set API expected to be implemented by different representations</td>
</tr>
<tr>
<td>Stream</td>
<td>Module for creating and composing streams</td>
</tr>
<tr>
<td>String</td>
<td>A String in Elixir is a UTF-8 encoded binary</td>
</tr>
<tr>
<td>StringIO</td>
<td>This module provides an IO device that wraps a string</td>
</tr>
<tr>
<td>Supervisor</td>
<td>A behaviour module for implementing supervision functionality</td>
</tr>
<tr>
<td>Supervisor.Spec</td>
<td>Convenience functions for defining a supervision specification</td>
</tr>
<tr>
<td>System</td>
<td>The System module provides access to variables used or maintained by the VM and to functions defined in the Code module.</td>
</tr>
<tr>
<td>Task</td>
<td>Conveniences for spawning and awaiting for tasks</td>
</tr>
<tr>
<td>Task.Supervisor</td>
<td>A task supervisor</td>
</tr>
<tr>
<td>Tuple</td>
<td>Functions for working with tuples</td>
</tr>
<tr>
<td>URI</td>
<td>Utilities for working with and creating URIs</td>
</tr>
<tr>
<td>Version</td>
<td>Functions for parsing and matching versions against requirements</td>
</tr>
</tbody>
</table>

#### Agent

**Overview**   Agents are a simple abstraction around state.

Often in Elixir there is a need to share or store state that must be accessed from different processes or by a same process in different points in time.

The Agent module provides a basic server implementation that allows state to be retrieved and updated via a simple API.

**Examples**   For example, in the Mix tool that ships with Elixir, we need to keep a set of all tasks executed by a given project. Since this set is shared, we can implement it with an Agent:
defmodule Mix.TasksServer do
  def start_link do
    Agent.start_link(fn -> HashSet.new end, name: __MODULE__)
  end

  @doc "Checks if the task has already executed"
  def executed?(task, project) do
    item = {task, project}
    Agent.get(__MODULE__, fn set ->
      item in set
    end)
  end

  @doc "Marks a task as executed"
  def put_task(task, project) do
    item = {task, project}
    Agent.update(__MODULE__, &Set.put(&1, item))
  end
end

Note that agents still provide a segregation in between the client and server APIs, as seen in GenServers. In particular, all code inside the function passed to the agent is executed by the agent. This distinction is important because you may want to avoid expensive operations inside the agent, as it will effectively block the agent until the request is fulfilled.

Consider these two examples:

```elixir
# Compute in the agent/server
def get_something(agent) do
  Agent.get(agent, fn state -> do_something_expensive(state) end)
end

# Compute in the agent/client
def get_something(agent) do
  Agent.get(agent, &(&1)) |> do_something_expensive()
end
```

The first one blocks the agent while the second one copies all the state to the client and executes the operation in the client. The trade-off here is exactly if the data is small enough to be sent to the client cheaply or large enough to require processing on the server (or at least some initial processing).

Name Registration

An Agent is bound to the same name registration rules as GenServers. Read more about it in the GenServer docs.

A word on distributed agents

It is important to consider the limitations of distributed agents. Agents work by sending anonymous functions in between the caller and the agent. In a distributed setup with multiple nodes, agents only work if the caller (client) and the agent have the same version of a given module.

This setup may exhibit issues when doing “rolling upgrades”. By rolling upgrades we mean the following situation: you wish to deploy a new version of your software by shutting down some of your nodes and replacing them by nodes running a new version of the software. In this setup, part of your environment will have one version of a given module and the other part another version (the newer one) of the same module; this may cause agents to crash. That said, if you plan to run in distributed environments, agents should likely be avoided.

Note, however, that agents work fine if you want to perform hot code swapping, as it keeps both the old and new versions of a given module. We detail how to do hot code swapping with agents in the next section.
**Hot code swapping**  An agent can have its code hot swapped live by simply passing a module, function and args tuple to the update instruction. For example, imagine you have an agent named `:sample` and you want to convert its inner state from some dict structure to a map. It can be done with the following instruction:

`{:update, :sample, {:advanced, {Enum, :into, [%{}][])}}`

The agent’s state will be added to the given list as the first argument.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cast/2</code></td>
<td>Performs a cast (fire and forget) operation on the agent state</td>
</tr>
<tr>
<td><code>get/3</code></td>
<td>Gets the agent value and executes the given function</td>
</tr>
<tr>
<td><code>get_and_update/3</code></td>
<td>Gets and updates the agent state in one operation</td>
</tr>
<tr>
<td><code>start/2</code></td>
<td>Starts an agent process without links (outside of a supervision tree)</td>
</tr>
<tr>
<td><code>start_link/2</code></td>
<td>Starts an agent linked to the current process</td>
</tr>
<tr>
<td><code>stop/2</code></td>
<td>Stops the agent</td>
</tr>
<tr>
<td><code>update/3</code></td>
<td>Updates the agent state</td>
</tr>
</tbody>
</table>

**Summary**

**Types**

- `on_start` :: {:ok, pid} | {:error, {:already_started, pid} | term}
  - Return values of `start*` functions

- `name` :: atom | {:global, term} | {:via, module, term}
  - The agent name

- `agent` :: pid | {atom, node} | name
  - The agent reference

- `state` :: term
  - The agent state

**Functions**

**cast (agent, fun)(function)**

Specs:

- `cast(agent, (state -> state)) :: :ok`
  - Performs a cast (fire and forget) operation on the agent state.
  - The function `fun` is sent to the `agent` which invokes the function passing the agent state. The function must return the new state.
  - Note that `cast` returns `:ok` immediately, regardless of whether the destination node or agent exists.

**get (agent, fun, timeout \ 5000)(function)**

Specs:

- `(get(agent, (state -> a), timeout) :: a) when a: var`
  - Gets the agent value and executes the given function.
  - The function `fun` is sent to the `agent` which invokes the function passing the agent state. The result of the function invocation is returned.
  - A timeout can also be specified (it has a default value of 5000).
get_and_update(agent, fun, timeout \ 5000)(function)
Specs:

*get_and_update(agent, (state -> {a, state}), timeout) :: a) when a: var

Gets and updates the agent state in one operation.

The function fun is sent to the agent which invokes the function passing the agent state. The function must return a tuple with two elements, the first being the value to return (i.e. the get value) and the second one is the new state.

A timeout can also be specified (it has a default value of 5000).

start(fun, options \ [])(function)
Specs:

*start((() -> term), GenServer.options) :: on_start

Starts an agent process without links (outside of a supervision tree).

See start_link/2 for more information.

start_link(fun, options \ [])(function)
Specs:

*start_link((() -> term), GenServer.options) :: on_start

Starts an agent linked to the current process.

This is often used to start the agent as part of a supervision tree.

Once the agent is spawned, the given function is invoked and its return value is used as the agent state. Note that start_link does not return until the given function has returned.

Options

The :name option is used for registration as described in the module documentation.

If the :timeout option is present, the agent is allowed to spend at most the given amount of milliseconds on initialization or it will be terminated and the start function will return {:error, :timeout}.

If the :debug option is present, the corresponding function in the ‘:sys module <http://www.erlang.org/doc/man/sys.html>’ will be invoked.

If the :spawn_opt option is present, its value will be passed as options to the underlying process as in Process.spawn/4.

Return values

If the server is successfully created and initialized, the function returns {:ok, pid}, where pid is the pid of the server. If there already exists an agent with the specified name, the function returns {:error, {:already_started, pid}} with the pid of that process.

If the given function callback fails with reason, the function returns {:error, reason}.

stop(agent, timeout \ 5000)(function)
Specs:

*stop(agent, timeout) :: :ok

Stops the agent.

Returns :ok if the agent is stopped within the given timeout.

update(agent, fun, timeout \ 5000)(function)

Updates the agent state.
The function `fun` is sent to the `agent` which invokes the function passing the agent state. The function must return the new state.

A timeout can also be specified (it has a default value of 5000). This function always returns `:ok`.

**Application**

**Overview** A module for working with applications and defining application callbacks.

In Elixir (actually, in Erlang/OTP), an application is a component implementing some specific functionality, that can be started and stopped as a unit, and which can be re-used in other systems as well.

Applications are defined with an application file named `APP.app` where `APP` is the APP name, usually in `underscore_case` convention. The application file must reside in the same `ebin` directory as the application’s modules bytecode.

In Elixir, Mix is responsible for compiling your source code and generating your application `.app` file. Furthermore, Mix is also responsible for configuring, starting and stopping your application and its dependencies. For this reason, this documentation will focus on the remaining aspects of your application: the application environment, and the application callback module.

You can learn more about Mix compilation of `.app` files by typing `mix help compile.app`.

**Application environment** Once an application is started, OTP provides an application environment that can be used to configure applications.

Assuming you are inside a Mix project, you can edit your application function in the `mix.exs` file to the following:

```elixir
def application do
  [env: [hello: :world]]
end
```

In the application function, we can define the default environment values for our application. By starting your application with `iex -S mix`, you can access the default value:

```elixir
Application.get_env(:APP_NAME, :hello)
#=> {:ok, :hello}
```

It is also possible to put and delete values from the application value, including new values that are not defined in the environment file (although those should be avoided).

In the future, we plan to support configuration files which allows developers to configure the environment of their dependencies.

Keep in mind that each application is responsible for its environment. Do not use the functions in this module for directly access or modify the environment of other application (as it may lead to inconsistent data in the application environment).

**Application module callback** Often times, an application defines a supervision tree that must be started and stopped when the application starts and stops. For such, we need to define an application module callback. The first step is to define the module callback in the application definition in the `mix.exs` file:

```elixir
def application do
  [mod: {MyApp, []}]
end
```

Our application now requires the `MyApp` module to provide an application callback. This can be done by invoking `use Application` in that module and defining a `start/2` callback, for example:
defmodule MyApp do
  use Application

  def start(_type, _args) do
    MyApp.Supervisor.start_link()
  end
end

start/2 most commonly returns {:ok, pid} or {:ok, pid, state} where pid identifies the supervision tree and the state is the application state. args is second element of the tuple given to the :mod option.

The type passed into start/2 is usually :normal unless in a distributed setup where applications takeover and failovers are configured. This particular aspect of applications can be read with more detail in the OTP documentation:

- http://www.erlang.org/doc/design_principles/applications.html

A developer may also implement the stop/1 callback (automatically defined by use Application) which does any application cleanup. It receives the application state and can return any value. Notice that shutting down the supervisor is automatically handled by the VM:

| app_dir/1 | Gets the directory for app |
| app_dir/2 | Returns the given path inside app_dir/1 |
| delete_env/3 | Deletes the key from the given app environment |
| ensure_all_started/2 | Ensures the given app and its applications are started |
| ensure_started/2 | Ensures the given app is started |
| fetch_env/2 | Returns the value for key in app’s environment in a tuple |
| format_error/1 | Formats the error reason returned by start/2, ensure_started/2, stop/1, load/1, and unload/1, returns a string |
| get_all_env/1 | Returns all key-value pairs for app |
| get_env/3 | Returns the value for key in app’s environment |
| load/1 | Loads the given app |
| put_env/4 | Puts the value in key for the given app |
| start/2 | Starts the given app |
| stop/1 | Stops the given app |
| unload/1 | Unloads the given app |

Types

app
  app :: atom

key
  key :: atom

value
  value :: term

start_type
  start_type :: :permanent | :transient | :temporary

Functions

app_dir(app) (function)
  Specs:
    • app_dir(app) :: String.t

3.2. API Reference (v0.14.0-dev)
Gets the directory for app.

This information is returned based on the code path. Here is an example:

```elixir
File.mkdir_p!("foo/ebin")
Code.prepend_path("foo/ebin")
Application.app_dir(:foo)
#=> "foo"
```

Even though the directory is empty and there is no .app file it is considered the application directory based on the name “foo/ebin”. The name may contain a dash – which is considered to be the app version and it is removed for the lookup purposes:

```elixir
File.mkdir_p!("bar-123/ebin")
Code.prepend_path("bar-123/ebin")
Application.app_dir(:bar)
#=> "bar-123"
```

For more information on code paths, check the Code module in Elixir and also Erlang’s :code module.

**app_dir(app, path)** (function)

Specs:

- `app_dir(app, String.t) :: String.t`

Returns the given path inside `app_dir/1`.

**delete_env(app, key, opts \ [\])** (function)

Specs:

- `delete_env(app, key, timeout: timeout, persistent: boolean) :: :ok`

Deletes the `key` from the given app environment.

See `put_env/4` for a description of the options.

**ensure_all_started(app, type \ :temporary)** (function)

Specs:

- `ensure_all_started(app, start_type) :: {:ok, [app]} | {:error, term}`

Ensures the given app and its applications are started.

Same as `start/2` but also starts the applications listed under :applications in the .app file in case they were not previously started.

**ensure_started(app, type \ :temporary)** (function)

Specs:

- `ensure_started(app, start_type) :: :ok | {:error, term}`

Ensures the given app is started.

Same as `start/2` but returns :ok if the application was already started. This is useful in scripts and in test setup, where test applications need to be explicitly started:

```elixir
:ok = Application.ensure_started(:my_test_dep)
```

**fetch_env(app, key)** (function)

Specs:

- `fetch_env(app, key) :: {:ok, value} | :error`

Returns the value for `key` in app’s environment in a tuple.

If the specified application is not loaded, or the configuration parameter does not exist, the function returns :error.
**format_error(reason)** (function)

Specs:

- *format_error(any)** :: **String.t*

Formats the error reason returned by `start/2, ensure_started/2, stop/1, load/1 and unload/1`, returning a string.

**get_all_env(app)** (function)

Specs:

- *get_all_env(app)** :: **[{key, value}]**

Returns all key-value pairs for `app`.

**get_env(app, key, default \ \ nil)** (function)

Specs:

- *get_env(app, key, value)** :: **value**

Returns the value for `key` in `app`'s environment.

If the specified application is not loaded, or the configuration parameter does not exist, the function returns the `default` value.

**load(app)** (function)

Specs:

- *load(app)** :: **:ok | {:error, term}**

Loads the given `app`.

In order to be loaded, an `.app` file must be in the load paths. All `:included_applications` will also be loaded.

Loading the application does not start it nor load its modules, but it does load its environment.

**put_env(app, key, value, opts \ \ [])** (function)

Specs:

- *put_env(app, key, value, timeout: timeout, persistent: boolean)** :: **:ok**

Puts the `value` in `key` for the given `app`.

**Options**

- **:timeout** - the timeout for the change (defaults to 5000ms);
- **:persistent** - persists the given value on application load and reloads;

If `put_env/4` is called before the application is loaded, the application environment values specified in the `.app` file will override the ones previously set.

The persistent option can be set to true when there is a need to guarantee parameters set with this function will not be overridden by the ones defined in the application resource file on load. This means persistent values will stick after the application is loaded and also on application reload.

**start(app, type \ \ :temporary)** (function)

Specs:

- *start(app, start_type)** :: **:ok | {:error, term}**

Starts the given `app`.

If the `app` is not loaded, the application will first be loaded using `load/1`. Any included application, defined in the `:included_applications` key of the `.app` file will also be loaded, but they won’t be started.
Furthermore, all applications listed in the :applications key must be explicitly started before this application is. If not, \{:error, {:not_started, app}\} is returned, where app is the name of the missing application.

In case you want to automatically load and start all of app's dependencies, see ensure_all_started/2.

The type argument specifies the type of the application:

- **:permanent** - if app terminates, all other applications and the entire node are also terminated;
- **:transient** - if app terminates with :normal reason, it is reported but no other applications are terminated. If a transient application terminates abnormally, all other applications and the entire node are also terminated;
- **:temporary** - if app terminates, it is reported but no other applications are terminated (the default);

Note that it is always possible to stop an application explicitly by calling stop/1. Regardless of the type of the application, no other applications will be affected.

Note also that the :transient type is of little practical use, since when a supervision tree terminates, the reason is set to :shutdown, not :normal.

**stop** (app) (function)
Specs:

- **stop(app)** :: :ok | {:error, term}

  Stops the given app.

  When stopped, the application is still loaded.

**unload** (app) (function)
Specs:

- **unload(app)** :: :ok | {:error, term}

  Unloads the given app.

  It will also unload all :included_applications. Note that the function does not purge the application modules.

**Atom**

Overview

**Summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_char_list/1</td>
<td>Converts an atom to a char list</td>
</tr>
<tr>
<td>to_string/1</td>
<td>Converts an atom to string</td>
</tr>
</tbody>
</table>

**Functions**

**to_char_list** (atom) (function)
Specs:

- **to_char_list(atom)** :: char_list

  Converts an atom to a char list.

  Inlined by the compiler.

**to_string** (atom) (function)
Specs:

- **to_string(atom)** :: String.t
Converts an atom to string.
Inlined by the compiler.

**Base**

**Overview**  This module provides data encoding and decoding functions according to RFC 4648.
This document defines the commonly used base 64, base 32, and base 16 encoding schemes.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode16!/1</td>
<td>Decodes a base 16 encoded string into a binary string</td>
</tr>
<tr>
<td>decode16/1</td>
<td>Decodes a base 16 encoded string into a binary string</td>
</tr>
<tr>
<td>decode32!/1</td>
<td>Decodes a base 32 encoded string into a binary string</td>
</tr>
<tr>
<td>decode32/1</td>
<td>Decodes a base 32 encoded string into a binary string</td>
</tr>
<tr>
<td>decode64!/1</td>
<td>Decodes a base 64 encoded string into a binary string</td>
</tr>
<tr>
<td>decode64/1</td>
<td>Decodes a base 64 encoded string into a binary string</td>
</tr>
<tr>
<td>encode16/1</td>
<td>Encodes a binary string into a base 16 encoded string</td>
</tr>
<tr>
<td>encode32/1</td>
<td>Encodes a binary string into a base 32 encoded string</td>
</tr>
<tr>
<td>encode64/1</td>
<td>Encodes a binary string into a base 64 encoded string</td>
</tr>
<tr>
<td>hex_decode32!/1</td>
<td>Decodes a base 32 encoded string with extended hexadecimal alphabet into a binary string</td>
</tr>
<tr>
<td>hex_decode32/1</td>
<td>Decodes a base 32 encoded string with extended hexadecimal alphabet into a binary string</td>
</tr>
<tr>
<td>hex_encode32/1</td>
<td>Encodes a binary string into a base 32 encoded string with an extended hexadecimal alphabet</td>
</tr>
<tr>
<td>url_decode64!/1</td>
<td>Decodes a base 64 encoded string with URL and filename safe alphabet into a binary string</td>
</tr>
<tr>
<td>url_decode64/1</td>
<td>Decodes a base 64 encoded string with URL and filename safe alphabet into a binary string</td>
</tr>
<tr>
<td>url_encode64/1</td>
<td>Encodes a binary string into a base 64 encoded string with URL and filename safe alphabet</td>
</tr>
</tbody>
</table>

**Functions**

**decode16(string)(function)**

Specs:

- decode16(binary) :: {:ok, binary} | :error

Decodes a base 16 encoded string into a binary string.

The following alphabet is used both for encoding and decoding:

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>12</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>9</td>
<td>13</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>A</td>
<td>14</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>B</td>
<td>15</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

```elixir```
```iex> Base.decode16("666F6F626172")```
```{:ok, "foobar"}```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Decodes a base 16 encoded string into a binary string.

An ArgumentError exception is raised if the padding is incorrect or a non-alphabet character is present in the string.

Examples

```iex> Base.decode16!("66F626172")
"foobar"
```

decode32(string)(function)

Specs:

- `decode32(binary) :: {:ok, binary} | :error`

Decodes a base 32 encoded string into a binary string.

The following alphabet is used both for encoding and decoding:

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>9</td>
<td>J</td>
<td>18</td>
<td>S</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>10</td>
<td>K</td>
<td>19</td>
<td>T</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>11</td>
<td>L</td>
<td>20</td>
<td>U</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>12</td>
<td>M</td>
<td>21</td>
<td>V</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>13</td>
<td>N</td>
<td>22</td>
<td>W</td>
<td>31</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>14</td>
<td>O</td>
<td>23</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>15</td>
<td>P</td>
<td>24</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>H</td>
<td>16</td>
<td>Q</td>
<td>25</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>I</td>
<td>17</td>
<td>R</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples

```iex> Base.decode32("MZXW6YTBOI======")
{:ok, "foobar"}
```

decode32!(string)(function)

Specs:

- `decode32!(binary) :: binary`

Decodes a base 32 encoded string into a binary string.

An ArgumentError exception is raised if the padding is incorrect or a non-alphabet character is present in the string.

Examples

```iex> Base.decode32!("MZXW6YTBOI======")
"foobar"
```

decode64(string)(function)

Specs:

- `decode64(binary) :: {:ok, binary} | :error`

Decodes a base 64 encoded string into a binary string.

The following alphabet is used both for encoding and decoding:
Examples

iex> Base.decode64("Zm9vYmFy")
{:ok, "foobar"}

decode64! (string)(function)
Specs:
  •decode64!(binary) :: binary

Decodes a base 64 encoded string into a binary string.

The following alphabet is used both for encoding and decoding:

An ArgumentError exception is raised if the padding is incorrect or a non-alphabet character is present in the string.

Examples

iex> Base.decode64!("Zm9vYmFy")
"foobar"

encode16 (data)(function)
Specs:
  •encode16(binary) :: binary

Encodes a binary string into a base 16 encoded string.

Examples

iex> Base.encode16("foobar")
"666f6f626172"

encode32 (data)(function)
Specs:
  •encode32(binary) :: binary

Encodes a binary string into a base 32 encoded string.
Examples

```iex> Base.encode32("foobar")
"MZXW6YTBOI======"
```

**encode64**(data) (function)

specs:

- `encode64(binary) :: binary`

Encodes a binary string into a base 64 encoded string.

Examples

```iex> Base.encode64("foobar")
"Zm9vYmFy"
```

**hex_decode32**(string) (function)

specs:

- `hex_decode32(binary) :: {:ok, binary} | :error`

Decodes a base 32 encoded string with extended hexadecimal alphabet into a binary string.

The following alphabet is used both for encoding and decoding:

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>18</td>
<td>I</td>
<td>27</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>A</td>
<td>19</td>
<td>J</td>
<td>28</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
<td>B</td>
<td>20</td>
<td>K</td>
<td>29</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>12</td>
<td>C</td>
<td>21</td>
<td>L</td>
<td>30</td>
<td>U</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>13</td>
<td>D</td>
<td>22</td>
<td>M</td>
<td>31</td>
<td>V</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>14</td>
<td>E</td>
<td>23</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>15</td>
<td>F</td>
<td>24</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>16</td>
<td>G</td>
<td>25</td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>17</td>
<td>H</td>
<td>26</td>
<td>Q</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples

```iex> Base.hex_decode32("CPNMUOJ1E8======")
{:ok, "foobar"}
```

**hex_decode32!**(string) (function)

specs:

- `hex_decode32!(binary) :: binary`

Decodes a base 32 encoded string with extended hexadecimal alphabet into a binary string.

An ArgumentError exception is raised if the padding is incorrect or a non-alphabet character is present in the string.

Examples

```iex> Base.hex_decode32!("CPNMUOJ1E8======")
"foobar"
```

**hex_encode32**(data) (function)

specs:

- `hex_encode32(binary) :: binary`

Encodes a binary string into a base 32 encoded string with an extended hexadecimal alphabet.
Examples

```
  iex> Base.hex_encode32("foobar")
  "CPNNUOJ1E8========"
```

**url_decode64(string)** (function)

Specs:

- `url_decode64(binary) :: {:ok, binary} | :error`

Decodes a base 64 encoded string with URL and filename safe alphabet into a binary string.

The following alphabet is used both for encoding and decoding:

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
</tr>
<tr>
<td>8</td>
<td>I</td>
</tr>
<tr>
<td>9</td>
<td>J</td>
</tr>
<tr>
<td>10</td>
<td>K</td>
</tr>
<tr>
<td>11</td>
<td>L</td>
</tr>
<tr>
<td>12</td>
<td>M</td>
</tr>
<tr>
<td>13</td>
<td>N</td>
</tr>
<tr>
<td>14</td>
<td>O</td>
</tr>
<tr>
<td>15</td>
<td>P</td>
</tr>
<tr>
<td>16</td>
<td>Q</td>
</tr>
<tr>
<td>17</td>
<td>R</td>
</tr>
<tr>
<td>18</td>
<td>S</td>
</tr>
<tr>
<td>19</td>
<td>T</td>
</tr>
<tr>
<td>20</td>
<td>U</td>
</tr>
<tr>
<td>21</td>
<td>V</td>
</tr>
<tr>
<td>22</td>
<td>W</td>
</tr>
<tr>
<td>23</td>
<td>X</td>
</tr>
<tr>
<td>24</td>
<td>Y</td>
</tr>
<tr>
<td>25</td>
<td>Z</td>
</tr>
<tr>
<td>26</td>
<td>a</td>
</tr>
<tr>
<td>27</td>
<td>b</td>
</tr>
<tr>
<td>28</td>
<td>c</td>
</tr>
<tr>
<td>29</td>
<td>d</td>
</tr>
<tr>
<td>30</td>
<td>e</td>
</tr>
<tr>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>32</td>
<td>g</td>
</tr>
<tr>
<td>33</td>
<td>h</td>
</tr>
<tr>
<td>34</td>
<td>i</td>
</tr>
<tr>
<td>35</td>
<td>j</td>
</tr>
<tr>
<td>36</td>
<td>k</td>
</tr>
<tr>
<td>37</td>
<td>l</td>
</tr>
<tr>
<td>38</td>
<td>m</td>
</tr>
<tr>
<td>39</td>
<td>n</td>
</tr>
<tr>
<td>40</td>
<td>o</td>
</tr>
<tr>
<td>41</td>
<td>p</td>
</tr>
<tr>
<td>42</td>
<td>q</td>
</tr>
<tr>
<td>43</td>
<td>r</td>
</tr>
<tr>
<td>44</td>
<td>s</td>
</tr>
<tr>
<td>45</td>
<td>t</td>
</tr>
<tr>
<td>46</td>
<td>u</td>
</tr>
<tr>
<td>47</td>
<td>v</td>
</tr>
<tr>
<td>48</td>
<td>w</td>
</tr>
<tr>
<td>49</td>
<td>x</td>
</tr>
<tr>
<td>50</td>
<td>y</td>
</tr>
<tr>
<td>51</td>
<td>z</td>
</tr>
<tr>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>53</td>
<td>1</td>
</tr>
<tr>
<td>54</td>
<td>2</td>
</tr>
<tr>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>56</td>
<td>4</td>
</tr>
<tr>
<td>57</td>
<td>5</td>
</tr>
<tr>
<td>58</td>
<td>6</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
</tr>
<tr>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>61</td>
<td>9</td>
</tr>
<tr>
<td>62</td>
<td>10</td>
</tr>
<tr>
<td>63</td>
<td>11</td>
</tr>
</tbody>
</table>

Examples

```
  iex> Base.url_decode64("_3-_A==")
  {:ok, <<255,127,254,252>>}
```

**url_decode64!(string)** (function)

Specs:

- `url_decode64!(binary) :: binary`

Decodes a base 64 encoded string with URL and filename safe alphabet into a binary string. An ArgumentError exception is raised if the padding is incorrect or a non-alphabet character is present in the string.

Examples

```
  iex> Base.url_decode64!("_3-_A==")
  <<255,127,254,252>>
```

**url_encode64(data)** (function)

Specs:
url_encode64(binary) :: binary

Encodes a binary string into a base 64 encoded string with URL and filename safe alphabet.

Examples

```
  iex> Base.url_encode64(<<255,127,254,252>>)
  ":-3__A=="
```

Behaviour

Overview

Utilities for defining behaviour interfaces.

Behaviours can be referenced by other modules to ensure they implement required callbacks.

For example, you can specify the URI.Parser behaviour as follows:

```
defmodule URI.Parser do
  use Behaviour

  @doc "Parses the given URL"
  defcallback parse(uri_info :: URI.t) :: URI.t

  @doc "Defines a default port"
  defcallback default_port() :: integer
end
```

And then a module may use it as:

```
defmodule URI.HTTP do
  @behaviour URI.Parser
  def default_port(), do: 80
  def parse(info), do: info
end
```

If the behaviour changes or URI.HTTP does not implement one of the callbacks, a warning will be raised.

Implementation

Since Erlang R15, behaviours must be defined via `@callback` attributes. `defcallback` is a simple mechanism that defines the `@callback` attribute according to the given type specification. `defcallback` allows documentation to be created for the callback and defines a custom function signature.

The callbacks and their documentation can be retrieved via the `__behaviour__` callback function.

<table>
<thead>
<tr>
<th>Summary</th>
<th>defcallback/1</th>
<th>defmacrocallback/1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Define a function callback according to the given type specification</td>
<td>Define a macro callback according to the given type specification</td>
</tr>
</tbody>
</table>

Macros

defcallback (spec) (macro)

Define a function callback according to the given type specification.

defmacrocallback (spec) (macro)

Define a macro callback according to the given type specification.

Bitwise

Overview

This module provides macros and operators for bitwise operators. These macros can be used in guards.
The easiest way to use is to simply import them into your module:

```iex
use Bitwise
iex> bnot 1
-2
iex> 1 &&& 1
1
```

You can select to include only or skip operators by passing options:

```iex
use Bitwise, only_operators: true
iex> 1 &&& 1
1
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;&amp;/2</code></td>
<td>Bitwise and as operator</td>
</tr>
<tr>
<td><code>&lt;&lt;&lt;/2</code></td>
<td>Arithmetic bitshift left as operator</td>
</tr>
<tr>
<td><code>&gt;&gt;&gt;/2</code></td>
<td>Arithmetic bitshift right as operator</td>
</tr>
<tr>
<td><code>^^^/2</code></td>
<td>Bitwise xor as operator</td>
</tr>
<tr>
<td><code>__using__/1</code></td>
<td>Allow a developer to use this module in their programs with the following options:</td>
</tr>
<tr>
<td><code>band/2</code></td>
<td>Bitwise and</td>
</tr>
<tr>
<td><code>bnot/1</code></td>
<td>Bitwise not</td>
</tr>
<tr>
<td><code>bor/2</code></td>
<td>Bitwise or</td>
</tr>
<tr>
<td><code>bsl/2</code></td>
<td>Arithmetic bitshift left</td>
</tr>
<tr>
<td><code>bsr/2</code></td>
<td>Arithmetic bitshift right</td>
</tr>
<tr>
<td><code>bxor/2</code></td>
<td>Bitwise xor</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>~~~/1</code></td>
<td>Bitwise not as operator</td>
</tr>
</tbody>
</table>

### Macros

- **left `&&&` right** *(macro)*
  - Bitwise and as operator.

- **left `<<<` right** *(macro)*
  - Arithmetic bitshift left as operator.

- **left `>>>` right** *(macro)*
  - Arithmetic bitshift right as operator.

- **left `^^^` right** *(macro)*
  - Bitwise xor as operator.

- **__using__**(options)**(macro)**
  - Allow a developer to use this module in their programs with the following options:
    - `:only_operators` - Include only operators;
    - `:skip_operators` - Skip operators;

- **band**(left, right)**(macro)**
  - Bitwise and.

- **bnot**(expr)**(macro)**
  - Bitwise not.

- **bor**(left, right)**(macro)**
  - Bitwise or.

- **bsl**(left, right)**(macro)**
  - Arithmetic bitshift left.
bsr(left, right)(macro)
Arithmetic bitshift right.

bxor(left, right)(macro)
Bitwise xor.

left ||| right(macro)
Bitwise or as operator.

~~~expr(macro)
Bitwise not as operator.

Code

Overview  Utilities for managing code compilation, code evaluation and code loading.

This module complements Erlang's code module to add behaviour which is specific to Elixir.

<table>
<thead>
<tr>
<th>Code Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append_path/1</td>
<td>Append a path to the Erlang VM code path</td>
</tr>
<tr>
<td>compile_quoted/2</td>
<td>Compiles the quoted expression</td>
</tr>
<tr>
<td>compile_string/2</td>
<td>Compiles the given string</td>
</tr>
<tr>
<td>compiler_options/0</td>
<td>Gets the compilation options from the code server</td>
</tr>
<tr>
<td>compiler_options/1</td>
<td>Sets compilation options</td>
</tr>
<tr>
<td>delete_path/1</td>
<td>Delete a path from the Erlang VM code path</td>
</tr>
<tr>
<td>ensure_compiled/1</td>
<td>Ensures the given module is compiled and loaded</td>
</tr>
<tr>
<td>ensureCompiled/1</td>
<td>Ensures the given module is compiled and loaded</td>
</tr>
<tr>
<td>ensure_loaded/1</td>
<td>Ensures the given module is loaded</td>
</tr>
<tr>
<td>ensure_loaded?/1</td>
<td>Ensures the given module is loaded</td>
</tr>
<tr>
<td>eval_file/2</td>
<td>Evals the given file</td>
</tr>
<tr>
<td>eval_quoted/3</td>
<td>Evaluate the quoted contents</td>
</tr>
<tr>
<td>eval_string/3</td>
<td>Evaluate the contents given by string</td>
</tr>
<tr>
<td>get_docs/2</td>
<td>Returns the docs for the given module</td>
</tr>
<tr>
<td>load_file/2</td>
<td>Load the given file</td>
</tr>
<tr>
<td>loaded_files/0</td>
<td>List all loaded files</td>
</tr>
<tr>
<td>prepend_path/1</td>
<td>Prepend a path to the Erlang VM code path</td>
</tr>
<tr>
<td>require_file/1</td>
<td>Requires the given file</td>
</tr>
<tr>
<td>string_to_quoted!/2</td>
<td>Convert the given string to its quoted form</td>
</tr>
<tr>
<td>string_to_quoted/2</td>
<td>Convert the given string to its quoted form</td>
</tr>
<tr>
<td>unload_files/1</td>
<td>Remove files from the loaded files list</td>
</tr>
</tbody>
</table>

Functions

append_path(path)(function)
Append a path to the Erlang VM code path.

The path is expanded with Path.expand/1 before being appended.

compile_quoted(quoted, file \ "nofile") (function)
Compiles the quoted expression.

Returns a list of tuples where the first element is the module name and the second one is its byte code (as a binary).

compile_string(string, file \ "nofile") (function)
Compiles the given string.

Returns a list of tuples where the first element is the module name and the second one is its byte code (as a binary).
For compiling many files at once, check `Kernel.ParallelCompiler.files/2`.

**compiler_options()** (function)

Gets the compilation options from the code server.

Check `compiler_options/1` for more information.

**compiler_options(opts)** (function)

Sets compilation options.

These options are global since they are stored by Elixir’s Code Server.

Available options are:

- `:docs` - when `true`, retain documentation in the compiled module, `true` by default;
- `:debug_info` - when `true`, retain debug information in the compiled module. This allows a developer to reconstruct the original source code, `false` by default;
- `:ignore_module_conflict` - when `true`, override modules that were already defined without raising errors, `false` by default;
- `:warnings_as_errors` - cause compilation to fail when warnings are generated;

**delete_path(path)** (function)

Delete a path from the Erlang VM code path.

The path is expanded with `Path.expand/1` before being deleted.

**ensure_compiled(module)** (function)

Ensures the given module is compiled and loaded.

If the module is already loaded, it works as no-op. If the module was not loaded yet, it checks if it needs to be compiled first and then tries to load it.

If it succeeds loading the module, it returns `{:module, module}`. If not, returns `{:error, reason}` with the error reason.

Check `ensure_loaded/1` for more information on module loading and when to use `ensure_loaded/1` or `ensure_compiled/1`.

**ensure_compiled?(module)** (function)

Ensures the given module is compiled and loaded.

Similar to `ensure_compiled/1`, but returns `true` if the module is already loaded or was successfully loaded and compiled. Returns `false` otherwise.

**ensure_loaded(module)** (function)

Ensures the given module is loaded.

If the module is already loaded, this works as no-op. If the module was not yet loaded, it tries to load it.

If it succeeds loading the module, it returns `{:module, module}`. If not, returns `{:error, reason}` with the error reason.

**Code loading on the Erlang VM**

Erlang has two modes to load code: interactive and embedded.

By default, the Erlang VM runs in interactive mode, where modules are loaded as needed. In embedded mode the opposite happens, as all modules need to be loaded upfront or explicitly.

Therefore, this function is used to check if a module is loaded before using it and allows one to react accordingly. For example, the `URI` module uses this function to check if a specific parser exists for a given URI scheme.

`:elixir:func:‘Code.ensure_compiled/1‘
Elixir also contains an `ensure_compiled/1` function that is a superset of `ensure_loaded/1`.

Since Elixir’s compilation happens in parallel, in some situations you may need to use a module that was not yet compiled, therefore it can’t even be loaded.

`ensure_compiled/1` halts the current process until the module we are depending on is available.

In most cases, `ensure_loaded/1` is enough. `ensure_compiled/1` must be used in rare cases, usually involving macros that need to invoke a module for callback information.

### `ensure_loaded?` (module) (function)
Ensures the given module is loaded.

Similar to `ensure_loaded/1`, but returns `true` if the module is already loaded or was successfully loaded. Returns `false` otherwise.

### `eval_file` (file, relative_to \ nil) (function)
Evals the given file.

Accepts `relative_to` as an argument to tell where the file is located.

While `load_file` loads a file and returns the loaded modules and their byte code, `eval_file` simply evaluates the file contents and returns the evaluation result and its bindings.

### `eval_quoted` (quoted, binding \ [], opts \ []) (function)
Evaluate the quoted contents.

See `eval_string/3` for a description of arguments and return values.

**Examples**

```
iex> contents = quote(do: var!(a) + var!(b))
iex> Code.eval_quoted(contents, [a: 1, b: 2], file: __ENV__.file, line: __ENV__.line) [3, [a: 1, b: 2]]
```

For convenience, you can pass `__ENV__` as the `opts` argument and all options will be automatically extracted from the current environment:

```
iex> contents = quote(do: var!(a) + var!(b))
iex> Code.eval_quoted(contents, [a: 1, b: 2], __ENV__) [3, [a: 1, b: 2]]
```

### `eval_string` (string, binding \ [], opts \ []) (function)
Evaluate the contents given by `string`.

The `binding` argument is a keyword list of variable bindings. The `opts` argument is a keyword list of environment options.

Those options can be:

- `:file` - the file to be considered in the evaluation
- `:line` - the line on which the script starts
- `:delegate_locals_to` - delegate local calls to the given module, the default is to not delegate

Additionally, the following scope values can be configured:

- `:aliases` - a list of tuples with the alias and its target
- `:requires` - a list of modules required
- `:functions` - a list of tuples where the first element is a module and the second a list of imported function names and arity. The list of function names and arity must be sorted
• :macros - a list of tuples where the first element is a module and the second a list of imported macro names and arity. The list of function names and arity must be sorted

Notice that setting any of the values above overrides Elixir’s default values. For example, setting :requires to [], will no longer automatically require the Kernel module; in the same way setting :macros will no longer auto-import Kernel macros like if, case, etc.

Returns a tuple of the form {value, binding}, where value is the value returned from evaluating string. If an error occurs while evaluating string an exception will be raised.

binding is a keyword list with the value of all variable bindings after evaluating string. The binding key is usually an atom, but it may be a tuple for variables defined in a different context.

Examples

```
iex> Code.eval_string("a + b", [a: 1, b: 2], file: __ENV__.file, line: __ENV__.line)
{3, [a: 1, b: 2]}

iex> Code.eval_string("c = a + b", [a: 1, b: 2], __ENV__)
{3, [a: 1, b: 2, c: 3]}

iex> Code.eval_string("a = a + b", [a: 1, b: 2])
{3, [a: 3, b: 2]}
```

For convenience, you can pass __ENV__ as the opts argument and all imports, requires and aliases defined in the current environment will be automatically carried over:

```
iex> Code.eval_string("a + b", [a: 1, b: 2], __ENV__)
{3, [a: 1, b: 2]}
```

get_docs(module, kind)(function)
Returns the docs for the given module.

When given a module name, it finds its BEAM code and reads the docs from it.

When given a path to a .beam file, it will load the docs directly from that file.

The return value depends on the kind value:

• :docs - list of all docstrings attached to functions and macros using the @doc attribute

• :moduledoc - tuple {<line>, <doc>} where line is the line on which module definition starts and doc is the string attached to the module using the @moduledoc attribute

• :all - a keyword list with both :docs and :moduledoc

load_file(file, relative_to \ \ nil)(function)
Load the given file.

Accepts relative_to as an argument to tell where the file is located. If the file was already required/loaded, loads it again.

It returns a list of tuples {ModuleName, <<byte_code>>}, one tuple for each module defined in the file.

Notice that if load_file is invoked by different processes concurrently, the target file will be loaded concurrently many times. Check require_file/2 if you don’t want a file to be loaded concurrently.

loaded_files() (function)
List all loaded files.

prepend_path(path)(function)
Prepend a path to the Erlang VM code path.

The path is expanded with Path.expand/1 before being prepended.
**require_file(file, relative_to \ nil)**(function)

Requires the given file.

Accepts `relative_to` as an argument to tell where the file is located. The return value is the same as that of `load_file/2`. If the file was already required/loaded, doesn’t do anything and returns `nil`.

Notice that if `require_file` is invoked by different processes concurrently, the first process to invoke `require_file` acquires a lock and the remaining ones will block until the file is available. I.e. if `require_file` is called N times with a given file, it will be loaded only once. The first process to call `require_file` will get the list of loaded modules, others will get `nil`.

Check `load_file/2` if you want a file to be loaded multiple times.

**string_to_quoted(string, opts \ \ [])**(function)

Convert the given string to its quoted form.

Returns `{:ok, quoted_form}` if it succeeds, `{:error, {line, error, token}}` otherwise.

**Options**

- `:file` - The filename to be used in stacktraces and the file reported in the `__ENV__` variable.
- `:line` - The line reported in the `__ENV__` variable.
- `:existing_atoms_only` - When `true`, raises an error when non-existing atoms are found by the tokenizer.

**Macro.to_string/2**

The opposite of converting a string to its quoted form is `Macro.to_string/2`, which converts a quoted form to a string/binary representation.

**string_to_quoted!(string, opts \ \ [])**(function)

Convert the given string to its quoted form.

It returns the ast if it succeeds, raises an exception otherwise. The exception is a `TokenMissingError` in case a token is missing (usually because the expression is incomplete), `SyntaxError` otherwise.

Check `string_to_quoted/2` for options information.

**unload_files(files)**(function)

Remove files from the loaded files list.

The modules defined in the file are not removed; calling this function only removes them from the list, allowing them to be required again.

**Dict**

**Overview** This module specifies the Dict API expected to be implemented by different dictionaries. It also provides functions that redirect to the underlying Dict, allowing a developer to work with different Dict implementations using one API.

To create a new dict, use the `new` functions defined by each dict type:

`HashDict.new  #=> creates an empty HashDict`

In the examples below, `dict_impl` means a specific Dict implementation, for example `HashDict` or `Map`.

**Protocols** Besides implementing the functions in this module, all dictionaries are required to implement the Access protocol:
As well as the `Enumerable` and `Collectable` protocols.

**Match**   Dictionaries are required to implement all operations using the match (===) operator.

**Default implementation**   Default implementations for some functions in the `Dict` module are provided via `use Dict`.

For example:

```elixir
defmodule MyDict do
  use Dict

  # implement required functions (see below)
  # override default implementations if optimization
  # is needed
end
```

The client module must contain the following functions:

- `delete/2`
- `fetch/2`
- `put/3`
- `reduce/3`
- `size/1`

All functions, except `reduce/3`, are required by the `Dict` behaviour. `reduce/3` must be implemented as per the `Enumerable` protocol.

Based on these functions, `Dict` generates default implementations for the following functions:

- `drop/2`
- `equal?/2`
- `fetch!/2`
- `get/2`
- `get/3`
- `has_key?/2`
- `keys/1`
- `merge/2`
- `merge/3`
- `pop/2`
- `pop/3`
- `put_new/3`
- `split/2`
• take/2
• to_list/1
• update/4
• update!/3
• values/1

All of these functions are defined as overridable, so you can provide your own implementation if needed.

Note you can also test your custom module via Dict’s doctests:

```elixir
defmodule MyDict do
  # ...
end

defmodule MyTests do
  use ExUnit.Case
doctest Dict
defp dict_impl, do: MyDict
end
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete/2</td>
<td>Removes the entry stored under the given key from dict. If dict does not contain key, returns the dictionary unchanged</td>
</tr>
<tr>
<td>drop/2</td>
<td>Returns a new dict where the given keys are removed from dict. Any non-member keys are ignored</td>
</tr>
<tr>
<td>equal?/2</td>
<td>Check if two dicts are equal using ===</td>
</tr>
<tr>
<td>fetch!/2</td>
<td>Returns the value associated with key in dict. If dict does not contain key, it raises KeyError</td>
</tr>
<tr>
<td>fetch/2</td>
<td>Returns {:ok, value} associated with key in dict. If dict does not contain key, returns :error</td>
</tr>
<tr>
<td>get/3</td>
<td>Returns the value associated with key in dict. If dict does not contain key, returns default (or nil if not provided)</td>
</tr>
<tr>
<td>has_key?</td>
<td>Returns whether the given key exists in the given dict</td>
</tr>
<tr>
<td>keys/1</td>
<td>Returns a list of all keys in dict. The keys are not guaranteed to be in any order</td>
</tr>
<tr>
<td>merge/3</td>
<td>Merges the dict b into dict a</td>
</tr>
<tr>
<td>pop/3</td>
<td>Returns the value associated with key in dict as well as the dict without key</td>
</tr>
<tr>
<td>put/3</td>
<td>Stores the given value under key in dict. If dict already has key, the stored value is replaced by the new one</td>
</tr>
<tr>
<td>put_new/3</td>
<td>Puts the given value under key in dict unless key already exists</td>
</tr>
<tr>
<td>size/1</td>
<td>Returns the number of elements in dict</td>
</tr>
<tr>
<td>split/2</td>
<td>Returns a tuple of two dicts, where the first dict contains only entries from dict with keys in keys, and the second dict contains only entries from dict with keys not in keys</td>
</tr>
<tr>
<td>take/2</td>
<td>Returns a new dict where only the keys in keys from dict are included</td>
</tr>
<tr>
<td>to_list/1</td>
<td>Returns a list of key-value pairs stored in dict. No particular order is enforced</td>
</tr>
<tr>
<td>update!/3</td>
<td>Update a value in dict by calling fun on the value to get a new value. An exception is generated if key is not present in the dict</td>
</tr>
<tr>
<td>update/4</td>
<td>Update a value in dict by calling fun on the value to get a new value. If key is not present in dict then initial will be stored as the first value</td>
</tr>
<tr>
<td>values/1</td>
<td>Returns a list of all values in dict. The values are not guaranteed to be in any order</td>
</tr>
</tbody>
</table>

**Summary**

- **key**
  - key:: any
- **value**
  - value:: any

---

124 Chapter 3. API reference
Functions

**delete(dict, key)** *(function)*

Specs:

- `delete(t, key) :: t`

Removes the entry stored under the given `key` from `dict`. If `dict` does not contain `key`, returns the dictionary unchanged.

**Examples**

```iex
d = Enum.into([a: 1, b: 2], dict_impl.new)
d = Dict.delete(d, :a)
dict.get(d, :a)  # nil

d = Enum.into([b: 2], dict_impl.new)
d = Dict.delete(d, :a)  # == d
true
```

**drop(dict, keys)** *(function)*

Specs:

- `drop(t, keys) :: t`

Returns a new dict where the given `keys` are removed from `dict`. Any non-member keys are ignored.

**Examples**

```iex
d = Enum.into([a: 1, b: 2], dict_impl.new)
d = Dict.drop(d, [:a, :c, :d])
dict.to_list(d)  # [b: 2]

d = Enum.into([a: 1, b: 2], dict_impl.new)
d = Dict.drop(d, [:c, :d])
dict.to_list(d)  # > Enum.sort [a: 1, b: 2]
```

**equal?(dict1, dict2)** *(function)*

Specs:

- `equal?(t, t) :: boolean`

Check if two dicts are equal using `===`. Notice this function is polymorphic as it compares dicts of any type. Each dict implementation also provides an equal? function, but they can only compare dicts of the same type.

**Examples**

```iex
da = Enum.into([a: 2, b: 3, f: 5, c: 123], dict_impl.new)
b = [a: 2, b: 3, f: 5, c: 123]
dict.equal?(a, b)  # true

da = Enum.into([a: 2, b: 3, f: 5, c: 123], dict_impl.new)
b = []
```
Elixir Documentation, Release

```elixir
dict = %{:a => 1}
dict.fetch(:a)  # {ok, 1}
dict.fetch(:b)  # :error
```

## fetch! (dict, key) (function)

**Specs:**

- `fetch!(t, key) :: value | no_return`

Returns the value associated with `key` in `dict`. If `dict` does not contain `key`, it raises `KeyError`.

**Examples**

```elixir
d = Enum.into([a: 1], dict_impl.new)
dict.fetch!(d, :a)  # 1
```

## get (dict, key, default \ nil) (function)

**Specs:**

- `get(t, key, value) :: value`

Returns the value associated with `key` in `dict`. If `dict` does not contain `key`, returns `default` (or `nil` if not provided).

**Examples**

```elixir
d = Enum.into([a: 1], dict_impl.new)
dict.get(d, :a)  # 1
dict.get(d, :b)  # nil
dict.get(d, :b, 3)  # 3
```

## has_key? (dict, key) (function)

**Specs:**

- `has_key?(t, key) :: boolean`

Returns whether the given `key` exists in the given `dict`.

**Examples**

```elixir
d = Enum.into([a: 1], dict_impl.new)
dict.has_key?(d, :a)  # true
dict.has_key?(d, :b)  # false
```
keys(dict)(function)
Specs:

•keys(t):: [key]

    Returns a list of all keys in dict. The keys are not guaranteed to be in any order.

Examples
iex> d = Enum.into([a: 1, b: 2], dict_impl.new)
iex> Enum.sort(Dict.keys(d))
[a, :b]

merge(dict1, dict2, fun \ fn _k, _v1, v2 -> v2 end)(function)
Specs:

•merge(t, t, (key, value, value -> value)):: t

Merges the dict b into dict a.

    If one of the dict b entries already exists in the dict, the functions in entries in b have higher precedence unless a function is given to resolve conflicts.

    Notice this function is polymorphic as it merges dicts of any type. Each dict implementation also provides a merge function, but they can only merge dicts of the same type.

Examples
iex> d1 = Enum.into([a: 1, b: 2], dict_impl.new)
iex> d2 = Enum.into([a: 3, d: 4], dict_impl.new)
iex> d = Dict.merge(d1, d2)
[a: Dict.get(d, :a), b: Dict.get(d, :b), d: Dict.get(d, :d)]
[a: 3, b: 2, d: 4]

iex> d1 = Enum.into([a: 1, b: 2], dict_impl.new)
iex> d2 = Enum.into([a: 3, d: 4], dict_impl.new)
iex> d = Dict.merge(d1, d2, fn(_k, v1, v2) -> v1 + v2
...> end)
iex> [a: Dict.get(d, :a), b: Dict.get(d, :b), d: Dict.get(d, :d)]
[a: 4, b: 2, d: 4]

pop(dict, key, default \ nil)(function)
Specs:

•pop(t, key, value):: {value, t}

    Returns the value associated with key in dict as well as the dict without key.

Examples
iex> dict = Enum.into([a: 1], dict_impl.new)
iex> {v, d} = Dict.pop dict, :a
iex> {v, Enum.sort(d)}
{1, []}

iex> dict = Enum.into([a: 1], dict_impl.new)
iex> {v, d} = Dict.pop dict, :b
iex> {v, Enum.sort(d)}
(nil, [a: 1])

iex> dict = Enum.into([a: 1], dict_impl.new)
iex> {v, d} = Dict.pop dict, :b, 3
Elixir Documentation, Release

```elixir
eix> {v, Enum.sort(d)}
{3, [a: 1]}
```

**put** (dict, key, val)

*Specs:*

```elixir
• put(t, key, value) :: t
```

Stores the given value under key in dict. If dict already has key, the stored value is replaced by the new one.

*Examples*

```elixir
eix> d = Enum.into([a: 1, b: 2], dict_impl.new)
eix> d = Dict.put(d, :a, 3)
eix> Dict.get(d, :a)
3
```

**put_new** (dict, key, val)

*Specs:*

```elixir
• put_new(t, key, value) :: t
```

Puts the given value under key in dict unless key already exists.

*Examples*

```elixir
eix> d = Enum.into([a: 1, b: 2], dict_impl.new)
eix> d = Dict.put_new(d, :a, 3)
eix> Dict.get(d, :a)
1
```

**size** (dict)

*Specs:*

```elixir
• size(t) :: non_neg_integer
```

Returns the number of elements in dict.

*Examples*

```elixir
eix> d = Enum.into([a: 1, b: 2], dict_impl.new)
eix> Dict.size(d)
2
```

**split** (dict, keys)

*Specs:*

```elixir
• split(t, [key]) :: {t, t}
```

Returns a tuple of two dicts, where the first dict contains only entries from dict with keys in keys, and the second dict contains only entries from dict with keys not in keys.

Any non-member keys are ignored.

*Examples*

```elixir
eix> d = Enum.into([a: 1, b: 2, c: 3, d: 4], dict_impl.new)
eix> {d1, d2} = Dict.split(d, [:a, :c, :e])
eix> {Dict.to_list(d1) |> Enum.sort, Dict.to_list(d2) |> Enum.sort}
([a: 1, c: 3], [b: 2, d: 4])
```

```elixir
eix> d = Enum.into([], dict_impl.new)
eix> {d1, d2} = Dict.split(d, [:a, :c])
```

128 Chapter 3. API reference
{Dict.to_list(d1), Dict.to_list(d2)}
([], [])

{{a: 1, b: 2}, [a: 1, b: 2], []}

d = Enum.into([a: 1, b: 2], dict_impl.new)
d1, d2 = Dict.split(d, [a, b, c])
{Dict.to_list(d1) |> Enum.sort, Dict.to_list(d2)}

\[a: 1, b: 2\]

\[
\text{take}(\text{dict}, \text{keys})(\text{function})
\]
Specs:

\[
\text{take}(\text{t}, [\text{key}]) :: \text{t}
\]

Returns a new dict where only the keys in keys from dict are included.
Any non-member keys are ignored.

**Examples**

iex> d = Enum.into([a: 1, b: 2], dict_impl.new)
iex> d = Dict.take(d, [a, c, d])
iex> Dict.to_list(d)
[a: 1]
iex> d = Dict.take(d, [c, d])
iex> Dict.to_list(d)
[]

to_list(dict)(function)
Specs:

\[
\text{to_list}(\text{t}) :: []
\]

Returns a list of key-value pairs stored in dict. No particular order is enforced.

update(dict, key, initial, fun)(function)
Specs:

\[
\text{update}(\text{t}, \text{key}, \text{value}, (\text{value} \rightarrow \text{value})) :: \text{t}
\]

Update a value in dict by calling fun on the value to get a new value. If key is not present in dict then initial will be stored as the first value.

**Examples**

iex> d = Enum.into([a: 1, b: 2], dict_impl.new)
iex> d = Dict.update(d, :c, 3, fn(val) -> -val end)
iex> Dict.get(d, :c)
3

update!(dict, key, fun)(function)
Specs:

\[
\text{update!}(\text{t}, \text{key}, (\text{value} \rightarrow \text{value})) :: \text{t}
\]

Update a value in dict by calling fun on the value to get a new value. An exception is generated if key is not present in the dict.

**Examples**

iex> d = Enum.into([a: 1, b: 2], dict_impl.new)
iex> d = Dict.update!(d, :a, fn(val) -> -val end)
iex> Dict.get(d, :a)
-1
values(dict)(function)
Specs:
  • values(t) :: [value]

Returns a list of all values in dict. The values are not guaranteed to be in any order.

Examples

iex> d = Enum.into([a: 1, b: 2], dict_impl.new)
iex> Enum.sort(Dict.values(d))
[1,2]

Callbacks
delete/2(callback)
Specs:
  • delete(t, key) :: t

drop/2(callback)
Specs:
  • drop(t, Enum.t) :: t

equal?/2(callback)
Specs:
  • equal?(t, t) :: boolean

fetch/2(callback)
Specs:
  • fetch(t, key) :: {ok, value} | :error

fetch!/2(callback)
Specs:
  • fetch!(t, key) :: value | no_return

get/2(callback)
Specs:
  • get(t, key) :: value

get/3(callback)
Specs:
  • get(t, key, value) :: value

has_key?/2(callback)
Specs:
  • has_key?(t, key) :: boolean

keys/1(callback)
Specs:
  • keys(t) :: [key]

merge/2(callback)
Specs:
  • merge(t, t) :: t
merge/3(callback)
   Specs:
   •merge(t, t, (key, value, value -> value)) :: t

new/0(callback)
   Specs:
   •new :: t

pop/2(callback)
   Specs:
   •pop(t, key):: {value, t}

pop/3(callback)
   Specs:
   •pop(t, key, value):: {value, t}

put/3(callback)
   Specs:
   •put(t, key, value):: t

put_new/3(callback)
   Specs:
   •put_new(t, key, value):: t

size/1(callback)
   Specs:
   •size(t):: non_neg_integer

split/2(callback)
   Specs:
   •split(t, Enum.t):: {t, t}

take/2(callback)
   Specs:
   •take(t, Enum.t):: t

to_list/1(callback)
   Specs:
   •to_list(t):: []

update/4(callback)
   Specs:
   •update(t, key, value, (value -> value)):: t

update!/3(callback)
   Specs:
   •update!(t, key, (value -> value)):: t | no_return

values/1(callback)
   Specs:
   •values(t):: [value]
Enum

Overview

Provides a set of algorithms that enumerate over collections according to the Enumerable protocol:

```
iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
[2,4,6]
```

Some particular types, like dictionaries, yield a specific format on enumeration. For dicts, the argument is always a \((key, value)\) tuple:

```
iex> dict = %{a: 1, b: 2}
iex> Enum.map(dict, fn (k, v) -> {k, v * 2} end)
[a: 2, b: 4]
```

Note that the functions in the Enum module are eager: they always start the enumeration of the given collection. The Stream module allows lazy enumeration of collections and provides infinite streams.

Since the majority of the functions in Enum enumerate the whole collection and return a list as result, infinite streams need to be carefully used with such functions, as they can potentially run forever. For example:

```
Enum.each Stream.cycle([1,2,3]), &IO.puts(&1)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all?/2</td>
<td>Invokes the given fun for each item in the collection and returns false if at least one invocation returns false</td>
</tr>
<tr>
<td>any?/2</td>
<td>Invokes the given fun for each item in the collection and returns true if at least one invocation returns true</td>
</tr>
<tr>
<td>at/3</td>
<td>Finds the element at the given index (zero-based). Returns default if index is out of bounds</td>
</tr>
<tr>
<td>chunk/2</td>
<td>Returns a collection of lists containing (n) items each, where each new chunk starts (n) elements into the collection</td>
</tr>
<tr>
<td>chunk/4</td>
<td>Splits (coll) on every element for which (fun) returns a new value</td>
</tr>
<tr>
<td>concat/1</td>
<td>Given an enumerable of enumerables, concatenate the enumerables into a single list</td>
</tr>
<tr>
<td>concat/2</td>
<td>Concatenates the enumerable on the right with the enumerable on the left</td>
</tr>
<tr>
<td>count/1</td>
<td>Returns the collection's size</td>
</tr>
<tr>
<td>count/2</td>
<td>Returns the count of items in the collection for which (fun) returns true</td>
</tr>
<tr>
<td>drop/2</td>
<td>Drops the first count items from (collection)</td>
</tr>
<tr>
<td>drop_while/2</td>
<td>Drops items at the beginning of (collection) while (fun) returns true</td>
</tr>
<tr>
<td>each/2</td>
<td>Invokes the given fun for each item in the collection. Returns (:ok)</td>
</tr>
<tr>
<td>empty?/1</td>
<td>Returns (true) if the collection is empty, otherwise (false)</td>
</tr>
<tr>
<td>fetch!/2</td>
<td>Finds the element at the given index (zero-based). Raises OutOfBoundsError if the given position is out of bounds</td>
</tr>
<tr>
<td>fetch/2</td>
<td>Finds the element at the given index (zero-based). Returns ({:ok, element}) if found, otherwise (:error)</td>
</tr>
<tr>
<td>filter/2</td>
<td>Filters the collection, i.e. returns only those elements for which (fun) returns true</td>
</tr>
<tr>
<td>filter_map/3</td>
<td>Filters the collection and maps its values in one pass</td>
</tr>
<tr>
<td>find/3</td>
<td>Returns the first item for which (fun) returns a truthy value. If no such item is found, returns ifnone</td>
</tr>
<tr>
<td>find_index/2</td>
<td>Similar to find/3, but returns the index (zero-based) of the element instead of the element itself</td>
</tr>
<tr>
<td>find_value/3</td>
<td>Similar to find/3, but returns the value of the function invocation instead of the element itself</td>
</tr>
<tr>
<td>flat_map/2</td>
<td>Returns a new collection appending the result of invoking (fun) on each corresponding item of (collection)</td>
</tr>
<tr>
<td>flat_map_reduce/3</td>
<td>Maps and reduces a collection, flattening the given results</td>
</tr>
<tr>
<td>group_by/3</td>
<td>Splits (collection) into groups based on (fun)</td>
</tr>
<tr>
<td>intersperse/2</td>
<td>Intersperses (element) between each element of the enumeration</td>
</tr>
<tr>
<td>into/2</td>
<td>Inserts the given enumerable into a collectable</td>
</tr>
<tr>
<td>into/3</td>
<td>Inserts the given enumerable into a collectable according to the transformation function</td>
</tr>
<tr>
<td>join/2</td>
<td>Joins the given (collection) according to (joiner). (joiner) can be either a binary or a list and the result must be a (list)</td>
</tr>
<tr>
<td>map/2</td>
<td>Returns a new collection, where each item is the result of invoking (fun) on each corresponding item of (collection)</td>
</tr>
<tr>
<td>map_join/3</td>
<td>Maps and joins the given (collection) in one pass. (joiner) can be either a binary or a list and the result must be a (list)</td>
</tr>
<tr>
<td>map_reduce/3</td>
<td>Maps the given (collection) in one pass. (reduce) can be either a binary or a list and the result must be an (element)</td>
</tr>
<tr>
<td>max/1</td>
<td>Invokes the given fun for each item in the collection while also keeping an accumulator. Returns a (list)</td>
</tr>
</tbody>
</table>

Returns the maximum value. Raises EmptyError if the collection is empty
### Table 3.2 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_by/2</td>
<td>Returns the maximum value as calculated by the given function. Raises EmptyError if the collection is empty</td>
</tr>
<tr>
<td>member?/2</td>
<td>Checks if value exists within the collection</td>
</tr>
<tr>
<td>min/1</td>
<td>Returns the minimum value. Raises EmptyError if the collection is empty</td>
</tr>
<tr>
<td>min_by/2</td>
<td>Returns the minimum value as calculated by the given function. Raises EmptyError if the collection is empty</td>
</tr>
<tr>
<td>partition/2</td>
<td>Partitions collection into two collections, where the first one contains elements for which fun returns true</td>
</tr>
<tr>
<td>reduce/2</td>
<td>Invokes fun for each element in the collection passing that element and the accumulator acc as arguments</td>
</tr>
<tr>
<td>reduce/3</td>
<td>Invokes fun for each element in the collection passing that element and the accumulator acc as arguments</td>
</tr>
<tr>
<td>reverse/1</td>
<td>Reverses the collection</td>
</tr>
<tr>
<td>reverse/2</td>
<td>Reverses the collection and appends the tail. This is an optimization for Enum.concat(Enum.reverse(collection), tail)</td>
</tr>
<tr>
<td>scan/2</td>
<td>Applies the given function to each element in the collection, storing the result in a list and passing it as the next accumulator argument</td>
</tr>
<tr>
<td>scan/3</td>
<td>Applies the given function to each element in the collection, storing the result in a list and passing it as the next accumulator argument</td>
</tr>
<tr>
<td>shuffle/1</td>
<td>Returns a list of collection elements shuffled</td>
</tr>
<tr>
<td>slice/2</td>
<td>Returns a subset list of the given collection. Drops elements until element position range.first, then takes elements until element position range.last (inclusive)</td>
</tr>
<tr>
<td>slice/3</td>
<td>Returns a subset list of the given collection. Drops elements until element position start, then takes count elements</td>
</tr>
<tr>
<td>sort/1</td>
<td>Sorts the collection according to Elixir’s term ordering</td>
</tr>
<tr>
<td>sort/2</td>
<td>Sorts the collection by the given function</td>
</tr>
<tr>
<td>split/2</td>
<td>Splits the enumerable into two collections, leaving count elements in the first one. If count is a negative number, counts from the end of the collection</td>
</tr>
<tr>
<td>split_while/2</td>
<td>Splits collection in two while fun returns true</td>
</tr>
<tr>
<td>sum/1</td>
<td>Returns the sum of all values</td>
</tr>
<tr>
<td>take/2</td>
<td>Takes the first count items from the collection</td>
</tr>
<tr>
<td>take_every/2</td>
<td>Returns a collection of every nth item in the collection, starting with the first element</td>
</tr>
<tr>
<td>take_while/2</td>
<td>Takes the items at the beginning of collection while fun returns true</td>
</tr>
<tr>
<td>to_list/1</td>
<td>Convert collection to a list</td>
</tr>
<tr>
<td>traverse/2</td>
<td>Traverses the given enumerable keeping its shape</td>
</tr>
<tr>
<td>uniq/2</td>
<td>Enumerates the collection, removing all duplicated items</td>
</tr>
<tr>
<td>with_index/1</td>
<td>Returns the collection with each element wrapped in a tuple alongside its index</td>
</tr>
<tr>
<td>zip/2</td>
<td>Zips corresponding elements from two collections into one list of tuples</td>
</tr>
</tbody>
</table>

### Summary

#### Types

- **t**: Enumerable.t
- **element**
  - element :: any
- **index**
  - index :: non_neg_integer
- **default**
  - default :: any

#### Functions

- **all?(collection, fun \ fn x -> x end)(function)**

  **Specs**:

  ```
  • all?(t.(element -> as Boolean(term))) :: Boolean
  ```

  Invokes the given fun for each item in the collection and returns false if at least one invocation returns false. Otherwise returns true.

  **Examples**
```elixir
eix> Enum.all?([2, 4, 6], fn(x) -> rem(x, 2) == 0 end)  
true

eix> Enum.all?([2, 3, 4], fn(x) -> rem(x, 2) == 0 end)  
false

If no function is given, it defaults to checking if all items in the collection evaluate to true.

eix> Enum.all?([1, 2, 3])  
true

eix> Enum.all?([1, nil, 3])  
false
```

### any?(collection, fun \ fn x -> x end)

Specs:

- `any?(t, (element -> as_boolean(term))) :: boolean`

Invokes the given `fun` for each item in the `collection` and returns `true` if at least one invocation returns `true`. Returns `false` otherwise.

**Examples**

```elixir
eix> Enum.any?([2, 4, 6], fn(x) -> rem(x, 2) == 1 end)  
false

eix> Enum.any?([2, 3, 4], fn(x) -> rem(x, 2) == 1 end)  
true
```

If no function is given, it defaults to checking if at least one item in the collection evaluates to true.

```elixir
eix> Enum.any?([false, false, false])  
false

eix> Enum.any?([false, true, false])  
true
```

### at(collection, n, default \ nil)

Specs:

- `at(t, integer, default) :: element | default`

Finds the element at the given index (zero-based). Returns `default` if index is out of bounds.

**Examples**

```elixir
eix> Enum.at([2, 4, 6], 0)  
2

eix> Enum.at([2, 4, 6], 2)  
6

eix> Enum.at([2, 4, 6], 4)  
nil

eix> Enum.at([2, 4, 6], 4, :none)  
:none
```

### chunk(coll, n)

Specs:

- `chunk(t, non_neg_integer) :: [[]]`
Shortcut to `chunk(coll, n, n)`.

`chunk(coll, n, step, pad \ nil)(function)`
Specs:
- \*chunk(t, non_neg_integer, non_neg_integer, t | nil) :: [[]]

Returns a collection of lists containing \( n \) items each, where each new chunk starts \( \text{step} \) elements into the collection.

\( \text{step} \) is optional and, if not passed, defaults to \( n \), i.e. chunks do not overlap. If the final chunk does not have \( n \) elements to fill the chunk, elements are taken as necessary from \( \text{pad} \) if it was passed. If \( \text{pad} \) is passed and does not have enough elements to fill the chunk, then the chunk is returned anyway with less than \( n \) elements. If \( \text{pad} \) is not passed at all or is \( \text{nil} \), then the partial chunk is discarded from the result.

**Examples**

```
iex> Enum.chunk([1, 2, 3, 4, 5, 6], 2)
[[1, 2], [3, 4], [5, 6]]
iex> Enum.chunk([1, 2, 3, 4, 5, 6], 3, 2)
[[1, 2, 3], [3, 4, 5]]
iex> Enum.chunk([1, 2, 3, 4, 5, 6], 3, 2, [7])
[[1, 2, 3], [3, 4, 5], [5, 6, 7]]
iex> Enum.chunk([1, 2, 3, 4, 5, 6], 3, 3, [])
[[1, 2, 3], [4, 5, 6]]
```

`chunk_by(coll, fun)(function)`
Specs:
- \*chunk_by(t, (element -> any)) :: [][]

Splits \( \text{coll} \) on every element for which \( \text{fun} \) returns a new value.

**Examples**

```
iex> Enum.chunk_by([1, 2, 2, 3, 4, 4, 6, 7, 7], &(rem(&1, 2) == 1))
[[1], [2, 2], [3], [4, 4, 6], [7, 7]]
```

`concat(enumerables)(function)`
Specs:
- \*concat(t) :: t

Given an enumerable of enumerables, concatenate the enumerables into a single list.

**Examples**

```
iex> Enum.concat([1..3, 4..6, 7..9])
[1,2,3,4,5,6,7,8,9]
iex> Enum.concat([[1, [2], 3], [4], [5, 6]])
[[1, [2], 3], 4, [5, 6]]
```

`concat(left, right)(function)`
Specs:
- \*concat(t, t) :: t

Concatenates the enumerable on the right with the enumerable on the left.

This function produces the same result as the `Kernel.++/2` operator for lists.
Examples

\[
\text{iex> Enum.concat(1..3, 4..6)} \\
[1,2,3,4,5,6]
\]

\[
\text{iex> Enum.concat([1, 2, 3], [4, 5, 6])} \\
[1,2,3,4,5,6]
\]

count \(\text{(collection)}\) \(\text{(function)}\)

Specs:

- \text{count(t)} :: non_neg_integer
  
  Returns the collection’s size.

Examples

\[
\text{iex> Enum.count([1, 2, 3])} \\
3
\]

count \(\text{(collection, fun)}\) \(\text{(function)}\)

Specs:

- \text{count(t, (element \(\rightarrow\) as_boolean(term)))} :: non_neg_integer
  
  Returns the count of items in the collection for which \text{fun} returns \text{true}.

Examples

\[
\text{iex> Enum.count([1, 2, 3, 4, 5], \text{fn}(x) \rightarrow \text{rem}(x, 2) == 0 \text{ end})} \\
2
\]

drop \(\text{(collection, count)}\) \(\text{(function)}\)

Specs:

- \text{drop(t, integer)} :: []
  
  Drops the first \text{count} items from \text{collection}.

Examples

\[
\text{iex> Enum.drop([1, 2, 3], 2)} \\
[3]
\]

\[
\text{iex> Enum.drop([1, 2, 3], 10)} \\
[]
\]

\[
\text{iex> Enum.drop([1, 2, 3], 0)} \\
[1,2,3]
\]

\[
\text{iex> Enum.drop([1, 2, 3], -1)} \\
[1,2]
\]

drop \(\text{while} \text{(collection, fun)}\) \(\text{(function)}\)

Specs:

- \text{drop\_while(t, (element \(\rightarrow\) as\_boolean(term)))} :: []
  
  Drops items at the beginning of \text{collection} while \text{fun} returns \text{true}.

Examples
```iex
Enum.drop_while([1, 2, 3, 4, 5], fn(x) -> x < 3 end)
[3, 4, 5]
```

each (collection, fun) (function)
Specs:
•each(t, (element -> any)) :: :ok

Invokes the given fun for each item in the collection. Returns :ok.

Examples
```iex
Enum.each(["some", "example"], fn(x) -> IO.puts x end)
"some"
"example"
#=> :ok
```

eempty? (collection) (function)
Specs:
•empty?(t) :: boolean

Returns true if the collection is empty, otherwise false.

Examples
```iex
Enum.empty?([[])
true
```
```iex
Enum.empty?([1, 2, 3])
false
```

fetch (collection, n) (function)
Specs:
•fetch(t, integer) :: {:ok, element} | :error

Finds the element at the given index (zero-based). Returns {:ok, element} if found, otherwise :error.

A negative index can be passed, which means the collection is enumerated once and the index is counted from the end (i.e. -1 fetches the last element).

Examples
```iex
Enum.fetch([2, 4, 6], 0)
{:ok, 2}
```
```iex
Enum.fetch([2, 4, 6], 2)
{:ok, 6}
```
```iex
Enum.fetch([2, 4, 6], 4)
:error
```

fetch! (collection, n) (function)
Specs:
•fetch!(t, integer) :: element | no_return

Finds the element at the given index (zero-based). Raises OutOfBoundsError if the given position is outside the range of the collection.

Examples
iex> Enum.fetch!([2, 4, 6], 0)
2
iex> Enum.fetch!([2, 4, 6], 2)
6
iex> Enum.fetch!([2, 4, 6], 4)
** (Enum.OutOfBoundsError) out of bounds error

**filter**<br/>
(collection, fun)<br/>
(function)

Specs:

- `filter(t, (element -> as_boolean(term))) :: []`

Filters the collection, i.e. returns only those elements for which `fun` returns `true`.

Examples

iex> Enum.filter([1, 2, 3], fn(x) -> rem(x, 2) == 0 end)
[2]

**filter_map**<br/>
(collection, filter, mapper)<br/>
(function)

Specs:

- `filter_map(t, (element -> as_boolean(term)), (element -> element)) :: []`

Filters the collection and maps its values in one pass.

Examples

iex> Enum.filter_map([1, 2, 3], fn(x) -> rem(x, 2) == 0 end, &(&1 * 2))
[4]

**find**<br/>
(collection, ifnone \ nil, fun)<br/>
(function)

Specs:

- `find(t, default, (element -> any)) :: element | default`

Returns the first item for which `fun` returns a truthy value. If no such item is found, returns `ifnone`.

Examples

iex> Enum.find([2, 4, 6], fn(x) -> rem(x, 2) == 1 end)
nil
iex> Enum.find([2, 4, 6], 0, fn(x) -> rem(x, 2) == 1 end)
0
iex> Enum.find([2, 3, 4], fn(x) -> rem(x, 2) == 1 end)
3

**find_index**<br/>
(collection, fun)<br/>
(function)

Specs:

- `find_index(t, (element -> any)) :: index | nil`

Similar to `find/3`, but returns the index (zero-based) of the element instead of the element itself.

Examples

iex> Enum.find_index([2, 4, 6], fn(x) -> rem(x, 2) == 1 end)
nil
iex> Enum.find_index([2, 3, 4], fn(x) -> rem(x, 2) == 1 end)
1

find_value(collection, ifnone \ \ nil, fun)(function)
Specs:
  •find_value(t, any, (element -> any)) :: any | nil

Similar to find/3, but returns the value of the function invocation instead of the element itself.

Examples
iex> Enum.find_value([2, 4, 6], fn(x) -> rem(x, 2) == 1 end)
nil
iex> Enum.find_value([2, 3, 4], fn(x) -> rem(x, 2) == 1 end)
true

flat_map(collection, fun)(function)
Specs:
  •flat_map(t, (element -> t)) :: []

Returns a new collection appending the result of invoking fun on each corresponding item of collection.

The given function should return an enumerable.

Examples
iex> Enum.flat_map([[:a, :b, :c], fn(x) -> [x, x] end])
[[:a, :a, :b, :b, :c, :c]]
iex> Enum.flat_map([[1,3], {4,6}], fn({x,y}) -> x..y end)
[1, 2, 3, 4, 5, 6]

flat_map_reduce(collection, acc, fun)(function)
Specs:
  *(flat_map_reduce(t, acc, fun) :: {[any], any}) when fun: (element, acc -> {t, acc} | {:halt, acc}), acc: any

Maps and reduces a collection, flattening the given results.

It expects an accumulator and a function that receives each stream item and an accumulator, and must return a tuple containing a new stream (often a list) with the new accumulator or a tuple with :halt as first element and the accumulator as second.

Examples
iex> enum = 1..100
iex> n = 3
iex> Enum.flat_map_reduce(enum, 0, fn i, acc ->
...>
...> if acc < n, do: ([i], acc + 1), else: (:halt, acc)
...>
...> end

([1,2,3], 3)

group_by(collection, dict \ \ %{}, fun)(function)
Specs:
  *(group_by(t, dict, (element -> any)) :: dict) when dict: Dict.t

Splits collection into groups based on fun.
The result is a dict (by default a map) where each key is a group and each value is a list of elements from collection for which fun returned that group. Ordering is not necessarily preserved.

**Examples**

```iex
Enum.group_by(~w(ant buffalo cat dingo), &String.length/1)
%{3 => ["cat", "ant"], 7 => ["buffalo"], 5 => ["dingo"]}
```

**intersperse(collection, element)(function)**

Specs:

- ```intersperse(t, element):: []```  

Intersperses element between each element of the enumeration.  

Complexity: O(n)

**Examples**

```iex
Enum.intersperse([1, 2, 3], 0)
[1, 0, 2, 0, 3]
```

```iex
Enum.intersperse([1], 0)
[1]
```

```iex
Enum.intersperse([], 0)
[]
```

**into(collection, list)(function)**

Specs:

- ```into(Enumerable.t, Collectable.t):: Collectable.t```  

Inserts the given enumerable into a collectable.

**Examples**

```iex
Enum.into([1, 2], [0])
[0, 1, 2]
```

```iex
Enum.into([a: 1, b: 2], %{})
%{a: 1, b: 2}
```

**into(collection, list, transform)(function)**

Specs:

- ```into(Enumerable.t, Collectable.t, (term -> term)):: Collectable.t```  

Inserts the given enumerable into a collectable according to the transformation function.

**Examples**

```iex
Enum.into([2, 3], [3], fn x -> x * 3 end)
[3, 6, 9]
```

**join(collection, joiner \ " ") (function)**

Specs:

- ```join(t, String.t):: String.t```  

Joins the given collection according to joiner. joiner can be either a binary or a list and the result will be of the same type as joiner. If joiner is not passed at all, it defaults to an empty binary.

All items in the collection must be convertible to a binary, otherwise an error is raised.
Examples

```iex
Enum.join([1, 2, 3])
"123"

Enum.join([1, 2, 3], " = ")
"1 = 2 = 3"
```

map(collection, fun)(function)

Specs:

- `map(t, (element -> any)) :: []`

Returns a new collection, where each item is the result of invoking `fun` on each corresponding item of `collection`.

For dicts, the function expects a key-value tuple.

Examples

```iex
Enum.map([1, 2, 3], fn(x) -> x * 2 end)
[2, 4, 6]

Enum.map([a: 1, b: 2], fn({k, v}) -> {k, -v} end)
[a: -1, b: -2]
```

map_join(collection, joiner \ " ", mapper)(function)

Specs:

- `map_join(t, String.t, (element -> any)) :: String.t`

Maps and joins the given `collection` in one pass. `joiner` can be either a binary or a list and the result will be of the same type as `joiner`. If `joiner` is not passed at all, it defaults to an empty binary.

All items in the collection must be convertible to a binary, otherwise an error is raised.

Examples

```iex
Enum.map_join([1, 2, 3], &(&1 * 2))
"246"

Enum.map_join([1, 2, 3], " = ", &(&1 * 2))
"2 = 4 = 6"
```

map_reduce(collection, acc, fun)(function)

Specs:

- `map_reduce(t, any, (element, any -> any)) :: any`

Invokes the given `fun` for each item in the `collection` while also keeping an accumulator. Returns a tuple where the first element is the mapped collection and the second one is the final accumulator.

For dicts, the first tuple element must be a `{key, value}` tuple.

Examples

```iex
Enum.map_reduce([1, 2, 3], 0, fn(x, acc) -> {x * 2, x + acc} end)
([2, 4, 6], 6)
```

max(collection)(function)

Specs:

- `max(t) :: element | no_return`
Returns the maximum value. Raises **EmptyError** if the collection is empty.

**Examples**

eix> Enum.max([1, 2, 3])
3

**max_by**(collection, fun)(function)

Specs:

*max_by(t. (element -> any)): element | no_return

Returns the maximum value as calculated by the given function. Raises **EmptyError** if the collection is empty.

**Examples**

eix> Enum.max_by(["a", "aa", "aaa"], fn(x) -> String.length(x) end) "aaa"

**member**(collection, value)(function)

Specs:

*member?(t. element): boolean

Checks if value exists within the collection.

Membership is tested with the match (===) operator, although enumerables like ranges may include floats inside the given range.

**Examples**

iex> Enum.member?(1..10, 5)
true

iex> Enum.member?([:a, :b, :c], :d)
false

**min**(collection)(function)

Specs:

*min(t): element | no_return

Returns the minimum value. Raises **EmptyError** if the collection is empty.

**Examples**

iex> Enum.min([1, 2, 3])
1

**min_by**(collection, fun)(function)

Specs:

*min_by(t, (element -> any)): element | no_return

Returns the minimum value as calculated by the given function. Raises **EmptyError** if the collection is empty.

**Examples**

iex> Enum.min_by(["a", "aa", "aaa"], fn(x) -> String.length(x) end) "a"

**partition**(collection, fun)(function)

Specs:

*partition(t, (element -> any)): {[], []}
Partitions `collection` into two collections, where the first one contains elements for which `fun` returns a truthy value, and the second one – for which `fun` returns `false` or `nil`.

**Examples**

```iex
eix> Enum.partition([1, 2, 3], fn(x) -> rem(x, 2) == 0 end)
([2], [1, 3])
```

**reduce**(collection, fun)(function)

Specs:

• `reduce(t, (element, any -> any)) :: any`

Invokes `fun` for each element in the collection passing that element and the accumulator `acc` as arguments. `fun`'s return value is stored in `acc`. The first element of the collection is used as the initial value of `acc`. Returns the accumulator.

**Examples**

```iex
eix> Enum.reduce([1, 2, 3, 4], fn(x, acc) -> x * acc end)
24
```

**reduce**(collection, acc, fun)(function)

Specs:

• `reduce(t, any, (element, any -> any)) :: any`

Invokes `fun` for each element in the collection passing that element and the accumulator `acc` as arguments. `fun`'s return value is stored in `acc`. Returns the accumulator.

**Examples**

```iex
eix> Enum.reduce([1, 2, 3], 0, fn(x, acc) -> x + acc end)
6
```

**reject**(collection, fun)(function)

Specs:

• `reject(t, (element -> as_boolean(term))) :: []`

Returns elements of collection for which `fun` returns `false`.

**Examples**

```iex
eix> Enum.reject([1, 2, 3], fn(x) -> rem(x, 2) == 0 end)
[1, 3]
```

**reverse**(collection)(function)

Specs:

• `reverse(t) :: []`

Reverses the collection.

**Examples**

```iex
eix> Enum.reverse([1, 2, 3])
[3, 2, 1]
```

**reverse**(collection, tail)(function)

Specs:

• `reverse(t, t) :: []`

```
Reverses the collection and appends the tail. This is an optimization for
\texttt{Enum.concat(}\texttt{Enum.reverse(collection), tail)}.

**Examples**

\begin{verbatim}
iex> Enum.reverse([1, 2, 3], [4, 5, 6])
[3, 2, 1, 4, 5, 6]
\end{verbatim}

\textbf{scan (enum, fun) (function)}

Specs:

\begin{verbatim}
  • scan(t, \langle element, any \rightarrow any \rangle) :: []
\end{verbatim}

Applies the given function to each element in the collection, storing the result in a list and passing it as the
accumulator for the next computation.

**Examples**

\begin{verbatim}
iex> Enum.scan(1..5, \&(\&1 + &2))
[1, 3, 6, 10, 15]
\end{verbatim}

\textbf{scan (enum, acc, fun) (function)}

Specs:

\begin{verbatim}
  • scan(t, \langle any, element, any \rightarrow any \rangle) :: []
\end{verbatim}

Applies the given function to each element in the collection, storing the result in a list and passing it as the
accumulator for the next computation. Uses the given \texttt{acc} as the starting value.

**Examples**

\begin{verbatim}
iex> Enum.scan(1..5, 0, \&(\&1 + &2))
[1, 3, 6, 10, 15]
\end{verbatim}

\textbf{shuffle (collection) (function)}

Specs:

\begin{verbatim}
  • shuffle(t) :: []
\end{verbatim}

Returns a list of collection elements shuffled.

Notice that you need to explicitly call \texttt{/random.seed/1'} \((http://www.erlang.org/doc/man/random.html#seed-1)\)
and set a seed value for the random algorithm. Otherwise, the default seed will be set which will always return
the same result. For example, one could do the following to set a seed dynamically:

\begin{verbatim}
:random.seed(:erlang.now)
\end{verbatim}

**Examples**

\begin{verbatim}
iex> Enum.shuffle([1, 2, 3])
[3, 2, 1]
iex> Enum.shuffle([1, 2, 3])
[3, 1, 2]
\end{verbatim}

\textbf{slice (coll, arg2) (function)}

Specs:

\begin{verbatim}
  • slice(t, \langle Range.t \rangle) :: []
\end{verbatim}

Returns a subset list of the given collection. Drops elements until element position \texttt{range.first}, then takes
elements until element position \texttt{range.last} (inclusive).

Positions are calculated by adding the number of items in the collection to negative positions (so position -3 in
a collection with count 5 becomes position 2).
The first position (after adding count to negative positions) must be smaller or equal to the last position.

**Examples**

```iex
iex> Enum.slice(1..100, 5..10)
[6, 7, 8, 9, 10, 11]
```

**slice(coll, start, count)(function)**

Specs:

- `slice(t, integer, non_neg_integer) :: []`

Returns a subset list of the given collection. Drops elements until element position `start`, then takes `count` elements.

**Examples**

```iex
iex> Enum.slice(1..100, 5, 10)
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

**sort(collection)(function)**

Specs:

- `sort(t) :: []`

Sorts the collection according to Elixir’s term ordering.

Uses the merge sort algorithm.

**Examples**

```iex
iex> Enum.sort([3, 2, 1])
[1, 2, 3]
```

**sort(collection, fun)(function)**

Specs:

- `sort(t, (element, element -> boolean)) :: []`

Sorts the collection by the given function.

This function uses the merge sort algorithm. The given function must return false if the first argument is less than right one.

**Examples**

```iex
iex> Enum.sort([1, 2, 3], &(&1 > &2))
[3, 2, 1]
```

The sorting algorithm will be stable as long as the given function returns true for values considered equal:

```iex
iex> Enum.sort ["some", "kind", "of", "monster"], &byte_size(&1) <= byte_size(&2)
["of", "some", "kind", "monster"]
```

If the function does not return true, the sorting is not stable and the order of equal terms may be shuffled:

```iex
iex> Enum.sort ["some", "kind", "of", "monster"], &byte_size(&1) < byte_size(&2)
["of", "kind", "some", "monster"]
```

**split(collection, count)(function)**

Specs:

- `split(t, integer) :: [[], []]`
Splits the enumerable into two collections, leaving \texttt{count} elements in the first one. If \texttt{count} is a negative number, it starts counting from the back to the beginning of the collection.

Be aware that a negative \texttt{count} implies the collection will be enumerated twice: once to calculate the position, and a second time to do the actual splitting.

\textbf{Examples}
\begin{verbatim}
iex> Enum.split([1, 2, 3], 2) 
([1,2], [3])

iex> Enum.split([1, 2, 3], 10) 
([1,2,3], [])

iex> Enum.split([1, 2, 3], 0) 
([], [1,2,3])

iex> Enum.split([1, 2, 3], -1) 
([1,2], [3])

iex> Enum.split([1, 2, 3], -5) 
([], [1,2,3])
\end{verbatim}

\begin{verbatim}
split_while-collection, fun(function)
\end{verbatim}

\textbf{Specs:}
\begin{verbatim}
•split_while(t, (element -> as_boolean(term))) :: [[], []]
\end{verbatim}

Splits \texttt{collection} in two while \texttt{fun} returns true.

\textbf{Examples}
\begin{verbatim}
iex> Enum.split_while([1, 2, 3, 4], fn(x) -> x < 3 end) 
([1, 2], [3, 4])
\end{verbatim}

\begin{verbatim}
sum-collection(function)
\end{verbatim}

\textbf{Specs:}
\begin{verbatim}
•sum(t) :: number
\end{verbatim}

Returns the sum of all values.

\textbf{Raises} \texttt{ArithmeticError} if \texttt{collection} contains a non-numeric value.

\textbf{Examples}
\begin{verbatim}
iex> Enum.sum([1, 2, 3])
6
\end{verbatim}

\begin{verbatim}
take-collection, count(function)
\end{verbatim}

\textbf{Specs:}
\begin{verbatim}
•take(t, integer) :: []
\end{verbatim}

Takes the first \texttt{count} items from the collection.

If a negative \texttt{count} is given, the last \texttt{count} values will be taken. For such, the collection is fully enumerated keeping up to \(2 \times \texttt{count}\) elements in memory. Once the end of the collection is reached, the last \texttt{count} elements are returned.

\textbf{Examples}
iex> Enum.take([1, 2, 3], 2)
[1, 2]

iex> Enum.take([1, 2, 3], 10)
[1, 2, 3]

iex> Enum.take([1, 2, 3], 0)
[]

iex> Enum.take([1, 2, 3], -1)
[3]

**take_every**(collection, nth)(function)

Specs:

- `take_every(t, integer) :: []`

Returns a collection of every `nth` item in the collection, starting with the first element.

**Examples**

iex> Enum.take_every(1..10, 2)
[1, 3, 5, 7, 9]

**take_while**(collection, fun)(function)

Specs:

- `take_while(t, (element -> as_boolean(term))) :: []`

Takes the items at the beginning of `collection` while `fun` returns `true`.

**Examples**

iex> Enum.take_while([1, 2, 3], fn(x) -> x < 3 end)
[1, 2]

**to_list**(collection)(function)

Specs:

- `to_list(t) :: [term]`

Convert `collection` to a list.

**Examples**

iex> Enum.to_list(1 .. 3)
[1, 2, 3]

**traverse**(collection, transform)(function)

Specs:

- `traverse(Enumerable.t, (term -> term)) :: Collectable.t`

Traverses the given enumerable keeping its shape.
It also expects the enumerable to implement the `Collectable` protocol.

**Examples**

iex> Enum.traverse(%{a: 1, b: 2}, fn {k, v} -> {k, v * 2} end)
%{a: 2, b: 4}

**uniq**(collection, fun \ fn x -> x end)(function)

Specs:
•uniq\(\text{(element } \rightarrow \text{term})\) :: []

Enumerates the collection, removing all duplicated items.

Examples

```iex
Enum.uniq([1, 2, 3, 2, 1])
[1, 2, 3]
```

```iex
Enum.uniq([\{1, :x\}, \{2, :y\}, \{1, :z\}], fn \(x, _\) \rightarrow x\ end)
[\{1, :x\}, \{2, :y\}]
```

with_index\(\text{(collection)}\)(function)

Specs:

•with_index\(\text{(t)}\) :: [{\text{element}, non_neg_integer}]

Returns the collection with each element wrapped in a tuple alongside its index.

Examples

```iex
Enum.with_index [1,2,3]
[[1,0],[2,1],[3,2]]
```

zip\(\text{coll1, coll2)}\)(function)

Specs:

•zip\(\text{(t, t)}\) :: [{\text{any, any}}]

Zips corresponding elements from two collections into one list of tuples.

The zipping finishes as soon as any enumerable completes.

Examples

```iex
Enum.zip([1, 2, 3], [:a, :b, :c])
[\{1, :a\}, \{2, :b\}, \{3, :c\}]
```

```iex
Enum.zip([1,2,3,4,5], [:a, :b, :c])
[\{1, :a\}, \{2, :b\}, \{3, :c\}]
```

Exception

Overview  Functions to format throw/catch/exit and exceptions.

Note that stacktraces in Elixir are updated on throw, errors and exits. For example, at any given moment, `System.stacktrace` will return the stacktrace for the last throw/error/exit that occurred in the current process.

Do not rely on the particular format returned by the `format` functions in this module. They may be changed in future releases in order to better suit Elixir’s tool chain. In other words, by using the functions in this module it is guarantee you will format exceptions as in the current Elixir version being used.
### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception?/1</td>
<td>Returns true if the given argument is an exception</td>
</tr>
<tr>
<td>format/3</td>
<td>Normalizes and formats throw/errors/exits and stacktrace</td>
</tr>
<tr>
<td>format_banner/3</td>
<td>Normalizes and formats any throw, error and exit</td>
</tr>
<tr>
<td>format_exit/1</td>
<td>Formats an exit, returns a string</td>
</tr>
<tr>
<td>format_fa/2</td>
<td>Receives an anonymous function and arity and formats it as shown in stacktraces. The arity may also be a list of arguments</td>
</tr>
<tr>
<td>format_file_line/2</td>
<td>Formats the given file and line as shown in stacktraces. If any of the values are nil, they are omitted</td>
</tr>
<tr>
<td>format_mfa/3</td>
<td>Receives a module, fun and arity and formats it as shown in stacktraces. The arity may also be a list of arguments</td>
</tr>
<tr>
<td>format_stacktrace/1</td>
<td>Receives a stacktrace entry and formats it into a string</td>
</tr>
<tr>
<td>format_stacktrace_entry/1</td>
<td>Gets the message for an exception</td>
</tr>
<tr>
<td>normalize/3</td>
<td>Normalizes an exception, converting Erlang exceptions to Elixir exceptions</td>
</tr>
</tbody>
</table>

### Types

- **t**
  
  \[t::\%module\{|\_exception\_: true\}\]
  
  The exception type (as generated by defexception)

- **kind**
  
  \[kind:::error \|:exit \|:throw \|{:EXIT, pid}\]
  
  The kind handled by formatting functions

- **stacktrace**
  
  \[stacktrace::[stacktrace_entry]\]

- **stacktrace_entry**
  
  \[stacktrace_entry::\{module, function, arity_or_args, location\} | \{function, arity_or_args, location\}\]

### Functions

#### exception?/1 (function)

Returns true if the given argument is an exception.

#### format(kind, payload, stacktrace \ \ nil)(function)

**Specs:**

\[
\text{•} \text{format(kind, any, stacktrace\|nil)}::\text{String.t}
\]

Normalizes and formats throw/errors/exits and stacktrace.

It relies on **format_banner/3** and **format_stacktrace/1** to generate the final format.

Note that \{:EXIT, pid\} do not generate a stacktrace though (as they are retrieved as messages without stacktraces).

#### format_banner(kind, exception, stacktrace \ \ nil)(function)

**Specs:**

\[
\text{•} \text{format_banner(kind, any, stacktrace\|nil)}::\text{String.t}
\]

Normalizes and formats any throw, error and exit.

The message is formatted and displayed in the same format as used by Elixir’s CLI.

The third argument, a stacktrace, is optional. If it is not supplied `System.stacktrace/0` will sometimes be used to get additional information for the kind :error. If the stacktrace is unknown and `System.stacktrace/0` would not return the stacktrace corresponding to the exception an empty stacktrace, [], must be used.
format_exit(reason)(function)
  Specs:
  • format_exit(any) :: String.t
  Formats an exit, returns a string.
  Often there are errors/exceptions inside exits. Exits are often wrapped by the caller and provide stacktraces too. This function formats exits in a way to nicely show the exit reason, caller and stacktrace.

format_fa(fun, arity)(function)
  Receives an anonymous function and arity and formats it as shown in stacktraces. The arity may also be a list of arguments.

Examples

```elixir
define exception
  Exception.format_fa(fn -> end, 1)
  #=> "#Function<...>/1"
```

format_file_line(file, line)(function)
  Formats the given file and line as shown in stacktraces. If any of the values are nil, they are omitted.

Examples

```elixir
iex> Exception.format_file_line("foo", 1)
"foo:1:

iex> Exception.format_file_line("foo", nil)
"foo:

iex> Exception.format_file_line(nil, nil)
"
```

format_mfa(module, fun, arity)(function)
  Receives a module, fun and arity and formats it as shown in stacktraces. The arity may also be a list of arguments.

Examples

```elixir
iex> Exception.format_mfa Foo, :bar, 1
"Foo.bar/1"

iex> Exception.format_mfa Foo, :bar, []
"Foo.bar()"

iex> Exception.format_mfa nil, :bar, []
"nil.bar()"

Anonymous functions are reported as -func/arity-anonfn-count-, where func is the name of the enclosing function. Convert to "anonymous fn in func/arity"

format_stacktrace(trace \ nil)(function)
  Formats the stacktrace.
  A stacktrace must be given as an argument. If not, the stacktrace is retrieved from Process.info/2.

format_stacktrace_entry(entry)(function)
  Specs:
  • format_stacktrace_entry(stacktrace_entry) :: String.t
  Receives a stacktrace entry and formats it into a string.

message(exception)(function)
  Gets the message for an exception.
normalize(kind, payload, stacktrace \ nil)(function)

Specs:

• (normalize(kind, payload, stacktrace) :: payload) when payload: var
• normalize(error, any, stacktrace) :: t

Normalizes an exception, converting Erlang exceptions to Elixir exceptions.

It takes the kind spilled by catch as an argument and normalizes only :error, returning the untouched payload for others.

The third argument, a stacktrace, is optional. If it is not supplied System.stacktrace/0 will sometimes be used to get additional information for the kind :error. If the stacktrace is unknown and System.stacktrace/0 would not return the stacktrace corresponding to the exception an empty stacktrace, [], must be used.

File

Overview  This module contains functions to manipulate files.

Some of those functions are low-level, allowing the user to interact with the file or IO devices, like open/2, copy/3 and others. This module also provides higher level functions that work with filenames and have their naming based on UNIX variants. For example, one can copy a file via cp/3 and remove files and directories recursively via rm_rf/1

Encoding  In order to write and read files, one must use the functions in the IO module. By default, a file is opened in binary mode which requires the functions IO.binread/2 and IO.binwrite/2 to interact with the file. A developer may pass :utf8 as an option when opening the file, then the slower IO.read/2 and IO.write/2 functions must be used as they are responsible for doing the proper conversions and data guarantees.

Note that filenames when given as char lists in Elixir are always treated as UTF-8. In particular, we expect that the shell and the operating system are configured to use UTF8 encoding. Binary filenames are considering raw and passed to the OS as is.

API  Most of the functions in this module return :ok or {:ok, result} in case of success, {:error, reason} otherwise. Those function are also followed by a variant that ends with ! which returns the result (without the {:ok, result} tuple) in case of success or raises an exception in case it fails. For example:

File.read("hello.txt")
#=> {:ok, "World"}

File.read("invalid.txt")
#=> {:error, :enoent}

File.read!("hello.txt")
#=> "World"

File.read!("invalid.txt")
#=> raises File.Error

In general, a developer should use the former in case he wants to react if the file does not exist. The latter should be used when the developer expects his software to fail in case the file cannot be read (i.e. it is literally an exception).

Processes and raw files  Every time a file is opened, Elixir spawns a new process. Writing to a file is equivalent to sending messages to that process that writes to the file descriptor.
This means files can be passed between nodes and message passing guarantees they can write to the same file in a network.

However, you may not always want to pay the price for this abstraction. In such cases, a file can be opened in :raw mode. The options :read_ahead and :delayed_write are also useful when operating large files or working with files in tight loops.

Check [http://www.erlang.org/doc/man/file.html#open-2](http://www.erlang.org/doc/man/file.html#open-2) for more information about such options and other performance considerations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd!/1</td>
<td>The same as cd/1, but raises an exception if it fails</td>
</tr>
<tr>
<td>cd!/2</td>
<td>Changes the current directory to the given path, executes the given function and then revert back to the previous path</td>
</tr>
<tr>
<td>cd/1</td>
<td>Sets the current working directory</td>
</tr>
<tr>
<td>chgrp!/2</td>
<td>Same as chgrp/2, but raises an exception in case of failure. Otherwise :ok</td>
</tr>
<tr>
<td>chgrp/2</td>
<td>Changes the user group given by the group id gid for a given file. Returns :ok on success, or {:error, reason} on failure</td>
</tr>
<tr>
<td>chmod!/2</td>
<td>Same as chmod/2, but raises an exception in case of failure. Otherwise :ok</td>
</tr>
<tr>
<td>chmod/2</td>
<td>Changes the unix file mode for a given file. Returns :ok on success, or {:error, reason} on failure</td>
</tr>
<tr>
<td>chown!/2</td>
<td>Same as chown/2, but raises an exception in case of failure. Otherwise :ok</td>
</tr>
<tr>
<td>chown/2</td>
<td>Changes the owner given by the user id uid for a given file. Returns :ok on success, or {:error, reason} on failure</td>
</tr>
<tr>
<td>close/1</td>
<td>Closes the file referenced by io_device. It mostly returns :ok, except for some severe errors such as out of memory.</td>
</tr>
<tr>
<td>copy!/3</td>
<td>The same as copy/3 but raises File.CopyError if it fails. Returns the bytes_copied otherwise</td>
</tr>
<tr>
<td>copy/3</td>
<td>Copies the contents of source to destination</td>
</tr>
<tr>
<td>cp!/3</td>
<td>The same as cp/3, but raises File.CopyError if it fails. Returns the list of copied files otherwise</td>
</tr>
<tr>
<td>cp/3</td>
<td>Copies the contents in source to destination preserving its mode</td>
</tr>
<tr>
<td>cp_r!/3</td>
<td>The same as cp_r/3, but raises File.CopyError if it fails. Returns the list of copied files otherwise</td>
</tr>
<tr>
<td>cp_r/3</td>
<td>Copies the contents in source to destination</td>
</tr>
<tr>
<td>cwd/0</td>
<td>The same as cwd/0, but raises an exception if it fails</td>
</tr>
<tr>
<td>cwd/0</td>
<td>Gets the current working directory</td>
</tr>
<tr>
<td>dir?/1</td>
<td>Returns true if the path is a directory</td>
</tr>
<tr>
<td>exists?/1</td>
<td>Returns true if the given path exists. It can be regular file, directory, socket, symbolic link, named pipe or device file</td>
</tr>
<tr>
<td>ln_s/2</td>
<td>Creates a symbolic link new to the file or directory existing</td>
</tr>
<tr>
<td>ls!/1</td>
<td>The same as ls/1 but raises File.Error in case of an error</td>
</tr>
<tr>
<td>ls/1</td>
<td>Returns list of files in the given directory</td>
</tr>
<tr>
<td>mkdir!/1</td>
<td>Same as mkdir/1, but raises an exception in case of failure. Otherwise :ok</td>
</tr>
<tr>
<td>mkdir/1</td>
<td>Tries to create the directory path. Missing parent directories are not created. Returns :ok if successful, or {:error, reason} on failure</td>
</tr>
<tr>
<td>mkdir_p!/1</td>
<td>Same as mkdir_p/1, but raises an exception in case of failure. Otherwise :ok</td>
</tr>
<tr>
<td>mkdir_p/1</td>
<td>Tries to create the directory path. Missing parent directories are created. Returns :ok if successful, or {:error, reason} on failure</td>
</tr>
<tr>
<td>open!/2</td>
<td>Same as open/2 but raises an error if file could not be opened. Returns the io_device otherwise</td>
</tr>
<tr>
<td>open/3</td>
<td>Same as open/3 but raises an error if file could not be opened. Returns the function result otherwise</td>
</tr>
<tr>
<td>open/2</td>
<td>Opens the given path according to the given list of modes</td>
</tr>
<tr>
<td>open/3</td>
<td>Similar to open/2 but expects a function as last argument</td>
</tr>
<tr>
<td>read!/1</td>
<td>Returns binary with the contents of the given filename or raises File.Error if an error occurs</td>
</tr>
<tr>
<td>read/1</td>
<td>Returns {:ok, binary}, where binary is a binary data object that contains the contents of path, or {:error, reason} if the path is a regular file</td>
</tr>
<tr>
<td>regular?/1</td>
<td>Returns true if the path is a regular file</td>
</tr>
<tr>
<td>rm!/1</td>
<td>Same as rm/1, but raises an exception in case of failure. Otherwise :ok</td>
</tr>
<tr>
<td>rm/1</td>
<td>Tries to delete the file path. Returns :ok if successful, or {:error, reason} if an error occurs</td>
</tr>
<tr>
<td>rm_rf!/1</td>
<td>Same as rm_rf/1 but raises File.Error in case of failures, otherwise the list of files or directories removed</td>
</tr>
<tr>
<td>rm_rf/1</td>
<td>Remove files and directories recursively at the given path. Symlinks are not followed but simply removed, not replaced. Returns the bytes_copied otherwise</td>
</tr>
<tr>
<td>rmdir!/1</td>
<td>Same as rmdir/1, but raises an exception in case of failure. Otherwise :ok</td>
</tr>
<tr>
<td>rmdir/1</td>
<td>Tries to delete the dir at path. Returns :ok if successful, or {:error, reason} if an error occurs</td>
</tr>
<tr>
<td>stat!/2</td>
<td>Same as stat/2 but returns the File.Stat directly and throws File.Error if an error is returned</td>
</tr>
<tr>
<td>stat/2</td>
<td>Returns information about the path. If it exists, it returns a {:ok, info} tuple, where info is a File.Stat struct. Returns a File.Stream for the given path with the given modes</td>
</tr>
<tr>
<td>stream!/3</td>
<td>Returns the list of files in the given directory, or raises File.Error if an error occurs</td>
</tr>
</tbody>
</table>

Check [http://www.erlang.org/doc/man/file.html#open-2](http://www.erlang.org/doc/man/file.html#open-2) for more information about such options and other performance considerations.
 touch!/2  
Same as touch/2 but raises an exception if it fails. Returns :ok otherwise

 touch/2  
Updates modification time (mtime) and access time (atime) of the given file. File is created if it doesn’t exist

 write!/3  
Same as write/3 but raises an exception if it fails, returns :ok otherwise

 write/3  
Writes content to the file path

 write_stat!/3  
Same as write_stat/3 but raises an exception if it fails. Returns :ok otherwise

 write_stat/3  
Writes the given File.Stat back to the filesystem at the given path. Returns :ok or {:error, reason}

---

Summary

Types

**posix**
- posix:::file.posix

**io_device**
- io_device:::file.io_device

**stat_options**
- stat_options:::[{time, :local | :universal | :posix}]

Functions

cd(path)(function)
Specs:

  • cd(Path.t):::ok | {:error, posix}

Sets the current working directory.

Returns :ok if successful, {:error, reason} otherwise.

cd!(path)(function)
Specs:

  • cd!(Path.t):::ok | no_return

The same as cd/1, but raises an exception if it fails.

cd!(path, function)(function)
Specs:

  • (cd!(Path.t, (res) -> res)):::res | no_return when res: var

Changes the current directory to the given path, executes the given function and then revert back to the previous path regardless if there is an exception.

Raises an error if retrieving or changing the current directory fails.

chgrp(path, gid)(function)
Specs:

  • chgrp(Path.t, integer):::ok | {:error, posix}

Changes the user group given by the group id gid for a given file. Returns :ok on success, or {:error, reason} on failure.

chgrp!(path, gid)(function)
Specs:

  • chgrp!(Path.t, integer):::ok | no_return

Same as chgrp/2, but raises an exception in case of failure. Otherwise :ok.
**Elixir Documentation, Release**

`chmod(path, mode)(function)`
Specs:

```
•chmod(Path.t, integer) :: :ok | {:error, posix}
```

Changes the unix file mode for a given file. Returns :ok on success, or {:error, reason} on failure.

`chmod!(path, mode)(function)`
Specs:

```
•chmod!(Path.t, integer) :: :ok | no_return
```

Same as chmod/2, but raises an exception in case of failure. Otherwise :ok.

`chown(path, uid)(function)`
Specs:

```
•chown(Path.t, integer) :: :ok | {:error, posix}
```

Changes the owner given by the user id uid for a given file. Returns :ok on success, or {:error, reason} on failure.

`chown!(path, uid)(function)`
Specs:

```
•chown!(Path.t, integer) :: :ok | no_return
```

Same as chown/2, but raises an exception in case of failure. Otherwise :ok.

`close(io_device)(function)`
Specs:

```
•close(io_device) :: :ok | {:error, posix | :badarg | :terminated}
```

Closes the file referenced by io_device. It mostly returns :ok, except for some severe errors such as out of memory.

Note that if the option :delayed_write was used when opening the file, close/1 might return an old write error and not even try to close the file. See open/2.

`copy(source, destination, bytes_count \ :infinity)(function)`
Specs:

```
•copy(Path.t, Path.t, pos_integer | :infinity) :: {:ok, non_neg_integer} | {:error, posix}
```

Copies the contents of source to destination.

Both parameters can be a filename or an io device opened with open/2. bytes_count specifies the number of bytes to copy, the default being :infinity.

If file destination already exists, it is overwritten by the contents in source.

Returns {:ok, bytes_copied} if successful, {:error, reason} otherwise.

Compared to the cp/3, this function is more low-level, allowing a copy from device to device limited by a number of bytes. On the other hand, cp/3 performs more extensive checks on both source and destination and it also preserves the file mode after copy.

Typical error reasons are the same as in open/2, read/1 and write/3.

`copy!(source, destination, bytes_count \ :infinity)(function)`
Specs:

```
•copy!(Path.t, Path.t, pos_integer | :infinity) :: non_neg_integer | no_return
```

The same as copy/3 but raises a File.CopyError if it fails. Returns the bytes_copied otherwise.
**Elixir Documentation, Release**

`cp(source, destination, callback \ fn _, _ -> true end)(function)`

**Specs:**

- `cp(Path.t, Path.t, (Path.t, Path.t -> boolean)) :: ok | {:error, posix}`

Copies the contents in `source` to `destination` preserving its mode.

If a file already exists in the destination, it invokes a callback which should return `true` if the existing file should be overwritten, `false` otherwise. It defaults to return `true`.

It returns `:ok` in case of success, returns `{:error, reason}` otherwise.

If you want to copy contents from an io device to another device or do a straight copy from a source to a destination without preserving modes, check `copy/3` instead.

**Note:** The command `cp` in Unix systems behaves differently depending if `destination` is an existing directory or not. We have chosen to explicitly disallow this behaviour. If destination is a directory, an error will be returned.

`cp!(source, destination, callback \ fn _, _ -> true end)(function)`

**Specs:**

- `cp!(Path.t, Path.t, (Path.t, Path.t -> boolean)) :: :ok | no_return`

The same as `cp/3`, but raises `File.CopyError` if it fails. Returns the list of copied files otherwise.

`cp_r(source, destination, callback \ fn _, _ -> true end)(function)`

**Specs:**

- `cp_r(Path.t, Path.t, (Path.t, Path.t -> boolean)) :: {:ok, [binary]} | {:error, posix, binary}`

Copies the contents in source to destination.

If the source is a file, it copies `source` to `destination`. If the source is a directory, it copies the contents inside source into the destination.

If a file already exists in the destination, it invokes a callback which should return `true` if the existing file should be overwritten, `false` otherwise. It defaults to return `true`.

If a directory already exists in the destination where a file is meant to be (or otherwise), this function will fail.

This function may fail while copying files, in such cases, it will leave the destination directory in a dirty state, where already copied files won’t be removed.

It returns `{:ok, files_and_directories}` in case of success with all files and directories copied in no specific order, `{:error, reason, file}` otherwise.

**Note:** The command `cp` in Unix systems behaves differently depending if `destination` is an existing directory or not. We have chosen to explicitly disallow this behaviour.

**Examples**

```elixir
# Copies "a.txt" to "tmp"
File.cp_r "a.txt", "tmp.txt"

# Copies all files in "samples" to "tmp"
File.cp_r "samples", "tmp"

# Same as before, but asks the user how to proceed in case of conflicts
File.cp_r "samples", "tmp", fn(source, destination) ->
  IO.gets("Overwriting #{destination} by #{source}. Type y to confirm.") == "y"
end
```

`cp_r!(source, destination, callback \ fn _, _ -> true end)(function)`

**Specs:**

Elixir Documentation, Release
• `cp_r(Path.t, Path.t, (Path.t, Path.t) -> boolean) :: [binary] | no_return`

The same as `cp_r/3`, but raises `File.CopyError` if it fails. Returns the list of copied files otherwise.

`cwd()` (function)
Specs:

• `cwd :: {:ok, binary} | {:error, posix}`

Gets the current working directory.

In rare circumstances, this function can fail on Unix. It may happen if read permission does not exist for the parent directories of the current directory. For this reason, returns `{:ok, cwd}` in case of success, `{:error, reason}` otherwise.

`cwd!()` (function)
Specs:

• `cwd! :: binary | no_return`

The same as `cwd/0`, but raises an exception if it fails.

`dir?(path)` (function)
Specs:

• `dir?(Path.t) :: boolean`

Returns `true` if the path is a directory.

`exists?(path)` (function)
Specs:

• `exists?(Path.t) :: boolean`

Returns `true` if the given path exists. It can be regular file, directory, socket, symbolic link, named pipe or device file.

Examples

```elixir
File.exists?("test/")
#=> true

File.exists?("missing.txt")
#=> false

File.exists?("/dev/null")
#=> true
```

`ln_s(existing, new)` (function)

Creates a symbolic link `new` to the file or directory `existing`.

Returns `:ok` if successful, `{:error, reason}` otherwise. If the operating system does not support symbolic links, returns `{:error, :enotsup}`.

`ls(path \ " ".")` (function)
Specs:

• `ls(Path.t) :: {:ok, [binary]} | {:error, posix}`

Returns list of files in the given directory.

It returns `{:ok, [files]}` in case of success, `{:error, reason}` otherwise.

`ls!(path \ " ".")` (function)
Specs:
ls!(Path.t) :: [binary] | no_return

The same as ls/1 but raises File.Error in case of an error.

mkdir(path) (function)
Specs:
•mkdir(Path.t) :: :ok | {:error, posix}

Tries to create the directory path. Missing parent directories are not created. Returns :ok if successful, or {:error, reason} if an error occurs.

Typical error reasons are:
•:eacces - Missing search or write permissions for the parent directories of path.
•:excl - There is already a file or directory named path.
•:enoent - A component of path does not exist.
•:nospc - There is a no space left on the device.
•:enotdir - A component of path is not a directory. On some platforms, :enoent is returned instead.

mkdir!(path) (function)
Specs:
•mkdir!(Path.t) :: :ok | no_return

Same as mkdir/1, but raises an exception in case of failure. Otherwise :ok.

mkdir_p(path) (function)
Specs:
•mkdir_p(Path.t) :: :ok | {:error, posix}

Tries to create the directory path. Missing parent directories are created. Returns :ok if successful, or {:error, reason} if an error occurs.

Typical error reasons are:
•:eacces - Missing search or write permissions for the parent directories of path.
•:nospc - There is a no space left on the device.
•:enotdir - A component of path is not a directory.

mkdir_p!(path) (function)
Specs:
•mkdir_p!(Path.t) :: :ok | no_return

Same as mkdir_p/1, but raises an exception in case of failure. Otherwise :ok.

open(path, modes \ [] ) (function)
Specs:
•open(Path.t, []) :: {:ok, io_device} | {:error, posix}

Opens the given path according to the given list of modes.

In order to write and read files, one must use the functions in the IO module. By default, a file is opened in binary mode which requires the functions IO.binread/2 and IO.binwrite/2 to interact with the file. A developer may pass :utf8 as an option when opening the file and then all other functions from IO are available, since they work directly with Unicode data.

The allowed modes:
- **:read** - The file, which must exist, is opened for reading.

- **:write** - The file is opened for writing. It is created if it does not exist. If the file exists, and if write is not combined with read, the file will be truncated.

- **:append** - The file will be opened for writing, and it will be created if it does not exist. Every write operation to a file opened with append will take place at the end of the file.

- **:exclusive** - The file, when opened for writing, is created if it does not exist. If the file exists, open will return {:error, :eexist}.

- **:char_list** - When this term is given, read operations on the file will return char lists rather than binaries;

- **:compressed** - Makes it possible to read or write gzip compressed files. The compressed option must be combined with either read or write, but not both. Note that the file size obtained with `stat/1` will most probably not match the number of bytes that can be read from a compressed file.

- **:utf8** - This option denotes how data is actually stored in the disk file and makes the file perform automatic translation of characters to and from utf-8. If data is sent to a file in a format that cannot be converted to the utf-8 or if data is read by a function that returns data in a format that cannot cope with the character range of the data, an error occurs and the file will be closed.


This function returns:

- {:ok, io_device} - The file has been opened in the requested mode. `io_device` is actually the pid of the process which handles the file. This process is linked to the process which originally opened the file. If any process to which the `io_device` is linked terminates, the file will be closed and the process itself will be terminated. An `io_device` returned from this call can be used as an argument to the `IO` module functions.

- {:error, reason} - The file could not be opened.

**Examples**

```elixir
{:ok, file} = File.open("foo.tar.gz", [:read, :compressed])
IO.read(file, :line)
File.close(file)
```

**open** *(path, modes, function)*

**Specs:**

- `(open(Path.t, [], (io_device -> res)) :: {:ok, res} | {:error, posix})` when res: var

Similar to `open/2` but expects a function as last argument.

The file is opened, given to the function as argument and automatically closed after the function returns, regardless if there was an error or not.

It returns {:ok, function_result} in case of success, {:error, reason} otherwise.

Do not use this function with `:delayed_write` option since automatically closing the file may fail (as writes are delayed).

**Examples**

```elixir
File.open("file.txt", [:read, :write], fn(file) ->
  IO.read(file, :line)
end)
```
open!(path, modes \ \
[])(function)
   Specs:
   •open!(Path.t, []) :: io_device | no_return
   Same as open/2 but raises an error if file could not be opened. Returns the io_device otherwise.

open!(path, modes, function)(function)
   Specs:
   •(open!(Path.t, [], (io_device -> res)) :: res | no_return) when res: var
   Same as open/3 but raises an error if file could not be opened. Returns the function result otherwise.

read(path)(function)
   Specs:
   •read(Path.t) :: {:ok, binary} | {:error, posix}
   Returns {:ok, binary}, where binary is a binary data object that contains the contents of path, or
   {:error, reason} if an error occurs.
   Typical error reasons:
   •:enoent - The file does not exist.
   •:eacces - Missing permission for reading the file, or for searching one of the parent directories.
   •:eisdir - The named file is a directory.
   •:enotdir - A component of the file name is not a directory. On some platforms, :enoent is returned
     instead.
   •:enomem - There is not enough memory for the contents of the file.
   You can use /:file.format_error/1\(\texttt{http://www.erlang.org/doc/man/file.html#format_error-1}\)’ to get a descriptive
   string of the error.

read!(path)(function)
   Specs:
   •read!(Path.t) :: binary | no_return
   Returns binary with the contents of the given filename or raises File.Error if an error occurs.

regular?(path)(function)
   Specs:
   •regular?(Path.t) :: boolean
   Returns true if the path is a regular file.

Examples
File.regular? __ENV__.file #=> true

rm(path)(function)
   Specs:
   •rm(Path.t) :: :ok | {:error, posix}
   Tries to delete the file path. Returns :ok if successful, or {:error, reason} if an error occurs.
   Typical error reasons are:
   •:enoent - The file does not exist.
   •:eacces - Missing permission for the file or one of its parents.
• :eperm - The file is a directory and user is not super-user.
• :enotdir - A component of the file name is not a directory. On some platforms, enoent is returned instead.
• :einv - Filename had an improper type, such as tuple.

Examples

File.rm('file.txt')
#=> :ok

File.rm('tmp_dir/专业化')
#=> {:error, :eperm}

rm!(path) (function)
Specs:

• rm!(Path.t) :: :ok | no_return

Same as rm/1, but raises an exception in case of failure. Otherwise :ok.

rm_rf(path) (function)
Specs:

• rm_rf(Path.t) :: {ok, [binary]} | {error, posix, binary}

Remove files and directories recursively at the given path. Symlinks are not followed but simply removed, non-existing files are simply ignored (i.e. doesn’t make this function fail).

Returns {:ok, files_and_directories} with all files and directories removed in no specific order, {:error, reason, file} otherwise.

Examples

File.rm_rf "samples"
#=> {:ok, ["samples", "samples/1.txt"]}

File.rm_rf "unknown"
#=> {:ok, []}

rm_rf!(path) (function)
Specs:

• rm_rf!(Path.t) :: [binary] | no_return

Same as rm_rf/1 but raises File.Error in case of failures, otherwise the list of files or directories removed.

rmdir(path) (function)
Specs:

• rmdir(Path.t) :: :ok | {:error, posix}

Tries to delete the dir at path. Returns :ok if successful, or {:error, reason} if an error occurs.

Examples

File.rmdir('tmp_dir')
#=> :ok

File.rmdir('file.txt')
#=> {:error, :enotdir}

rmdir!(path) (function)
Specs:
• rmdir!(Path.t) :: :ok | {:error, posix}

Same as rmdir/1, but raises an exception in case of failure. Otherwise :ok.

stat(path, opts \ [[]](function)
Specs:
• stat(Path.t, stat_options) :: {:ok, File.Stat.t} | {:error, posix}

Returns information about the path. If it exists, it returns a {:ok, info} tuple, where info is a File.Stat struct. Returns {:error, reason} with the same reasons as read/1 if a failure occurs.

Options
The accepted options are:
• :time if the time should be :local, :universal or :posix. Default is :local.

stat!(path, opts \ [[]](function)
Specs:
• stat!(Path.t, stat_options) :: File.Stat.t | no_return

Same as stat/2 but returns the File.Stat directly and throws File.Error if an error is returned.

stream!(path, modes \ [[]], line_or_bytes \ :line)(function)
Returns a File.Stream for the given path with the given modes.

The stream implements both Enumerable and Collectable protocols, which means it can be used both for read and write.

The line_or_byte argument configures how the file is read when streaming, by :line (default) or by a given number of bytes.

Operating the stream can fail on open for the same reasons as File.open!/2. Note that the file is automatically opened only and every time streaming begins. There is no need to pass :read and :write modes, as those are automatically set by Elixir.

Raw files
Since Elixir controls when the streamed file is opened, the underlying device cannot be shared and as such it is convenient to open the file in raw mode for performance reasons. Therefore, Elixir will open streams in :raw mode with the :read_ahead option unless an encoding is specified.

One may also consider passing the :delayed_write option if the stream is meant to be written to under a tight loop.

touch(path, time \ :calendar.local_time())(function)
Specs:
• touch(Path.t, :calendar.datetime) :: :ok | {:error, posix}

Updates modification time (mtime) and access time (atime) of the given file. File is created if it doesn’t exist.

touch!(path, time \ :calendar.local_time())(function)
Specs:
• touch!(Path.t, :calendar.datetime) :: :ok | no_return

Same as touch/2 but raises an exception if it fails. Returns :ok otherwise.

write(path, content, modes \ [[]](function)
Specs:
• write(Path.t, iodata, []) :: :ok | {:error, posix}
Writes content to the file path.

The file is created if it does not exist. If it exists, the previous contents are overwritten. Returns :ok if successful, or {:error, reason} if an error occurs.

**Warning:** Every time this function is invoked, a file descriptor is opened and a new process is spawned to write to the file. For this reason, if you are doing multiple writes in a loop, opening the file via File.open/2 and using the functions in IO to write to the file will yield much better performance then calling this function multiple times.

Typical error reasons are:

- :enoent - A component of the file name does not exist.
- :enotdir - A component of the file name is not a directory. On some platforms, enoent is returned instead.
- :enospc - There is a no space left on the device.
- :eaccess - Missing permission for writing the file or searching one of the parent directories.
- :eisdir - The named file is a directory.

The writing is automatically done in :raw mode. Check File.open/2 for other available options.

```elixir
write!(path, content, modes \ \ []) (function)
```

**Specs:**

```elixir
•write!(Path.t, iodata, []) :: :ok \ no_return
```

Same as write/3 but raises an exception if it fails, returns :ok otherwise.

```elixir
write_stat!(path, stat, opts \ \ []) (function)
```

**Specs:**

```elixir
•write_stat!(Path.t, File.Stat.t, stat_options) :: :ok \ [{:error, posix}]
```

Defines the given File.Stat back to the filesystem at the given path. Returns :ok or {:error, reason}.

```elixir
write_stat!(path, stat, opts \ \ []) (function)
```

**Specs:**

```elixir
•write_stat!(Path.t, File.Stat.t, stat_options) :: :ok \ no_return
```

Same as write_stat/3 but raises an exception if it fails. Returns :ok otherwise.

**File.Stat**

**Overview** A struct responsible to hold file information.

In Erlang, this struct is represented by a :file_info record. Therefore this module also provides functions for converting in between the Erlang record and the Elixir struct.

Its fields are:

- **size** - Size of file in bytes.
- **access** - :read, :write, :read_write, :none. The current system access to the file.
- **atime** - The last time the file was read.
- **mtime** - The last time the file was written.
- **ctime** - The interpretation of this time field depends on the operating system. On Unix, it is the last time the file or the inode was changed. In Windows, it is the create time.
• mode - The file permissions.
• links - The number of links to this file. This is always 1 for file systems which have no concept of links.
• major_device - Identifies the file system where the file is located. In windows, the number indicates a drive as follows: 0 means A:, 1 means B:, and so on.
• minor_device - Only valid for character devices on Unix. In all other cases, this field is zero.
• inode - Gives the inode number. On non-Unix file systems, this field will be zero.
• uid - Indicates the owner of the file.
• gid - Gives the group that the owner of the file belongs to. Will be zero for non-Unix file systems.

The time type returned in atime, mtime, and ctime is dependent on the time type set in options. \{:time, type\} where type can be :local, :universal, or :posix. Default is :local.

### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_record/1</td>
<td>Converts a :file_info record into a File.Stat</td>
</tr>
<tr>
<td>to_record/1</td>
<td>Converts a File.Stat struct to a :file_info record</td>
</tr>
</tbody>
</table>

### Types


### Functions

**from_record(arg1)** (function)


**to_record(stat)** (function)

Converts a File.Stat struct to a :file_info record.

### File.Stream

**Overview**

Defines a File.Stream struct returned by File.stream!/2.

The following fields are public:

• path - the file path
• modes - the file modes
• raw - a boolean indicating if bin functions should be used
• line_or_bytes - if reading should read lines or a given amount of bytes

### Types

t :: %File.Stream{path: term, modes: term, line_or_bytes: term, raw: term}

### Float

**Overview**

Functions for working with floating point numbers.
Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil/1</td>
<td>Rounds a float to the largest integer greater than or equal to <code>num</code></td>
</tr>
<tr>
<td>floor/1</td>
<td>Rounds a float to the largest integer less than or equal to <code>num</code></td>
</tr>
<tr>
<td>parse/1</td>
<td>Parses a binary into a float</td>
</tr>
<tr>
<td>round/2</td>
<td>Rounds a floating point value to an arbitrary number of fractional digits (between 0 and 15)</td>
</tr>
<tr>
<td>to_char_list/1</td>
<td>Returns a char list which corresponds to the text representation of the given float</td>
</tr>
<tr>
<td>to_char_list/2</td>
<td>Returns a list which corresponds to the text representation of <code>float</code></td>
</tr>
<tr>
<td>to_string/1</td>
<td>Returns a binary which corresponds to the text representation of <code>some_float</code></td>
</tr>
<tr>
<td>to_string/2</td>
<td>Returns a binary which corresponds to the text representation of <code>float</code></td>
</tr>
</tbody>
</table>

Functions

**ceil (num) (function)**

Specs:

- ceil(float | integer) :: integer

Rounds a float to the largest integer greater than or equal to `num`.

**Examples**

```
iex> Float.ceil(34)
34
iex> Float.ceil(34.25)
35
iex> Float.ceil(-56.5)
-56
```

**floor (num) (function)**

Specs:

- floor(float | integer) :: integer

Rounds a float to the largest integer less than or equal to `num`.

**Examples**

```
iex> Float.floor(34)
34
iex> Float.floor(34.25)
34
iex> Float.floor(-56.5)
-57
```

**parse (binary) (function)**

Specs:

- parse(binary) :: {float, binary} | :error

Parses a binary into a float.

If successful, returns a tuple of the form `{float, remainder_of_binary}`. Otherwise :error.

**Examples**

```
iex> Float.parse("34")
{34.0, ""}
iex> Float.parse("34.25")
{34.25, ""}
```
iex > Float.parse("56.5xyz")
(56.5, "xyz")

iex > Float.parse("pi")
:error

round(number, precision)(function)
Specs:

•round(float, integer) :: float

Rounds a floating point value to an arbitrary number of fractional digits (between 0 and 15).

Examples

iex > Float.round(5.5674, 3)
5.567

iex > Float.round(5.5675, 3)
5.568

iex > Float.round(-5.5674, 3)
-5.567

iex > Float.round(-5.5675, 3)
-5.568

to_char_list(number)(function)
Specs:

•to_char_list(float) :: char_list

Returns a char list which corresponds to the text representation of the given float.

Inlined by the compiler.

Examples

iex > Float.to_char_list(7.0)
'7.00000000000000000000e+00'

to_char_list(float, options)(function)
Specs:

•to_char_list(float, []) :: char_list

Returns a list which corresponds to the text representation of float.

Options

•:decimals — number of decimal points to show

•:scientific — number of decimal points to show, in scientific format

•:compact — when true, use the most compact representation (ignored with the scientific option)

Examples

iex > Float.to_char_list 7.1, [:decimals: 2, :compact: true]
'7.1'

to_string(some_float)(function)
Specs:
•to_string(float) :: String.t

Returns a binary which corresponds to the text representation of some_float.

Inlined by the compiler.

Examples

iex> Float.to_string(7.0)
"7.00000000000000000000e+00"

to_string(float, options)(function)

Specs:

•to_string(float, []) :: String.t

Returns a binary which corresponds to the text representation of float.

Options

•:decimals — number of decimal points to show
•:scientific — number of decimal points to show, in scientific format
•:compact — when true, use the most compact representation (ignored with the scientific option)

Examples

iex> Float.to_string 7.1, [decimals: 2, compact: true]
"7.1"

GenEvent

Overview  A behaviour module for implementing event handling functionality.

The event handling model consists of a generic event manager process with an arbitrary number of event handlers which are added and deleted dynamically.

An event manager implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an supervision tree.

Example  There are many use cases for event handlers. For example, a logging system can be built using event handlers where which log message is an event and different event handlers can be plugged to handle the log messages. One handler may print error messages on the terminal, another can write it to a file, while a third one can keep the messages in memory (like a buffer) until they are read.

As an example, let’s have a GenEvent that accumulates messages until they are collected by an explicit call.

defmodule LoggerHandler do
  use GenEvent

  # Callbacks

  def handle_event({:log, x}, messages) do
    {:ok, [x|messages]}
  end

  def handle_call(:messages, messages) do
    {:ok, Enum.reverse(messages), []}
  end
end
Elixir Documentation, Release

{:ok, pid} = GenEvent.start_link()

GenEvent.add_handler(pid, LoggerHandler, [])
#=> :ok

GenEvent.notify(pid, {:log, 1})
#=> :ok

GenEvent.notify(pid, {:log, 2})
#=> :ok

GenEvent.call(pid, LoggerHandler, :messages)
#=> [1, 2]

GenEvent.call(pid, LoggerHandler, :messages)
#=> []

We start a new event manager by calling GenEvent.start_link/0. Notifications can be sent to the event manager which will then invoke handle_event/0 for each registered handler.

We can add new handlers with add_handler/4. Calls can also be made to specific handlers by using call/3.

**Callbacks** There are 6 callbacks required to be implemented in a GenEvent. By adding use GenEvent to your module, Elixir will automatically define all 6 callbacks for you, leaving it up to you to implement the ones you want to customize. The callbacks are:

- **init(args)** - invoked when the event handler is added
  It must return:
  - {:ok, state}
  - {:ok, state, :hibernate}
  - {:error, reason}
  - handle_event(msg, state) - invoked whenever an event is sent via notify/2 or sync_notify/2.

  It must return:
  - {:ok, new_state}
  - {:ok, new_state, :hibernate}
  - {:swap_handler, arg1, new_state, handler2, arg2}
  - :remove_handler
  - handle_call(msg, state) - invoked when a call/3 is done to a specific handler.

  It must return:
  - {:ok, reply, new_state}
  - {:ok, reply, new_state, :hibernate}
  - {:swap_handler, reply, arg1, new_state, handler2, arg2}
  - {:remove_handler, reply}
  - handle_info(msg, state) - invoked to handle all other messages which are received by the process.

  Must return the same values as handle_event/2;

  It must return:
Elixir Documentation, Release

- {:noreply, state}
- {:noreply, state, timeout}
- {:stop, reason, state}
- terminate(reason, state) - called when the event handler is removed or the event manager is terminating. It can return any term.
- code_change(old_vsn, state, extra) - called when the application code is being upgraded live (hot code swapping).

It must return:
- {:ok, new_state}

Name Registration  A GenEvent is bound to the same name registration rules as a GenServer. Read more about it in the GenServer docs.

Streaming  GenEvents can be streamed from and streamed with the help of stream/2. Here are some examples:

```elixir
stream = GenEvent.stream(pid)

# Take the next 10 events
Enum.take(stream, 10)

# Print all remaining events
for event <- stream do
  IO.inspect event
end
```

A stream may also be given an id, which allows all streams with the given id to be cancelled at any moment via cancel_streams/1.

Learn more  If you wish to find out more about gen events, Elixir getting started guides provide a tutorial-like introduction. The documentation and links in Erlang can also provide extra insight.

- http://elixir-lang.org/getting_started/mix/1.html

<table>
<thead>
<tr>
<th><strong>struct</strong>/0</th>
<th>Defines a GenEvent stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_handler/4</td>
<td>Makes a new event handler to the event manager</td>
</tr>
<tr>
<td>call/4</td>
<td>Cancels all streams currently running with the given :id</td>
</tr>
<tr>
<td>cancel_streams/1</td>
<td>Sends an event notification to the event manager</td>
</tr>
<tr>
<td>notify/2</td>
<td>Removes an event handler from the event manager</td>
</tr>
<tr>
<td>remove_handler/3</td>
<td>Starts an event manager process without links (outside of a supervision tree)</td>
</tr>
<tr>
<td>start/1</td>
<td>Starts an event manager linked to the current process</td>
</tr>
<tr>
<td>start_link/1</td>
<td>Terminates the event manager</td>
</tr>
<tr>
<td>stop/1</td>
<td>Returns a stream that consumes and notifies events to the manager</td>
</tr>
<tr>
<td>stream/2</td>
<td>Replaces an old event handler with a new one in the event manager</td>
</tr>
<tr>
<td>swap_handler/6</td>
<td>Sends a sync event notification to the event manager</td>
</tr>
<tr>
<td>sync_notify/2</td>
<td>Returns a list of all event handlers installed in the manager</td>
</tr>
<tr>
<td>which_handlers/1</td>
<td></td>
</tr>
</tbody>
</table>

Summary
Types

**on_start**

```elixir
on_start :: {:ok, pid} | {:error, {:already_started, pid}}
```

Return values of `start*` functions

**name**

```elixir
name :: atom | {:global, term} | {:via, module, term}
```

The GenEvent manager name

**options**

```elixir
options :: [{name, name}]
```

Options used by the `start*` functions

**manager**

```elixir
manager :: pid | name | {atom, node}
```

The event manager reference

**handler**

```elixir
handler :: module | {module, term}
```

Supported values for new handlers

**t**

```elixir
t :: %GenEvent{manager: term, id: term, timeout: term, duration: term}
```

Functions

**__struct__()** (function)

Specs:

```
•__struct__ :: t
```

Defines a GenEvent stream.

This is a struct returned by `stream/2`. The struct is public and contains the following fields:

• `:manager` - the manager reference given to `GenEvent.stream/2`
• `:id` - the event stream id for cancellation
• `:timeout` - the timeout in between events, defaults to `:infinity`
• `:duration` - the duration of the subscription, defaults to `:infinity`

**add_handler(manager, handler, args, options \ \ [])** (function)

Specs:

```
•add_handler(manager, handler, term, [{:link, boolean}]) :: :ok | {:EXIT, term} | {:error, term}
```

Adds a new event handler to the event manager.

The event manager will call the `init/1` callback with `args` to initiate the event handler and its internal state. If `init/1` returns a correct value indicating successful completion, the event manager adds the event handler and this function returns `:ok`. If the callback fails with `reason` or returns `{:error, reason}`, the event handler is ignored and this function returns `{:EXIT, reason}` or `{:error, reason}`, respectively.

**Linked handlers**

When adding a handler, a `:link` option with value `true` can be given. This means the event handler and the calling process are now linked.

If the calling process later terminates with `reason`, the event manager will delete the event handler by calling the `terminate/2` callback with `{:stop, reason}` as argument. If the event handler later is deleted,
the event manager sends a message \{:gen_event_EXIT, handler, reason\} to the calling process. Reason is one of the following:

- **:normal** - if the event handler has been removed due to a call to \texttt{remove_handler/3}, or \texttt{remove_handler} has been returned by a callback function;
- **:shutdown** - if the event handler has been removed because the event manager is terminating;
- **{:swapped, new_handler, pid}** - if the process \texttt{pid} has replaced the event handler by another;
- a term - if the event handler is removed due to an error. Which term depends on the error;

\texttt{call(manager, handler, request, timeout \ \ 5000)}(function)

Specs:

\texttt{call(manager, handler, term, timeout) :: term | {:error, term}}

Makes a synchronous call to the event handler installed in \texttt{manager}.

The given \texttt{request} is sent and the caller waits until a reply arrives or a timeout occurs. The event manager will call \texttt{handle_call/2} to handle the request.

The return value \texttt{reply} is defined in the return value of \texttt{handle_call/2}. If the specified event handler is not installed, the function returns \{:error, :bad_module\}.

\texttt{cancel_streams(genevent)}(function)

Specs:

\texttt{cancel_streams(t)::ok}

Cancels all streams currently running with the given \texttt{id}.

In order for a stream to be cancelled, an \texttt{id} must be passed when the stream is created via \texttt{stream/2}. Passing a stream without an id leads to an argument error.

\texttt{notify(manager, event)}(function)

Specs:

\texttt{notify(manager, term)::ok}

Sends an event notification to the event manager.

The event manager will call \texttt{handle_event/2} for each installed event handler.

\texttt{notify} is asynchronous and will return immediately after the notification is sent. \texttt{notify} will not fail even if the specified event manager does not exist, unless it is specified as \texttt{name} (atom).

\texttt{remove_handler(manager, handler, args)}(function)

Specs:

\texttt{remove_handler(manager, handler, term)::term | {:error, term}}

Removes an event handler from the event manager.

The event manager will call \texttt{terminate/2} to terminate the event handler and return the callback value. If the specified event handler is not installed, the function returns \{:error, :module_not_found\}.

\texttt{start(options \ \ [])}(function)

Specs:

\texttt{start(options)::on_start}

Starts an event manager process without links (outside of a supervision tree).

See \texttt{start_link/1} for more information.
start_link(options \ \
[])(function)
Specs:
  •start_link(options):: on_start
Starts an event manager linked to the current process.
This is often used to start the GenEvent as part of a supervision tree.
It accepts the :name option which is described under the Name Registration section in the GenServer module docs.
If the event manager is successfully created and initialized, the function returns {:ok, pid}, where pid is the pid of the server. If there already exists a process with the specified server name, the function returns {:error, {already_started, pid}} with the pid of that process.

stop(manager)(function)
Specs:
  •stop(manager):: :ok
Terminates the event manager.
Before terminating, the event manager will call terminate(:stop, ...) for each installed event handler.

stream(manager, options \ \
[])(function)
Returns a stream that consumes and notifies events to the manager.
The stream is a GenEvent struct that implements the Enumerable protocol. The supported options are:
  •:id - an id to identify all live stream instances. When an :id is given, existing streams can be called with via cancel_streams;
  •:timeout (Enumerable) - raises if no event arrives in X milliseconds;
  •:duration (Enumerable) - only consume events during the X milliseconds from the streaming start;

swap_handler(manager, handler1, args1, handler2, args2, options \ \
[])(function)
Specs:
  •swap_handler(manager, handler, term, handler, term, [{:link, boolean}]):: :ok | {error, term}
Replaces an old event handler with a new one in the event manager.
First, the old event handler is deleted by calling terminate/2 with the given args1 and collects the return value. Then the new event handler is added and initiated by calling init({args2, term}), where term is the return value of calling terminate/2 in the old handler. This makes it possible to transfer information from one handler to another.
The new handler will be added even if the specified old event handler is not installed in which case term = :error or if the handler fails to terminate with a given reason.
If there was a linked connection between handler1 and a process pid, there will be a link connection between handler2 and pid instead. A new link in between the caller process and the new handler can also be set with by giving link: true as option. See add_handler/4 for more information.
If init/1 in the second handler returns a correct value, this function returns :ok.

sync_notify(manager, event)(function)
Specs:
  •sync_notify(manager, term):: :ok
Sends a sync event notification to the event manager.
In other words, this function only returns :ok after the event manager invokes the handle_event/2 on each installed event handler.

See notify/2 for more info.

which_handlers(manager) (function)

Specs:

- which_handlers(manager):: [handler]

Returns a list of all event handlers installed in the manager.

GenServer

Overview A behaviour module for implementing the server of a client-server relation.

A GenServer is a process as any other Elixir process and it can be used to keep state, execute code asynchronously and so on. The advantage of using a generic server process (GenServer) implemented using this module is that it will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into a supervision tree.

Example The GenServer behaviour abstracts the common client-server interaction. Developer are only required to implement the callbacks and functionality they are interested in.

Let's start with a code example and then explore the available callbacks. Imagine we want a GenServer that works like a stack, allowing us to push and pop items:

```elixir
defmodule Stack do
  use GenServer

  # Callbacks

  def handle_call(:pop, _from, [h|t]) do
    {:reply, h, t}
  end

  def handle_cast({:push, item}, state) do
    {:noreply, [item|state]}
  end
end

# Start the server
{:ok, pid} = GenServer.start_link(Stack, [:hello])

# This is the client
GenServer.call(pid, :pop)
#=> :hello

GenServer.cast(pid, {:push, :world})
#=> :ok

GenServer.call(pid, :pop)
#=> :world
```

We start our Stack by calling start_link/3, passing the module with the server implementation and its initial argument (a list representing the stack containing the item :hello). We can primarily interact with the server by sending two types of messages. call messages expect a reply from the server (and are therefore synchronous) while cast messages do not.
Every time you do a `GenServer.call/3`, the client will send a message that must be handled by the `handle_call/3` callback in the GenServer. A `cast/2` message must be handled by `handle_cast/2`.

**Callbacks** There are 6 callbacks required to be implemented in a `GenServer`. By adding `use GenServer` to your module, Elixir will automatically define all 6 callbacks for you, leaving it up to you to implement the ones you want to customize. The callbacks are:

- `init(args)` - invoked when the server is started
  
  It must return:
  
  - `{:ok, state}
  - `{:ok, state, timeout}
  - `ignore`
  - `{:stop, reason}
  - `handle_call(msg, {from, ref}, state)` and `handle_cast(msg, state)` - invoked to handle call (sync) and cast (async) messages.

  It must return:
  
  - `{reply, reply, new_state}
  - `{reply, reply, new_state, timeout}
  - `{reply, reply, new_state, :hibernate}
  - `{noreply, new_state}
  - `{noreply, new_state, timeout}
  - `{noreply, new_state, :hibernate}
  - `{stop, reason, new_state}
  - `{stop, reason, reply, new_state}
  - `handle_info(msg, state)` - invoked to handle all other messages which are received by the process.

  It must return:
  
  - `{noreply, state}
  - `{noreply, state, timeout}
  - `{stop, reason, state}
  - `terminate(reason, state)` - called when the server is about to terminate, useful for cleaning up. It must return `:ok`
  - `code_change(old_vsn, state, extra)` - called when the application code is being upgraded live (hot code swapping).

  It must return:
  
  - `{ok, new_state}
  - `{error, reason}`
Name Registration  Both `start_link/3` and `start/3` support the GenServer to register a name on start via the :name option. Registered names are also automatically cleaned up on termination. The supported values are:

- an atom - the GenServer is registered locally with the given name using `Process.register/2`;
- {:global, term} - the GenServer is registered globally with the given term using the functions in the :global module;
- {:via, module, term} - the GenServer is registered with the given mechanism and name. The :via option expects a module name to control the registration mechanism alongside a name which can be any term;

For example, we could start and register our Stack server locally as follows:

```elixir
# Start the server and register it locally with name MyStack
{:ok, _} = GenServer.start_link(Stack, [:hello], name: MyStack)

# Now messages can be sent directly to MyStack
GenServer.call(MyStack, :pop) #=> :hello
```

Once the server is started, the remaining functions in this module (`call/3`, `cast/2`, and friends) will also accept an atom, or any :global or :via tuples. In general, the following formats are supported:

- a pid
- an atom if the server is locally registered
- {atom, node} if the server is locally registered at another node
- {:global, term} if the server is globally registered
- {:via, module, name} if the server is registered through an alternative registry

Client / Server APIs  Although in the example above we have used `GenServer.start_link/3` and friends to directly start and communicate with the server, most of the time we don’t call the GenServer functions directly. Instead, we wrap the calls in new functions representing the public API of the server.

Here is a better implementation of our Stack module:

```elixir
defmodule Stack do
  use GenServer

  # Client
  def start_link(default) do
    GenServer.start_link(__MODULE__, default)
  end

  def push(pid, item) do
    GenServer.cast(pid, {:push, item})
  end

  def pop(pid) do
    GenServer.call(pid, :pop)
  end

  # Server (callbacks)
  def handle_call(:pop, _from, [hit]) do
    {:reply, h, t}
  end
end
```

174 Chapter 3. API reference
def handle_call(request, from, state) do
  # Call the default implementation from GenServer
  super(request, from, state)
end

def handle_cast({:push, item}, state) do
  {:noreply, [item|state]}
end

def handle_cast(request, state) do
  super(request, state)
end
end

In practice, it is common to have both server and client functions in the same module. If the server and/or client implementations are growing complex, you may want to have them in different modules.

Learn more  If you wish to find out more about gen servers, Elixir getting started guides provide a tutorial-like introduction. The documentation and links in Erlang can also provide extra insight.

- [http://elixir-lang.org/getting_started/mix/1.html](http://elixir-lang.org/getting_started/mix/1.html)
- [http://www.erlang.org/doc/design_principles/gen_server_concepts.html](http://www.erlang.org/doc/design_principles/gen_server_concepts.html)

<table>
<thead>
<tr>
<th>abcast/3</th>
<th>Casts all servers locally registered as name at the specified nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/3</td>
<td>Makes a synchronous call to the server and waits for its reply</td>
</tr>
<tr>
<td>cast/2</td>
<td>Sends an asynchronous request to the server</td>
</tr>
<tr>
<td>multi_call/4</td>
<td>Calls all servers locally registered as name at the specified nodes</td>
</tr>
<tr>
<td>reply/2</td>
<td>Replies to a client</td>
</tr>
<tr>
<td>start/3</td>
<td>Starts a GenServer process without links (outside of a supervision tree)</td>
</tr>
<tr>
<td>start_link/3</td>
<td>Starts a GenServer process linked to the current process</td>
</tr>
</tbody>
</table>

Types

on_start

on_start :: {:ok, pid} | :ignore | {:error, {:already_started, pid} | term}

Return values of start* functions

name

ame :: atom | {:global, term} | {:via, module, term}

The GenServer name

options

options :: [:debug: debug, name: name, timeout: timeout, spawn_opt: Process.spawn_opt]

Options used by the start* functions

data

data :: [trace | :log | statistics | {:log_to_file, Path.t}]

debug options supported by the start* functions

server

server :: pid | name | {atom, node}
The server reference

Functions

**abcast\(\text{nodes} \ \backslash\land\backslash \text{nodes()}\), \text{name}, \text{request})(\text{function})**

Specs:

*abcast([\text{node}], \text{name} :: \text{atom}, \text{term}) :: :abcast*

Casts all servers locally registered as \text{name} at the specified nodes.

The function returns immediately and ignores nodes that do not exist, or where the server name does not exist.

See **multi_call/4** for more information.

**call\(\text{server}, \text{request}, \text{timeout} \ \backslash\land\backslash \text{5000})(\text{function})**

Specs:

*call(\text{server}, \text{term}, \text{timeout}) :: \text{term}*

Makes a synchronous call to the \text{server} and waits for its reply.

The client sends the given \text{request} to the server and waits until a reply arrives or a timeout occurs. \text{handle_call/3} will be called on the server to handle the request.

The server can be any of the values described in the Name Registration section of the module documentation.

**Timeouts**

The \text{timeout} is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom \text{:infinity} to wait indefinitely. The default value is 5000. If no reply is received within the specified time, the function call fails. If the caller catches the failure and continues running, and the server is just late with the reply, it may arrive at any time later into the caller’s message queue. The caller must in this case be prepared for this and discard any such garbage messages that are two element tuples with a reference as the first element.

**cast\(\text{server}, \text{request})(\text{function})**

Specs:

*cast(\text{server}, \text{term}) :: :ok*

Sends an asynchronous request to the \text{server}.

This function returns \text{:ok} immediately, regardless of whether the destination node or server does exists. \text{handle_cast/2} will be called on the server to handle the request.

**multi_call\(\text{nodes} \ \backslash\land\backslash \text{nodes()}\), \text{name}, \text{request}, \text{timeout} \ \backslash\land\backslash \text{:infinity})(\text{function})**

Specs:

*multi_call([\text{node}], \text{name} :: \text{atom}, \text{term}, \text{timeout}) :: [{\text{replies} :: [{\text{node}, \text{term}}]}, \text{bad_nodes} :: [\text{node}]\})*

Calls all servers locally registered as \text{name} at the specified \text{nodes}.

The \text{request} is first sent to every node and then we wait for the replies. This function returns a tuple containing the node and its reply as first element and all bad nodes as second element. The bad nodes is a list of nodes that either did not exist, or where a server with the given \text{name} did not exist or did not reply.

Nodes is a list of node names to which the request is sent. The default value is the list of all known nodes.

To avoid that late answers (after the timeout) pollute the caller’s message queue, a middleman process is used to do the actual calls. Late answers will then be discarded when they arrive to a terminated process.

**reply\(\text{client}, \text{reply})(\text{function})**

Specs:

*reply([{\text{pid}}, \text{reference}]), \text{term}) :: :ok*
Replies to a client.

This function can be used by a server to explicitly send a reply to a client that called \texttt{call/3} or \texttt{multi_call/4}. When the reply cannot be defined in the return value of \texttt{handle_call/3}.

The \texttt{client} must be the \texttt{from} argument (the second argument) received in \texttt{handle_call/3} callbacks. Reply is an arbitrary term which will be given back to the client as the return value of the call.

This function always returns \texttt{:ok}.

\begin{verbatim}
start(module, args, options \(\[\]\))(function)
  Specs:
  \texttt{\*start(module, any, options) :: on_start}

  Starts a \texttt{GenServer} process without links (outside of a supervision tree).
  See \texttt{start_link/3} for more information.
\end{verbatim}

\begin{verbatim}
start_link(module, args, options \(\[\]\))(function)
  Specs:
  \texttt{\*start_link(module, any, options) :: on_start}

  Starts a \texttt{GenServer} process linked to the current process.

  This is often used to start the \texttt{GenServer} as part of a supervision tree.

  Once the server is started, it calls the \texttt{init/1} function in the given \texttt{module} passing the given \texttt{args} to initialize it. To ensure a synchronized start-up procedure, this function does not return until \texttt{init/1} has returned.

  Options

  The \texttt{:name} option is used for name registration as described in the module documentation. If the option \texttt{:timeout} option is present, the server is allowed to spend the given milliseconds initializing or it will be terminated and the start function will return \texttt{(:error, :timeout)}.

  If the \texttt{:debug} option is present, the corresponding function in the \texttt{\'sys\' module <http://www.erlang.org/doc/man/sys.html>\'} will be invoked.

  If the \texttt{:spawn_opt} option is present, its value will be passed as options to the underlying process as in \texttt{Process.spawn/4}.

  Return values

  If the server is successfully created and initialized, the function returns \texttt{\{:ok, pid\}}, where \texttt{pid} is the \texttt{pid} of the server. If there already exists a process with the specified server name, the function returns \texttt{\{:error, \{already_started, pid\}\}} with the \texttt{pid} of that process.

  If the \texttt{init/1} callback fails with \texttt{reason}, the function returns \texttt{\{:error, reason\}}. Otherwise, if it returns \texttt{\{:stop, reason\}} or \texttt{\ignore\}, the process is terminated and the function returns \texttt{\{:error, reason\}} or \texttt{\ignore\}, respectively.
\end{verbatim}

\textbf{HashDict}

\textbf{Overview} \; A key-value store.

The \texttt{HashDict} is represented internally as a struct, therefore \texttt{\%HashDict\{\}} can be used whenever there is a need to match on any \texttt{HashDict}. Note though the struct fields are private and must not be accessed directly. Instead, use the functions on this or in the \texttt{Dict} module.

Implementation-wise, \texttt{HashDict} is implemented using tries, which grows in space as the number of keys grows, working well with both small and large set of keys. For more information about the functions and their APIs, please consult the \texttt{Dict} module.
Types

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete/2</td>
<td>Callback implementation of Dict.delete/2</td>
</tr>
<tr>
<td>drop/2</td>
<td>Callback implementation of Dict.drop/2</td>
</tr>
<tr>
<td>equal?/2</td>
<td>Callback implementation of Dict.equal?/2</td>
</tr>
<tr>
<td>fetch!/2</td>
<td>Callback implementation of Dict.fetch!/2</td>
</tr>
<tr>
<td>fetch/2</td>
<td>Callback implementation of Dict.fetch/2</td>
</tr>
<tr>
<td>get/3</td>
<td>Callback implementation of Dict.get/3</td>
</tr>
<tr>
<td>has_key?/2</td>
<td>Callback implementation of Dict.has_key?/2</td>
</tr>
<tr>
<td>keys/1</td>
<td>Callback implementation of Dict.keys/1</td>
</tr>
<tr>
<td>merge/3</td>
<td>Callback implementation of Dict.merge/3</td>
</tr>
<tr>
<td>put/3</td>
<td>Callback implementation of Dict.put/3</td>
</tr>
<tr>
<td>put_new/3</td>
<td>Callback implementation of Dict.put_new/3</td>
</tr>
<tr>
<td>size/1</td>
<td>Callback implementation of Dict.size/1</td>
</tr>
<tr>
<td>split/2</td>
<td>Callback implementation of Dict.split/2</td>
</tr>
<tr>
<td>take/2</td>
<td>Callback implementation of Dict.take/2</td>
</tr>
<tr>
<td>to_list/1</td>
<td>Callback implementation of Dict.to_list/1</td>
</tr>
<tr>
<td>update!/3</td>
<td>Callback implementation of Dict.update!/3</td>
</tr>
<tr>
<td>update/4</td>
<td>Callback implementation of Dict.update/4</td>
</tr>
<tr>
<td>values/1</td>
<td>Callback implementation of Dict.values/1</td>
</tr>
</tbody>
</table>

Functions

delete(dict, key) (function)
Callback implementation of Dict.delete/2.

drop(dict, keys) (function)
Callback implementation of Dict.drop/2.

equal?(dict1, dict2) (function)
Callback implementation of Dict.equal?/2.

fetch(hashdict, key) (function)
Callback implementation of Dict.fetch/2.

fetch!(dict, key) (function)
Callback implementation of Dict.fetch!/2.

get(dict, key, default \ nil) (function)
Callback implementation of Dict.get/3.

has_key?(dict, key) (function)
Callback implementation of Dict.has_key?/2.

keys(dict) (function)
Callback implementation of Dict.keys/1.

merge(dict1, dict2, fun \ fn _k, _v1, v2 -> v2 end) (function)
Callback implementation of Dict.merge/3.

new() (function)
Specs:
  *new :: Dict.t

Creates a new empty dict.
pop(dict, key, default \ nil)(function)
Callback implementation of Dict.pop/3.

put(hashdict, key, value)(function)
Callback implementation of Dict.put/3.

put_new(dict, key, value)(function)
Callback implementation of Dict.put_new/3.

size(hashdict)(function)
Callback implementation of Dict.size/1.

split(dict, keys)(function)
Callback implementation of Dict.split/2.

take(dict, keys)(function)
Callback implementation of Dict.take/2.

to_list(dict)(function)
Callback implementation of Dict.to_list/1.

update(dict, key, initial, fun)(function)
Callback implementation of Dict.update/4.

update!(dict, key, fun)(function)
Callback implementation of Dict.update!/3.

values(dict)(function)
Callback implementation of Dict.values/1.

**HashSet**

**Overview**  
A set store.

The **HashSet** is represented internally as a struct, therefore `%HashSet{}` can be used whenever there is a need to match on any **HashSet**. Note though the struct fields are private and must not be accessed directly. Instead, use the functions on this or in the **Set** module.

The **HashSet** is implemented using tries, which grows in space as the number of keys grows, working well with both small and large set of keys. For more information about the functions and their APIs, please consult the **Set** module.

<table>
<thead>
<tr>
<th>Function</th>
<th>Callback implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete/2</td>
<td>Set.delete/2</td>
</tr>
<tr>
<td>difference/2</td>
<td>Set.difference/2</td>
</tr>
<tr>
<td>disjoint?/2</td>
<td>Set.disjoint?/2</td>
</tr>
<tr>
<td>equal?/2</td>
<td>Set.equal?/2</td>
</tr>
<tr>
<td>intersection/2</td>
<td>Set.intersection/2</td>
</tr>
<tr>
<td>member?/2</td>
<td>Set.member?/2</td>
</tr>
<tr>
<td>new/0</td>
<td>Set.new/0</td>
</tr>
<tr>
<td>put/2</td>
<td>Set.put/2</td>
</tr>
<tr>
<td>size/1</td>
<td>Set.size/1</td>
</tr>
<tr>
<td>subset?/2</td>
<td>Set.subset?/2</td>
</tr>
<tr>
<td>to_list/1</td>
<td>Set.to_list/1</td>
</tr>
<tr>
<td>union/2</td>
<td>Set.union/2</td>
</tr>
</tbody>
</table>

**Types**

**t**

3.2. API Reference (v0.14.0-dev)
Functions

delete(set, term)(function)
Callback implementation of Set.delete/2.

difference(set1, set2)(function)
Callback implementation of Set.difference/2.

disjoint?(set1, set2)(function)
Callback implementation of Set.disjoint?/2.

equal?(set1, set2)(function)
Callback implementation of Set.equal?/2.

intersection(set1, set2)(function)
Callback implementation of Set.intersection/2.

member?(hashset, term)(function)
Callback implementation of Set.member?/2.

new()(function)
Specs:
• new :: Set.t
  Creates a new empty set.

put(hashset, term)(function)
Callback implementation of Set.put/2.

size(hashset)(function)
Callback implementation of Set.size/1.

subset?(set1, set2)(function)
Callback implementation of Set.subset?/2.

to_list(set)(function)
Callback implementation of Set.to_list/1.

union(set1, set2)(function)
Callback implementation of Set.union/2.

IO

Overview  Functions handling IO.

Many functions in this module expects an IO device as argument. An IO device must be a pid or an atom representing
a process. For convenience, Elixir provides :stdio and :stderr as shortcuts to Erlang’s :standard_io and :
standard_error.

The majority of the functions expect char data, i.e. strings or lists of characters and strings. In case another type is
given, it will do a conversion to string via the String.Chars protocol (as shown in typespecs).

The functions starting with bin* expects iodata as argument, i.e. binaries or lists of bytes and binaries.

IO devices  An IO device may be an atom or a pid. In case it is an atom, the atom must be the name of a registered
process. However, there are three exceptions for this rule:

• :standard_io - when the :standard_io atom is given, it is treated as a shortcut for Process.group_leader
• :stdio - is a shortcut for :standard_io
• :stderr - is a shortcut for :standard_error

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binread/2</td>
<td>Reads <code>count</code> bytes from the IO device or until the end of the line if :line is given. It returns:</td>
</tr>
<tr>
<td>binstream/2</td>
<td>Converts the IO device into a IO.Stream</td>
</tr>
<tr>
<td>binwrite/2</td>
<td>Writes the given argument to the given device as a binary, no unicode conversion happens</td>
</tr>
<tr>
<td>chardata_to_string/1</td>
<td>Converts chardata (a list of integers representing codepoints, lists and strings) into a string</td>
</tr>
<tr>
<td>getn/2</td>
<td>Gets a number of bytes from the io device. If the io device is a unicode device, <code>count</code> implies the number of unicode codepoints to be retrieved. Otherwise, <code>count</code> is the number of raw bytes to be retrieved. It returns:</td>
</tr>
<tr>
<td>getn/3</td>
<td>Gets a number of bytes from the io device. If the io device is a unicode device, <code>count</code> implies the number of unicode codepoints to be retrieved. Otherwise, <code>count</code> is the number of raw bytes to be retrieved.</td>
</tr>
<tr>
<td>gets/2</td>
<td>Reads a line from the IO device. It returns:</td>
</tr>
<tr>
<td>inspect/2</td>
<td>Inspects and writes the given argument to the device</td>
</tr>
<tr>
<td>inspect/3</td>
<td>Inspects the item with options using the given device</td>
</tr>
<tr>
<td>iodata_length/1</td>
<td>Returns the size of an iodata</td>
</tr>
<tr>
<td>iodata_to_binary/1</td>
<td>Converts iodata (a list of integers representing bytes, lists and binaries) into a binary</td>
</tr>
<tr>
<td>puts/2</td>
<td>Writes the argument to the device, similar to write/2, but adds a newline at the end. The argument is expected to be a chardata</td>
</tr>
<tr>
<td>read/2</td>
<td>Reads <code>count</code> characters from the IO device or until the end of the line if :line is given. It returns:</td>
</tr>
<tr>
<td>stream/2</td>
<td>Converts the io device into a IO.Stream</td>
</tr>
<tr>
<td>write/2</td>
<td>Writes the given argument to the given device</td>
</tr>
</tbody>
</table>

**Types**

device :: atom | pid

nodata :: {:error, term} | :eof

chardata :: :unicode.chardata

**Functions**

binread\(device \ \ \ :\text{erlang.group_leader}(), \ \ chars\_or\_line\)\(\)\(function\)

Specs:

- \(\text{binread}(device, :line \ | \ \nonneginteger) :: \text{iodata} \ | \ \text{nodata}\)

Reads `count` bytes from the IO device or until the end of the line if :line is given. It returns:

- `data` - The input characters.
- `:eof` - End of file was encountered.
- `{:error, reason}` - Other (rare) error condition, for instance `{:error, :estale}` if reading from an NFS file system.

binstream\(device, \ \ line\_or\_bytes\)\(\)\(function\)

Specs:

- \(\text{binstream}(device, :line \ | \ \posinteger) :: \text{Enumerable}\_t\)

Converts the IO device into a IO.Stream.

An IO.Stream implements both Enumerable and Collectable, allowing it to be used for both read and write.
The device is iterated line by line or by a number of bytes. This reads the IO device as a raw binary.

Note that an IO stream has side effects and every time you go over the stream you may get different results.

```elixir
device \ :erlang.group_leader(), item\(function\)
```

**Specs:**

- `binwrite(device, iodata) :: :ok | {:error, term}

Writes the given argument to the given device as a binary, no unicode conversion happens.

Check `write/2` for more information.

```elixir
chardata_to_string(string)\(function\)
```

**Specs:**

- `chardata_to_string(chardata) :: String.t | no_return`

Converts chardata (a list of integers representing codepoints, lists and strings) into a string.

In case the conversion fails, it raises a `UnicodeConversionError`. If a string is given, returns the string itself.

**Examples**

```elixir
iex> IO.chardata_to_string([0x00E6, 0x00DF])
"æß"

iex> IO.chardata_to_string([0x0061, "bc"])
"abc"
```

```elixir
getn(prompt, count \ 1)\(function\)
```

**Specs:**

- `getn(device, chardata|String.Chars.t) :: chardata|nodata
- `getn(chardata|String.Chars.t, pos_integer) :: chardata|nodata`

Gets a number of bytes from the IO device. If the IO device is a unicode device, `count` implies the number of unicode codepoints to be retrieved. Otherwise, `count` is the number of raw bytes to be retrieved. It returns:

- `data` - The input characters.
- `:eof` - End of file was encountered.
- `{:error, reason}` - Other (rare) error condition, for instance `{:error, :estale}` if reading from an NFS file system.

```elixir
getn(device, prompt, count)\(function\)
```

**Specs:**

- `getn(device, chardata|String.Chars.t, pos_integer) :: chardata|nodata`

Gets a number of bytes from the IO device. If the IO device is a unicode device, `count` implies the number of unicode codepoints to be retrieved. Otherwise, `count` is the number of raw bytes to be retrieved.

```elixir
gets(device \ :erlang.group_leader(), prompt)\(function\)
```

**Specs:**

- `gets(device, chardata|String.Chars.t) :: chardata|nodata

Reads a line from the IO device. It returns:

- `data` - The characters in the line terminated by a LF (or end of file).
- `:eof` - End of file was encountered.
•{:error, reason} - Other (rare) error condition, for instance {:error, :estale} if reading from an NFS file system.

`inspect(item, opts \ ([]))` (function)
Specs:

•inspect(term, Keyword.t) :: term

Inspect and writes the given argument to the device.

It sets by default pretty printing to true and returns the item itself.

Note this function does not use the IO device width because some IO devices does not implement the appropriate functions. Setting the width must be done explicitly by passing the :width option.

Examples

```iex
IO.inspect Process.list
```

`inspect(device, item, opts)` (function)
Specs:

•inspect(device, term, Keyword.t) :: term

Inspect the item with options using the given device.

`iodata_length(item)` (function)
Specs:

•iodata_length(iodata) :: non_neg_integer

Returns the size of an iodata.

Inlined by the compiler.

Examples

```iex
iex> IO.iodata_length([1, 2<<3, 4>>])
4
```

`iodata_to_binary(item)` (function)
Specs:

•iodata_to_binary(iodata) :: binary

Converts iodata (a list of integers representing bytes, lists and binaries) into a binary.

Notice that this function treats lists of integers as raw bytes and does not perform any kind of encoding conversion. If you want to convert from a char list to a string (UTF-8 encoded), please use `chardata_to_string/1` instead.

If this function receives a binary, the same binary is returned.

Inlined by the compiler.

Examples

```iex
iex> bin1 = <<1, 2, 3>>
iex> bin2 = <<4, 5>>
iex> bin3 = <<6>>
iex> IO.iodata_to_binary([bin1, 1, [2, 3, bin2], 4|bin3])
<<1,2,3,1,2,3,4,5,6>>

iex> bin = <<1, 2, 3>>
iex> IO.iodata_to_binary(bin)
<<1,2,3>>
```
puts(device \ :erlang.group_leader(), item)(function)
Specs:
  •puts(device, chardata | String.Chars.t) :: ok

Writes the argument to the device, similar to write/2, but adds a newline at the end. The argument is expected to be a chardata.

read(device \ :erlang.group_leader(), chars_or_line)(function)
Specs:
  •read(device, :line | non_neg_integer) :: chardata | nodata

Reads count characters from the IO device or until the end of the line if :line is given. It returns:
  • data - The input characters.
  • :eof - End of file was encountered.
  • {:error, reason} - Other (rare) error condition, for instance {:error, :estale} if reading from an NFS file system.

stream(device, line_or_codepoints)(function)
Specs:
  • stream(device, :line | pos_integer) :: Enumerable.t

Converts the io device into a IO.Stream.

An IO.Stream implements both Enumerable and Collectable, allowing it to be used for both read and write.

The device is iterated line by line if :line is given or by a given number of codepoints.

This reads the IO as utf-8. Check out IO.binstream/2 to handle the IO as a raw binary.

Note that an IO stream has side effects and every time you go over the stream you may get different results.

Examples
Here is an example on how we mimic an echo server from the command line:

Enum.each IO.stream(:stdio, :line), &IO.write(&1)

write(device \ :erlang.group_leader(), item)(function)
Specs:
  • write(device, chardata | String.Chars.t) :: ok

Writes the given argument to the given device.

By default the device is the standard output. It returns :ok if it succeeds.

Examples

IO.write "sample"
#=> "Sample"

IO.write :stderr, "error"
#=> "error"

IO.ANSI

Overview  Functionality to render ANSI escape sequences (http://en.wikipedia.org/wiki/ANSI_escape_code) — characters embedded in text used to control formatting, color, and other output options on video text terminals.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>black/0</td>
<td>Sets foreground color to black</td>
</tr>
<tr>
<td>black_background/0</td>
<td>Sets background color to black</td>
</tr>
<tr>
<td>blink_off/0</td>
<td>Blink: off</td>
</tr>
<tr>
<td>blink_rapid/0</td>
<td>Blink: Rapid. MS-DOS ANSI.SYS; 150 per minute or more; not widely supported</td>
</tr>
<tr>
<td>blink_slow/0</td>
<td>Blink: Slow. Less than 150 per minute</td>
</tr>
<tr>
<td>blue/0</td>
<td>Sets foreground color to blue</td>
</tr>
<tr>
<td>blue_background/0</td>
<td>Sets background color to blue</td>
</tr>
<tr>
<td>bright/0</td>
<td>Bright (increased intensity) or Bold</td>
</tr>
<tr>
<td>clear/0</td>
<td>Clear screen</td>
</tr>
<tr>
<td>conceal/0</td>
<td>Conceal. Not widely supported</td>
</tr>
<tr>
<td>crossed_out/0</td>
<td>Crossed-out. Characters legible, but marked for deletion. Not widely supported</td>
</tr>
<tr>
<td>cyan/0</td>
<td>Sets foreground color to cyan</td>
</tr>
<tr>
<td>cyan_background/0</td>
<td>Sets background color to cyan</td>
</tr>
<tr>
<td>default_background/0</td>
<td>Default background color</td>
</tr>
<tr>
<td>default_color/0</td>
<td>Default text color</td>
</tr>
<tr>
<td>encircled/0</td>
<td>Encircled</td>
</tr>
<tr>
<td>escape/2</td>
<td>Escapes a string by converting named ANSI sequences into actual ANSI codes</td>
</tr>
<tr>
<td>escape_fragment/2</td>
<td>Escapes a string by converting named ANSI sequences into actual ANSI codes</td>
</tr>
<tr>
<td>faint/0</td>
<td>Faint (decreased intensity), not widely supported</td>
</tr>
<tr>
<td>font_1/0</td>
<td>Sets alternative font 1</td>
</tr>
<tr>
<td>font_2/0</td>
<td>Sets alternative font 2</td>
</tr>
<tr>
<td>font_3/0</td>
<td>Sets alternative font 3</td>
</tr>
<tr>
<td>font_4/0</td>
<td>Sets alternative font 4</td>
</tr>
<tr>
<td>font_5/0</td>
<td>Sets alternative font 5</td>
</tr>
<tr>
<td>font_6/0</td>
<td>Sets alternative font 6</td>
</tr>
<tr>
<td>font_7/0</td>
<td>Sets alternative font 7</td>
</tr>
<tr>
<td>font_8/0</td>
<td>Sets alternative font 8</td>
</tr>
<tr>
<td>font_9/0</td>
<td>Sets alternative font 9</td>
</tr>
<tr>
<td>framed/0</td>
<td>Framed</td>
</tr>
<tr>
<td>green/0</td>
<td>Sets foreground color to green</td>
</tr>
<tr>
<td>green_background/0</td>
<td>Sets background color to green</td>
</tr>
<tr>
<td>home/0</td>
<td>Send cursor home</td>
</tr>
<tr>
<td>inverse/0</td>
<td>Image: Negative. Swap foreground and background</td>
</tr>
<tr>
<td>italic/0</td>
<td>Italic: on. Not widely supported. Sometimes treated as inverse</td>
</tr>
<tr>
<td>magenta/0</td>
<td>Sets foreground color to magenta</td>
</tr>
<tr>
<td>magenta_background/0</td>
<td>Sets background color to magenta</td>
</tr>
<tr>
<td>no_underline/0</td>
<td>Underline: None</td>
</tr>
<tr>
<td>normal/0</td>
<td>Normal color or intensity</td>
</tr>
<tr>
<td>not_framed_encircled/0</td>
<td>Not framed or encircled</td>
</tr>
<tr>
<td>not_italic/0</td>
<td>Not italic</td>
</tr>
<tr>
<td>not_overlined/0</td>
<td>Not overlined</td>
</tr>
<tr>
<td>overlined/0</td>
<td>Overlined</td>
</tr>
<tr>
<td>primary_font/0</td>
<td>Sets primary (default) font</td>
</tr>
<tr>
<td>red/0</td>
<td>Sets foreground color to red</td>
</tr>
<tr>
<td>red_background/0</td>
<td>Sets background color to red</td>
</tr>
<tr>
<td>reset/0</td>
<td>Resets all attributes</td>
</tr>
<tr>
<td>reverse/0</td>
<td>Image: Negative. Swap foreground and background</td>
</tr>
<tr>
<td>terminal?/1</td>
<td>Checks whether the default I/O device is a terminal or a file</td>
</tr>
<tr>
<td>underline/0</td>
<td>Underline: Single</td>
</tr>
<tr>
<td>white/0</td>
<td>Sets foreground color to white</td>
</tr>
<tr>
<td>white_background/0</td>
<td>Sets background color to white</td>
</tr>
<tr>
<td>yellow/0</td>
<td>Sets foreground color to yellow</td>
</tr>
</tbody>
</table>

Continued on next page
Summary

**Functions**

- **black() (function)**
  - Sets foreground color to black

- **black_background() (function)**
  - Sets background color to black

- **blink_off() (function)**
  - Blink: off

- **blink_rapid() (function)**
  - Blink: Rapid. MS-DOS ANSI.SYS; 150 per minute or more; not widely supported

- **blink_slow() (function)**
  - Blink: Slow. Less than 150 per minute

- **blue() (function)**
  - Sets foreground color to blue

- **blue_background() (function)**
  - Sets background color to blue

- **bright() (function)**
  - Bright (increased intensity) or Bold

- **clear() (function)**
  - Clear screen

- **conceal() (function)**
  - Conceal. Not widely supported

- **crossed_out() (function)**
  - Crossed-out. Characters legible, but marked for deletion. Not widely supported.

- **cyan() (function)**
  - Sets foreground color to cyan

- **cyan_background() (function)**
  - Sets background color to cyan

- **default_background() (function)**
  - Default background color

- **default_color() (function)**
  - Default text color

- **encircled() (function)**
  - Encircled

- **escape(string, emit \ terminal?()) (function)**
  - Specs:
    - `*escape(String.t, emit :: boolean) :: String.t`
  - Escapes a string by converting named ANSI sequences into actual ANSI codes.
  - The format for referring to sequences is `%{red}` and `%{red, bright}` (for multiple sequences).
It will also append a `{reset}` to the string. If you don’t want this behaviour, use `escape_fragment/2`. An optional boolean parameter can be passed to enable or disable emitting actual ANSI codes. When `false`, no ANSI codes will emitted. By default, standard output will be checked if it is a terminal capable of handling these sequences (using `terminal?/1` function).

Examples

```elixir
iex> IO.ANSI.escape("Hello %{red,bright,green}yes", true)
"Hello \e[31m\e[1m\e[32myes\e[0m"
```

**escape_fragment**(string, emit \ \ terminal?())

Escapes a string by converting named ANSI sequences into actual ANSI codes.

The format for referring to sequences is `{red}` and `{red, bright}` (for multiple sequences).

An optional boolean parameter can be passed to enable or disable emitting actual ANSI codes. When `false`, no ANSI codes will emitted. By default, standard output will be checked if it is a terminal capable of handling these sequences (using `terminal?/1` function)

Examples

```elixir
iex> IO.ANSI.escape_fragment("Hello %{red,bright,green}yes", true)
"Hello \e[31m\e[1m\e[32myes"

iex> IO.ANSI.escape_fragment("%{reset}bye", true)
"\e[0mbye"
```

**faint()**

Faint (decreased intensity), not widely supported

**font_1()**

Sets alternative font 1

**font_2()**

Sets alternative font 2

**font_3()**

Sets alternative font 3

**font_4()**

Sets alternative font 4

**font_5()**

Sets alternative font 5

**font_6()**

Sets alternative font 6

**font_7()**

Sets alternative font 7

**font_8()**

Sets alternative font 8

**font_9()**

Sets alternative font 9

**framed()**

Framed
green() (function)
    Sets foreground color to green

green_background() (function)
    Sets background color to green

home() (function)
    Send cursor home

inverse() (function)
    Image: Negative. Swap foreground and background

italic() (function)
    Italic: on. Not widely supported. Sometimes treated as inverse.

magenta() (function)
    Sets foreground color to magenta

magenta_background() (function)
    Sets background color to magenta

no_underline() (function)
    Underline: None

normal() (function)
    Normal color or intensity

not_framed_encircled() (function)
    Not framed or encircled

not_italic() (function)
    Not italic

not_overlined() (function)
    Not overlined

overlined() (function)
    Overlined

primary_font() (function)
    Sets primary (default) font

red() (function)
    Sets foreground color to red

red_background() (function)
    Sets background color to red

reset() (function)
    Resets all attributes

reverse() (function)
    Image: Negative. Swap foreground and background

terminal? (device \ :erlang.group_leader()) (function)

    Specs:
    ^terminal?(io.device) :: boolean

    Checks whether the default I/O device is a terminal or a file.

    Used to identify whether printing ANSI escape sequences will likely be displayed as intended. This is checked
    by sending a message to the group leader. In case the group leader does not support the message, it will likely
    lead to a timeout (and a slow down on execution time).
underline() (function)
Underline: Single

white() (function)
Sets foreground color to white

white_background() (function)
Sets background color to white

yellow() (function)
Sets foreground color to yellow

yellow_background() (function)
Sets background color to yellow

**IO.Stream**

**Overview** Defines a **IO.Stream** struct returned by **IO.stream/2** and **IO.binstream/2**.

The following fields are public:

- **device** - the IO device
- **raw** - a boolean indicating if bin functions should be used
- **line_or_bytes** - if reading should read lines or a given amount of bytes

**Types**

```elixir
t :: %IO.Stream{device: term, raw: term, line_or_bytes: term}
```

**Inspect.Algebra**

**Overview** A set of functions for creating and manipulating algebra documents, as described in “Strictly Pretty” (2000) by Christian Lindig.

An algebra document is represented by an **Inspect.Algebra** node or a regular string.

```
iex> Inspect.Algebra.empty
:doc_nil

iex> "foo"
"foo"
```

With the functions in this module, we can concatenate different elements together and render them:

```
iex> doc = Inspect.Algebra.concat(Inspect.Algebra.empty, "foo")
iex> Inspect.Algebra.pretty(doc, 80)
"foo"
```

The functions **nest/2**, **space/2** and **line/2** help you put the document together into a rigid structure. However, the document algebra gets interesting when using functions like **break/2**, which converts the given string into a line break depending on how much space there is to print. Let’s glue two docs together with a break and then render it:

```
iex> doc = Inspect.Algebra.glue("a", " ", "b")
iex> Inspect.Algebra.pretty(doc, 80)
"a b"
```
Notice the break was represented as is, because we haven’t reached a line limit. Once we do, it is replaced by a newline:

```iex> doc = Inspect.Algebra.glue(String.duplicate("a", 20), " ", "b")
iex> Inspect.Algebra.pretty(doc, 10)
"aaaaaaaaaaaaaaaaaaaa
b"
```

Finally, this module also contains Elixir related functions, a bit tied to Elixir formatting, namely `surround/3` and `surround_many/5`.

**Implementation details** The original Haskell implementation of the algorithm by Wadler relies on lazy evaluation to unfold document groups on two alternatives: `:flat` (breaks as spaces) and `:break` (breaks as newlines). Implementing the same logic in a strict language such as Elixir leads to an exponential growth of possible documents, unless document groups are encoded explicitly as `:flat` or `:break`. Those groups are then reduced to a simple document, where the layout is already decided, per Lindig.

This implementation slightly changes the semantic of Lindig’s algorithm to allow elements that belong to the same group to be printed together in the same line, even if they do not fit the line fully. This was achieved by changing `:break` to mean a possible break and `:flat` to force a flat structure. Then deciding if a break works as a newline is just a matter of checking if we have enough space until the next break that is not inside a group (which is still flat).

Custom pretty printers can be implemented using the documents returned by this module and by providing their own rendering functions.

<table>
<thead>
<tr>
<th><code>break/0</code></th>
<th>Document entity representing a break. This break can be rendered as a linebreak or as spaces, depending on the <code>mode</code> of the chosen layout or the provided separator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>break/1</code></td>
<td>Concatenates a list of documents</td>
</tr>
<tr>
<td><code>concat/1</code></td>
<td>Concatenates two document entities. Takes two arguments: left doc and right doc. Returns a DocCons doc</td>
</tr>
<tr>
<td><code>concat/2</code></td>
<td>Returns <code>:doc_nil</code> which is a document entity used to represent nothingness. Takes no arguments</td>
</tr>
<tr>
<td><code>empty/0</code></td>
<td>Folds a list of document entities into a document entity using a function that is passed as the first argument</td>
</tr>
<tr>
<td><code>folddoc/2</code></td>
<td>Inserts a break between two docs. See <code>break/1</code> for more info</td>
</tr>
<tr>
<td><code>glue/2</code></td>
<td>Inserts a break, passed as the second argument, between two docs, the first and the third arguments</td>
</tr>
<tr>
<td><code>glue/3</code></td>
<td>Returns a group containing the specified document</td>
</tr>
<tr>
<td><code>group/1</code></td>
<td>Inserts a mandatory linebreak between two document entities</td>
</tr>
<tr>
<td><code>line/2</code></td>
<td>Nests document entity x positions deep. Nesting will be appended to the line breaks</td>
</tr>
<tr>
<td><code>nest/2</code></td>
<td>The pretty printing function</td>
</tr>
<tr>
<td><code>pretty/2</code></td>
<td>Inserts a mandatory single space between two document entities</td>
</tr>
<tr>
<td><code>space/2</code></td>
<td>Surrounds a document with characters</td>
</tr>
<tr>
<td><code>surround/3</code></td>
<td>Maps and glues a collection of items together using the given separator and surrounds them. A limit can be passed which, once reached, stops gluing and outputs ”...” instead</td>
</tr>
<tr>
<td><code>surround_many/5</code></td>
<td>Converts an Elixir structure to an algebra document according to the inspect protocol</td>
</tr>
</tbody>
</table>

**Summary**

**Types**

```
t :: :doc_nil | :doc_line | doc_cons | doc_nest | doc_break | doc_group | binary
```

**Functions**

`break/0` (function)

Specs:
**break** (s)(function)

Specs:

- `break(binary) :: doc_break`

Document entity representing a break. This break can be rendered as a linebreak or as spaces, depending on the mode of the chosen layout or the provided separator.

**Examples**

Let's glue two docs together with a break and then render it:

```iex
iex> doc = Inspect.Algebra.glue("a", " ", "b")
iex> Inspect.Algebra.pretty(doc, 80)
"a b"
```

Notice the break was represented as is, because we haven’t reached a line limit. Once we do, it is replaced by a newline:

```iex
iex> doc = Inspect.Algebra.glue(String.duplicate("a", 20), " ", "b")
iex> Inspect.Algebra.pretty(doc, 10)
"aaaaaaaaaaaaaaaaa\nb"
```

**concat (docs)(function)**

Specs:

- `concat([]) :: doc_cons`

Concatenates a list of documents.

**concat (x, y)(function)**

Specs:

- `concat(t, t) :: doc_cons`

Concatenates two document entities. Takes two arguments: left doc and right doc. Returns a DocCons doc

**Examples**

```iex
iex> doc = Inspect.Algebra.concat "Tasteless", "Artosis"
iex> Inspect.Algebra.pretty(doc, 80)
"TastelessArtosis"
```

**empty () (function)**

Specs:

- `empty :: :doc_nil`

Returns :doc_nil which is a document entity used to represent nothingness. Takes no arguments.

**Examples**

```iex
iex> Inspect.Algebra.empty
:doc_nil
```

**folddoc (list1, f)(function)**

Specs:

- `folddoc([t], (t, t -> t)) :: t`

Folds a list of document entities into a document entity using a function that is passed as the first argument.

**Examples**
glue(x, y)(function)

Specs:

•glue(t, t) :: doc_cons

Inserts a break between two docs. See break/1 for more info.

glue(x, g, y)(function)

Specs:

•glue(t, binary, t) :: doc_cons

Inserts a break, passed as the second argument, between two docs, the first and the third arguments.

group(d)(function)

Specs:

•group(t) :: doc_group

Returns a group containing the specified document.

Examples

iex> doc = Inspect.Algebra.group{
...> Inspect.Algebra.concat{
...> Inspect.Algebra.group{
...> Inspect.Algebra.concat{
...> "Hello,",
...> Inspect.Algebra.concat{
...> Inspect.Algebra.break,
...> "A"
...> }
...> },
...> Inspect.Algebra.concat{
...> Inspect.Algebra.break,
...> "B"
...> }
...> }
...> Inspect.Algebra.pretty(doc, 80)
"Hello, A B"

iex> Inspect.Algebra.pretty(doc, 6)
"Hello,\nA B"

line(x, y)(function)

Specs:

•line(t, t) :: doc_cons

Inserts a mandatory linebreak between two document entities.

Examples

iex> doc = Inspect.Algebra.line "Hughes", "Wadler"
iex> Inspect.Algebra.pretty(doc, 80)
"Hughes\nWadler"
nest(x, i)(function)
   Specs:
   •nest(t, non_neg_integer) :: doc_nest
   Nesting entity x positions deep. Nesting will be appended to the line breaks.

Examples
iex> Inspect.Algebra.pretty(doc, 80)
" 6"

pretty(d, w)(function)
   Specs:
   •pretty(t, non_neg_integer | :infinity) :: binary
   The pretty printing function.
   Takes the maximum width and a document to print as its arguments and returns the string representation of the best layout for the document to fit in the given width.

space(x, y)(function)
   Specs:
   •space(t, t) :: doc_cons
   Inserts a mandatory single space between two document entities.

Examples
iex> doc = Inspect.Algebra.space "Hughes", "Wadler"
iex> Inspect.Algebra.pretty(doc, 80)
"Hughes Wadler"

surround(left, doc, right)(function)
   Specs:
   •surround(binary, t, binary) :: t
   Surrounds a document with characters.
   Puts the document between left and right enclosing and nesting it. The document is marked as a group, to show the maximum as possible concisely together.

Examples
iex> doc = Inspect.Algebra.surround ":[", Inspect.Algebra.glue("a", "b"), "]"
iex> Inspect.Algebra.pretty(doc, 3)
"[a
 b]"

surround_many(left, docs, right, limit, fun, separator \ ", ") (function)
   Specs:
   •surround_many(binary, [any], binary, integer | :infinity, (term -> t), binary) :: t
   Maps and glues a collection of items together using the given separator and surrounds them. A limit can be passed which, once reached, stops gluing and outputs "..." instead.

Examples
iex> doc = Inspect.Algebra.surround_many("[", Enum.to_list(1..5), "]", :infinity, &Integer.to_string/1)
iex> Inspect.Algebra.pretty(doc, 5)
"[1,\n 2,\n 3,\n 4,\n 5]"

3.2. API Reference (v0.14.0-dev)
Elixir Documentation, Release

```elixir
iex> doc = Inspect.Algebra.surround_many("[", Enum.to_list(1..5), "]", 3, &Integer.to_string(&1))
iex> Inspect.Algebra.pretty(doc, 20)
"[1, 2, 3, ...]"

iex> doc = Inspect.Algebra.surround_many("[", Enum.to_list(1..5), "]", 3, &Integer.to_string(&1), ")
"[1! 2! 3! ...]"
```

to_doc(map, opts) (function)

Specs:

- to_doc(any, Inspect.Opts.t):: t

Converts an Elixir structure to an algebra document according to the inspect protocol.

Inspect.Opts

Overview  Defines the Inspect.Opts used by the Inspect protocol.

The following fields are available:

- structs - when false, structs are not formatted by the inspect protocol, they are instead printed as maps, defaults to true;
- binaries - when :as_strings all binaries will be printed as strings, non-printable bytes will be escaped; when :as_binaries all binaries will be printed in bit syntax; when the default :infer, the binary will be printed as a string if it is printable, otherwise in bit syntax;
- char_lists - when :as_char_lists all lists will be printed as char lists, non-printable elements will be escaped; when :as_lists all lists will be printed as lists; when the default :infer, the list will be printed as a char list if it is printable, otherwise as list;
- limit - limits the number of items that are printed for tuples, bitstrings, and lists, does not apply to strings nor char lists, defaults to 50;
- pretty - if set to true enables pretty printing, defaults to false;
- width - defaults to the 80 characters;

Types  


Integer

Overview  Functions for working with integers.
even?/1
odd?/1
determines if an integer is even
determines if an integer is odd

parse/1
converts a binary to an integer

Summary
to_char_list/1
returns a char list which corresponds to the text representation of the given integer

to_char_list/2
returns a char list which corresponds to the text representation of the given integer in the given case

to_string/1
returns a binary which corresponds to the text representation of some_integer

to_string/2
returns a binary which corresponds to the text representation of some_integer in base base

Functions

parse (bin) (function)

Specs:

•parse(binary) :: {integer, binary} | :error

Converts a binary to an integer.
If successful, returns a tuple of the form {integer, remainder_of_binary}. Otherwise :error.

Examples

iex> Integer.parse("34")
{34,""}

iex> Integer.parse("34.5")
{34,".5"}

iex> Integer.parse("three")
:error

to_char_list (number) (function)

Specs:

•to_char_list(integer) :: []

Returns a char list which corresponds to the text representation of the given integer.
Inlined by the compiler.

Examples

iex> Integer.to_char_list(7)
'7'

to_char_list (number, base) (function)

Specs:

•to_char_list(integer, pos_integer) :: []

Returns a char list which corresponds to the text representation of the given integer in the given case.
Inlined by the compiler.

Examples

iex> Integer.to_char_list(1023, 16)
'3FF'

to_string (some_integer) (function)

Specs:

•to_string(integer) :: String.t
Returns a binary which corresponds to the text representation of `some_integer`.
Inlined by the compiler.

**Examples**
```
 iex> Integer.to_string(123)  
 "123"
```

### to_string(some_integer, base)(function)

Specs:
```
•to_string(integer, pos_integer) :: String.t
```

Returns a binary which corresponds to the text representation of `some_integer` in base `base`.
Inlined by the compiler.

**Examples**
```
 iex> Integer.to_string(100, 16)  
 "64"
```

#### Macros

**even? (n) (macro)**

Determines if an integer is even.

Returns `true` if `n` is an even number, otherwise `false`. Implemented as a macro so it is allowed in guard clauses.

**odd? (n) (macro)**

Determines if an integer is odd.

Returns `true` if `n` is an odd number, otherwise `false`. Implemented as a macro so it is allowed in guard clauses.

**Kernel**

**Overview**  _Kernel_ provides the default macros and functions Elixir imports into your environment. These macros and functions can be skipped or cherry-picked via the `import` macro. For instance, if you want to tell Elixir not to import the `if` macro, you can do:
```
import Kernel, except: [if: 2]
```

Elixir also has special forms that are always imported and cannot be skipped. These are described in _Kernel.SpecialForms_.

Some of the functions described in this module are inlined by the Elixir compiler into their Erlang counterparts in the `_erlang` module. Those functions are called BIFs (builtin internal functions) in Erlang-land and they exhibit interesting properties, as some of them are allowed in guards and others are used for compiler optimizations.

Most of the inlined functions can be seen in effect when capturing the function:
```
iex> &Kernel.is_atom/1  
&:erlang.is_atom/1
```

Those functions will be explicitly marked in their docs as “inlined by the compiler”.

Continued on next page
Elixir Documentation, Release

Table 3.5 – continued from previous page

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!/1</td>
<td>Receives any argument and returns true if it is false or nil. Returns false otherwise. Not allowed in guard clauses.</td>
</tr>
<tr>
<td>!=/2</td>
<td>Returns true if the two items are not equal</td>
</tr>
<tr>
<td>!==/2</td>
<td>Returns true if the two items do not match</td>
</tr>
<tr>
<td>&amp;&amp;/2</td>
<td>Provides a short-circuit operator that evaluates and returns the second expression only if the first one is not nil nor false.</td>
</tr>
<tr>
<td>*/2</td>
<td>Arithmetic multiplication</td>
</tr>
<tr>
<td>++/2</td>
<td>Concatenates two lists</td>
</tr>
<tr>
<td>+/1</td>
<td>Arithmetic unary plus</td>
</tr>
<tr>
<td>+/2</td>
<td>Arithmetic plus</td>
</tr>
<tr>
<td>--/2</td>
<td>Removes the first occurrence of an item on the left for each item on the right</td>
</tr>
<tr>
<td>-/1</td>
<td>Arithmetic unary minus</td>
</tr>
<tr>
<td>-/2</td>
<td>Arithmetic minus</td>
</tr>
<tr>
<td>./.2</td>
<td>Returns a range with the specified start and end.Includes both ends</td>
</tr>
<tr>
<td>//2</td>
<td>Arithmetic division</td>
</tr>
<tr>
<td>&lt;/2</td>
<td>Returns true if left is less than right</td>
</tr>
<tr>
<td>&lt;=/2</td>
<td>Returns true if left is less than or equal to right</td>
</tr>
<tr>
<td>=&gt;/2</td>
<td>Concatenates two binaries</td>
</tr>
<tr>
<td>==/2</td>
<td>Returns true if the two items are equal</td>
</tr>
<tr>
<td>===/2</td>
<td>Returns true if the two items are match</td>
</tr>
<tr>
<td>=~/2</td>
<td>Matches the term on the left against the regular expression or string on the right. Returns true if left matches right (if it's a regular expression) or contains right (if it's a string).</td>
</tr>
<tr>
<td>&gt;/2</td>
<td>Returns true if left is more than right</td>
</tr>
<tr>
<td>&gt;=/2</td>
<td>Returns true if left is more than or equal to right</td>
</tr>
<tr>
<td>@/1</td>
<td>Read and write attributes of the current module</td>
</tr>
<tr>
<td>abs/1</td>
<td>Returns an integer or float which is the arithmetical absolute value of number</td>
</tr>
<tr>
<td>alias!/1</td>
<td>When used inside quoting, marks that the alias should not be hygienezed. This means the alias will be expanded even if the alias is redefined.</td>
</tr>
<tr>
<td>and/2</td>
<td>Boolean and. Requires only the first argument to be a boolean since it short-circuits</td>
</tr>
<tr>
<td>apply/2</td>
<td>Invokes the given fun with the array of arguments args</td>
</tr>
<tr>
<td>apply/3</td>
<td>Invokes the given fun from module with the array of arguments args</td>
</tr>
<tr>
<td>binary_part/3</td>
<td>Extracts the part of the binary starting at start with length length. Binaries are zero-indexed</td>
</tr>
<tr>
<td>binding/0</td>
<td>Returns the binding as a keyword list where the variable name is the key and the variable value is the value.</td>
</tr>
<tr>
<td>binding/1</td>
<td>Receives a list of atoms at compilation time and returns the binding of the given variables as a keyword list.</td>
</tr>
<tr>
<td>binding/2</td>
<td>Receives a list of atoms at compilation time and returns the binding of the given variables in the given context as a keyword list.</td>
</tr>
<tr>
<td>bit_size/1</td>
<td>Returns an integer which is the size in bits of bitstring</td>
</tr>
<tr>
<td>byte_size/1</td>
<td>Returns the number of bytes needed to contain bitstring</td>
</tr>
<tr>
<td>cond/1</td>
<td>Evaluates the expression corresponding to the first clause that evaluates to true. Raises an error if all clauses are false.</td>
</tr>
<tr>
<td>def/2</td>
<td>Defines a function with the given name and contents</td>
</tr>
<tr>
<td>defdelegate/2</td>
<td>Defines the given functions in the current module that will delegate to the given target. Functions defined with defdelegate are public and are allowed to be invoked from external. If you find yourself wishing to define a delegation as private, you should likely use import instead.</td>
</tr>
<tr>
<td>defexception/1</td>
<td>Defines an exception</td>
</tr>
<tr>
<td>defimpl/3</td>
<td>Defines an implementation for the given protocol. See defprotocol/2 for examples</td>
</tr>
<tr>
<td>defmacro/2</td>
<td>Defines a macro with the given name and contents</td>
</tr>
<tr>
<td>defmacrop/2</td>
<td>Defines a macro that is private. Private macros are only accessible from the same module in which they are defined.</td>
</tr>
<tr>
<td>defmodule/2</td>
<td>Defines a module given by name with the given contents</td>
</tr>
<tr>
<td>defoverridable/1</td>
<td>Makes the given functions in the current module overridable. An overridable function is lazily defined.</td>
</tr>
<tr>
<td>defp/2</td>
<td>Defines a function that is private. Private functions are only accessible from within the module in which they are defined.</td>
</tr>
<tr>
<td>defprotocol/2</td>
<td>Defines a protocol</td>
</tr>
<tr>
<td>defstruct/1</td>
<td>Defines a struct for the current module</td>
</tr>
<tr>
<td>destructure/2</td>
<td>Allows you to destructure two lists, assigning each term in the right to the matching term in the left. Use \ to perform an integer division</td>
</tr>
<tr>
<td>div/2</td>
<td>Get the element at the zero-based index in tuple</td>
</tr>
<tr>
<td>exit/1</td>
<td>Stops the execution of the calling process with the given reason</td>
</tr>
<tr>
<td>function_exported?/3</td>
<td>Returns true if the module is loaded and contains a public function with the given arity, otherwise false.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>returns the head of a list, raises badarg if the list is empty</td>
<td></td>
</tr>
<tr>
<td>Provides an if macro. This macro expects the first argument to be a condition and the rest are keyword arguments.</td>
<td></td>
</tr>
<tr>
<td>Checks if the element on the left side is member of the collection on the right side.</td>
<td></td>
</tr>
<tr>
<td>Inspect the given argument according to the <code>Inspect</code> protocol. The second argument is a keywords list with options to control inspection.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is an atom; otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is a binary; otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is a bitstring (including a binary); otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is either the atom <code>true</code> or the atom <code>false</code> (i.e. a boolean); otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is a floating point number; otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is a function; otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is a list with zero or more elements; otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>term</code> is a tuple; otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns the length of <code>list</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns true if the module is loaded and contains a public macro with the given <code>arity</code>, otherwise false.</td>
<td></td>
</tr>
<tr>
<td>Returns an almost unique reference.</td>
<td></td>
</tr>
<tr>
<td>Returns the size of a map.</td>
<td></td>
</tr>
<tr>
<td>A convenient macro that checks if the right side matches the left side. The left side is allowed to be a tuple, otherwise returns <code>false</code>.</td>
<td></td>
</tr>
<tr>
<td>Return the biggest of the two given terms according to Erlang’s term ordering. If the terms compare equal, the first one is returned.</td>
<td></td>
</tr>
<tr>
<td>Return the smallest of the two given terms according to Erlang’s term ordering. If the terms compare equal, the first one is returned.</td>
<td></td>
</tr>
<tr>
<td>Checks if the given argument is nil or not. Allowed in guard clauses.</td>
<td></td>
</tr>
<tr>
<td>Returns an atom representing the name of the local node. If the node is not alive, <code>:nonode@nohost</code>.</td>
<td></td>
</tr>
<tr>
<td>Returns the node where the given argument is located. The argument can be a pid, a reference, or a port. Boolean not. Argument must be a boolean.</td>
<td></td>
</tr>
<tr>
<td>Boolean or. Requires only the first argument to be a boolean since it short-circuits.</td>
<td></td>
</tr>
<tr>
<td>Sets the element in <code>tuple</code> at the zero-based <code>index</code> to the given <code>value</code>.</td>
<td></td>
</tr>
<tr>
<td>Raises an exception.</td>
<td></td>
</tr>
<tr>
<td>Calculates the remainder of an integer division.</td>
<td></td>
</tr>
<tr>
<td>Raises an exception preserving a previous stacktrace.</td>
<td></td>
</tr>
<tr>
<td>Raises an exception preserving a previous stacktrace.</td>
<td></td>
</tr>
<tr>
<td>Returns an integer by rounding the given number.</td>
<td></td>
</tr>
<tr>
<td>Returns the pid (process identifier) of the calling process.</td>
<td></td>
</tr>
<tr>
<td>Sends a message to the given <code>dest</code> and returns the message.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~C</code>. It simply returns a char list without escaping characters and without interpolating.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~R</code>. It returns a Regex pattern without escaping nor interpreting interpolations.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~S</code>. It simply returns a string without escaping characters and without interpolations.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~W</code>. It returns a list of “words” split by whitespace without escaping nor interpreting.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~c</code>. It returns a char list as if it were a single quoted string, unescaping characters and replacing interpolations.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~r</code>. It returns a Regex pattern.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~s</code>. It returns a string as if it was double quoted string, unescaping characters and replacing interpolations.</td>
<td></td>
</tr>
<tr>
<td>Handles the sigil <code>~w</code>. It returns a list of “words” split by whitespace.</td>
<td></td>
</tr>
<tr>
<td>Returns the size of the given argument, which must be a tuple or a binary.</td>
<td></td>
</tr>
<tr>
<td>Spawns the given function and returns its pid.</td>
<td></td>
</tr>
<tr>
<td>Spawns the given function and returns its pid.</td>
<td></td>
</tr>
<tr>
<td>Spawns the given function, links it to the current process and returns its pid.</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spawn_link/3</td>
<td>Spawns the given module and function passing the given args, links it to the current process and returns its pid</td>
</tr>
<tr>
<td>spawn_monitor/1</td>
<td>Spawns the given function, monitors it and returns its pid and monitoring reference</td>
</tr>
<tr>
<td>spawn_monitor/3</td>
<td>Spawns the given module and function passing the given args, monitors it and returns its pid and monitoring reference</td>
</tr>
<tr>
<td>struct/2</td>
<td>Creates and updates structs</td>
</tr>
<tr>
<td>throw/1</td>
<td>A non-local return from a function. Check Kernel.SpecialForms.try/1 for more information</td>
</tr>
<tr>
<td>tl/1</td>
<td>Returns the tail of a list. Raises ArgumentError if the list is empty</td>
</tr>
<tr>
<td>to_char_list/1</td>
<td>Convert the argument to a list according to the List.Chars protocol</td>
</tr>
<tr>
<td>to_string/1</td>
<td>Converts the argument to a string according to the String.Chars protocol. This is the function invoked when there is string interpolation</td>
</tr>
<tr>
<td>trunc/1</td>
<td>Returns an integer by truncating the given number</td>
</tr>
<tr>
<td>tuple_size/1</td>
<td>Returns the size of a tuple</td>
</tr>
<tr>
<td>unless/2</td>
<td>Evaluates and returns the do-block passed in as a second argument unless clause evaluates to true. Returns nil otherwise. See also if</td>
</tr>
<tr>
<td>use/2</td>
<td>use is a simple mechanism for using a given module into the current context</td>
</tr>
<tr>
<td>xor/2</td>
<td>Boolean exclusive-or. Arguments must be booleans. Returns true if and only if both arguments are different</td>
</tr>
<tr>
<td></td>
<td>&gt;/2</td>
</tr>
<tr>
<td></td>
<td>!/2</td>
</tr>
</tbody>
</table>

**Summary**

**Functions**

**left != right(function)**

Specs:

• term != term :: boolean

Returns true if the two items are not equal.

This operator considers 1 and 1.0 to be equal. For match comparison, use !== instead.

All terms in Elixir can be compared with each other.

Allowed in guard tests. Inlined by the compiler.

**Examples**

```iex
iex> 1 != 2
true

iex> 1 != 1.0
false
```

**left !== right(function)**

Specs:

• term !== term :: boolean

Returns true if the two items do not match.

All terms in Elixir can be compared with each other.

Allowed in guard tests. Inlined by the compiler.

**Examples**

```iex
iex> 1 !== 2
true

iex> 1 !== 1.0
true
```
left * right(function)
Specs:
  • number * number :: number
Arithmetic multiplication.
Allowed in guard tests. Inlined by the compiler.

Examples
  iex> 1 * 2
   2

+value(function)
Specs:
  • +number :: number
Arithmetic unary plus.
Allowed in guard tests. Inlined by the compiler.

Examples
  iex> +1
   1

left + right(function)
Specs:
  • number + number :: number
Arithmetic plus.
Allowed in guard tests. Inlined by the compiler.

Examples
  iex> 1 + 2
   3

left ++ right(function)
Specs:
  • [] ++ term :: maybe_improper_list
Concatenates two lists.
Allowed in guard tests. Inlined by the compiler.

Examples
  iex> [1] ++ [2, 3]
   [1,2,3]
  iex> 'foo' ++ 'bar'
   'foobara'
Examples

```elixir
iex> -2
-2
```

**left - right(function)**

Specs:

- number - number :: number

Arithmetic minus.

Allowed in guard tests. Inlined by the compiler.

Examples

```elixir
iex> 1 - 2
-1
```

**left -- right(function)**

Specs:

- [] - [] :: []

Removes the first occurrence of an item on the left for each item on the right.

Allowed in guard tests. Inlined by the compiler.

Examples

```elixir
iex> [1, 2, 3] -- [1, 2]
[3]

iex> [1, 2, 3, 2, 1] -- [1, 2, 2]
[3, 1]
```

**left / right(function)**

Specs:

- number / number :: float

Arithmetic division.

The result is always a float. Use `div` and `rem` if you want a natural division or the remainder.

Allowed in guard tests. Inlined by the compiler.

Examples

```elixir
iex> 1 / 2
0.5

iex> 2 / 1
2.0
```

**left < right(function)**

Specs:

- term < term :: boolean

Returns `true` if left is less than right.

All terms in Elixir can be compared with each other.

Allowed in guard tests. Inlined by the compiler.

Examples
iex> 1 < 2
true

**left <= right**(function)
Specs:

- •term <= term :: boolean
Returns `true` if left is less than or equal to right.
All terms in Elixir can be compared with each other.
Allowed in guard tests. Inlined by the compiler.

**Examples**
iex> 1 <= 2
true

**left == right**(function)
Specs:

- •term == term :: boolean
Returns `true` if the two items are equal.
This operator considers 1 and 1.0 to be equal. For match semantics, use `===` instead.
All terms in Elixir can be compared with each other.
Allowed in guard tests. Inlined by the compiler.

**Examples**
iex> 1 == 2
false

iex> 1.0 == 1.0
true

**left === right**(function)
Specs:

- •term === term :: boolean
Returns `true` if the two items are match.
This operator gives the same semantics as the one existing in pattern matching, i.e., 1 and 1.0 are equal, but they do not match.
All terms in Elixir can be compared with each other.
Allowed in guard tests. Inlined by the compiler.

**Examples**
iex> 1 === 2
false

iex> 1.0 === 1.0
false

**left =~ right**(function)
Matches the term on the left against the regular expression or string on the right. Returns `true` if `left` matches `right` (if it’s a regular expression) or contains `right` (if it’s a string).
Examples

```iex
  iex> "abcd" =~ ~r/c(d)/
  true

  iex> "abcd" =~ ~r/e/
  false

  iex> "abcd" =~ "bc"
  true

  iex> "abcd" =~ "ad"
  false
```

**left > right(function)**

Specs:

- `term > term :: boolean`

Returns `true` if left is more than right.

All terms in Elixir can be compared with each other.

Allowed in guard tests. Inlined by the compiler.

Examples

```iex
  iex> 1 > 2
  false
```

**left >= right(function)**

Specs:

- `term >= term :: boolean`

Returns `true` if left is more than or equal to right.

All terms in Elixir can be compared with each other.

Allowed in guard tests. Inlined by the compiler.

Examples

```iex
  iex> 1 >= 2
  false
```

**abs(number) (function)**

Specs:

- `abs(number) :: number`

Returns an integer or float which is the arithmetical absolute value of `number`.

Allowed in guard tests. Inlined by the compiler.

Examples

```iex
  iex> abs(-3.33)
  3.33

  iex> abs(-3)
  3
```

**apply(fun, args) (function)**

Specs:
Invokes the given fun with the array of arguments args.
Inlined by the compiler.

**Examples**

```iex
def fn x -> x * 2 end
apply(fn x -> x * 2 end, [2])
```

4

---

### apply(module, fun, args)(function)

**Specs:**

- apply(module, atom, [any]) :: any

Invokes the given fun from module with the array of arguments args.
Inlined by the compiler.

**Examples**

```iex
apply(Enum, :reverse, [[1, 2, 3]])
```

`[3,2,1]`

---

### binary_part(binary, start, length)(function)

**Specs:**

- binary_part(binary, pos_integer, integer) :: binary

Extracts the part of the binary starting at start with length length. Binaries are zero-indexed.
If start or length references in any way outside the binary, an ArgumentError exception is raised.
Allowed in guard tests. Inlined by the compiler.

**Examples**

```iex
binary_part("foo", 1, 2)
```

"oo"

A negative length can be used to extract bytes at the end of a binary:

```iex
binary_part("foo", 3, -1)
```

"o"

---

### bit_size(bitstring)(function)

**Specs:**

- bit_size(bitstring) :: non_neg_integer

Returns an integer which is the size in bits of bitstring.
Allowed in guard tests. Inlined by the compiler.

**Examples**

```iex
bit_size(<<433::16, 3::3>>)
```

19

```iex
bit_size(<<1, 2, 3>>)  
```

24

---

### byte_size(bitstring)(function)

**Specs:**

- byte_size(bitstring) :: non_neg_integer


Returns the number of bytes needed to contain `bitstring`.

That is, if the number of bits in `bitstring` is not divisible by 8, the resulting number of bytes will be rounded up. This operation happens in constant time.

Allowed in guard tests. Inlined by the compiler.

**Examples**

```
  iex> byte_size(<433::16, 3::3>)
  3

  iex> byte_size(<1, 2, 3>)
  3
```

**div(left, right)** (function)

Specs:

- `div(integer, integer) :: integer`

Performs an integer division.

Raises an error if one of the arguments is not an integer.

Allowed in guard tests. Inlined by the compiler.

**Examples**

```
  iex> div(5, 2)
  2
```

**elem(tuple, index)** (function)

Specs:

- `elem(tuple, non_neg_integer) :: term`

Get the element at the zero-based `index` in `tuple`.

Allowed in guard tests. Inlined by the compiler.

**Example**

```
  iex> tuple = {:foo, :bar, 3}
  iex> elem(tuple, 1)
  :bar
```

**exit(reason)** (function)

Specs:

- `exit(term) :: no_return`

Stops the execution of the calling process with the given reason.

Since evaluating this function causes the process to terminate, it has no return value.

Inlined by the compiler.

**Examples**

```
  exit(:normal)
  exit(:seems_bad)
```

**function_exported?(module, function, arity)** (function)

Specs:

- `function_exported?(atom | tuple, atom, integer) :: boolean`
Returns true if the module is loaded and contains a public function with the given arity, otherwise false.

Notice that this function does not load the module in case it is not loaded. Check `Code.ensure_loaded/1` for more information.

hd(list)(function)
Specs:
- `hd([]) :: term`

Returns the head of a list, raises `badarg` if the list is empty.

Inlined by the compiler.

inspect(arg, opts \ [[]]) (function)
Specs:
- `inspect(Inspect.t, Keyword.t) :: String.t`

Inspect the given argument according to the `Inspect` protocol. The second argument is a keywords list with options to control inspection.

Options

`inspect/2` accepts a list of options that are internally translated to an `Inspect.Opts` struct. Check the docs for `Inspect.Opts` to see the supported options.

Examples

```iex
iex> inspect(:foo)
":foo"

iex> inspect([1, 2, 3, 4, 5], limit: 3)
"[1, 2, 3, ...]"

iex> inspect("jose" <> <<0>>)
"<<106, 111, 115, 195, 169, 0>>"

iex> inspect("jose" <> <<0>>, binaries: :as_strings)
"\"jose\000\"

iex> inspect("jose", binaries: :as_binaries)
"<<106, 111, 115, 195, 169>>"
```

Note that the inspect protocol does not necessarily return a valid representation of an Elixir term. In such cases, the inspected result must start with `#`. For example, inspecting a function will return:

```iex
inspect fn a, b -> a + b end
#=> #Function<...>
```

is_atom(term) (function)
Specs:
- `is_atom(term) :: boolean`

Returns `true` if `term` is an atom; otherwise returns `false`.

Allowed in guard tests. Inlined by the compiler.

is_binary(term)(function)
Specs:
- `is_binary(term) :: boolean`
Returns true if term is a binary; otherwise returns false.
A binary always contains a complete number of bytes.
Allowed in guard tests. Inlined by the compiler.

is_bitstring(term) (function)
Specs:
  •is_bitstring(term) :: boolean
Returns true if term is a bitstring (including a binary); otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_boolean(term) (function)
Specs:
  •is_boolean(term) :: boolean
Returns true if term is either the atom true or the atom false (i.e. a boolean); otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_float(term) (function)
Specs:
  •is_float(term) :: boolean
Returns true if term is a floating point number; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_function(term) (function)
Specs:
  •is_function(term) :: boolean
Returns true if term is a function; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_function(term, arity) (function)
Specs:
  •is_function(term, non_neg_integer) :: boolean
Returns true if term is a function that can be applied with arity number of arguments; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_integer(term) (function)
Specs:
  •is_integer(term) :: boolean
Returns true if term is an integer; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_list(term) (function)
Specs:
  •is_list(term) :: boolean
Returns true if term is a list with zero or more elements; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.
is_map(term) (function)
Specs:
  • is_map(term) :: boolean
Returns true if term is a map; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_number(term) (function)
Specs:
  • is_number(term) :: boolean
Returns true if term is either an integer or a floating point number; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_pid(term) (function)
Specs:
  • is_pid(term) :: boolean
Returns true if term is a pid (process identifier); otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_port(term) (function)
Specs:
  • is_port(term) :: boolean
Returns true if term is a port identifier; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_reference(term) (function)
Specs:
  • is_reference(term) :: boolean
Returns true if term is a reference; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

is_tuple(term) (function)
Specs:
  • is_tuple(term) :: boolean
Returns true if term is a tuple; otherwise returns false.
Allowed in guard tests. Inlined by the compiler.

length(list) (function)
Specs:
  • length([]) :: non_neg_integer
Returns the length of list.
Allowed in guard tests. Inlined by the compiler.

Examples
iex> length([1, 2, 3, 4, 5, 6, 7, 8, 9])
9
macro_exported?(module, macro, arity) (function)

Specs:

- `macro_exported?(atom, atom, integer) :: boolean`

Returns true if the module is loaded and contains a public macro with the given arity, otherwise false.

Notice that this function does not load the module in case it is not loaded. Check `Code.ensure_loaded/1` for more information.

make_ref() (function)

Specs:

- `make_ref :: reference`

Returns an almost unique reference.

The returned reference will re-occur after approximately $2^{82}$ calls; therefore it is unique enough for practical purposes.

Inlined by the compiler.

Examples

```
make_ref() #=> #Reference<0.0.0.135>
```

map_size(map) (function)

Specs:

- `map_size(%{}) :: non_neg_integer`

Returns the size of a map.

This operation happens in constant time.

Allowed in guard tests. Inlined by the compiler.

max(first, second) (function)

Specs:

- `max(term, term) :: term`

Return the biggest of the two given terms according to Erlang’s term ordering. If the terms compare equal, the first one is returned.

Inlined by the compiler.

Examples

```
iex> max(1, 2)
2
```

min(first, second) (function)

Specs:

- `min(term, term) :: term`

Return the smallest of the two given terms according to Erlang’s term ordering. If the terms compare equal, the first one is returned.

Inlined by the compiler.

Examples

```
iex> min(1, 2)
1
```
node () (function)
Specs:

•node :: node

Returns an atom representing the name of the local node. If the node is not alive, :nonode@nohost is returned instead.
Allowed in guard tests. Inlined by the compiler.

node (arg) (function)
Specs:

•node(pid | reference | port) :: node

Returns the node where the given argument is located. The argument can be a pid, a reference, or a port. If the local node is not alive, nonode@nohost is returned.
Allowed in guard tests. Inlined by the compiler.

not arg (function)
Specs:

•not boolean :: boolean

Boolean not. Argument must be a boolean.
Allowed in guard tests. Inlined by the compiler.

Examples
iex> not false
true

put_elem (tuple, index, value) (function)
Specs:

•put_elem(tuple, non_neg_integer, term) :: tuple

Sets the element in tuple at the zero-based index to the given value.
Inlined by the compiler.

Example
iex> tuple = {:foo, :bar, 3}
iex> put_elem(tuple, 0, :baz)
{:baz, :bar, 3}

rem (left, right) (function)
Specs:

•rem(integer, integer) :: integer

Calculates the remainder of an integer division.
Raises an error if one of the arguments is not an integer.
Allowed in guard tests. Inlined by the compiler.

Examples
iex> rem(5, 2)
1

round (number) (function)
Specs:
•round(number) :: integer
Returns an integer by rounding the given number.
Allowed in guard tests. Inlined by the compiler.

Examples
iex> round(5.5)
6

self() (function)
Specs:
  •self :: pid
Returns the pid (process identifier) of the calling process.
Allowed in guard clauses. Inlined by the compiler.

send(dest, msg) (function)
Specs:
  •(send(dest :: pid | port | atom | {atom, node}, msg) :: msg) when msg: any
Sends a message to the given dest and returns the message.
dest may be a remote or local pid, a (local) port, a locally registered name, or a tuple {registered_name, node} for a registered name at another node.
Inlined by the compiler.

Examples
iex> send self(), :hello
:hello

size(arg) (function)
Specs:
  •size(tuple | binary) :: non_neg_integer
Returns the size of the given argument, which must be a tuple or a binary.
Prefer using tuple_size/1 or byte_size/1 instead.
Allowed in guard tests. Inlined by the compiler.

spawn(fun) (function)
Specs:
  •spawn((() -> any)) :: pid
Spawns the given function and returns its pid.
Check the modules Process and Node for other functions to handle processes, including spawning functions in nodes.
Inlined by the compiler.

Examples
current = Kernel.self
child = spawn(fn -> send current, {Kernel.self, 1 + 2} end)

receive do
spawn(module, fun, args)(function)
Specs:
•spawn(module, atom, []) :: pid
Spawns the given module and function passing the given args and returns its pid.
Check the modules Process and Node for other functions to handle processes, including spawning functions in nodes.
Inlined by the compiler.
Examples
spawn(SomeModule, :function, [1, 2, 3])

spawn_link(fun)(function)
Specs:
•spawn_link((() -> any)) :: pid
Spawns the given function, links it to the current process and returns its pid.
Check the modules Process and Node for other functions to handle processes, including spawning functions in nodes.
Inlined by the compiler.
Examples
current = Kernel.self
child  = spawn_link(fn -> send current, (Kernel.self, 1 + 2) end)
receive do
  (^child, 3) -> IO.puts "Received 3 back"
end

spawn_link(module, fun, args)(function)
Specs:
•spawn_link(module, atom, []) :: pid
Spawns the given module and function passing the given args, links it to the current process and returns its pid.
Check the modules Process and Node for other functions to handle processes, including spawning functions in nodes.
Inlined by the compiler.
Examples
spawn_link(SomeModule, :function, [1, 2, 3])

spawn_monitor(fun)(function)
Specs:
•spawn_monitor((() -> any)) :: {pid, reference}
Spawns the given function, monitors it and returns its pid and monitoring reference.
Check the modules Process and Node for other functions to handle processes, including spawning functions in nodes.
Inlined by the compiler.

**Examples**

```elixir
current = Kernel.self
spawn_monitor(fn -> send current, {Kernel.self, 1 + 2} end)
```

### spawn_monitor(module, fun, args)(function)

**Specs:**

- `spawn_monitor(module, atom, []) :: {pid, reference}`

Spawns the given module and function passing the given args, monitors it and returns its pid and monitoring reference.

Check the modules `Process` and `Node` for other functions to handle processes, including spawning functions in nodes.

Inlined by the compiler.

**Examples**

```elixir
spawn_monitor(SomeModule, :function, [1, 2, 3])
```

### struct(struct, kv \ [])(function)

**Specs:**

- `struct(module | %{}, Enum.t) :: %{}`

Creates and updates structs.

The struct argument may be an atom (which defines `defstruct`) or a struct itself. The second argument is any Enumerable that emits two-item tuples (key-value) during enumeration.

If one of the keys in the Enumerable does not exist in the struct, they are automatically discarded.

This function is useful for dynamically creating and updating structs.

**Example**

```elixir
defmodule User do
  defstruct name: "jose"
end

struct(User) #=> %User{name: "jose"}
```

```elixir
opts = [name: "eric"]
user = struct(User, opts) #=> %User{name: "eric"}
```

```elixir
struct(user, unknown: "value") #=> %User{name: "eric"}
```

### throw(term)(function)

**Specs:**

- `throw(term) :: no_return`

A non-local return from a function. Check `Kernel.SpecialForms.try/1` for more information.

Inlined by the compiler.

### tl(list)(function)

**Specs:**
### \texttt{tl} (maybe_improper_list) :: maybe_improper_list

Returns the tail of a list. Raises \texttt{ArgumentError} if the list is empty. Allowed in guard tests. Inlined by the compiler.

### \texttt{trunc(number)} (function)

Specs:

- \texttt{trunc(number)} :: integer

Returns an integer by truncating the given number. Allowed in guard tests. Inlined by the compiler.

**Examples**

```elixir
ing> \texttt{trunc(5.5)}
5
```

### \texttt{tuple_size(tuple)} (function)

Specs:

- \texttt{tuple_size(tuple)} :: non_neg_integer

Returns the size of a tuple. This operation happens in constant time. Allowed in guard tests. Inlined by the compiler.

### \texttt{left xor right} (function)

Specs:

- boolean xor boolean :: boolean

Boolean exclusive-or. Arguments must be booleans. Returns \texttt{true} if and only if both arguments are different. Allowed in guard tests. Inlined by the compiler.

**Examples**

```elixir
ing> \texttt{true xor false}
true

iex> \texttt{true xor true}
false
```

### Macros

\texttt{!arg(macro)}

Receives any argument and returns \texttt{true} if it is \texttt{false} or \texttt{nil}. Returns \texttt{false} otherwise. Not allowed in guard clauses.

**Examples**

```elixir
ing> \texttt{!Enum.empty?([])}
false

iex> \texttt{!List.first([])}
true
```

\texttt{left && right (macro)}

Provides a short-circuit operator that evaluates and returns the second expression only if the first one evaluates to \texttt{true} (i.e. it is not \texttt{nil} nor \texttt{false}). Returns the first expression otherwise.
Examples

```iex
Enum.empty?([]) && Enum.empty?([])
true

List.first([]) && true
nil

Enum.empty?([]) && List.first([1])
1

false && throw(:bad)
false
```

Notice that, unlike Erlang’s `and` operator, this operator accepts any expression as an argument, not only booleans, however it is not allowed in guards.

### first .. last(macro)

Returns a range with the specified start and end. Includes both ends.

**Examples**

```iex
0 in 1..3
false

1 in 1..3
true

2 in 1..3
true

3 in 1..3
true
```

### left <> right(macro)

Concatenates two binaries.

**Examples**

```iex
"foo" <> "bar"
"foobar"
```

The `<>` operator can also be used in guard clauses as long as the first part is a literal binary:

```iex
"foo" <> x = "foobar"
iex> x
"bar"
```

### @expr(macro)

Read and write attributes of the current module.

The canonical example for attributes is annotating that a module implements the OTP behaviour called `gen_server`:

```elixir
defmodule MyServer do
  @behaviour :gen_server
  # ... callbacks ...
end
```

By default Elixir supports all Erlang module attributes, but any developer can also add custom attributes:
Unlike Erlang, such attributes are not stored in the module by default since it is common in Elixir to use such attributes to store temporary data. A developer can configure an attribute to behave closer to Erlang by calling `Module.register_attribute/3`.

Finally, notice that attributes can also be read inside functions:

```elixir
defmodule MyServer do
  @my_data 11
  def first_data, do: @my_data
  @my_data 13
  def second_data, do: @my_data
end
```

```
MyServer.first_data #=> 11
MyServer.second_data #=> 13
```

It is important to note that reading an attribute takes a snapshot of its current value. In other words, the value is read at compilation time and not at runtime. Check the module `Module` for other functions to manipulate module attributes.

**alias! (alias) (macro)**

When used inside quoting, marks that the alias should not be hygienezed. This means the alias will be expanded when the macro is expanded.

Check `Kernel.SpecialForms.quote/2` for more information.

**left and right (macro)**

Boolean and. Requires only the first argument to be a boolean since it short-circuits.

Allowed in guard tests.

**Examples**

```
iex> true and false
false
```

**binding () (macro)**

Returns the binding as a keyword list where the variable name is the key and the variable value is the value.

**Examples**

```
iex> x = 1
iex> binding()
[x: 1]
iex> x = 2
iex> binding()
[x: 2]
```

**binding(list) (macro)**

Receives a list of atoms at compilation time and returns the binding of the given variables as a keyword list where the variable name is the key and the variable value is the value.

In case a variable in the list does not exist in the binding, it is not included in the returned result.

**Examples**
iex> x = 1
iex> binding([:x, :y])
[x: 1]

**binding(list, context)** *(macro)*

Receives a list of atoms at compilation time and returns the binding of the given variables in the given context as a keyword list where the variable name is the key and the variable value is the value.

In case a variable in the list does not exist in the binding, it is not included in the returned result.

**Examples**

iex> var!(x, :foo) = 1
iex> binding([:x, :y])
[]
iex> binding([:x, :y], :foo)
[x: 1]

**cond(list1)** *(macro)*

Evaluates the expression corresponding to the first clause that evaluates to true. Raises an error if all conditions evaluate to to nil or false.

**Examples**

```elixir
cond do
  1 + 1 == 1 ->
    "This will never match"
  2 * 2 != 4 ->
    "Nor this"
  true ->
    "This will"
end
```

**def(call, expr \ nil)** *(macro)*

Defines a function with the given name and contents.

**Examples**

```elixir
defmodule Foo do
  def bar, do: :baz
end
```

```elixir
Foo.bar #=> :baz
```

A function that expects arguments can be defined as follow:

```elixir
defmodule Foo do
  def sum(a, b) do
    a + b
  end
end
```

In the example above, we defined a function `sum` that receives two arguments and sums them.

**defdelegate(funs, opts)** *(macro)*

Defines the given functions in the current module that will delegate to the given target. Functions defined with `defdelegate` are public and are allowed to be invoked from external. If you find yourself wishing to define a delegation as private, you should likely use import instead.

Delegation only works with functions, delegating to macros is not supported.

**Options**

3.2. API Reference (v0.14.0-dev) 217
• :to - The expression to delegate to. Any expression is allowed and its results will be calculated on runtime;
• :as - The function to call on the target given in :to. This parameter is optional and defaults to the name being delegated.
• :append_first - If true, when delegated, first argument passed to the delegate will be relocated to the end of the arguments when dispatched to the target. The motivation behind this is because Elixir normalizes the “handle” as a first argument and some Erlang modules expect it as last argument.

Examples

defmodule MyList do
defdelegate reverse(list), to: :lists
defdelegate [reverse(list), map(callback, list)], to: :lists
defdelegate other_reverse(list), to: :lists, as: :reverse
end

MyList.reverse([1, 2, 3])
#=> [3, 2, 1]

MyList.other_reverse([1, 2, 3])
#=> [3, 2, 1]


defexception(fields) (macro)

Defines an exception.

Exceptions are structs backed by a module that implements the Exception behaviour. The Exception behaviour requires two functions to be implemented:

• exception/1 - that receives the arguments given to raise/2 and returns the exception struct. The default implementation accepts a set of keyword arguments that is merged into the struct;
• message/1 - receives the exception struct and must return its message. Most commonly exceptions have a message field which by default is accessed by this function. However, if your exception does not have a message field, this function must be explicitly implemented;

Since exceptions are structs, all the API supported by defstruct/1 is also available in defexception/1.

Raising exceptions

The most common way to raise an exception is via the raise/2 function:

defmodule MyAppError do
defexception [:message]
end

raise MyAppError,
message: "did not get what was expected, got: #{inspect value}"

In many cases it is more convenient to pass the expected value to raise and generate the message in the exception/1 callback:

defmodule MyAppError do
defexception [:message]

def exception(value) do
  msg = "did not get what was expected, got: #{inspect value}"
  %MyAppError{message: msg}
end
end
raise MyAppError, value

The example above is the preferred mechanism for customizing exception messages.

```elixir
defimpl(name, opts, do_block \ \ [[]](macro)
  Defines an implementation for the given protocol. See defprotocol/2 for examples.

  Inside an implementation, the name of the protocol can be accessed via @protocol and the current target as @for.


defmacro(call, expr \ \ nil)(macro)
  Defines a macro with the given name and contents.

  Examples

  defmodule Mylogic do
    defmacro unless(expr, opts) do
      quote do
        if :unquote(expr), unquote(opts)
          end
      end
    end

    require Mylogic
    MyLogic.unless false do
      IO.puts "It works"
    end


defmacrop(call, expr \ \ nil)(macro)
  Defines a macro that is private. Private macros are only accessible from the same module in which they are defined.

  Check defmacro/2 for more information


defmodule(alias, list2)(macro)
  Defines a module given by name with the given contents.

  It returns the module name, the module binary and the block contents result.

  Examples

  iex> defmodule Foo do
      ...
      def bar, do: :baz
      ...
    end
  iex> Foo.bar
  :baz

  Nesting

  Nesting a module inside another module affects its name:

  defmodule Foo do
    defmodule Bar do
      end
    end

  In the example above, two modules Foo and Foo.Bar are created. When nesting, Elixir automatically creates an alias, allowing the second module Foo.Bar to be accessed as Bar in the same lexical scope.

  This means that, if the module Bar is moved to another file, the references to Bar needs to be updated or an alias needs to be explicitly set with the help of Kernel.SpecialForms.alias/2.
```
Dynamic names

Elixir module names can be dynamically generated. This is very useful for macros. For instance, one could write:

```elixir
defmodule String.to_atom("Foo#{1}") do
  # contents ...
end
```

Elixir will accept any module name as long as the expression returns an atom. Note that, when a dynamic name is used, Elixir won’t nest the name under the current module nor automatically set up an alias.

`defoverridable(tuples)` (macro)

Makes the given functions in the current module overridable. An overridable function is lazily defined, allowing a developer to customize it.

Example

```elixir
defmodule DefaultMod do
  defmacro __using__(_opts) do
    quote do
      def test(x, y) do
        x + y
      end

      defoverridable [test: 2]
    end
  end
end
```

```elixir
defmodule InheritMod do
  use DefaultMod

  def test(x, y) do
    x * y + super(x, y)
  end
end
```

As seen as in the example `super` can be used to call the default implementation.

`defp(call, expr \ nil)` (macro)

Defines a function that is private. Private functions are only accessible from within the module in which they are defined.

Check `def/2` for more information

Examples

```elixir
defmodule Foo do
  def bar do
    sum(1, 2)
  end

  defp sum(a, b), do: a + b
end
```

In the example above, `sum` is private and accessing it through `Foo.sum` will raise an error.

`defprotocol(name, list2)` (macro)

Defines a protocol.

A protocol specifies an API that should be defined by its implementations.
Examples

In Elixir, only `false` and `nil` are considered falsy values. Everything else evaluates to true in `if` clauses. Depending on the application, it may be important to specify a `blank?` protocol that returns a boolean for other data types that should be considered `blank?`. For instance, an empty list or an empty binary could be considered blanks.

We could implement this protocol as follow:

```elixir
defprotocol Blank do
  @doc "Returns true if data is considered blank/empty"
  def blank?(data)
end
```

Now that the protocol is defined, we can implement it. We need to implement the protocol for each Elixir type. For example:

```elixir
# Integers are never blank
defimpl Blank, for: Integer do
  def blank?(number), do: false
end

# Just empty list is blank
defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_), do: false
end

# Just the atoms false and nil are blank
defimpl Blank, for: Atom do
  def blank?(false), do: true
  def blank?(nil), do: true
  def blank?(_), do: false
end
```

And we would have to define the implementation for all types. The supported types available are:

- Structs (see below)
- Tuple
- Atom
- List
- BitString
- Integer
- Float
- Function
- PID
- Map
- Port
- Reference
- Any (see below)

Protocols + Structs
The real benefit of protocols comes when mixed with structs. For instance, Elixir ships with many data types implemented as structs, like `HashDict` and `HashSet`. We can implement the `Blank` protocol for those types as well:

```elixir
defimpl Blank, for: [HashDict, HashSet] do
def blank?(enum_like), do: Enum.empty?(enum_like)
end
```

If a protocol is not found for a given type, it will fallback to `Any`.

**Fallback to any**

In some cases, it may be convenient to provide a default implementation for all types. This can be achieved by setting `@fallback_to_any` to `true` in the protocol definition:

```elixir
defprotocol Blank do
  @fallback_to_any true
  def blank?(data)
end
```

Which can now be implemented as:

```elixir
defimpl Blank, for: Any do
def blank?(_,), do: true
end
```

One may wonder why such fallback is not true by default.

It is two-fold: first, the majority of protocols cannot implement an action in a generic way for all types. In fact, providing a default implementation may be harmful, because users may rely on the default implementation instead of providing a specialized one.

Second, falling back to `Any` adds an extra lookup to all types, which is unnecessary overhead unless an implementation for `Any` is required.

**Types**

Defining a protocol automatically defines a type named `t`, which can be used as:

```elixir
@spec present?(Blank.t) :: boolean
def present?(blank) do
  not Blank.blank?(blank)
end
```

The `@spec` above expresses that all types allowed to implement the given protocol are valid argument types for the given function.

**Reflection**

Any protocol module contains three extra functions:

- `__protocol__/1` - returns the protocol name when `:name` is given, and a keyword list with the protocol functions when `:functions` is given;
- `impl_for/1` - receives a structure and returns the module that implements the protocol for the structure, nil otherwise;
- `impl_for!/1` - same as above but raises an error if an implementation is not found

**Consolidation**

In order to cope with code loading in development, protocols in Elixir provide a slow implementation of protocol dispatching specific to development.
In order to speed up dispatching in production environments, where all implementations are known up-front, Elixir provides a feature called protocol consolidation. For this reason, all protocols are compiled with `debug_info` set to true, regardless of the option set by `elixirc` compiler. The debug info though may be removed after consolidation.

For more information on how to apply protocol consolidation to a given project, please check the functions in the `Protocol` module or the `mix compile.protocols` task.

**defstruct (kv) (macro)**

Defines a struct for the current module.

A struct is a tagged map that allows developers to provide default values for keys, tags to be used in polymorphic dispatches and compile time assertions.

To define a struct, a developer needs to only define a function named `__struct__/0` that returns a map with the structs field. This macro is a convenience for defining such function, with the addition of a type `t` and deriving conveniences.

For more information about structs, please check `Kernel.SpecialForms.%/2`.

**Examples**

```elixir
defmodule User do  
  defstruct name: nil, age: nil 
end
```

Struct fields are evaluated at definition time, which allows them to be dynamic. In the example below, `10 + 11` will be evaluated at compilation time and the age field will be stored with value `21`:

```elixir
defmodule User do  
  defstruct name: nil, age: 10 + 11 
end
```

**Deriving**

Although structs are maps, by default structs do not implement any of the protocols implemented for maps. For example, if you attempt to use the access protocol with the User struct, it will lead to an error:

```elixir
%User{}[:age]
** (Protocol.UndefinedError) protocol Access not implemented for %User{...}
```

However, `defstruct/2` allows implementation for protocols to derived by defining a `@derive` attribute as a list before `defstruct/2` is invoked:

```elixir
defmodule User do  
  @derive [Access]
  defstruct name: nil, age: 10 + 11 
end
```

```elixir
%User{}[:age] #=> 21
```

For each protocol given to `@derive`, Elixir will assert there is an implementation of that protocol for maps and check if the map implementation defines a `__deriving__/2` callback. If so, the callback is invoked, otherwise an implementation that simply points to the map one is automatically derived.

**Types**

`defstruct` automatically generates a type `t` unless one exists. The following definition:

```elixir
defmodule User do  
  defstruct name: "José" :: String.t,  
         age: 25 :: integer  
end
```
Generates a type as follows:

@type t :: %User{name: String.t, age: integer}

In case a struct does not declare a field type, it defaults to \texttt{term}.

\texttt{destructure(left, right) (macro)}

Allows you to destruct two lists, assigning each term in the right to the matching term in the left. Unlike pattern matching via \texttt{=}, if the sizes of the left and right lists don’t match, destructuring simply stops instead of raising an error.

\textbf{Examples}

\begin{verbatim}
iex> destructure([x, y, z], [1, 2, 3, 4, 5])
iex> {x, y, z}
{1, 2, 3}
\end{verbatim}

Notice in the example above, even though the right size has more entries than the left, destructuring works fine. If the right size is smaller, the remaining items are simply assigned to nil:

\begin{verbatim}
iex> destructure([x, y, z], [1])
iex> {x, y, z}
{1, nil, nil}
\end{verbatim}

The left side supports any expression you would use on the left side of a match:

\begin{verbatim}
x = 1
destructure([\^x, y, z], [1, 2, 3])
\end{verbatim}

The example above will only work if \texttt{x} matches the first value from the right side. Otherwise, it will raise a \texttt{CaseClauseError}.

\texttt{if(condition, clauses) (macro)}

Provides an \texttt{if} macro. This macro expects the first argument to be a condition and the rest are keyword arguments.

\textbf{One-liner examples}

\begin{verbatim}
if(foo, do: bar)
\end{verbatim}

In the example above, \texttt{bar} will be returned if \texttt{foo} evaluates to true (i.e. it is neither \texttt{false} nor \texttt{nil}). Otherwise, \texttt{nil} will be returned.

An \texttt{else} option can be given to specify the opposite:

\begin{verbatim}
if(foo, do: bar, else: baz)
\end{verbatim}

\textbf{Blocks examples}

Elixir also allows you to pass a block to the \texttt{if} macro. The first example above would be translated to:

\begin{verbatim}
if foo do
  bar
end
\end{verbatim}

Notice that \texttt{do/end} becomes delimiters. The second example would then translate to:

\begin{verbatim}
if foo do
  bar
else
  baz
end
\end{verbatim}
If you want to compare more than two clauses, you can use the `cond/1` macro.

**left in right**(macro)

Checks if the element on the left side is member of the collection on the right side.

**Examples**

```
iex> x = 1
iex> x in [1, 2, 3]
true
```

This macro simply translates the expression above to:

```elixir
Enum.member?([1,2,3], x)
```

**Guards**

The `in` operator can be used on guard clauses as long as the right side is a range or a list. Elixir will then expand the operator to a valid guard expression. For example:

```elixir
when x in [1,2,3]
```

Translates to:

```elixir
when x === 1 or x === 2 or x === 3
```

When using ranges:

```elixir
when x in 1..3
```

Translates to:

```elixir
when x >= 1 and x <= 3
```

**match?**(pattern, expr)(macro)

A convenient macro that checks if the right side matches the left side. The left side is allowed to be a match pattern.

**Examples**

```
iex> match?(1, 1)
true
iex> match?(1, 2)
false
iex> match?({:a, _}, {1, 2})
true
```

Match can also be used to filter or find a value in an enumerable:

```
list = [{:a, 1}, {:b, 2}, {:a, 3}]
Enum.filter list, &match?({:a, _}, &1)
```

Guard clauses can also be given to the match:

```
list = [{:a, 1}, {:b, 2}, {:a, 3}]
Enum.filter list, &match?({:a, x} when x < 2, &1)
```

However, variables assigned in the match will not be available outside of the function call:
nil? \(x\) (macro)
Checks if the given argument is nil or not. Allowed in guard clauses.

Examples
iex> nil?(1)
false

iex> nil?(nil)
true

left or right (macro)
Boolean or. Requires only the first argument to be a boolean since it short-circuits.
Allowed in guard tests.

Examples
iex> true or false
true

raise (msg) (macro)
Raises an exception.
If the argument is a binary, it raises `RuntimeError` using the given argument as message.
If an atom, it will become a call to `raise(atom, [])`.
If anything else, it will just raise the given exception.

Examples
raise "Given values do not match"

try do
  1 + :foo
rescue
  x in [ArithmeticError] ->
    IO.puts "that was expected"
    raise x
end

raise(exception, attrs) (macro)
 Raises an exception.
Calls `.exception` on the given argument passing the attributes in order to retrieve the appropriate exception structure.

Any module defined via `defexception/1` automatically implements `exception(attrs)` callback expected by `raise/2`.

Examples
iex> raise(ArgumentError, message: "Sample")
** (ArgumentError) Sample
reraise(msg, stacktrace)(macro)

Raises an exception preserving a previous stacktrace.

Works like raise/1 but does not generate a new stacktrace.

Notice that System.stacktrace returns the stacktrace of the last exception. That said, it is common to assign the stacktrace as the first expression inside a rescue clause as any other exception potentially raised (and rescued) in between the rescue clause and the raise call may change the System.stacktrace value.

Examples

```elixir
try do
  raise "Oops"
rescue
  exception ->
    stacktrace = System.stacktrace
    if Exception.message(exception) == "Oops" do
      reraise exception, stacktrace
    end
end
```

reraise(exception, attrs, stacktrace)(macro)

Raises an exception preserving a previous stacktrace.

Works like raise/2 but does not generate a new stacktrace.

See reraise/2 for more details.

Examples

```elixir
try do
  raise "Oops"
rescue
  exception ->
    stacktrace = System.stacktrace
    reraise WrapperError, [exception: exception], stacktrace
end
```

sigil_C(arg1, list2)(macro)

Handles the sigil ~C. It simply returns a char list without escaping characters and without interpolations.

Examples

```
iex> ~C(foo)
'foo'
iex> ~C(f#{o}o)
'f\#{o}o'
```

sigil_R(arg1, options)(macro)

Handles the sigil ~R. It returns a Regex pattern without escaping nor interpreting interpolations.

Examples

```
iex> Regex.match?(~R(f#{1,3}o), "f#o")
true
```

sigil_S(string, list2)(macro)

Handles the sigil ~S. It simply returns a string without escaping characters and without interpolations.

Examples

```
```
iex> ~S(foo)
"foo"

iex> ~S(f#{o}o)
"f\#{o}o"

**sigil_W(arg1, modifiers)(macro)**
Handles the sigil ~W. It returns a list of “words” split by whitespace without escaping nor interpreting interpolations.

**Modifiers**
- `s`: strings (default)
- `a`: atoms
- `c`: char lists

**Examples**
```iex
iex> ~W(foo #{bar} baz)
["foo", "\#{bar}", "baz"]
```

**sigil_c(arg1, list2)(macro)**
Handles the sigil ~c. It returns a char list as if it were a single quoted string, unescaping characters and replacing interpolations.

**Examples**
```iex
iex> ~c(foo)
'foo'

iex> ~c(f#{:o}o)
'foo'
```

**sigil_r(arg1, options)(macro)**
Handles the sigil ~r. It returns a Regex pattern.

**Examples**
```iex
iex> Regex.match?(~r(foo), "foo")
true
```

**sigil_s(arg1, list2)(macro)**
Handles the sigil ~s. It returns a string as if it was double quoted string, unescaping characters and replacing interpolations.

**Examples**
```iex
iex> ~s(foo)
"foo"

iex> ~s(f#{:o}o)
"foo"
```

**sigil_w(arg1, modifiers)(macro)**
Handles the sigil ~w. It returns a list of “words” split by whitespace.

**Modifiers**
- `s`: strings (default)
- `a`: atoms
•c: char lists  

Examples

iex> ~w(foo #{:bar} baz)
["foo", "bar", "baz"]

iex> ~w(--source test/enum_test.exs)
["--source", "test/enum_test.exs"]

iex> ~w(foo bar baz)
[:foo, :bar, :baz]

to_char_list(arg) (macro)
Convert the argument to a list according to the List.Chars protocol.

Examples

iex> to_char_list(:foo)
'foo'

to_string(arg) (macro)
Converts the argument to a string according to the String.Chars protocol. This is the function invoked when there is string interpolation.

Examples

iex> to_string(:foo)
"foo"

unless(clause, options) (macro)
Evaluates and returns the do-block passed in as a second argument unless clause evaluates to true. Returns nil otherwise. See also if.

Examples

iex> unless(Enum.empty?([]), do: "Hello")
nil

iex> unless(Enum.empty?([1,2,3]), do: "Hello")
"Hello"

use(module, opts = \[])(macro)
use is a simple mechanism for using a given module into the current context.

Examples

For example, in order to write tests using the ExUnit framework, a developer should use the ExUnit.Case module:

```elixir
defmodule AssertionTest do
  use ExUnit.Case, async: true

  test "always pass" do
    assert true
  end
end
```

By calling use, a hook called __using__ will be invoked in ExUnit.Case which will then do the proper setup.

Simply put, use is simply a translation to:
Elixir Documentation, Release

```elixir
defmodule AssertionTest do
  require ExUnit.Case
  ExUnit.Case.__using__([async: true])

  test "always pass" do
    assert true
  end
end
```

`var!(var, context \ nil)` (macro)
When used inside quoting, marks that the variable should not be hygienized. The argument can be either a variable unquoted or an atom representing the variable name.

Check `Kernel.SpecialForms.quote/2` for more information.

`left |> right` (macro)

|> is the pipe operator.

This operator introduces the expression on the left as the first argument to the function call on the right.

**Examples**

```elixir
iex> [1, [2], 3] |> List.flatten()
[1, 2, 3]
```

The example above is the same as calling `List.flatten([1, [2], 3])`, i.e. the argument on the left side of |> is introduced as the first argument of the function call on the right side.

This pattern is mostly useful when there is a desire to execute a bunch of operations, resembling a pipeline:

```elixir
iex> [1, [2], 3] |> List.flatten |> Enum.map(fn x -> x * 2 end)
[2, 4, 6]
```

The example above will pass the list to `List.flatten/1`, then get the flattened list and pass to `Enum.map/2`, which will multiply each entry in the list per two.

In other words, the expression above simply translates to:

`Enum.map(List.flatten([1, [2], 3]), fn x -> x * 2 end)`

Beware of operator precedence when using the pipe operator. For example, the following expression:

```elixir
String.graphemes "Hello" |> Enum.reverse
```

Translates to:

`String.graphemes("Hello" |> Enum.reverse)`

Which will result in an error as Enumerable protocol is not defined for binaries. Adding explicit parenthesis resolves the ambiguity:

`String.graphemes("Hello") |> Enum.reverse`

Or, even better:

"Hello" |> String.graphemes |> Enum.reverse

`left || right` (macro)

Provides a short-circuit operator that evaluates and returns the second expression only if the first one does not evaluate to true (i.e. it is either nil or false). Returns the first expression otherwise.

**Examples**
iex> Enum.empty?([1]) || Enum.empty?([1])
false

iex> List.first([]) || true
true

iex> Enum.empty?([1]) || 1
1

iex> Enum.empty?([]) || throw(:bad)
true

Notice that, unlike Erlang’s or operator, this operator accepts any expression as an argument, not only booleans, however it is not allowed in guards.

Kernel.ParallelCompiler

Overview  A module responsible for compiling files in parallel.

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>files/2</td>
<td>Compiles the given files</td>
</tr>
<tr>
<td>files_to_path/3</td>
<td>Compiles the given files to the given path. Read files/2 for more information</td>
</tr>
</tbody>
</table>

Functions

files(files, callbacks \ [[])(function)
Compiles the given files.

Those files are compiled in parallel and can automatically detect dependencies between them. Once a dependency is found, the current file stops being compiled until the dependency is resolved.

If there is an error during compilation or if warnings_as_errors is set to true and there is a warning, this function will fail with an exception.

This function receives a set of callbacks as options:

- :each_file - for each file compiled, invokes the callback passing the file
- :each_module - for each module compiled, invokes the callback passing the file, module and the module bytecode

The compiler doesn’t care about the return values of the callbacks. Returns the modules generated by each compiled file.

files_to_path(files, path, callbacks \ [[])(function)
Compiles the given files to the given path. Read files/2 for more information.

Kernel.ParallelRequire

Overview  A module responsible for requiring files in parallel.

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>files/2</td>
<td>Requires the given files</td>
</tr>
</tbody>
</table>
Files

defs(files, callback \ fn x -> x end)(function)

Requires the given files.

A callback that is invoked every time a file is required can be optionally given as argument.

Returns the modules generated by each required file.

Kernel.SpecialForms

Overview

In this module we define Elixir special forms. Special forms cannot be overridden by the developer and are the basic building blocks of Elixir code.

Some of those forms are lexical (like alias, case, etc). The macros { } and <<< are also special forms used to define tuple and binary data structures respectively.

This module also documents Elixir’s pseudo variables (__ENV__, __MODULE__, __DIR__ and __CALLER__). Pseudo variables return information about Elixir’s compilation environment and can only be read, never assigned to.

Finally, it also documents 2 special forms, __block__ and __aliases__, which are not intended to be called directly by the developer but they appear in quoted contents since they are essential in Elixir’s constructs.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%/2</td>
<td>Creates a struct</td>
</tr>
<tr>
<td>%}/1</td>
<td>Creates a map</td>
</tr>
<tr>
<td>&amp;/1</td>
<td>Captures or creates an anonymous function</td>
</tr>
<tr>
<td>./2</td>
<td>Defines a remote call or an alias</td>
</tr>
<tr>
<td>&lt;&lt;&lt;&gt;/1</td>
<td>Defines a new bitstring</td>
</tr>
<tr>
<td>^/1</td>
<td>Accesses an already bound variable in match clauses</td>
</tr>
<tr>
<td><strong>DIR</strong>/0</td>
<td>Returns the current directory as a binary</td>
</tr>
<tr>
<td><strong>ENV</strong>/0</td>
<td>Returns the current environment information as a Macro.Env struct</td>
</tr>
<tr>
<td><strong>MODULE</strong>/0</td>
<td>Returns the current module name as an atom or nil otherwise</td>
</tr>
<tr>
<td><strong>aliases</strong>/1</td>
<td>Internal special form to hold aliases information</td>
</tr>
<tr>
<td><strong>block</strong>/1</td>
<td>Internal special form for block expressions</td>
</tr>
<tr>
<td>alias/2</td>
<td>alias is used to setup aliases, often useful with modules names</td>
</tr>
<tr>
<td>case/2</td>
<td>Matches the given expression against the given clauses</td>
</tr>
<tr>
<td>fn/1</td>
<td>Defines an anonymous function</td>
</tr>
<tr>
<td>for/1</td>
<td>Comprehensions allow you to quickly build a data structure from an enumerable or a bitstring</td>
</tr>
<tr>
<td>import/2</td>
<td>Imports function and macros from other modules</td>
</tr>
<tr>
<td>quote/2</td>
<td>Gets the representation of any expression</td>
</tr>
<tr>
<td>receive/1</td>
<td>Checks if there is a message matching the given clauses in the current process mailbox</td>
</tr>
<tr>
<td>require/2</td>
<td>Requires a given module to be compiled and loaded</td>
</tr>
<tr>
<td>super/1</td>
<td>Calls the overridden function when overriding it with defoverridable. See Kernel.\texttt{defoverridable} for more information and documentation</td>
</tr>
<tr>
<td>try/1</td>
<td>Evaluate the given expressions and handle any error, exit or throw that may have happened</td>
</tr>
<tr>
<td>unquote/1</td>
<td>Unquotes the given expression from inside a macro</td>
</tr>
<tr>
<td>unquote_split/1</td>
<td>Unquotes the given list expanding its arguments. Similar to unquote</td>
</tr>
</tbody>
</table>

Macros

%(macro)

Creates a struct.

A struct is a tagged map that allows developers to provide default values for keys, tags to be used in polymorphic dispatches and compile time assertions.
To define a struct, you just need to implement the `__struct__/0` function in a module:

```elixir
defmodule User do
def __struct__ do
  %{name: "José", age: 27}
end
end
```

In practice though, structs are usually defined with the `Kernel.defstruct/2` macro:

```elixir
defmodule User do
defstruct name: "José", age: 27 end
end
```

Now a struct can be created as follow:

```elixir
%User{}
```

Underneath a struct is just a map with a `__struct__` field pointing to the User module:

```elixir
%User{} == %{__struct__: User, name: "José", age: 27}
```

A struct also validates the given keys are part of the defined struct. The example below will fail because there is no key `:full_name` in the user struct:

```elixir
%User{full_name: "José Valim"}
```

Note that a struct specifies a minimum set of keys required for operations. Other keys can be added to structs via the regular map operations:

```elixir
user = %User{}
Map.put(user, :a_non_struct_key, :value)
```

An update operation specific for structs is also available:

```elixir
%User{user | age: 28}
```

The syntax above will guarantee the given keys are valid at compilation time and it will guarantee at runtime the given argument is a struct, failing with `BadStructError` otherwise.

Although structs are maps, by default structs do not implement any of the protocols implemented for maps. Check `Kernel.defprotocol/2` for more information on how structs can be used with protocols for polymorphic dispatch. Also see `Kernel.struct/2` for examples on how to create and update structs dynamically.

`%{}` (macro)

Creates a map.

Maps are key-value stores where keys are compared using the match operator (`===`). Maps can be created with the `%{}` special form where keys are associated via `=>`:

```elixir
%{1 => 2}
```

Maps also support the keyword notation, as other special forms, as long as they are at the end of the argument list:

```elixir
%{hello: :world, with: :keywords}
%{:hello => :world, with: :keywords}
```

If a map has duplicated keys, the last key will always have higher precedence:
Conveniences for manipulating maps can be found in the `Map` module.

Access syntax

Besides the access functions available in the `Map` module, like `Map.get/3` and `Map.fetch/2`, a map can be accessed using the `. ` operator:

```iex
map = %{a: :b}
map.a
```

Note that the `. ` operator expects the field to exist in the map. If not, an `ArgumentError` is raised.

Update syntax

Maps also support an update syntax:

```iex
map = %{a => :b}
%{map | :a => :c}
```

Notice the update syntax requires the given keys to exist. Trying to update a key that does not exist will raise an `ArgumentError`.

AST representation

Regardless if `=>` or the keywords syntax is used, Maps are always represented internally as a list of two-items tuples for simplicity:

```iex
quote do: %{a => :b, c: :d}
```

Captures or creates an anonymous function.

Capture

The capture operator is most commonly used to capture a function with given name and arity from a module:

```iex
fun = &Kernel.is_atom/1
fun.(:atom)
```

In the example above, we captured `Kernel.is_atom/1` as an anonymous function and then invoked it.

The capture operator can also be used to capture local functions, including private ones, and imported functions by omitting the module name:

```iex
&local_function/1
```

Anonymous functions

The capture operator can also be used to partially apply functions, where `&1, &2` and so on can be used as value placeholders. For example:

```iex
double = &(x * 2)
double.(2)
```

4
In other words, \( \& (\&1 \ast 2) \) is equivalent to \( \text{fn } x \rightarrow x \ast 2 \text{ end} \). Another example using a local function:

```elixir
iex> fun = \&is_atom(\&1)
iex> fun.(\&atom)
true
```

The \( \& \) operator can be used with more complex expressions:

```elixir
iex> fun = \&(\&1 + \&2 + \&3)
iex> fun.(1, 2, 3)
6
```

As well as with lists and tuples:

```elixir
iex> fun = \&{\&1, \&2}
iex> fun.(1, 2)
{1, 2}

iex> fun = \&{\&1|\&2}
iex> fun.(1, 2)
[1|2]
```

The only restrictions when creating anonymous functions is that at least one placeholder must be present, i.e. it must contain at least \( \&1 \):

```elixir
# No placeholder fails to compile
\&var

# Block expressions are also not supported
\&{foo(\&1, \&2); \&3 + \&4}
```

**left . right (macro)**

Defines a remote call or an alias.

The dot (.) in Elixir can be used for remote calls:

```elixir
iex> String.downcase("FOO")
"foo"
```

In this example above, we have used . to invoke `downcase` in the `String` alias, passing “FOO” as argument. We can also use the dot for creating aliases:

```elixir
iex> Hello.World
Hello.World
```

This time, we have joined two aliases, defining the final alias `Hello.World`.

**Syntax**

The right side of . may be a word starting in upcase, which represents an alias, a word starting with lowercase or underscore, any valid language operator or any name wrapped in single- or double-quotes. Those are all valid examples:

```elixir
iex> Kernel.Sample
Kernel.Sample

iex> Kernel.length([1,2,3])
3

iex> Kernel.+(1, 2)
3
```
Elixir Documentation, Release

```iex
iex> Kernel."length"([1,2,3])
3

iex> Kernel.‘+’(1, 2)
3

Note that Kernel."HELLO" will be treated as a remote call and not an alias. This choice was done so every time single- or double-quotes are used, we have a remote call irregardless of the quote contents. This decision is also reflected in the quoted expressions discussed below.

Runtime (dynamic) behaviour

The result returned by . is always specified by the right-side:

```iex
x = String
x.downcase("FOO")
"foo"
x.Sample
String.Sample
```

In case the right-side is also dynamic, .‘s behaviour can be reproduced at runtime via apply/3 and Module.concat/2:

```iex
apply(:erlang, :+, [1,2])
3

Module.concat(Kernel, Sample)
Kernel.Sample
```

Quoted expression

When . is used, the quoted expression may take two distinct forms. When the right side starts with a lowercase letter (or underscore):

```iex
quote do: String.downcase("FOO")
{{:, []}, [#{:__aliases__, [alias: false], [:String]}, :downcase]}, [], ["FOO"]}
```

Notice we have an inner tuple, containing the atom :. representing the dot as first element:

{{:, []}, [#{:__aliases__, [alias: false], [:String]}, :downcase]}

This tuple follows the general quoted expression structure in Elixir, with the name as first argument, some keyword list as metadata as second, and the number of arguments as third. In this case, the arguments is the alias String and the atom :downcase. The second argument is always an atom:

```iex
quote do: String."downcase"("FOO")
{{:, []}, [#{:__aliases__, [alias: false], [:String]}, :downcase]}, [], ["FOO"]}
```

The tuple containing :. is wrapped in another tuple, which actually represents the function call, and has "FOO" as argument.

When the right side is an alias (i.e. starts with uppercase), we get instead:

```iex
quote do: Hello.World
#{:__aliases__, [alias: false], [:Hello, :World]}
```

We got into more details about aliases in the __aliases__ special form documentation.

Unquoting

We can also use unquote to generate a remote call in a quoted expression:
Elixir Documentation, Release

```elixir
define(x) do
  String.downcase(x)
end
```

Similar to `Kernel."HELLO", unquote(x)` will always generate a remote call, independent of the value of `x`. To generate an alias via the quoted expression, one needs to rely on `Module.concat/2`:

```elixir
define(x) do
  Module.concat(String, unquote(x))
end
```

<<args>>>(macro)

Defines a new bitstring.

**Examples**

```elixir
<< 1, 2, 3 >>
```

**Bitstring types**

A bitstring is made of many segments. Each segment has a type, which defaults to integer:

```elixir
<<1, 2, 3>>
```

Elixir also accepts by default the segment to be a literal string or a literal char list, which are by expanded to integers:

```elixir
<<0, "foo">>
```

Any other type needs to be explicitly tagged. For example, in order to store a float type in the binary, one has to do:

```elixir
<<3.14 :: float>>
```

This also means that variables need to be explicitly tagged, otherwise Elixir defaults to integer:

```elixir
rest = "oo"
define <<102, rest>>
```

**ArgumentError** argument error

We can solve this by explicitly tagging it as a binary:

```elixir
<<102, rest :: binary>>
```

The type can be integer, float, bitstring/bits, binary/bytes, utf8, utf16 or utf32, e.g.:

```elixir
<<102 :: float, rest :: binary>>
```

An integer can be any arbitrary precision integer. A float is an IEEE 754 binary32 or binary64 floating point number. A bitstring is an arbitrary series of bits. A binary is a special case of bitstring that has a total size divisible by 8.

The utf8, utf16, and utf32 types are for UTF code points. They can also be applied to literal strings and char lists:
The bits type is an alias for bitstring. The bytes type is an alias for binary.

The signedness can also be given as signed or unsigned. The signedness only matters for matching. If unspecified, it defaults to unsigned. Example:

```
iex> <-100 :: signed, _rest :: binary> = <<-100, "foo">>
<<156,102,111,111>>
```

This match would have failed if we did not specify that the value -100 is signed. If we’re matching into a variable instead of a value, the signedness won’t be checked; rather, the number will simply be interpreted as having the given (or implied) signedness, e.g.:

```
iex> <<val, _rest :: binary>> = <<-100, "foo>>
iex> val
156
```

Here, `val` is interpreted as unsigned.

Signedness is only relevant on integers.

The endianness of a segment can be big, little or native (the latter meaning it will be resolved at VM load time). Passing many options can be done by giving a list:

```
<<102 :: [integer, native], rest :: binary>>
```

Or:

```
<<102 :: [unsigned, big, integer], rest :: binary>>
```

And so on.

Endianness only makes sense for integers and some UTF code point types (utf16 and utf32).

Finally, we can also specify size and unit for each segment. The unit is multiplied by the size to give the effective size of the segment:

```
iex> <<102, _rest :: [size(2), unit(8)]>> = "foo"
"foo"
iex> <<102, _rest :: size(16)>> = "foo"
"foo"
iex> <<102, _rest :: size(32)>> = "foo"
** (MatchError) no match of right hand side value: "foo"
```

In the example above, the first two expressions matches because the string “foo” takes 24 bits and we are matching against a segment of 24 bits as well, 8 of which are taken by the integer 102 and the remaining 16 bits are specified on the rest. On the last example, we expect a rest with size 32, which won’t match.

Size and unit are not applicable to utf8, utf16, and utf32.

The default size for integers is 8. For floats, it is 64. For binaries, it is the size of the binary. Only the last binary in a binary match can use the default size (all others must have their size specified explicitly). Bitstrings do not have a default size.

Size can also be specified using a syntax shortcut. Instead of writing `size(8)`, one can write just 8 and it will be interpreted as `size(8)`.
iex> << 1 :: 3 >> == << 1 :: size(3)>>
true

The default unit for integers, floats, and bitstrings is 1. For binaries, it is 8.
For floats, unit * size must result in 32 or 64, corresponding to binary32 and binary64, respectively.

^var(macro)
Accesses an already bound variable in match clauses.

Examples
Elixir allows variables to be rebound via static single assignment:

iex> x = 1
iex> x = 2
iex> x
2

However, in some situations, it is useful to match against an existing value, instead of rebinding. This can be done with the ^ special form:

iex> x = 1
iex> ^x = List.first([1])
iex> ^x = List.first([2])
** (MatchError) no match of right hand side value: 2

Note that ^ always refers to the value of x prior to the match. The following example will match:

iex> x = 0
iex> {x, ^x} = {1, 0}
iex> x
1

__DIR__(macro)
Returns the current directory as a binary.

Although the directory can be accessed as Path.dirname(__ENV__.file), this macro is a convenient shortcut.

__ENV__(macro)
Returns the current environment information as a Macro.Env struct.

In the environment you can access the current filename, line numbers, set up aliases, the current function and others.

__MODULE__(macro)
Returns the current module name as an atom or nil otherwise.

Although the module can be accessed in the __ENV__, this macro is a convenient shortcut.

__aliases__(args)(macro)
Internal special form to hold aliases information.

It is usually compiled to an atom:

iex> quote do: Foo.Bar
{:__aliases__, [alias: false], [Foo, :Bar]}

Elixir represents Foo.Bar as __aliases__ so calls can be unambiguously identified by the operator ::.
For example:
Whenever an expression iterator sees a `. as the tuple key, it can be sure that it represents a call and the second argument in the list is an atom.

On the other hand, aliases holds some properties:

1. The head element of aliases can be any term;
2. The tail elements of aliases are guaranteed to always be atoms;
3. When the head element of aliases is the atom :Elixir, no expansion happen;
4. When the head element of aliases is not an atom, it is expanded at runtime:

   ```iex
   quote do: some_var.Foo {:
   aliases, [], [[:some_var, [], Elixir], :Foo]}
   ```

Since `some_var` is not available at compilation time, the compiler expands such expression to:

```iex
Module.concat [some_var, Foo]
```

__block__(args)(macro)

Internal special form for block expressions.

This is the special form used whenever we have a block of expressions in Elixir. This special form is private and should not be invoked directly:

```iex
quote do: (1; 2; 3)
{:__block__, [], [1, 2, 3]}
```

alias(module, opts)(macro)

`alias` is used to setup aliases, often useful with modules names.

**Examples**

`alias` can be used to setup an alias for any module:

```iex
defmodule Math do
  alias MyKeyword, as: Keyword
end
```

In the example above, we have set up `MyKeyword` to be aliased as `Keyword`. So now, any reference to `Keyword` will be automatically replaced by `MyKeyword`.

In case one wants to access the original `Keyword`, it can be done by accessing `Elixir`:

```iex
Keyword.values #=> uses MyKeyword.values
Elixir.Keyword.values #=> uses Keyword.values
```

Notice that calling `alias` without the `as:` option automatically sets an alias based on the last part of the module. For example:

```iex
alias Foo.Bar.Baz
```

Is the same as:

```iex
alias Foo.Bar.Baz, as: Baz
```

**Lexical scope**

`import`, `require` and `alias` are called directives and all have lexical scope. This means you can set up aliases inside specific functions and it won’t affect the overall scope.

**Warnings**
If you alias a module and you don’t use the alias, Elixir is going to issue a warning implying the alias is not being used.

In case the alias is generated automatically by a macro, Elixir won’t emit any warnings though, since the alias was not explicitly defined.

Both warning behaviours could be changed by explicitly setting the :warn option to true or false.

case(condition, blocks)(macro)
Matches the given expression against the given clauses.

Examples

```elixir
case thing do
  {:selector, i, value} when is_integer(i) ->
    value
  value ->
    value
end
```

In the example above, we match thing against each clause “head” and execute the clause “body” corresponding to the first clause that matches. If no clause matches, an error is raised.

Variables handling

Notice that variables bound in a clause “head” do not leak to the outer context:

```elixir
case data do
  {:ok, value} -> value
  :error -> nil
end
value
```

However, variables explicitly bound in the clause “body” are accessible from the outer context:

```elixir
value = 7

case lucky? do
  false -> value = 13
  true -> true
end
value
```

In the example above, value is going to be 7 or 13 depending on the value of lucky?. In case value has no previous value before case, clauses that do not explicitly bind a value have the variable bound to nil.

fn [clauses] end(macro)
Defines an anonymous function.

Examples

```elixir
iex> add = fn a, b -> a + b end
iex> add.(1, 2)
3
```

for(args)(macro)
Comprehensions allow you to quickly build a data structure from an enumerable or a bitstring.

Let’s start with an example:
iex> for n <- [1, 2, 3, 4], do: n * 2
[2, 4, 6, 8]

A comprehension accepts many generators and filters. Enumerable generators are defined using <-:

# A list generator:
iex> for n <- [1, 2, 3, 4], do: n * 2
[2, 4, 6, 8]

# A comprehension with two generators
iex> for x <- [1, 2], y <- [2, 3], do: x*y
[2, 3, 4, 6]

Filters can also be given:

# A comprehension with a generator and a filter
iex> for n <- [1, 2, 3, 4, 5, 6], rem(n, 2) == 0, do: n
[2, 4, 6]

Note generators can also be used to filter as it removes any value that doesn’t match the left side of <-:

iex> for {:user, name} <- [user: "jose", admin: "john", user: "eric"] do
...>   String.upcase(name)
...> end
["JOSE", "ERIC"]

Bitstring generators are also supported and are very useful when you need to organize bitstring streams:

iex> pixels = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234, 32, 15>>
iex> for <<r::8, g::8, b::8 <- pixels>>, do: {r, g, b}
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]

Variable assignments inside the comprehension, be it in generators, filters or inside the block, are not reflected outside of the comprehension.

Into

In the examples above, the result returned by the comprehension was always a list. The returned result can be configured by passing an :into option, that accepts any structure as long as it implements the Collectable protocol.

For example, we can use bitstring generators with the :into option to easily remove all spaces in a string:

iex> for <<c <- " hello world ">>, c != ?\s, into: "", do: <<c>>
"helloworld"

The IO module provides streams, that are both Enumerable and Collectable, here is an upcase echo server using comprehensions:

for line <- IO.stream(:stdio, :line), into: IO.stream(:stdio, :line) do
  String.upcase(line)
end

import (module, opts)(macro)

Imports function and macros from other modules.

import allows one to easily access functions or macros from others modules without using the qualified name.

Examples

If you are using several functions from a given module, you can import those functions and reference them as local functions, for example:
iex> import List
iex> flatten([1, [2], 3])
[1,2,3]

Selector

By default, Elixir imports functions and macros from the given module, except the ones starting with underscore (which are usually callbacks):

```
import List
```

A developer can filter to import only macros or functions via the only option:

```
import List, only: :functions
import List, only: :macros
```

Alternatively, Elixir allows a developer to pass pairs of name/arities to :only or :except as a fine grained control on what to import (or not):

```
import List, only: [flatten: 1]
import String, except: [split: 2]
```

Notice that calling except for a previously declared import simply filters the previously imported elements. For example:

```
import List, only: [flatten: 1, keyfind: 3]
import List, except: [flatten: 1]
```

After the two import calls above, only List.keyfind/3 will be imported.

Lexical scope

It is important to notice that import is lexical. This means you can import specific macros inside specific functions:

```
defmodule Math do
  def some_function do
    # 1) Disable 'if/2' from Kernel
    import Kernel, except: [if: 2]

    # 2) Require the new 'if' macro from MyMacros
    import MyMacros

    # 3) Use the new macro
    if do_something, it_works
  end
end
```

In the example above, we imported macros from MyMacros, replacing the original if/2 implementation by our own within that specific function. All other functions in that module will still be able to use the original one.

Warnings

If you import a module and you don’t use any of the imported functions or macros from this module, Elixir is going to issue a warning implying the import is not being used.

In case the import is generated automatically by a macro, Elixir won’t emit any warnings though, since the import was not explicitly defined.

Both warning behaviours could be changed by explicitly setting the :warn option to true or false.

Ambiguous function/macro names
If two modules \texttt{A} and \texttt{B} are imported and they both contain a \texttt{foo} function with an arity of \texttt{1}, an error is only emitted if an ambiguous call to \texttt{foo/1} is actually made; that is, the errors are emitted lazily, not eagerly.

\textbf{quote}(\texttt{opts, block})(\texttt{macro})

Gets the representation of any expression.

\textbf{Examples}

\begin{verbatim}
quote do: sum(1, 2, 3)
#=> {:sum, [], [1, 2, 3]}
\end{verbatim}

\textbf{Explanation}

Any Elixir code can be represented using Elixir data structures. The building block of Elixir macros is a tuple with three elements, for example:

\begin{verbatim}
{:sum, [], [1, 2, 3]}
\end{verbatim}

The tuple above represents a function call to \texttt{sum} passing 1, 2 and 3 as arguments. The tuple elements are:

- The first element of the tuple is always an atom or another tuple in the same representation;
- The second element of the tuple represents metadata;
- The third element of the tuple are the arguments for the function call. The third argument may be an atom, which is usually a variable (or a local call);

\textbf{Options}

- \texttt{:unquote} - When false, disables unquoting. Useful when you have a quote inside another quote and want to control what quote is able to unquote;
- \texttt{:location} - When set to :keep, keeps the current line and file from quote. Read the Stacktrace information section below for more information;
- \texttt{:context} - Sets the resolution context;
- \texttt{:bind_quoted} - Passes a binding to the macro. Whenever a binding is given, \texttt{unquote} is automatically disabled;

\textbf{Quote literals}

Besides the tuple described above, Elixir has a few literals that when quoted return themselves. They are:

\begin{verbatim}
:sum    #=> Atoms
1       #=> Integers
2.0     #=> Floats
[1, 2]  #=> Lists
"strings" #=> Strings
(key, value) #=> Tuples with two elements
\end{verbatim}

\textbf{Quote and macros}

\texttt{quote} is commonly used with macros for code generation. As an exercise, let’s define a macro that multiplies a number by itself (squared). Note there is no reason to define such as a macro (and it would actually be seen as a bad practice), but it is simple enough that it allows us to focus on the important aspects of quotes and macros:

\begin{verbatim}
defmodule Math do
defmacro squared(x) do
  quote do
    unquote(x) * unquote(x)
  end
end
end
\end{verbatim}
We can invoke it as:

```elixir
import  Math
IO.puts "Got #{squared(5)}"
```

At first, there is nothing in this example that actually reveals it is a macro. But what is happening is that, at
compilation time, `squared(5)` becomes `5 * 5`. The argument `5` is duplicated in the produced code, we can
see this behaviour in practice though because our macro actually has a bug:

```elixir
import  Math
my_number = fn ->
  IO.puts "Returning 5"
  5
end
IO.puts "Got #{squared(my_number.())}"
```

The example above will print:

```
Returning 5
Returning 5
25
```

Notice how “Returning 5” was printed twice, instead of just once. This is because a macro receives an expression
and not a value (which is what we would expect in a regular function). This means that:

```
squared(my_number.())
```

Actually expands to:

```
my_number.() * my_number.()
```

Which invokes the function twice, explaining why we get the printed value twice! In the majority of the cases,
this is actually unexpected behaviour, and that’s why one of the first things you need to keep in mind when it
comes to macros is to not unquote the same value more than once.

Let’s fix our macro:

```elixir
defmodule Math do
  defmacro squared(x) do
    quote do
      x = unquote(x)
      x * x
    end
  end
end
```

Now invoking `squared(my_number.())` as before will print the value just once.

In fact, this pattern is so common that most of the times you will want to use the `bind_quoted` option with
quote:

```elixir
defmodule Math do
  defmacro squared(x) do
    quote bind_quoted: [x: x] do
      x * x
    end
  end
end
```

`:bind_quoted` will translate to the same code as the example above. `:bind_quoted` can be used in many
cases and is seen as good practice, not only because it helps us from running into common mistakes but also
because it allows us to leverage other tools exposed by macros, such as unquote fragments discussed in some sections below.

Before we finish this brief introduction, you will notice that, even though we defined a variable `x` inside our quote:

```elixir
quote do
  x = unquote(x)
  x * x
end
```

When we call:

```elixir
import Math
squared(5)
x #=> ** (RuntimeError) undefined function or variable: x
```

We can see that `x` did not leak to the user context. This happens because Elixir macros are hygienic, a topic we will discuss at length in the next sections as well.

**Hygiene in variables**

Consider the following example:

```elixir
defmodule Hygiene do
  defmacro no_interference do
    quote do
      :a = 1
    end
  end
end
```

```elixir
require Hygiene
a = 10
Hygiene.no_interference
a #=> 10
```

In the example above, `a` returns 10 even if the macro is apparently setting it to 1 because variables defined in the macro does not affect the context the macro is executed in. If you want to set or get a variable in the caller’s context, you can do it with the help of the `var!` macro:

```elixir
defmodule NoHygiene do
  defmacro interference do
    quote do
      var!(a) = 1
    end
  end
end
```

```elixir
require NoHygiene
a = 10
NoHygiene.interference
a #=> 1
```

Note that you cannot even access variables defined in the same module unless you explicitly give it a context:

```elixir
defmodule Hygiene do
  defmacro write do
    quote do
      a = 1
    end
  end
end
```

```elixir
require Hygiene
a = 10
Hygiene.write
a #=> 10
```
defmacro read do
  quote do
    a
  end
end

Hygiene.write
Hygiene.read
#=> ** (RuntimeError) undefined function or variable: a

For such, you can explicitly pass the current module scope as argument:

defmodule ContextHygiene do
  defmacro write do
    quote do
      var!(a, ContextHygiene) = 1
    end
  end
end

ContextHygiene.write
ContextHygiene.read
#=> 1

Hygiene in aliases

Aliases inside quote are hygienic by default. Consider the following example:

defmodule Hygiene do
  alias HashDict, as: D

  defmacro no_interference do
    quote do
      D.new
    end
  end
end

require Hygiene
Hygiene.no_interference #=> #HashDict<[]>

Notice that, even though the alias D is not available in the context the macro is expanded, the code above works because D still expands to HashDict.

Similarly, even if we defined an alias with the same name before invoking a macro, it won’t affect the macro’s result:

defmodule Hygiene do
  alias HashDict, as: D

  defmacro no_interference do
    quote do
      D.new
    end
  end
end
In some cases, you want to access an alias or a module defined in the caller. For such, you can use the `alias!` macro:

```elixir
defmodule Hygiene do
  defmacro no_interference do
    quote do: Nested.hello
  end

  defmacro interference do
    quote do: alias!(Nested).hello
  end
end
defmodule Parent do
  defmodule Nested do
    def hello, do: "world"
  end

  require Hygiene
  Hygiene.no_interference #=> ** (UndefinedFunctionError) ...
  Hygiene.interference #=> "world"
end
```

**Hygiene in imports**

Similar to aliases, imports in Elixir are hygienic. Consider the following code:

```elixir
defmodule Hygiene do
  defmacrop get_size do
    quote do
      size("hello")
    end
  end

  def return_size do
    import Kernel, except: [size: 1]
    get_size
  end
end

Hygiene.return_size #=> 5
```

Notice how `return_size` returns 5 even though the `size/1` function is not imported. In fact, even if `return_size` imported a function from another module, it wouldn’t affect the function result:
Calling this new `return_size` will still return 5 as result.

Elixir is smart enough to delay the resolution to the latest moment possible. So, if you call `size("hello")` inside quote, but no `size/1` function is available, it is then expanded in the caller:

```elixir
defmodule Lazy do
  defmacrop get_size do
    import Kernel, except: [size: 1]

    quote do
      size([a: 1, b: 2])
    end
  end

  def return_size do
    import Kernel, except: [size: 1]
    import Dict, only: [size: 1]
    get_size
  end
end

Lazy.return_size #=> 2
```

Stacktrace information

When defining functions via macros, developers have the option of choosing if runtime errors will be reported from the caller or from inside the quote. Let’s see an example:

```
# adder.ex
defmodule Adder do
  @doc "Defines a function that adds two numbers"
  defmacrop defadd do
    quote location: :keep do
      def add(a, b), do: a + b
    end
  end
end

# sample.ex
defmodule Sample do
  import Adder
  defadd
end

When using `location: :keep` and invalid arguments are given to `Sample.add/2`, the stacktrace information will point to the file and line inside the quote. Without `location: :keep`, the error is reported to where `defadd` was invoked. Note `location: :keep` affects only definitions inside the quote.

Binding and unquote fragments

Elixir quote/unquote mechanisms provides a functionality called unquote fragments. Unquote fragments provide an easy way to generate functions on the fly. Consider this example:

```elixir
kv = [foo: 1, bar: 2]
Enum.each kv, fn {k, v} ->
  def unquote(k)(), do: unquote(v)
end
```

In the example above, we have generated the functions `foo/0` and `bar/0` dynamically. Now, imagine that, we want to convert this functionality into a macro:
defmacro defkv(kv) do
  Enum.map kv, fn {k, v} ->
    quote do
      def unquote(k)(), do: unquote(v)
    end
  end
end

We can invoke this macro as:

defkv [foo: 1, bar: 2]

However, we can’t invoke it as follows:

e = [foo: 1, bar: 2]

defkv e

This is because the macro is expecting its arguments to be a keyword list at compilation time. Since in the
example above we are passing the representation of the variable kv, our code fails.

This is actually a common pitfall when developing macros. In practice, we want to avoid doing work at compilation
time as much as possible. That said, let’s attempt to improve our macro:

defmacro defkv(kv) do
  quote do
    Enum.each unquote(kv), fn {k, v} ->
      def unquote(k)(), do: unquote(v)
    end
  end
end

If you try to run our new macro, you will notice it won’t even compile, complaining that the variables k and v
does not exist. This is because of the ambiguity: unquote(k) can either be an unquote fragment, as
previously, or a regular unquote as in unquote(kv).

One solution to this problem is to disable unquoting in the macro, however, doing that would make it impossible
to inject the kv representation into the tree. That’s when the :bind_quoted option comes to the rescue
(again!). By using :bind_quoted, we can automatically disable unquoting while still injecting the desired
variables into the tree:

defmacro defkv(kv) do
  quote bind_quoted: [kv: kv] do
    Enum.each kv, fn {k, v} ->
      def unquote(k)(), do: unquote(v)
    end
  end
end

In fact, the :bind_quoted option is recommended every time one desires to inject a value into the quote.

receive(args)(macro)
Checks if there is a message matching the given clauses in the current process mailbox.

In case there is no such message, the current process hangs until a message arrives or waits until a given timeout
value.

Examples

receive do
  {:selector, i, value} when is_integer(i) ->
    value
value when is_atom(value) ->
  value
  _  -> IO.puts :stderr, "Unexpected message received"
end

An optional after clause can be given in case the message was not received after the specified period of time:

receive do
  {:selector, i, value} when is_integer(i) ->
    value
  value when is_atom(value) ->
    value
    _  -> IO.puts :stderr, "Unexpected message received"
  after 5000 ->
    IO.puts :stderr, "No message in 5 seconds"
end

The after clause can be specified even if there are no match clauses. There are two special cases for the timeout value given to after:

- • :infinity - The process should wait indefinitely for a matching message, this is the same as not using a timeout.
- • 0 - if there is no matching message in the mailbox, the timeout will occur immediately.

Variables handling

The receive special form handles variables exactly as the case special macro. For more information, check the docs for case/2.

require(module, opts)(macro)

Requires a given module to be compiled and loaded.

Examples

Notice that usually modules should not be required before usage, the only exception is if you want to use the macros from a module. In such cases, you need to explicitly require them.

Let’s suppose you created your own if implementation in the module MyMacros. If you want to invoke it, you need to first explicitly require the MyMacros:

```elixir
defmodule Math do
  require MyMacros
  MyMacros.if do_something, it_works
end```

An attempt to call a macro that was not loaded will raise an error.

Alias shortcut

require also accepts as: as an option so it automatically sets up an alias. Please check alias for more information.

super(args)(macro)

Calls the overriden function when overriding it with defoverridable. See Kernel.defoverridable for more information and documentation.

try(args)(macro)

Evaluate the given expressions and handle any error, exit or throw that may have happened.
Examples

```elixir
try do
  do_something_that_may_fail(some_arg)
rescue
  ArgumentError ->
    IO.puts "Invalid argument given"
catch
  value ->
    IO.puts "caught #{value}"
else
  value ->
    IO.puts "Success! The result was #{value}"
after
  IO.puts "This is printed regardless if it failed or succeed"
end
```

The rescue clause is used to handle exceptions, while the catch clause can be used to catch thrown values. The else clause can be used to control flow based on the result of the expression. Catch, rescue and else clauses work based on pattern matching.

Note that calls inside `try` are not tail recursive since the VM needs to keep the stacktrace in case an exception happens.

Rescue clauses

Besides relying on pattern matching, rescue clauses provides some conveniences around exceptions that allows one to rescue an exception by its name. All the following formats are valid rescue expressions:

```elixir
try do
  UndefinedModule.undefined_function
rescue
  UndefinedFunctionError -> nil
end

try do
  UndefinedModule.undefined_function
rescue
  [UndefinedFunctionError] -> nil
end

# rescue and bind to x
try do
  UndefinedModule.undefined_function
rescue
  x in [UndefinedFunctionError] -> nil
end

# rescue all and bind to x
try do
  UndefinedModule.undefined_function
rescue
  x -> nil
end
```

Erlang errors

Erlang errors are transformed into Elixir ones during rescue:

```elixir
try do
  :erlang.error(:badarg)
end
```
The most common Erlang errors will be transformed into their Elixir counter-part. Those which are not will be transformed into **ErlangError**:

```elixir
try do
  :erlang.error(:unknown)
rescue
  ErlangError -> :ok
end
```

In fact, **ErlangError** can be used to rescue any error that is not an Elixir error proper. For example, it can be used to rescue the earlier **:badarg** error too, prior to transformation:

```elixir
try do
  :erlang.error(:badarg)
rescue
  ErlangError -> :ok
end
```

**Catching throws and exits**

The catch clause can be used to catch throws values and exits.

```elixir
try do
  exit(1)
catch
  :exit, 1 -> IO.puts "Exited with 1"
end
```

```elixir
try do
  throw(:sample)
catch
  :throw, :sample ->
    IO.puts "sample thrown"
end
```

Catch values also support **:error**, as in Erlang, although it is commonly avoided in favor of raise/rescue control mechanisms.

**Else clauses**

Else clauses allow the result of the expression to be pattern matched on:

```elixir
x = 2
try do
  1 / x
rescue
  ArithmeticError ->
    :infinity
else
  y when y < 1 and y > -1 ->
    :small
  _ ->
    :large
end
```

If an else clause is not present the result of the expression will be return, if no exceptions are raised:
x = 1
"x =
  try do
    1 / x
  rescue
    ArithmeticError ->
      :infinity
  end

However when an else clause is present but the result of the expression does not match any of the patterns an exception will be raised. This exception will not be caught by a catch or rescue in the same try:

x = 1
try do
  try do
    1 / x
  rescue
    # The TryClauseError can not be rescued here:
    TryClauseError ->
      :error_a
  else
    0 ->
      :small
  end
rescue
    # The TryClauseError is rescued here:
    TryClauseError ->
      :error_b
end

Similarly an exception inside an else clause is not caught or rescued inside the same try:

try do
  try do
    nil
  catch
    # The exit(1) call below can not be caught here:
    :exit, _ ->
      :exit_a
  else
    _ ->
      exit(1)
  end
catch
    # The exit is caught here:
    :exit, _ ->
      :exit_b
end

This means the VM no longer needs to keep the stacktrace once inside an else clause and so tail recursion is possible when using `try` with a tail call as the final call inside an else clause. The same is true for rescue and catch clauses.

**Variable handling**

Since an expression inside `try` may not have been evaluated due to an exception, any variable created inside `try` cannot be accessed externally. For instance:

```
try do
  x = 1
```

```
```elixir
do_something_that_may_fail(same_arg)
  :ok
  catch
  _ = _ -> :failed
end
```

```
x  #=> unbound variable 'x'
```

In the example above, `x` cannot be accessed since it was defined inside the `try` clause. A common practice to address this issue is to return the variables defined inside `try`:

```elixir
x =
  try do
    x = 1
    do_something_that_may_fail(same_arg)
    x
  catch
    _ = _ -> :failed
end
```

### unquote(expr) (macro)

Unquotes the given expression from inside a macro.

**Examples**

Imagine the situation you have a variable `name` and you want to inject it inside some quote. The first attempt would be:

```elixir
value = 13
quote do: sum(1, value, 3)
```

Which would then return:

```
{:sum, [], [1, {:value, [], quoted}, 3]}
```

Which is not the expected result. For this, we use `unquote`:

```elixir
value = 13
quote do: sum(1, unquote(value), 3)
 #=> {:sum, [], [1, 13, 3]}
```

### unquote_splicing(expr) (macro)

Unquotes the given list expanding its arguments. Similar to `unquote`.

**Examples**

```elixir
values = [2, 3, 4]
quote do: sum(1, unquote_splicing(values), 5)
 #=> {:sum, [], [1, 2, 3, 4, 5]}
```

### {args} (macro)

Creates a tuple.

Only two item tuples are considered literals in Elixir. Therefore all other tuples are represented in the AST as a call to the special form `{}`.

Conveniences for manipulating tuples can be found in the `Tuple` module. Some functions for working with tuples are also available in `Kernel`, namely `Kernel.elem/2`, `Kernel.put_elem/3` and `Kernel.tuple_size/1`.

**Examples**
Elixir comes with a notation for declaring types and specifications. Elixir is dynamically typed, as such typespecs are never used by the compiler to optimize or modify code. Still, using typespecs is useful as documentation and tools such as Dialyzer can analyze the code with typespecs to find bugs.

The attributes @type, @opaque, @typep, @spec and @callback available in modules are handled by the equivalent macros defined by this module. See sub-sections “Defining a type” and “Defining a specification” below.

The type syntax provided by Elixir is fairly similar to the one in Erlang.

Most of the built-in types provided in Erlang (for example, pid()) are expressed the same way: pid() or simply pid. Parametrized types are also supported (list(integer)) and so are remote types (Enum.t).

Integers and atom literals are allowed as types (ex. 1, :atom or false). All other types are built of unions of predefined types. Certain shorthands are allowed, such as [...] and [ ... ].

**Predefined types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>the top type, the set of all terms</td>
</tr>
<tr>
<td>none</td>
<td>the bottom type, contains no terms</td>
</tr>
<tr>
<td>pid</td>
<td></td>
</tr>
<tr>
<td>port</td>
<td></td>
</tr>
<tr>
<td>reference</td>
<td></td>
</tr>
<tr>
<td>Atom</td>
<td></td>
</tr>
<tr>
<td>Bitstring</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>Fun</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
</tr>
<tr>
<td>Tuple</td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td></td>
</tr>
</tbody>
</table>
| UserDefined | # Described in section "Defining a type"

**Atom**

- atom
  - ElixirAtom # ':foo', ':bar', ...

**Bitstring**

- <<= M >>
  - M is a positive integer
- <<= _ :: _ * N >>
  - N is a positive integer

**Fun**

- (... -> any)
  - any function
- (... -> Type)
  - any arity, returning Type
- (() -> Type)
- (TList -> Type)

**Integer**

- integer
  - ElixirInteger # ...,
List :: list(Type) # proper list ([]-terminated)
| improper_list(Type1, Type2) # Type1=contents, Type2=termination
| maybe_improper_list(Type1, Type2) # Type1 and Type2 as above
| nonempty_list(Type) # proper non-empty list
| [] # empty list
| [Type] # shorthand for list(Type)
| [Type, ...] # shorthand for nonempty_list(Type)

Tuple :: tuple # a tuple of any size
| {} # empty tuple
| {TList}

TList :: Type
| Type, TList

Union :: Type1 | Type2

Bit strings  Bit string with a base size of 3:
<< _ :: 3 >>

Bit string with a unit size of 8:
<< _ :: _ * 8 >>

Anonymous functions  Any anonymous function:
((...) -> any)
(... -> any)

Anonymous function with arity of zero:
(() -> type)

Anonymous function with some arity:
((type, type) -> type)
(type, type -> type)
### Built-in types

<table>
<thead>
<tr>
<th>Built-in type</th>
<th>Defined as</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>any</td>
</tr>
<tr>
<td>binary</td>
<td>&lt;&lt; _ :: _ * 8 &gt;&gt;</td>
</tr>
<tr>
<td>bitstring</td>
<td>&lt;&lt; _ :: _ * 1 &gt;&gt;</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>byte</td>
<td>0..255</td>
</tr>
<tr>
<td>char</td>
<td>0..0xffff</td>
</tr>
<tr>
<td>number</td>
<td>integer</td>
</tr>
<tr>
<td>list</td>
<td>[any]</td>
</tr>
<tr>
<td>maybe_improper_list</td>
<td>maybe_improper_list(any, any)</td>
</tr>
<tr>
<td>nonempty_list</td>
<td>nonempty_list(any)</td>
</tr>
<tr>
<td>iodata</td>
<td>iolist</td>
</tr>
<tr>
<td>iolist</td>
<td>maybe_improper_list(byte</td>
</tr>
<tr>
<td>module</td>
<td>atom</td>
</tr>
<tr>
<td>mfa</td>
<td>{atom, atom, arity}</td>
</tr>
<tr>
<td>arity</td>
<td>0..255</td>
</tr>
<tr>
<td>node</td>
<td>atom</td>
</tr>
<tr>
<td>timeout</td>
<td>:infinity</td>
</tr>
<tr>
<td>no_return</td>
<td>none</td>
</tr>
<tr>
<td>fun</td>
<td>(... -&gt; any)</td>
</tr>
</tbody>
</table>

Some built-in types cannot be expressed with valid syntax according to the language defined above.

<table>
<thead>
<tr>
<th>Built-in type</th>
<th>Can be interpreted as</th>
</tr>
</thead>
<tbody>
<tr>
<td>non_neg_integer</td>
<td>0..</td>
</tr>
<tr>
<td>pos_integer</td>
<td>1..</td>
</tr>
<tr>
<td>neg_integer</td>
<td>..-1</td>
</tr>
</tbody>
</table>

Types defined in other modules are referred to as “remote types”, they are referenced as Module.type_name (ex. Enum.t or String.t).

### Defining a type

- **@type type_name :: type**
- **@typep type_name :: type**
- **@opaque type_name :: type**

A type defined with @typep is private. An opaque type, defined with @opaque is a type where the internal structure of the type will not be visible, but the type is still public.

Types can be parametrised by defining variables as parameters, these variables can then be used to define the type.

- **@type dict(key, value) :: [{key, value}]**

### Defining a specification

- **@spec function_name(type1, type2) :: return_type**
- **@callback function_name(type1, type2) :: return_type**

Callbacks are used to define the callbacks functions of behaviours (see Behaviour).

Guards can be used to restrict type variables given as arguments to the function.

- **@spec function(arg) :: [arg] when arg: atom**

Type variables with no restriction can also be defined.
Specifications can be overloaded just like ordinary functions.

@spec function(integer) :: atom
@spec function(atom) :: integer

Notes  Elixir discourages the use of type string as it might be confused with binaries which are referred to as “strings” in Elixir (as opposed to character lists). In order to use the type that is called string in Erlang, one has to use the char_list type which is a synonym for string. If you use string, you’ll get a warning from the compiler.

If you want to refer to the “string” type (the one operated on by functions in the String module), use String.t type instead.

Functions

beam_callbacks(module)(function)
Specs:

•beam_callbacks(module | binary) :: [tuple] | nil

Returns all callbacks available from the module’s beam code.

The result is returned as a list of tuples where the first element is spec name and arity and the second is the spec.

The module must have a corresponding beam file which can be located by the runtime system.

beam_specs(module)(function)
Specs:

•beam_specs(module | binary) :: [tuple] | nil

Returns all specs available from the module’s beam code.

The result is returned as a list of tuples where the first element is spec name and arity and the second is the spec.

The module must have a corresponding beam file which can be located by the runtime system.
beam_typedocs(module)(function)
   Specs:
      •beam_typedocs(module | binary) :: [tuple] | nil
   Returns all type docs available from the module’s beam code.
   The result is returned as a list of tuples where the first element is the pair of type name and arity and the second
   element is the documentation.
   The module must have a corresponding beam file which can be located by the runtime system.

beam_types(module)(function)
   Specs:
      •beam_types(module | binary) :: [tuple] | nil
   Returns all types available from the module’s beam code.
   The result is returned as a list of tuples where the first element is the type (:typep, :type and :opaque).
   The module must have a corresponding beam file which can be located by the runtime system.

define_callback(module, tuple, definition)(function)
   Defines a callback by receiving Erlang’s typespec.

define_spec(module, tuple, definition)(function)
   Defines a spec by receiving Erlang’s typespec.

define_type(caller, kind, type)(function)
   Defines a type, typep or opaque by receiving Erlang’s typespec.

defines_callback?(module, name, arity)(function)
   Returns true if the current module defines a callback. This function is only available for modules being
   compiled.

defines_spec?(module, name, arity)(function)
   Returns true if the current module defines a given spec. This function is only available for modules being
   compiled.

defines_type?(module, name, arity)(function)
   Returns true if the current module defines a given type (private, opaque or not). This function is only available
   for modules being compiled.

spec_to_ast(name, arg2)(function)
   Converts a spec clause back to Elixir AST.

type_to_ast(arg1)(function)
   Converts a type clause back to Elixir AST.

Macros

defcallback(spec)(macro)
   Defines a callback. This macro is responsible for handling the attribute @callback.

   Examples

      @callback add(number, number) :: number

defopaque(type)(macro)
   Defines an opaque type. This macro is responsible for handling the attribute @opaque.

   Examples
@opaque my_type :: atom

defspec (spec) (macro)
   Defines a spec. This macro is responsible for handling the attribute @spec.

    Examples
    @spec add(number, number) :: number

deftype (type) (macro)
   Defines a type. This macro is responsible for handling the attribute @type.

    Examples
    @type my_type :: atom

deftypep (type) (macro)
   Defines a private type. This macro is responsible for handling the attribute @typep.

    Examples
    @typep my_type :: atom

Keyword

Overview    A keyword is a list of tuples where the first element of the tuple is an atom and the second element can be any value.

A keyword may have duplicated keys so it is not strictly a dictionary. However most of the functions in this module behave exactly as a dictionary and mimic the API defined by the Dict behaviour.

For example, Keyword.get will get the first entry matching the given key, regardless if duplicated entries exist. Similarly, Keyword.put and Keyword.delete ensure all duplicated entries for a given key are removed when invoked.

A handful of functions exist to handle duplicated keys, in particular, from_enum allows creating a new keywords without removing duplicated keys, get_values returns all values for a given key and delete_first deletes just one of the existing entries.

Since a keyword list is simply a list, all the operations defined in Enum and List can also be applied.
### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>delete(keywords, key)</code></td>
<td>Deletes all entries in the keyword list for a specific <code>key</code>. If the <code>key</code> does not exist, returns the keyword list unchanged. Use <code>delete_first</code> to delete just the first entry in case of duplicated keys.</td>
</tr>
<tr>
<td><code>delete(keywords, key, value)</code></td>
<td>Deletes the entry in the keyword list for a <code>key</code> with <code>value</code>. If no <code>key</code> with <code>value</code> exists, returns the keyword list unchanged.</td>
</tr>
<tr>
<td><code>delete_first(keywords, key)</code></td>
<td>Deletes the first entry in the keyword list for a specific <code>key</code>. If the <code>key</code> does not exist, returns the keyword list unchanged.</td>
</tr>
<tr>
<td><code>drop(keywords)</code></td>
<td>Drops the given keys from the dict</td>
</tr>
<tr>
<td><code>equal?(keywords1, keywords2)</code></td>
<td>Checks if two keywords are equal. I.e. they contain the same keys and those keys contain the same values.</td>
</tr>
<tr>
<td><code>fetch!(key)</code></td>
<td>Fetches the value for specific <code>key</code>. If <code>key</code> does not exist, a <code>KeyError</code> is raised.</td>
</tr>
<tr>
<td><code>fetch(key)</code></td>
<td>Fetches the value for a specific <code>key</code> and returns it in a tuple. If the <code>key</code> does not exist, returns <code>:error</code>.</td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td>Gets the value for a specific <code>key</code></td>
</tr>
<tr>
<td><code>get_values(key)</code></td>
<td>Gets all values for a specific <code>key</code></td>
</tr>
<tr>
<td><code>has_key?(keywords, key)</code></td>
<td>Returns whether a given <code>key</code> exists in the given <code>keywords</code>.</td>
</tr>
<tr>
<td><code>keys(keywords)</code></td>
<td>Returns all keys from the keyword list. Duplicated keys appear duplicated in the final list of keys.</td>
</tr>
<tr>
<td><code>keyword?(term)</code></td>
<td>Checks if the given argument is a keyword list or not</td>
</tr>
<tr>
<td><code>merge(keywords1, keywords2)</code></td>
<td>Merges two keyword lists into one. If they have duplicated entries, the one given as second argument wins.</td>
</tr>
<tr>
<td><code>merge(keywords1, keywords2, conflicts)</code></td>
<td>Merges two keyword lists into one. If they have duplicated entries, the given function is invoked to solve conflicts.</td>
</tr>
<tr>
<td><code>new</code></td>
<td>Returns an empty keyword list, i.e. an empty list</td>
</tr>
<tr>
<td><code>new(keywords)</code></td>
<td>Creates a keyword from an enumerable</td>
</tr>
<tr>
<td><code>new(keywords, transformation_function)</code></td>
<td>Creates a keyword from an enumerable via the transformation function.</td>
</tr>
<tr>
<td><code>pop(keywords, key)</code></td>
<td>Returns the first value associated with <code>key</code> in the keyword list as well as the keyword list without that particular occurrence of <code>key</code></td>
</tr>
<tr>
<td><code>pop_first(keywords)</code></td>
<td>Returns the first value associated with <code>key</code> in the keyword list as well as the keyword list without that particular occurrence of <code>key</code></td>
</tr>
<tr>
<td><code>put(keywords, key)</code></td>
<td>Puts the given value under <code>key</code></td>
</tr>
<tr>
<td><code>put_new(keywords, key, value)</code></td>
<td>Puts the given value under <code>key</code> unless the entry <code>key</code> already exists.</td>
</tr>
<tr>
<td><code>split(keywords)</code></td>
<td>Takes all entries corresponding to the given keys and extracts them into a separate keyword list. Returns a tuple with the new list and the old list with removed keys.</td>
</tr>
<tr>
<td><code>take(keywords)</code></td>
<td>Takes all entries corresponding to the given keys and returns them in a new keyword list</td>
</tr>
<tr>
<td><code>update!(keywords, key)</code></td>
<td>Updates the <code>key</code> with the given function. If the <code>key</code> does not exist, raises <code>KeyError</code>.</td>
</tr>
<tr>
<td><code>update(keywords, key, function)</code></td>
<td>Updates the <code>key</code> with the given function. If the <code>key</code> does not exist, inserts the given initial value</td>
</tr>
<tr>
<td><code>values(keywords)</code></td>
<td>Returns all values from the keyword list</td>
</tr>
</tbody>
</table>

### Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>key</code></td>
<td>atom</td>
</tr>
<tr>
<td><code>value</code></td>
<td>any</td>
</tr>
<tr>
<td><code>t</code></td>
<td><code>[[key, value]]</code></td>
</tr>
</tbody>
</table>

### Functions

#### `delete(keywords, key)` (function)

Specs:
**delete(t, key)**:: t
Delete all entries in the keyword list for a specific `key`. If the `key` does not exist, returns the keyword list unchanged. Use `delete_first` to delete just the first entry in case of duplicated keys.

**Examples**

```iex
iex> Keyword.delete([a: 1, b: 2], :a)
    [b: 2]
iex> Keyword.delete([a: 1, b: 2, a: 3], :a)
    [a: 1, b: 2]
iex> Keyword.delete([b: 2], :a)
    [b: 2]
```

**delete(keywords, key, value)**(function)
Specs:

- **delete(t, key, value)**:: t
Delete the entry in the keyword list for a `key` with `value`. If no `key` with `value` exists, returns the keyword list unchanged.

**Examples**

```iex
iex> Keyword.delete([a: 1, b: 2], :a, 1)
    [b: 2]
iex> Keyword.delete([a: 1, b: 2, a: 3], :a, 3)
    [a: 1, b: 2]
iex> Keyword.delete([b: 2], :a, 5)
    [b: 2]
```

**delete_first(keywords, key)**(function)
Specs:

- **delete_first(t, key)**:: t
Delete the first entry in the keyword list for a specific `key`. If the `key` does not exist, returns the keyword list unchanged.

**Examples**

```iex
iex> Keyword.delete_first([a: 1, b: 2, a: 3], :a)
    [b: 2, a: 3]
iex> Keyword.delete_first([b: 2], :a)
    [b: 2]
```

**drop(keywords, keys)**(function)
Drops the given keys from the dict.

Duplicated keys are preserved in the new keyword list.

**Examples**

```iex
iex> d = [a: 1, b: 2, c: 3, d: 4]
iex> Keyword.drop(d, [:b, :d])
    [a: 1, c: 3]
iex> d = [a: 1, b: 2, b: 3, c: 3, d: 4, a: 5]
```
iex> Keyword.drop(d, [:b, :d])
[a: 1, c: 3, a: 5]

equal?(left, right)(function)
Specs:
  *equal?(t, t) :: boolean
Checks if two keywords are equal. I.e. they contain the same keys and those keys contain the same values.
Examples
iex> Keyword.equal?([a: 1, b: 2], [b: 2, a: 1])
true

fetch(keywords, key)(function)
Specs:
  *fetch(t, key) :: {:ok, value}
Fetches the value for a specific key and returns it in a tuple. If the key does not exist, returns :error.
Examples
iex> Keyword.fetch([a: 1], :a)
{:ok, 1}
iex> Keyword.fetch([a: 1], :b)
:error

fetch!(keywords, key)(function)
Specs:
  *fetch!(t, key) :: value | no_return
Fetches the value for specific key. If key does not exist, a KeyError is raised.
Examples
iex> Keyword.fetch!([a: 1], :a)
1
iex> Keyword.fetch!([a: 1], :b)
** (KeyError) key :b not found in: [a: 1]

get(keywords, key, default \ nil)(function)
Specs:
  *get(t, key, value) :: value
Gets the value for a specific key.
If key does not exist, return default value (nil if no default value).
If duplicated entries exist, the first one is returned. Use get_values/2 to retrieve all entries.
Examples
iex> Keyword.get([a: 1], :a)
1
iex> Keyword.get([a: 1], :b)
nil
iex> Keyword.get([a: 1], :b, 3)
3

`get_values(keywords, key)` (function)
Specs:
  * `get_values(keywords, key)` :: [value]

Gets all values for a specific `key`.

Examples
iex> Keyword.get_values([a: 1, a: 2], :a)
[1, 2]

`has_key?(keywords, key)` (function)
Specs:
  * `has_key?(keywords, key)` :: boolean

Returns whether a given `key` exists in the given `keywords`.

Examples
iex> Keyword.has_key?([a: 1], :a)
true
iex> Keyword.has_key?([a: 1], :b)
false

`keys(keywords)` (function)
Specs:
  * `keys(keywords)` :: [key]

Returns all keys from the keyword list. Duplicated keys appear duplicated in the final list of keys.

Examples
iex> Keyword.keys([a: 1, b: 2])
[:a, :b]
iex> Keyword.keys([a: 1, b: 2, a: 3])
[:a, :b, :a]

`keyword?` (arg1) (function)
Specs:
  * `keyword?(term)` :: boolean

Checks if the given argument is a keywords list or not.

`merge(d1, d2)` (function)
Specs:
  * `merge(keywords, keywords)` :: keywords

Merges two keyword lists into one. If they have duplicated entries, the one given as second argument wins.

Examples
iex> Keyword.merge([a: 1, b: 2], [a: 3, d: 4]) |> Enum.sort
[a: 3, b: 2, d: 4]
merge(d1, d2, fun)(function)
Specs:
  •merge(t, t, (key, value, value -> value)) :: t
Merges two keyword lists into one. If they have duplicated entries, the given function is invoked to solve conflicts.

Examples
iex> Keyword.merge([a: 1, b: 2], [a: 3, d: 4], fn (k, v1, v2) -> v1 + v2) #=> [a: 4, b: 2, d: 4]

new() (function)
Specs:
  •new :: t
Returns an empty keyword list, i.e. an empty list.

new(pairs) (function)
Specs:
  •new(Enum.t) :: t
Creates a keyword from an enumerable.
Duplicated entries are removed, the latest one prevails. I.e. differently from Enum.into(enumerable, []), Keyword.new(enumerable) guarantees the keys are unique.

Examples
iex> Keyword.new([{:b, 1}, {:a, 2}]) #=> [a: 2, b: 1]

new(pairs, transform) (function)
Specs:
  •new(Enum.t, (key, value) -> {key, value}) :: t
Creates a keyword from an enumerable via the transformation function.
Duplicated entries are removed, the latest one prevails. I.e. differently from Enum.into(enumerable, [], fun), Keyword.new(enumerable, fun) guarantees the keys are unique.

Examples
iex> Keyword.new([{:a, b}], fn (x) -> {x, x} end) | Enum.sort
    [a: a, b: b]

pop(keywords, key, default \ nil) (function)
Returns the first value associated with key in the keyword list as well as the keyword list without key.
All duplicated entries are removed. See pop_first/3 for removing only the first entry.

Examples
iex> Keyword.pop [a: 1], :a
    [{l, []}]

iex> Keyword.pop [a: 1], :b
    [{nil, [a: 1]}]
pop_first(keywords, key, default \ nil)(function)

Returns the first value associated with key in the keyword list as well as the keyword list without that particular occurrence of key.

Duplicated entries are not removed.

Examples

```iex
iex> Keyword.pop_first [a: 1], :a
{1,[a: 2]}

iex> Keyword.pop_first [a: 1], :b
(nil,[a: 1])

iex> Keyword.pop_first [a: 1], :b, 3
{3,[a: 1]}

iex> Keyword.pop_first [a: 1], :b, 3
{3,[a: 1]}

iex> Keyword.pop_first [a: 1, a: 2], :a
{1,[a: 2]}
```

put(keywords, key, value)(function)

Specs:

```iex
•put(t, key, value):: t
```

Puts the given value under key.

If a previous value is already stored, all entries are removed and the value is overridden.

Examples

```iex
iex> Keyword.put([a: 1, b: 2], :a, 3)
[a: 3, b: 2]

iex> Keyword.put([a: 1, b: 2, a: 4], :a, 3)
[a: 3, b: 2]
```

put_new(keywords, key, value)(function)

Specs:

```iex
•put_new(t, key, value):: t
```

Puts the given value under key unless the entry key already exists.

Examples

```iex
iex> Keyword.put_new([a: 1], :b, 2)
{b: 2, a: 1}
```
iex> Keyword.put_new([a: 1, b: 2], :a, 3)
[a: 1, b: 2]

**split(keywords, keys)**

Takes all entries corresponding to the given keys and extracts them into a separate keyword list. Returns a tuple with the new list and the old list with removed keys.

Keys for which there are no entries in the keyword list are ignored.

Entries with duplicated keys end up in the same keyword list.

**Examples**

```elixir
d = [a: 1, b: 2, c: 3, d: 4]
iex> Keyword.split(d, [:a, :c, :e])
[[a: 1, c: 3], [b: 2, d: 4]]

d = [a: 1, b: 2, c: 3, d: 4, a: 5]
iex> Keyword.split(d, [:a, :c, :e])
[[a: 1, c: 3, a: 5], [b: 2, d: 4]]
```

**take(keywords, keys)**

Takes all entries corresponding to the given keys and returns them in a new keyword list.

Duplicated keys are preserved in the new keyword list.

**Examples**

```elixir
d = [a: 1, b: 2, c: 3, d: 4]
iex> Keyword.take(d, [:a, :c, :e])
[a: 1, c: 3]

d = [a: 1, b: 2, c: 3, d: 4, a: 5]
iex> Keyword.take(d, [:a, :c, :e])
[a: 1, c: 3, a: 5]
```

**update(list1, key, initial, fun)**

Specs:

- `update(t, key, value, (value -> value)) :: t`

Updates the key with the given function. If the key does not exist, inserts the given initial value.

If there are duplicated entries, they are all removed and only the first one is updated.

**Examples**

```elixir
Keyword.update([a: 1], :a, 13, &(&1 * 2))
a: 2

Keyword.update([a: 1], :b, 11, &(&1 * 2))
a: b: 11
```

**update!(keywords, key, fun)**

Specs:

- `update!(t, key, value, (value -> value)) :: t | no_return`

Updates the key with the given function. If the key does not exist, raises **KeyError**.

If there are duplicated entries, they are all removed and only the first one is updated.

**Examples**

```elixir
```
```elixir
eix> Keyword.update!([:a: 1], :a, &(&1 * 2))
[a: 2]

eix> Keyword.update!([:a: 1], :b, &(&1 * 2))
** (KeyError) key :b not found in: [a: 1]
```

values(keywords)(function)

Specs:

- values(t) :: [value]

Returns all values from the keyword list.

Examples

```elixir
iex> Keyword.values([a: 1, b: 2])
[1,2]
```

List

Overview  Implements functions that only make sense for lists and cannot be part of the Enum protocol. In general, favor using the Enum API instead of List. Some functions in this module expect an index. Index access for list is linear. Negative indexes are also supported but they imply the list will be iterated twice, one to calculate the proper index and another to the operation. A decision was taken to delegate most functions to Erlang’s standard library but follow Elixir’s convention of receiving the target (in this case, a list) as the first argument.
Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete/2</td>
<td>Deletes the given item from the list. Returns a list without the item. If the item occurs more than once in the list, just the first occurrence is removed.</td>
</tr>
<tr>
<td>delete_at/2</td>
<td>Produces a new list by removing the value at the specified index. Negative indices indicate an offset from the end of the list. If index is out of bounds, the original list is returned.</td>
</tr>
<tr>
<td>duplicate/2</td>
<td>Duplicates the given element n times in a list.</td>
</tr>
<tr>
<td>first/1</td>
<td>Returns the first element in list or nil if list is empty.</td>
</tr>
<tr>
<td>flatten/1</td>
<td>Flattens the given list of nested lists.</td>
</tr>
<tr>
<td>flatten/2</td>
<td>Flattens the given list of nested lists. The list tail will be added at the end of the flattened list.</td>
</tr>
<tr>
<td>foldl/3</td>
<td>Folds (reduces) the given list to the left with a function. Requires an accumulator.</td>
</tr>
<tr>
<td>foldr/3</td>
<td>Folds (reduces) the given list to the right with a function. Requires an accumulator.</td>
</tr>
<tr>
<td>insert_at/3</td>
<td>Returns a list with value inserted at the specified index. Negative indices indicate an offset from the end of the list.</td>
</tr>
<tr>
<td>keydelete/3</td>
<td>Receives a list of tuples and deletes the first tuple where the item at position matches the given item. Returns the new list.</td>
</tr>
<tr>
<td>keyfind/4</td>
<td>Receives a list of tuples and returns the first tuple where the item at position in the tuple matches the given item.</td>
</tr>
<tr>
<td>keymember?/3</td>
<td>Receives a list of tuples and returns true if there is a tuple where the item at position in the tuple matches the given item</td>
</tr>
<tr>
<td>keyreplace/1</td>
<td>Receives a list of tuples and replaces the item identified by key at position if it exists.</td>
</tr>
<tr>
<td>keysort/2</td>
<td>Receives a list of tuples and sorts the items at position of the tuples. The sort is stable.</td>
</tr>
<tr>
<td>keystore/4</td>
<td>Receives a list of tuples and replaces the item identified by key at position. If the item does not exist, it is added to the end of the list.</td>
</tr>
<tr>
<td>last/1</td>
<td>Returns the last element in list or nil if list is empty.</td>
</tr>
<tr>
<td>replace_at/3</td>
<td>Returns a list with a replaced value at the specified index. Negative indices indicate an offset from the end of the list. If index is out of bounds, the original list is returned.</td>
</tr>
<tr>
<td>to_atom/1</td>
<td>Converts a char list to an atom.</td>
</tr>
<tr>
<td>to_existing_atom</td>
<td>Converts a char list to an existing atom.</td>
</tr>
<tr>
<td>to_float/1</td>
<td>Returns the float whose text representation is char_list.</td>
</tr>
<tr>
<td>to_integer/1</td>
<td>Returns an integer whose text representation is char_list.</td>
</tr>
<tr>
<td>to_integer/1</td>
<td>Returns an integer whose text representation is char_list in base base.</td>
</tr>
<tr>
<td>to_string/1</td>
<td>Converts a list of integers representing codepoints, lists or strings into a string.</td>
</tr>
<tr>
<td>to_tuple/1</td>
<td>Converts a list to a tuple.</td>
</tr>
<tr>
<td>unzip/1</td>
<td>Unzips the given list of lists or tuples into separate lists and returns a list of lists.</td>
</tr>
<tr>
<td>update_at/3</td>
<td>Returns a list with an updated value at the specified index. Negative indices indicate an offset from the end of the list. If index is out of bounds, the original list is returned.</td>
</tr>
<tr>
<td>wrap/1</td>
<td>Wraps the argument in a list. If the argument is already a list, returns the list. If the argument is nil, returns an empty list.</td>
</tr>
<tr>
<td>zip/1</td>
<td>Zips corresponding elements from each list in list_of_lists.</td>
</tr>
</tbody>
</table>

Functions

delete(list, item)(function)

Specs:

• delete([], any) :: []

Deletes the given item from the list. Returns a list without the item. If the item occurs more than once in the list, just the first occurrence is removed.

Examples

iex> List.delete([1, 2, 3], 1)
[2,3]

iex> List.delete([1, 2, 3], 2)
[1, 2, 3]

**delete_at(list, index)(function)**

Specs:

- `delete_at([], integer) :: []`

  Produces a new list by removing the value at the specified `index`. Negative indices indicate an offset from the end of the list. If `index` is out of bounds, the original `list` is returned.

**Examples**

```iex
iex> List.delete_at([1, 2, 3], 0)
[2, 3]
iex> List.delete_at([1, 2, 3], 10)
[1, 2, 3]
iex> List.delete_at([1, 2, 3], -1)
[1, 2]
```

**duplicate(elem, n)(function)**

Specs:

- `(duplicate(elem, non_neg_integer) :: [elem]) when elem: var`

  Duplicates the given element `n` times in a list.

**Examples**

```iex
iex> List.duplicate("hello", 3)
["hello","hello","hello"]
iex> List.duplicate([1, 2], 2)
[[1,2],[1,2]]
```

**first(list1)(function)**

Specs:

- `(first([elem]) :: nil | elem) when elem: var`

  Returns the first element in `list` or `nil` if `list` is empty.

**Examples**

```iex
iex> List.first([])
nil
iex> List.first([1])
1
iex> List.first([1, 2, 3])
1
```

**flatten(list)(function)**

Specs:

- `(flatten(deep_list) :: []) when deep_list: [any | deep_list]`

  Flattens the given `list` of nested lists.

**Examples**
flaten(list, tail)(function)
Specs:
  • (flaten(deep_list, [elem]) :: [elem]) when deep_list: [elem | deep_list], elem: var
Flattens the given list of nested lists. The list tail will be added at the end of the flattened list.

Examples
iex> List.flatten([1, [[2], 3]], [4, 5])
[1,2,3,4,5]

foldl(list, acc, function)(function)
Specs:
  • (foldl([elem], acc, (elem, acc -> acc)) :: acc) when elem: var, acc: var
Folds (reduces) the given list to the left with a function. Requires an accumulator.

Examples
iex> List.foldl([5, 5], 10, fn (x, acc) -> x + acc end)
20
iex> List.foldl([1, 2, 3, 4], 0, fn (x, acc) -> x - acc end)
2

foldr(list, acc, function)(function)
Specs:
  • (foldr([elem], acc, (elem, acc -> acc)) :: acc) when elem: var, acc: var
Folds (reduces) the given list to the right with a function. Requires an accumulator.

Examples
iex> List.foldr([1, 2, 3, 4], 0, fn (x, acc) -> x - acc end)
-2

insert_at(list, index, value)(function)
Specs:
  • insert_at([], integer, any) :: []
Returns a list with value inserted at the specified index. Note that index is capped at the list length. Negative indices indicate an offset from the end of the list.

Examples
iex> List.insert_at([1, 2, 3, 4], 2, 0)
[1, 2, 0, 3, 4]
iex> List.insert_at([1, 2, 3], 10, 0)
[1, 2, 3, 0]
iex> List.insert_at([1, 2, 3], -1, 0)
[1, 2, 3, 0]
iex> List.insert_at([1, 2, 3], -10, 0)
[0, 1, 2, 3]
keydelete(list, key, position)(function)
Specs:
  • keydelete([tuple], any, non_neg_integer) :: [tuple]

Receives a list of tuples and deletes the first tuple where the item at position matches the given item. Returns the new list.

Examples
iex> List.keydelete([a: 1, b: 2], :a, 0)
[b: 2]
iex> List.keydelete([a: 1, b: 2], 2, 1)
[a: 1]
iex> List.keydelete([a: 1, b: 2], :c, 0)
[a: 1, b: 2]

keyfind(list, key, position, default \ nil)(function)
Specs:
  • keyfind([tuple], any, non_neg_integer, any) :: any

Receives a list of tuples and returns the first tuple where the item at position in the tuple matches the given item.

Examples
iex> List.keyfind([a: 1, b: 2], :a, 0)
{:a, 1}
iex> List.keyfind([a: 1, b: 2], 2, 1)
{:b, 2}
iex> List.keyfind([a: 1, b: 2], :c, 0)
nil

keymember?(list, key, position)(function)
Specs:
  • keymember?([tuple], any, non_neg_integer) :: any

Receives a list of tuples and returns true if there is a tuple where the item at position in the tuple matches the given item.

Examples
iex> List.keymember?([a: 1, b: 2], :a, 0)
true
iex> List.keymember?([a: 1, b: 2], 2, 1)
true
iex> List.keymember?([a: 1, b: 2], :c, 0)
false

keyreplace(list, key, position, new_tuple)(function)
Specs:
  • keyreplace([tuple], any, non_neg_integer, tuple) :: [tuple]

Receives a list of tuples and replaces the item identified by key at position if it exists.
Examples

```iex
iex> List.keyreplace([a: 1, b: 2], :a, 0, {:a, 3})
{a: 3, b: 2}
```

**keysort**(list, position)(function)

Specs:

- `keysort([tuple], non_neg_integer) :: [tuple]`

Receives a list of tuples and sorts the items at `position` of the tuples. The sort is stable.

Examples

```iex
iex> List.keysort([a: 5, b: 1, c: 3], 1)
{b: 1, c: 3, a: 5}
iex> List.keysort([a: 5, c: 1, b: 3], 0)
{a: 5, b: 3, c: 1}
```

**keystore**(list, key, position, new_tuple)(function)

Specs:

- `keystore([tuple], any, non_neg_integer, tuple) :: [tuple]`

Receives a list of tuples and replaces the item identified by `key` at `position`. If the item does not exist, it is added to the end of the list.

Examples

```iex
iex> List.keystore([a: 1, b: 2], :a, 0, {:a, 3})
{a: 3, b: 2}
iex> List.keystore([a: 1, b: 2], :c, 0, {:c, 3})
{a: 1, b: 2, c: 3}
```

**last**(list1)(function)

Specs:

- `last([elem]) :: nil | elem` when `elem: var`

Returns the last element in `list` or `nil` if `list` is empty.

Examples

```iex
iex> List.last([])
nil
iex> List.last([1])
1
iex> List.last([1, 2, 3])
3
```

**replace_at**(list, index, value)(function)

Specs:

- `replace_at([], integer, any) :: []`

Returns a list with a replaced value at the specified `index`. Negative indices indicate an offset from the end of the list. If `index` is out of bounds, the original `list` is returned.

Examples
iex> List.replace_at([1, 2, 3], 0, 0)  
[0, 2, 3]  
iex> List.replace_at([1, 2, 3], 10, 0)  
[1, 2, 3]  
iex> List.replace_at([1, 2, 3], -1, 0)  
[1, 2, 0]  
iex> List.replace_at([1, 2, 3], -10, 0)  
[1, 2, 3]  

to_atom(char_list)(function)  
Specs:  
•to_atom(char_list) :: atom  
Converts a char list to an atom.  
Currently Elixir does not support conversions from char lists which contains Unicode codepoints greater than 0xFF.  
Inlined by the compiler.  
Examples  
iex> List.to_atom('elixir')  
:elixir  

to_existing_atom(char_list)(function)  
Specs:  
•to_existing_atom(char_list) :: atom  
Converts a char list to an existing atom.  
Currently Elixir does not support conversions from char lists which contains Unicode codepoints greater than 0xFF.  
Inlined by the compiler.  

to_float(char_list)(function)  
Specs:  
•to_float(char_list) :: float  
Returns the float whose text representation is char_list.  
Inlined by the compiler.  
Examples  
iex> List.to_float('2.2017764e+0')  
2.2017764  

to_integer(char_list)(function)  
Specs:  
•to_integer(char_list) :: integer  
Returns an integer whose text representation is char_list.  
Inlined by the compiler.  
Examples
Elixir Documentation, Release

```elixir
iex> List.to_integer('123
123

**to_integer(char_list, base)**
Specs:

- `to_integer(char_list, non_neg_integer) :: integer`

Returns an integer whose text representation is `char_list` in base `base`.
Inlined by the compiler.

Examples
```
iex> List.to_integer('3FF', 16)
1023
```

**to_string(list)**
Specs:

- `to_string(unicode.char_list) :: String.t`

Converts a list of integers representing codepoints, lists or strings into a string.
Notice that this function expect a list of integer representing UTF-8 codepoints. If you have a list of bytes, you must instead use the `:binary` module [http://erlang.org/doc/man/binary.html](http://erlang.org/doc/man/binary.html).

Examples
```
iex> List.to_string([0x00E6, 0x00DF])
"æß"
iex> List.to_string([0x0061, "bc"])
"abc"
```

**to_tuple(list)**
Specs:

- `to_tuple([]) :: tuple`

Converts a list to a tuple.
Inlined by the compiler.

Examples
```
iex> List.to_tuple([[:share, [:elixir, 163]])
{:share, [:elixir, 163]}
```

**unzip(list)**
Specs:

- `unzip(tuple) :: [[]]

Unzips the given list of lists or tuples into separate lists and returns a list of lists.

Examples
```
iex> List.unzip([{1, 2}, {3, 4}])
[[1, 3], [2, 4]]
iex> List.unzip([{:a, "apple"}, {:b, "banana"}, {3, :c}])
[[1, :a], [2, :b], [3, :c]]
```
update_at(list, index, fun)(function)
Specs:

- \texttt{(update\_at([elem], integer, (elem \rightarrow any)) :: [])} when \texttt{elem: var}

Returns a list with an updated value at the specified \texttt{index}. Negative indices indicate an offset from the end of the list. If \texttt{index} is out of bounds, the original \texttt{list} is returned.

Examples

\begin{verbatim}
 iex> List.update_at([1, 2, 3], 0, &(&1 + 10))
[11, 2, 3]
 iex> List.update_at([1, 2, 3], 10, &(&1 + 10))
[1, 2, 3]
 iex> List.update_at([1, 2, 3], -1, &(&1 + 10))
[1, 2, 13]
 iex> List.update_at([1, 2, 3], -10, &(&1 + 10))
[1, 2, 3]
\end{verbatim}

wrap(list)(function)
Specs:

- \texttt{wrap([| any]) :: []}

Wraps the argument in a list. If the argument is already a list, returns the list. If the argument is \texttt{nil}, returns an empty list.

Examples

\begin{verbatim}
 iex> List.wrap("hello")
["hello"]
 iex> List.wrap([1, 2, 3])
[1, 2, 3]
 iex> List.wrap(nil)
[]
\end{verbatim}

zip(list_of_lists)(function)
Specs:

- \texttt{zip([[]]) :: [tuple]}

Zips corresponding elements from each list in \texttt{list\_of\_lists}.

Examples

\begin{verbatim}
 iex> List.zip([[1, 2], [3, 4], [5, 6]])
[[1, 3, 5], [2, 4, 6]]
 iex> List.zip([[1, 2], [3], [5, 6]])
[[1, 3, 5]]
\end{verbatim}

Macro

Overview Conveniences for working with macros.
decompose_call/1

Decomposes a local or remote call into its remote part (when provided), function name and argument list

escape/2

Recursively escapes a value so it can be inserted into a syntax tree

expand/2

Receives an AST node and expands it until it can no longer be expanded

expand_once/2

Receives an AST node and expands it once

pipe/3

Pipes expr into the call_args at the given position

postwalk/2

Performs a depth-first, post-order traversal of quoted expressions

postwalk/3

Performs a depth-first, post-order traversal of quoted expressions using an accumulator

to_string/2

Converts the given expression to a binary

unescape_string/1

Unescape the given chars

unescape_string/2

Unescape the given chars according to the map given

unescape_tokens/1

Unescape the given tokens according to the default map

unescape_tokens/2

Unescape the given tokens according to the given map

unpipe/1

Breaks a pipeline expression into a list

update_meta/2

Applies the given function to the node metadata if it contains one

---

Types

t

t :: expr | t | atom | number | binary | pid | (... -> any) | t

Abstract Syntax Tree (AST)

expr

expr :: {expr | atom, Keyword.t, atom | t}

Expr node (remaining ones are literals)

---

Functions

decompose_call(arg1)(function)

Specs:

• decompose_call(Macro.t):: {atom, [Macro.t]} | {Macro.t, atom, [Macro.t]} | :error

Decomposes a local or remote call into its remote part (when provided), function name and argument list.

Returns :error when an invalid call syntax is provided.

Examples

iex> Macro.decompose_call(quote do: foo)
{:foo, []}

iex> Macro.decompose_call(quote do: foo())
{:foo, []}

iex> Macro.decompose_call(quote do: foo(1, 2, 3))
{:foo, [1, 2, 3]}

iex> Macro.decompose_call(quote do: Elixir.M.foo(1, 2, 3))
{{:__aliases__, [], [:Elixir, :M]}, :foo, [1, 2, 3]}

iex> Macro.decompose_call(quote do: 42)
:error

escape(expr, opts \ \ [])(function)

Specs:

• escape(term, Keyword.t):: Macro.t
Recursively escapes a value so it can be inserted into a syntax tree.

One may pass `unquote: true` to `escape/2` which leaves `unquote` statements unescaped, effectively unquoting the contents on escape.

**Examples**

```iex
eval > Macro.escape(:foo)
:foo

eval > Macro.escape({:a, :b, :c})
{:{}, [], [:a, :b, :c]}

eval > Macro.escape({:unquote, [], [1]}, unquote: true)
1
```

### expand/2 (tree, env) (function)

Receives an AST node and expands it until it can no longer be expanded.

This function uses `expand_once/2` under the hood. Check `expand_once/2` for more information and examples.

### expand_once/2 (ast, env) (function)

Receives an AST node and expands it once.

The following contents are expanded:

- Macros (local or remote);
- Aliases are expanded (if possible) and return atoms;
- Pseudo-variables (`__ENV__`, `__MODULE__` and `__DIR__`);
- Module attributes reader (`@foo`);

If the expression cannot be expanded, it returns the expression itself. Notice that `expand_once/2` performs the expansion just once and it is not recursive. Check `expand/2` for expansion until the node can no longer be expanded.

**Examples**

In the example below, we have a macro that generates a module with a function named `name_length` that returns the length of the module name. The value of this function will be calculated at compilation time and not at runtime.

Consider the implementation below:

```elixir
defmacro defmodule_with_length(name, do: block) do
  length = length(Atom.to_char_list(name))
  quote do
    defmodule unquote(name) do
      def name_length, do: unquote(length)
      unquote(block)
    end
  end
end
```

When invoked like this:

```elixir
defmodule_with_length My.Module do
  def other_function, do: ...
end
```
The compilation will fail because `MyModule` when quoted is not an atom, but a syntax tree as follow:

```
{:__aliases__, [], [:My, :Module]}
```

That said, we need to expand the aliases node above to an atom, so we can retrieve its length. Expanding the node is not straight-forward because we also need to expand the caller aliases. For example:

```
alias MyHelpers, as: My

defmodule_with_length My.Module do
  def other_function, do: ...
end
```

The final module name will be `MyHelpers.Module` and not `My.Module`. With `Macro.expand/2`, such aliases are taken into consideration. Local and remote macros are also expanded. We could rewrite our macro above to use this function as:

```
defmacro defmodule_with_length(name, do: block) do
  expanded = Macro.expand(name, __CALLER__)
  length = length(Atom.to_char_list(expanded))
  quote do
    defmodule unquote(name) do
      def name_length, do: unquote(length)
      unquote(block)
    end
  end
end
```

pipe(expr, call_args, position)(function)

Specs:

```
•pipe(Macro.t, Macro.t, integer) :: Macro.t | no_return
```

Pipes `expr` into the `call_args` at the given `position`.

postwalk(ast, fun)(function)

Specs:

```
•postwalk(t, (t -> t)) :: t
```

Performs a depth-first, post-order traversal of quoted expressions.

postwalk(ast, acc, fun)(function)

Specs:

```
•postwalk(t, any, (t, any -> {t, any})) :: {t, any}
```

Performs a depth-first, post-order traversal of quoted expressions using an accumulator.

prewalk(ast, fun)(function)

Specs:

```
•prewalk(t, (t -> t)) :: t
```

Performs a depth-first, pre-order traversal of quoted expressions.

prewalk(ast, acc, fun)(function)

Specs:

```
•prewalk(t, any, (t, any -> {t, any})) :: {t, any}
```

Performs a depth-first, pre-order traversal of quoted expressions using an accumulator.
to_string(tree, fun \ fn _ast, string -> string end)(function)
   Specs:
   •to_string(Macro.t, (Macro.t, String.t -> String.t)):: String.t

   Converts the given expression to a binary.

   Examples
   iex> Macro.to_string(quote do: foo.bar(1, 2, 3))
   "foo.bar(1, 2, 3)"

unescape_string(chars)(function)
   Specs:
   •unescape_string(String.t):: String.t

   Unescape the given chars.

   This is the unescaping behaviour used by default in Elixir single- and double-quoted strings. Check
    unescape_string/2 for information on how to customize the escaping map.

   In this setup, Elixir will escape the following: \a, \b, \d, \e, \f, \n, \r, \s, \t and \v. Octals are also
    escaped according to the latin1 set they represent.

   This function is commonly used on sigil implementations (like ~r, ~s and others) which receive a raw, un-
    escaped string.

   Examples
   iex> Macro.unescape_string("example\\n")
   "example\n"

   In the example above, we pass a string with \n escaped and return a version with it unescaped.

unescape_string(chars, map)(function)
   Specs:
   •unescape_string(String.t, (non_neg_integer -> non_neg_integer | false)):: String.t

   Unescape the given chars according to the map given.

   Check unescape_string/1 if you want to use the same map as Elixir single- and double-quoted strings.

   Map
   The map must be a function. The function receives an integer representing the codepoint of the character it
    wants to unescape. Here is the default mapping function implemented by Elixir:

   def unescape_map(?a), do: ?\a
   def unescape_map(?b), do: ?\b
   def unescape_map(?d), do: ?\d
   def unescape_map(?e), do: ?\e
   def unescape_map(?f), do: ?\f
   def unescape_map(?n), do: ?\n
   If the unescape_map function returns false. The char is not escaped and \ is kept in the char list.

Octals
Octals will by default be escaped unless the map function returns `false` for `?0`.

**Hex**

Hexadecimal will by default be escaped unless the map function returns `false` for `?x`.

**Examples**

Using the `unescape_map` function defined above is easy:

```elixir
Macro.unescape_string "example\n", &unescape_map(&1)
```

### `unescape_tokens(tokens)(function)`

**Specs:**

```elixir
• unescape_tokens([Macro.t]) :: [Macro.t]
```

Unescape the given tokens according to the default map.

Check `unescape_string/1` and `unescape_string/2` for more information about unescaping.

Only tokens that are binaries are unescaped, all others are ignored. This function is useful when implementing your own sigils. Check the implementation of `Kernel.sigil_s/2` for examples.

### `unescape_tokens(tokens, map)(function)`

**Specs:**

```elixir
• unescape_tokens([Macro.t], (non_neg_integer -> non_neg_integer | false)) :: [Macro.t]
```

Unescape the given tokens according to the given map.

Check `unescape_tokens/1` and `unescape_string/2` for more information.

### `unpipe(other)(function)`

**Specs:**

```elixir
• unpipe(Macro.t) :: [Macro.t]
```

Breaks a pipeline expression into a list.

Raises if the pipeline is ill-formed.

### `update_meta(quoted, fun)(function)`

**Specs:**

```elixir
• update_meta(t, (Keyword.t -> Keyword.t)) :: t
```

Applies the given function to the node metadata if it contains one.

This is often useful when used with `Macro.prewalk/1` to remove information like lines and hygienic counters from the expression for either storage or comparison.

**Examples**

```elixir
iex> quoted = quote line: 10, do: sample()
{:sample, [line: 10], []}
iex> Macro.update_meta(quoted, &Keyword.delete(&1, :line))
{:sample, [], []}
```

---

**Macro.Env**

**Overview**

A struct that holds compile time environment information.

The current environment can be accessed at any time as `__ENV__`. Inside macros, the caller environment can be accessed as `__CALLER__`. It contains the following fields:
• module - the current module name.
• file - the current file name as a binary
• line - the current line as an integer
• function - a tuple as {atom, integer}, where the first element is the function name and the seconds its arity. Returns nil if not inside a function
• context - the context of the environment. It can be nil (default context), inside a guard or inside an assign
• aliases - a list of two item tuples, where the first item is the aliased name and the second the actual name
• requires - the list of required modules
• functions - a list of functions imported from each module
• macros - a list of macros imported from each module
• macro_aliases - a list of aliases defined inside the current macro
• context_modules - a list of modules defined in the current context
• vars - a list keeping all defined variables as {var, context}
• export_vars - a list keeping all variables to be exported in a construct (may be nil)
• lexical_tracker - PID to the lexical tracker which is responsible to keep user info
• local - the module to expand local functions to

<table>
<thead>
<tr>
<th>Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_guard?/1</td>
<td>Returns whether the compilation environment is currently inside a guard</td>
</tr>
<tr>
<td>in_match?/1</td>
<td>Returns whether the compilation environment is currently inside a match clause</td>
</tr>
<tr>
<td>location/1</td>
<td>Returns a keyword list containing the file and line information as keys</td>
</tr>
<tr>
<td>stacktrace/1</td>
<td>Returns the environment stacktrace</td>
</tr>
</tbody>
</table>

Types

**name arity**

name_arity :: {atom, non_neg_integer}

**file**

file :: binary

**line**

line :: non_neg_integer

**aliases**

aliases :: [[module, module]]

**macro aliases**

macro_aliases :: [[module, {integer, module}]]

**context**

context :: :match | :guard | nil

**requires**

requires :: [module]

**functions**

functions :: [[module, [name arity]]]

**macros**

macros :: [[module, [name arity]]]
context_modules
    context_modules :: [module]

vars
    vars :: [{atom, atom | non_neg_integer}]

export_vars
    export_vars :: vars | nil

lexical_tracker
    lexical_tracker :: pid

local
    local :: module | nil

t

Functions

in_guard?(arg1)(function)
    Returns whether the compilation environment is currently inside a guard.

in_match?(arg1)(function)
    Returns whether the compilation environment is currently inside a match clause.

location(arg1)(function)
    Returns a keyword list containing the file and line information as keys.

stacktrace(env)(function)
    Returns the environment stacktrace.

Map

Overview
    A Dict implementation that works on maps.

Maps are key-value stores where keys are compared using the match operator (==). Maps can be created with the %() special form defined in the Kernel.SpecialForms module.

For more information about the functions in this module and their APIs, please consult the Dict module.
Functions

delete(map, key)(function)
Callback implementation of Dict.delete/2.

drop(dict, keys)(function)
Callback implementation of Dict.drop/2.

equal?(dict1, dict2)(function)
Callback implementation of Dict.equal?/2.

fetch(map, key)(function)
Callback implementation of Dict.fetch/2.

fetch!(dict, key)(function)
Callback implementation of Dict.fetch!/2.

get(dict, key, default \ nil)(function)
Callback implementation of Dict.get/3.

has_key?(dict, key)(function)
Callback implementation of Dict.has_key?/2.

keys(dict)(function)
Callback implementation of Dict.keys/1.

merge(map1, map2)(function)
Callback implementation of Dict.merge/2.

merge(dict1, dict2, fun \ fn _k, _v1, v2 -> v2 end)(function)
Callback implementation of Dict.merge/3.

new()(function)
Returns a new empty map.

pop(dict, key, default \ nil)(function)
Callback implementation of Dict.pop/3.
Module Overview

This module provides many functions to deal with modules during compilation time. It allows a developer to dynamically attach documentation, add, delete and register attributes and so forth.

After a module is compiled, using many of the functions in this module will raise errors, since it is out of their scope to inspect runtime data. Most of the runtime data can be inspected via the __info__(attr) function attached to each compiled module.

Module attributes

Each module can be decorated with one or more attributes. The following ones are currently defined by Elixir:

- @after_compile
  
  A hook that will be invoked right after the current module is compiled.

  Accepts a module or a tuple \( \langle \text{module}, \langle \text{function \ atom} \rangle \rangle \). The function must take two arguments: the module environment and its bytecode. When just a module is provided, the function is assumed to be __after_compile__/2.

  Example

  ```elixir
defmodule M do
    @after_compile __MODULE__

    def __after_compile__(env, _bytecode) do
      IO.inspect env
    end
  end
```

- @before_compile
  
  A hook that will be invoked before the module is compiled.

```elixir
defmodule M do
  @before_compile __MODULE__

  def __before_compile__(env, _bytecode) do
    IO.puts "Before compile: \"\"\""
  end
end
```
Accepts a module or a tuple `{<module>, <function/macro atom>}`. The function/macro must take one argument: the module environment. If it’s a macro, its returned value will be injected at the end of the module definition before the compilation starts.

When just a module is provided, the function/macro is assumed to be `__before_compile__/1`.

Note: unlike `@after_compile`, the callback function/macro must be placed in a separate module (because when the callback is invoked, the current module does not yet exist).

**Example**

```elixir
defmodule A do
defmacro __before_compile__(_env) do
  quote do
    def hello, do: "world"
  end
end
end

defmodule B do
  @before_compile A
end
```

- `@behaviour` *(notice the British spelling)*

Specify an OTP or user-defined behaviour.

**Example**

```elixir
defmodule M do
  @behaviour gen_event
  # ...
end
```

- `@compile`

Define options for module compilation that are passed to the Erlang compiler.

Accepts an atom, a tuple, or a list of atoms and tuples.

See [http://www.erlang.org/doc/man/compile.html](http://www.erlang.org/doc/man/compile.html) for the list of supported options.

**Example**

```elixir
defmodule M do
  @compile {:inline, myfun: 1}
  def myfun(arg) do
    to_string(arg)
  end
end
```

- `@doc`

Provide documentation for the function or macro that follows the attribute.

Accepts a string (often a heredoc) or `false` where `@doc false` will make the function/macro invisible to the documentation extraction tools like ExDoc.

Can be invoked more than once.

**Example**


```elixir
defmodule M do
  @doc "Hello world"
  def hello do
    "world"
  end

  @doc ""
  Sum.
  ""
  def sum(a, b) do
    a + b
  end
end
```

- **@file**
  Change the filename used in stacktraces for the function or macro that follows the attribute.
  Accepts a string. Can be used more than once.

**Example**

```elixir
defmodule M do
  @doc "Hello world"
  @file "hello.ex"
  def hello do
    "world"
  end
end
```

- **@moduledoc**
  Provide documentation for the current module.
  Accepts a string (which is often a heredoc) or `false` where `@moduledoc false` will make the module invisible to the documentation extraction tools like ExDoc.

**Example**

```elixir
defmodule M do
  @moduledoc "A very useful module"
end
```

- **@on_definition**
  A hook that will be invoked when each function or macro in the current module is defined. Useful when annotating functions.
  Accepts a module or a tuple `{<module>, <function atom>}`. The function must take 6 arguments:
  - the module environment
  - kind: `:def`, `:defp`, `:defmacro`, or `:defmacrop`
  - function/macro name
  - list of expanded arguments
  - list of expanded guards
  - expanded function body
Note the hook receives the expanded arguments and it is invoked before the function is stored in the module. So `Module.defines?/2` will return false for the first clause of every function.

If the function/macro being defined has multiple clauses, the hook will be called for each clause.

Unlike other hooks, `@on_definition` will only invoke functions and never macros. This is because the hook is invoked inside the context of the function (and nested function definitions are not allowed in Elixir).

When just a module is provided, the function is assumed to be `__on_definition__/6`.

**Example**

```elixir
defmodule H do
def on_def(_env, kind, name, args, guards, body) do
  IO.puts "Defining #{kind} named #{name} with args:
  IO.inspect args
  IO.puts "and guards
  IO.inspect guards
  IO.puts "and body"
  IO.puts Macro.to_string(body)
end
end
defmodule M do
  @on_definition {H, :on_def}
  def hello(arg) when is_binary(arg) or is_list(arg) do
    "Hello" <> to_string(arg)
  end
  def hello(_) do
    :ok
  end
end
```

• `@on_load`

A hook that will be invoked whenever the module is loaded.

Accepts a function atom of a function in the current module. The function must have arity 0 (no arguments) and has to return `:ok`, otherwise the loading of the module will be aborted.

**Example**

```elixir
defmodule M do
  @on_load :load_check
  def load_check do
    if some_condition() do
      :ok
    else
      nil
    end
  end
  def some_condition do
    false
  end
end
```

• `@vsn`
Specify the module version. Accepts any valid Elixir value.

**Example**

```elixir
defmodule M do
  @vsn "1.0"
end
```

The following attributes are part of typespecs and are also reserved by Elixir (see `Kernel.Typespec` for more information about typespecs):

- `@type` - defines a type to be used in `@spec`
- `@typep` - defines a private type to be used in `@spec`
- `@opaque` - defines an opaque type to be used in `@spec`
- `@spec` - provides a specification for a function
- `@callback` - provides a specification for the behaviour callback

In addition to the built-in attributes outlined above, custom attributes may also be added. A custom attribute is any valid identifier prefixed with an `@` and followed by a valid Elixir value:

```elixir
defmodule M do
  @custom_attr [some: "stuff"]
end
```

For more advanced options available when defining custom attributes, see `register_attribute/3`.

**Runtime information about a module**  It is possible to query a module at runtime to find out which functions and macros it defines, extract its docstrings, etc. See `__info__/1`.
## Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>info</strong>/1</td>
<td>Provides runtime information about functions and macros defined by the module, enables docstring extraction, etc.</td>
</tr>
<tr>
<td>add_doc/6</td>
<td>Attaches documentation to a given function or type. It expects the module the function/type belongs to, the line (a non negative integer), the kind (def or defmacro), a tuple representing the function and its arity, the function signature (the signature should be omitted for types) and the documentation, which should be either a binary or a boolean.</td>
</tr>
<tr>
<td>concat/1</td>
<td>Concatenates a list of aliases and returns a new alias</td>
</tr>
<tr>
<td>concat/2</td>
<td>Concatenates two aliases and returns a new alias</td>
</tr>
<tr>
<td>create/3</td>
<td>Creates a module with the given name and defined by the given quoted expressions. The line where the module is defined and its file can be passed as options</td>
</tr>
<tr>
<td>defines?/2</td>
<td>Checks if the module defines the given function or macro. Use defines?/3 to assert for a specific type</td>
</tr>
<tr>
<td>defines?/3</td>
<td>Checks if the module defines a function or macro of the given kind. kind can be any of :def, :defp, :defmacro or :defmacrop</td>
</tr>
<tr>
<td>definitions_in/1</td>
<td>Return all functions defined in module</td>
</tr>
<tr>
<td>definitions_in/2</td>
<td>Returns all functions defined in module, according to its kind</td>
</tr>
<tr>
<td>delete_attribute/2</td>
<td>Deletes all attributes that match the given key</td>
</tr>
<tr>
<td>eval_quoted/1</td>
<td>Evaluates the quoted contents in the given module’s context</td>
</tr>
<tr>
<td>function/3</td>
<td>Gets an anonymous function from the given module, function and arity. The module and function are not verified to exist</td>
</tr>
<tr>
<td>get_attribute/3</td>
<td>Gets the given attribute from a module. If the attribute was marked with accumulate with Module.register_attribute/3, a list is always returned</td>
</tr>
<tr>
<td>make_overridable/1</td>
<td>Makes the given functions in module overridable. An overridable function is lazily defined, allowing a developer to customize it. See Kernel.defoverridable/1 for more information and documentation</td>
</tr>
<tr>
<td>open?/1</td>
<td>Check if a module is open, i.e. it is currently being defined and its attributes and functions can be modified</td>
</tr>
<tr>
<td>overridable?/1</td>
<td>Returns true if tuple in module is marked as overridable</td>
</tr>
<tr>
<td>put_attribute/3</td>
<td>Puts an Erlang attribute to the given module with the given key and value. The semantics of putting the attribute depends if the attribute was registered or not via register_attribute/3</td>
</tr>
<tr>
<td>register_attribute/3</td>
<td>Registers an attribute. By registering an attribute, a developer is able to customize how Elixir will store and accumulate the attribute values</td>
</tr>
<tr>
<td>safe_concat/1</td>
<td>Concatenates a list of aliases and returns a new alias only if the alias was already referenced. If the alias was not referenced yet, fails with ArgumentError. It handles char lists, binaries and atoms</td>
</tr>
<tr>
<td>safe_concat/2</td>
<td>Concatenates two aliases and returns a new alias only if the alias was already referenced. If the alias was not referenced yet, fails with ArgumentError. It handles char lists, binaries and atoms</td>
</tr>
<tr>
<td>split/1</td>
<td>Split the given module name into binary parts</td>
</tr>
</tbody>
</table>

## Functions

__info__/1 (kind) (function)

Specs:

- __info__/ (atom) :: term

Provides runtime information about functions and macros defined by the module, enables docstring extraction, etc.

Each module gets an __info__/1 function when it’s compiled. The function takes one of the following atoms:

- :functions - keyword list of public functions along with their arities
- :macros - keyword list of public macros along with their arities
In addition to the above, you may also pass to `__info__/1` any atom supported by Erlang’s `module_info` function which also gets defined for each compiled module. See http://erlang.org/doc/reference_manual/modules.html#id69430 for more information.

**add_doc(module, line, kind, tuple, signature \ 
\[
\], doc)(function)**

Attaches documentation to a given function or type. It expects the module the function/type belongs to, the line (a non negative integer), the kind (def or defmacro), a tuple representing the function and its arity, the function signature (the signature should be omitted for types) and the documentation, which should be either a binary or a boolean.

**Examples**

```elixir
defmodule MyModule do
  Module.add_doc(__MODULE__, __ENV__.line + 1, :def, {:version, 0}, [], "Manually added docs")
  def version, do: 1
end
```

**concat(list)(function)**

Specs:

- `concat([binary | atom]) :: atom`

Concatenates a list of aliases and returns a new alias.

**Examples**

```elixir
iex> Module.concat([Foo, Bar])
Foo.Bar
iex> Module.concat([Foo, "Bar"])
Foo.Bar
```

**concat(left, right)(function)**

Specs:

- `concat(binary | atom, binary | atom) :: atom`

Concatenates two aliases and returns a new alias.

**Examples**

```elixir
iex> Module.concat(Foo, Bar)
Foo.Bar
iex> Module.concat(Foo, "Bar")
Foo.Bar
```

**create(module, quoted, opts \ 
\[
\])(function)**

Creates a module with the given name and defined by the given quoted expressions. The line where the module is defined and its file can be passed as options.

**Examples**

```elixir
contents =
  quote do
    def world, do: true
  end

Module.create(Hello, contents, Macro.Env.location(__ENV__))

Hello.world #=> true
```
Differences from “defmodule“

Module.create works similarly to defmodule and return the same results. While one could also use defmodule to define modules dynamically, this function is preferred when the module body is given by a quoted expression.

Another important distinction is that Module.create allows you to control the environment variables used when defining the module, while defmodule automatically shares the same environment.

defines?(module, tuple)(function)
Checks if the module defines the given function or macro. Use defines?/3 to assert for a specific type.

Examples

defmodule Example do
  def version, do: 1
  def version, do: 1
  Module.defines? __MODULE__, {:version, 0} #=> false
  Module.defines? __MODULE__, {:version, 0} #=> true
end

defines?(module, tuple, kind)(function)
Checks if the module defines a function or macro of the given kind. kind can be any of :def, :defp, :defmacro or :defmacrop.

Examples

defmodule Example do
  def version, do: 1
  def version, do: 1
  Module.defines? __MODULE__, {:version, 0}, :defp #=> false
  Module.defines? __MODULE__, {:version, 0}, :defp #=> false
end

definitions_in(module)(function)
Return all functions defined in module.

Examples

defmodule Example do
  def version, do: 1
  Module.definitions_in __MODULE__ #=> [{:version,0}]
end

definitions_in(module, kind)(function)
Returns all functions defined in module, according to its kind.

Examples

defmodule Example do
  def version, do: 1
  Module.definitions_in __MODULE__, :def #=> [{:version,0}]
  Module.definitions_in __MODULE__, :defp #=> []
end

delete_attribute(module, key)(function)
Deletes all attributes that match the given key.

Examples

defmodule MyModule do
  Module.put_attribute __MODULE__, :custom_threshold_for_lib, 10
  Module.delete_attribute __MODULE__, :custom_threshold_for_lib
end
Evaluates the quoted contents in the given module’s context.

A list of environment options can also be given as argument. See `Code.eval_string/3` for more information.

Raises an error if the module was already compiled.

**Examples**

```elixir
defmodule Foo do
  contents = quote do
    def sum(a, b), do: a + b
  end
  Module.eval_quoted __MODULE__, contents
end

Foo.sum(1, 2) #=> 3
```

For convenience, you can my pass `__ENV__` as argument and all options will be automatically extracted from the environment:

```elixir
defmodule Foo do
  contents = quote do
    def sum(a, b), do: a + b
  end
  Module.eval_quoted __MODULE__, contents, [], __ENV__
end

Foo.sum(1, 2) #=> 3
```

**function (mod, fun, arity)**

Gets an anonymous function from the given module, function and arity. The module and function are not verified to exist.

```iex
fun = Module.function(Kernel, :is_atom, 1)
fun.(:hello)
true
```

**get_attribute (module, key, warn \ nil)**

**Specs:**

* `get_attribute(module, atom, warn :: nil | [tuple]) :: term`

Gets the given attribute from a module. If the attribute was marked with `accumulate` with `Module.register_attribute/3`, a list is always returned.

The `@` macro compiles to a call to this function. For example, the following code:

```elixir
@foo
```

Expands to:

```elixir
Module.get_attribute(__MODULE__, :foo, true)
```

Notice the third argument may be given to indicate a stacktrace to be emitted when the attribute was not previously defined. The default value for `warn` is `nil` for direct calls but the `@foo` macro sets it to the proper stacktrace automatically, warning every time `@foo` is used but not set previously.

**Examples**

```elixir
defmodule Foo do
  Module.put_attribute __MODULE__, :value, 1
  Module.get_attribute __MODULE__, :value #=> 1
  Module.register_attribute __MODULE__, :value, accumulate: true
```
make_overridable(module, tuples)(function)
Makes the given functions in module overridable. An overridable function is lazily defined, allowing a developer to customize it. See Kernel.defoverridable/1 for more information and documentation.

open?(module)(function)
Check if a module is open, i.e. it is currently being defined and its attributes and functions can be modified.

overridable?(module, tuple)(function)
Returns true if tuple in module is marked as overridable.

put_attribute(module, key, value)(function)
Puts an Erlang attribute to the given module with the given key and value. The semantics of putting the attribute depends if the attribute was registered or not via register_attribute/3.

Examples

defmodule MyModule do
  Module.put_attribute __MODULE__, :custom_threshold_for_lib, 10
end

register_attribute(module, new, opts)(function)
Registers an attribute. By registering an attribute, a developer is able to customize how Elixir will store and accumulate the attribute values.

Options
When registering an attribute, two options can be given:

• :accumulate - Several calls to the same attribute will accumulate instead of override the previous one. New attributes are always added to the top of the accumulated list.

• :persist - The attribute will be persisted in the Erlang Abstract Format. Useful when interfacing with Erlang libraries.

By default, both options are false.

Examples

defmodule MyModule do
  Module.register_attribute __MODULE__, :custom_threshold_for_lib, accumulate: true, persist: false
  @custom_threshold_for_lib 10
  @custom_threshold_for_lib 20
  @custom_threshold_for_lib #=> [20, 10]
end

safe_concat(list)(function)
Specs:

• safe_concat([binary | atom]) :: atom | no_return

Concatenates a list of aliases and returns a new alias only if the alias was already referenced. If the alias was not referenced yet, fails with ArgumentError. It handles char lists, binaries and atoms.

Examples
iex> Module.safe_concat([Unknown, Module])
** (ArgumentError) argument error

iex> Module.safe_concat([List, Chars])
List.Chars

**safe_concat(left, right)** (function)

Specs:

•safe_concat(binary | atom, binary | atom) :: atom | no_return

Concatenates two aliases and returns a new alias only if the alias was already referenced. If the alias was not referenced yet, fails with ArgumentError. It handles char lists, binaries and atoms.

**Examples**

iex> Module.safe_concat(Unknown, Module)
** (ArgumentError) argument error

iex> Module.safe_concat(List, Chars)
List.Chars

**split**(module) (function)

Split the given module name into binary parts.

**Examples**

#=> ["Very", "Long", "Module", "Name", "And", "Even", "Longer"]

**Node**

**Overview** Functions related to VM nodes.

Some of the functions in this module are inlined by the compiler, similar to functions in the Kernel module and they are explicitly marked in their docs as “inlined by the compiler”. For more information about inlined functions, check out the Kernel module.
### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alive?/0</td>
<td>Returns true if the local node is alive</td>
</tr>
<tr>
<td>connect/1</td>
<td>Establishes a connection to node</td>
</tr>
<tr>
<td>disconnect/1</td>
<td>Forces the disconnection of a node</td>
</tr>
<tr>
<td>get_cookie/0</td>
<td>Returns the magic cookie of the local node</td>
</tr>
<tr>
<td>list/0</td>
<td>Returns a list of all visible nodes in the system, excluding the local node</td>
</tr>
<tr>
<td>list/1</td>
<td>Returns a list of nodes according to argument given</td>
</tr>
<tr>
<td>monitor/2</td>
<td>Monitors the status of the node</td>
</tr>
<tr>
<td>monitor/3</td>
<td>Behaves as monitor/2 except that it allows an extra option to be given, namely :allow_passive_connect</td>
</tr>
<tr>
<td>ping/1</td>
<td>Tries to set up a connection to node</td>
</tr>
<tr>
<td>self/0</td>
<td>Returns the current node</td>
</tr>
<tr>
<td>set_cookie/2</td>
<td>Sets the magic cookie of node to the atom cookie</td>
</tr>
<tr>
<td>spawn/2</td>
<td>Returns the pid of a new process started by the application of fun on node</td>
</tr>
<tr>
<td></td>
<td>If node does not exist, a useless pid is returned</td>
</tr>
<tr>
<td>spawn/3</td>
<td>Returns the pid of a new process started by the application of fun on node</td>
</tr>
<tr>
<td>spawn/4</td>
<td>Returns the pid of a new process started by the application of module.function(args) on node</td>
</tr>
<tr>
<td>spawn/5</td>
<td>Returns the pid of a new process started by the application of module.function(args) on node</td>
</tr>
<tr>
<td>spawn_link/2</td>
<td>Returns the pid of a new linked process started by the application of fun on node</td>
</tr>
<tr>
<td>spawn_link/4</td>
<td>Returns the pid of a new linked process started by the application of module.function(args) on node</td>
</tr>
</tbody>
</table>

### Types

\[
t \quad t :: \text{node}
\]

### Functions

**alive?/0 (function)**

Specs:

- \[\text{alive?} :: \text{boolean}\]

Returns true if the local node is alive.

That is, if the node can be part of a distributed system.

**connect/1 (function)**

Specs:

- \[\text{connect(t)} :: \text{boolean} | :\text{ignored}\]

Establishes a connection to node.

Returns true if successful, false if not, and the atom :\text{ignored} if the local node is not alive.

See [http://erlang.org/doc/man/net_kernel.html#connect_node-1](http://erlang.org/doc/man/net_kernel.html#connect_node-1) for more info.

**disconnect/1 (function)**

Specs:

- \[\text{disconnect(t)} :: \text{boolean} | :\text{ignored}\]

Forces the disconnection of a node.

This will appear to the node as if the local node has crashed. This function is mainly used in the Erlang network authentication protocols. Returns true if disconnection succeeds, otherwise false. If the local node is not alive, the function returns :\text{ignored}.

See [http://www.erlang.org/doc/man/erlang.html#disconnect_node-1](http://www.erlang.org/doc/man/erlang.html#disconnect_node-1) for more info.
get_cookie() (function)
Returns the magic cookie of the local node.
Returns the cookie if the node is alive, otherwise :nocookie.

list() (function)
Specs:
- list :: [t]
Returns a list of all visible nodes in the system, excluding the local node.
Same as list(:visible).

list(args) (function)
Specs:
- list(state | [state]) :: [t]
Returns a list of nodes according to argument given.
The result returned when the argument is a list, is the list of nodes satisfying the disjunction(s) of the list elements.
See http://www.erlang.org/doc/man/erlang.html#nodes-1 for more info.

monitor(node, flag) (function)
Specs:
- monitor(t, boolean) :: true
Monitors the status of the node.
If flag is true, monitoring is turned on. If flag is false, monitoring is turned off.
See http://www.erlang.org/doc/man/erlang.html#monitor_node-2 for more info.

monitor(node, flag, options) (function)
Specs:
- monitor(t, boolean, [:allow_passive_connect]) :: true
Behaves as monitor/2 except that it allows an extra option to be given, namely :allow_passive_connect.
See http://www.erlang.org/doc/man/erlang.html#monitor_node-3 for more info.

ping(node) (function)
Specs:
- ping(t) :: :pong | :pang
Tries to set up a connection to node.
Returns :pang if it fails, or :pong if it is successful.

Examples
iex> Node.ping(:unknown_node)
:pang

self() (function)
Specs:
- self :: t
Returns the current node.

It returns the same as the built-in `node()`.

```plaintext
set_cookie(node \ Node.self(), cookie)(function)
```

Sets the magic cookie of `node` to the atom `cookie`.

The default node is `Node.self`, the local node. If `node` is the local node, the function also sets the cookie of all other unknown nodes to `cookie`.

This function will raise `FunctionClauseError` if the given `node` is not alive.

```plaintext
spawn(node, fun)(function)
```

Specs:

- `spawn(t, (\() \to \text{any})::\text{pid}`

Retruns the pid of a new process started by the application of `fun` on `node`. If `node` does not exist, a useless pid is returned.

Check [http://www.erlang.org/doc/man/erlang.html#spawn-2](http://www.erlang.org/doc/man/erlang.html#spawn-2) for the list of available options.

Inlined by the compiler.

```plaintext
spawn(node, fun, opts)(function)
```

Specs:

- `spawn(t, (\() \to \text{any}), Process.spawn_opts) :: \text{pid} | \{\text{pid}, \text{reference}\}`

Retruns the pid of a new process started by the application of `fun` on `node`.

If `node` does not exist, a useless pid is returned. Check [http://www.erlang.org/doc/man/erlang.html#spawn_opt-3](http://www.erlang.org/doc/man/erlang.html#spawn_opt-3) for the list of available options.

Inlined by the compiler.

```plaintext
spawn(node, module, fun, args)(function)
```

Specs:

- `spawn(t, module, atom, [\text{any}]) :: \text{pid}`

Retruns the pid of a new process started by the application of `module.function(args)` on `node`.

If `node` does not exist, a useless pid is returned. Check [http://www.erlang.org/doc/man/erlang.html#spawn-4](http://www.erlang.org/doc/man/erlang.html#spawn-4) for the list of available options.

Inlined by the compiler.

```plaintext
spawn(node, module, fun, args, opts)(function)
```

Specs:

- `spawn(t, module, atom, any, Process.spawn_opts) :: \text{pid} | \{\text{pid}, \text{reference}\}`

Retruns the pid of a new process started by the application of `module.function(args)` on `node`.

If `node` does not exist, a useless pid is returned. Check [http://www.erlang.org/doc/man/erlang.html#spawn_opt-5](http://www.erlang.org/doc/man/erlang.html#spawn_opt-5) for the list of available options.

Inlined by the compiler.

```plaintext
spawn_link(node, fun)(function)
```

Specs:

- `spawn_link(t, (\() \to \text{any}) :: \text{pid}`
Returns the pid of a new linked process started by the application of `fun` on `node`.

A link is created between the calling process and the new process, atomically. If `node` does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason `:noconnection` will be received).

Inlined by the compiler.

```
spawn_link(node, module, fun, args)(function)
```

Specs:

```
•spawn_link(t, module, atom, [any]) :: pid
```

Returns the pid of a new linked process started by the application of `module.function(args)` on `node`.

A link is created between the calling process and the new process, atomically. If `node` does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason `:noconnection` will be received).

Inlined by the compiler.

### OptionParser

**Overview**  
This module contains functions to parse command line arguments.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>next/2</code></td>
<td>Low-level function that parses one option</td>
</tr>
<tr>
<td><code>parse/2</code></td>
<td>Parses <code>argv</code> into a keywords list</td>
</tr>
<tr>
<td><code>parse_head/2</code></td>
<td>Similar to <code>parse/2</code> but only parses the head of <code>argv</code>; as soon as it finds a non-switch, it stops parsing</td>
</tr>
</tbody>
</table>

**Types**

- `argv :: [String.t]`
- `parsed :: Keyword.t`
- `errors :: [{String.t, String.t} | nil]`
- `options :: [switches: Keyword.t, strict: Keyword.t, aliases: Keyword.t]`

**Functions**

```
next(argv, opts \ \ [])(function)
```

Specs:

```
•next(argv, options) :: {:ok, key :: atom, value :: term, argv} | {:invalid, key :: atom, value :: term, argv} | {:undefined, key :: atom, value :: term, argv} | {:error, argv}
```

Low-level function that parses one option.

It accepts the same options as `parse/2` and `parse_head/2` as both functions are built on top of `next`. This function may return:

- `{:ok, key, value, rest}` - the option `key` with `value` was successfully parsed
- `{:invalid, key, value, rest}` - the option `key` is invalid with `value` (returned when the switch type does not match the one given via the command line)
- `{:undefined, key, value, rest}` - the option `key` is undefined (returned on strict cases and the switch is unknown)
parse(argv, opts \ [\]) (function)

Specs:

parse(argv, options) :: {parsed, argv, errors}

Parses argv into a keywords list.
It returns the parsed values, remaining arguments and the invalid options.

Examples

iex> OptionParser.parse(["--debug"])
{[debug: true], [\], [\]}

iex> OptionParser.parse(["--source", "lib"])
{[source: "lib"], [\], [\]}

iex> OptionParser.parse(["--source-path", "lib", "test/enum_test.exs", "--verbose"])
{[source_path: "lib", verbose: true], ["test/enum_test.exs"], [\]}

By default, Elixir will try to automatically parse switches. Switches without an argument, like --debug will automatically be set to true. Switches followed by a value will be assigned to the value, always as strings.

Note Elixir also converts the switches to underscore atoms, as --source-path becomes :source_path, to better suit Elixir conventions.

Switches

Many times though, it is better to explicitly list the available switches and their formats. The switches can be specified via two different options:

* :strict - the switches are strict. Any switch that does not exist in the switch list is treated as an error;
* :switches - configure some switches. Switches that does not exist in the switch list are still attempted to be parsed;

Note only :strict or :switches may be given at once.

For each switch, the following types are supported:

* :boolean - Marks the given switch as a boolean. Boolean switches never consume the following value unless it is true or false;
* :integer - Parses the switch as an integer;
* :float - Parses the switch as a float;
* :string - Returns the switch as a string;

If a switch can’t be parsed or is not specified in the strict case, the option is returned in the invalid options list (third element of the returned tuple).

The following extra “types” are supported:

* :keep - Keeps duplicated items in the list instead of overriding;

Examples:

iex> OptionParser.parse(["--unlock", "path/to/file"], strict: [unlock: :boolean])
{[unlock: true], ["path/to/file"], [\]}

iex> OptionParser.parse(["--unlock", "--limit", "0", "path/to/file"],
...> strict: [unlock: :boolean, limit: :integer]
{[unlock: true, limit: 0], ["path/to/file"], [\]}

Elixir Documentation, Release
Elixir Documentation, Release

iex> OptionParser.parse(["--limit", "3"], strict: [limit: :integer])
{{limit: 3}, [], []}
iex> OptionParser.parse(["--limit", "xyz"], strict: [limit: :integer])
{{}, [], [{"--limit", "xyz"}]}  
iex> OptionParser.parse(["--unknown", "xyz"], strict: [])
{{}, ["xyz"], [{"--unknown", nil}]}  
iex> OptionParser.parse(["--limit", "3", "--unknown", "xyz"],
...> switches: [limit: :integer])
{{limit: 3, unknown: "xyz"}, [], []}

Negation switches

All switches starting with `--no-` are considered to be booleans and never parse the next value:
iex> OptionParser.parse(["--no-op", "path/to/file"])
{{no_op: true}, ["path/to/file"], []}

However, in case the base switch exists, it sets that particular switch to false:
iex> OptionParser.parse(["--no-op", "path/to/file"], switches: [op: :boolean])
{{op: false}, ["path/to/file"], []}

Aliases

A set of aliases can be given as options too:
iex> OptionParser.parse(["-d"], aliases: [d: :debug])
{{debug: true}, [], []}

parse_head(argv, opts \ [])(function)

Specs:

•parse_head(argv, options) :: {parsed, argv, errors}

Similar to `parse/2` but only parses the head of `argv`; as soon as it finds a non-switch, it stops parsing.

See `parse/2` for more information.

Example

iex> OptionParser.parse_head(["--source", "lib", "test/enum_test.exs", "--verbose"])
{{source: "lib"}, ["test/enum_test.exs", "--verbose"], []}
iex> OptionParser.parse_head(["--verbose", "--source", "lib", "test/enum_test.exs", "--unlock"])
{{verbose: true, source: "lib"}, ["test/enum_test.exs", "--unlock"], []}

Path

Overview   This module provides conveniences for manipulating or retrieving file system paths.

The functions in this module may receive a char data as argument (i.e. a string or a list of characters / string) and will always return a string (encoded in UTF-8).

The majority of the functions in this module do not interact with the file system, except for a few functions that require it (like `wildcard/1` and `expand/1`).

### absname/1
Converts the given path to an absolute one. Unlike `expand/1`, no attempt is made to resolve `..`, `.`, or `~`.

### absname/2
Builds a path from `relative_to` to `path`. If `path` is already an absolute path, `relative_to` is ignored. See also `relative_to/2`.

### basename/1
Returns the last component of the path or the path itself if it does not contain any directory separators.

### basename/2
Returns the last component of `path` with the extension stripped. This function should be used to remove a specific extension which may, or may not, be there.

### dirname/1
Returns the directory component of `path`.

### expand/1
Converts the path to an absolute one and expands any `.`, `..` characters and a leading `~`.

### expand/2
Expands the path relative to the path given as the second argument expanding any `.`, `..` characters. If the path is already an absolute path, `relative_to` is ignored.

### extname/1
Returns the extension of the last component of `path`.

### join/1
Returns a string with one or more path components joined by the path separator.

### join/2
Joins two paths.

### relative/1
Forces the path to be a relative path.

### relative_to/1
Returns the given `path` relative to the given `from` path. In other words, it tries to strip the prefix from `path`.

### relative_to/2
Convenience to get the path relative to the current working directory. If, for some reason, the current working directory cannot be retrieved, returns the full path.

### rootname/1
Returns the path with the extension stripped.

### rootname/2
Returns the path with the extension stripped. This function should be used to remove a specific extension which might, or might not, be there.

### split/1
Returns a list with the path split by the path separator. If an empty string is given, returns the root path.

### type/1
Returns the path type.

### wildcard/1
Traverses paths according to the given `glob` expression.

### Summary
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>absname(path)</td>
<td>Converts the given path to an absolute one. Unlike <code>expand/1</code>, no attempt is made to resolve <code>..</code>, <code>.</code>, or <code>~</code>.</td>
</tr>
<tr>
<td>absname(path, relative_to)</td>
<td></td>
</tr>
<tr>
<td>join(path)</td>
<td>Returns a string with one or more path components joined by the path separator.</td>
</tr>
<tr>
<td>relative_to(path)</td>
<td>Returns the path relative to the given from path. In other words, it tries to strip the prefix from path.</td>
</tr>
<tr>
<td>relative_to/2</td>
<td>Convenience to get the path relative to the current working directory. If, for some reason, the current working directory cannot be retrieved, returns the full path.</td>
</tr>
</tbody>
</table>

### Types

- `t` :: :unicode.chardata

### Functions

#### absname(path) (function)

- **Specs:**
  - `absname(t) :: binary`

  Converts the given path to an absolute one. Unlike `expand/1`, no attempt is made to resolve `..`, `.`, or `~`.

  **Unix examples**
  ```elixir
Pth.absname("foo")
#=> "/usr/local/foo"

Pth.absname("../x")
#=> "/usr/local/../x"
  ```

  **Windows**
  ```elixir
Pth.absname("foo")
"D:/usr/local/foo"
Pth.absname("../x")
"D:/usr/local/../x"
  ```

#### absname(path, relative_to)(function)

- **Specs:**

---

3.2. API Reference (v0.14.0-dev)  

---

303
•absname(t, t) :: binary
Builds a path from relative_to to path. If path is already an absolute path, relative_to is ignored. See also relative_to/2.
Unlike expand/2, no attempt is made to resolve .., . , or ~.

Examples
iex> Path.absname("foo", "bar")
"bar/foo"
iex> Path.absname("../x", "bar")
"bar/..\x"

basename(path) (function)
Specs:
•basename(t) :: binary
Returns the last component of the path or the path itself if it does not contain any directory separators.

Examples
iex> Path.basename("foo")
"foo"
iex> Path.basename("foo/bar")
"bar"
iex> Path.basename("/")
"

basename(path, extension) (function)
Specs:
•basename(t, t) :: binary
Returns the last component of path with the extension stripped. This function should be used to remove a specific extension which may, or may not, be there.

Examples
iex> Path.basename(~/foo/bar.ex", ".ex")
"bar"
iex> Path.basename(~/foo/bar.exs", ".ex")
"bar.exs"
iex> Path.basename(~/foo/bar.old.e", ".ex")
"bar.old"

dirname(path) (function)
Specs:
•dirname(t) :: binary
Returns the directory component of path.

Examples
Path.dirname("/foo/bar.e")
#=> "/foo"
Path.dirname("/foo/bar/baz.ex")
 #=> "/foo/bar"

**expand (path) (function)**

Specs:

- expand(t) :: binary

Converts the path to an absolute one and expands any . and .. characters and a leading ~.

**Examples**

Path.expand("/foo/bar/../bar")
"/foo/bar"

**expand (path, relative_to) (function)**

Specs:

- expand(t, t) :: binary

Expands the path relative to the path given as the second argument expanding any . and .. characters. If the path is already an absolute path, relative_to is ignored.

Note, that this function treats path with a leading ~ as an absolute one.

The second argument is first expanded to an absolute path.

**Examples**

# Assuming that the absolute path to baz is /quux/baz
Path.expand("foo/bar/..//bar", "baz")
 #=> "/quux/baz/foo/bar"

Path.expand("foo/bar/..//bar", "/baz")
"/baz/foo/bar"

Path.expand("/foo/bar/..//bar", "/baz")
"/foo/bar"

**extname (path) (function)**

Specs:

- extname(t) :: binary

Returns the extension of the last component of path.

**Examples**

iex> Path.extname("foo.erl")
".erl"

iex> Path.extname("~/foo/bar")
"

**join (list1) (function)**

Specs:

- join([t]) :: binary

Returns a string with one or more path components joined by the path separator.

This function should be used to convert a list of strings to a path. Note that any trailing slash is removed on join.

**Examples**
join(left, right)(function)

Specs:

• join(t, t) :: binary

Joins two paths.

Examples

iex> Path.join("foo", "bar")
"foo/bar"

relative(name)(function)

Specs:

• relative(t) :: binary

Forces the path to be a relative path.

Unix examples

Path.relative("/usr/local/bin")  #=> "usr/local/bin"
Path.relative("usr/local/bin")  #=> "usr/local/bin"
Path.relative("../usr/local/bin")  #=> "../usr/local/bin"

Windows examples

Path.relative("D:/usr/local/bin")  #=> "usr/local/bin"
Path.relative("usr/local/bin")  #=> "usr/local/bin"
Path.relative("D:bar.ex")  #=> "bar.ex"
Path.relative("/bar/foo.ex")  #=> "bar/foo.ex"

relative_to(path, from)(function)

Specs:

• relative_to(t, t) :: binary

Returns the given path relative to the given from path. In other words, it tries to strip the from prefix from path.

This function does not query the file system, so it assumes no symlinks in between the paths.

In case a direct relative path cannot be found, it returns the original path.

Examples

iex> Path.relative_to("/usr/local/foo", "/usr/local")
"foo"

iex> Path.relative_to("/usr/local/foo", "/")
"usr/local/foo"

iex> Path.relative_to("/usr/local/foo", "/etc")
"usr/local/foo"
relative_to_cwd(path)(function)
Specs:
  • relative_to_cwd(t) :: binary

Convenience to get the path relative to the current working directory. If, for some reason, the current working directory cannot be retrieved, returns the full path.

rootname(path)(function)
Specs:
  • rootname(t) :: binary

Returns the path with the extension stripped.

Examples
iex> Path.rootname("/foo/bar")
"/foo/bar"
iex> Path.rootname("/foo/bar.ex")
"/foo/bar"

rootname(path, extension)(function)
Specs:
  • rootname(t, t) :: binary

Returns the path with the extension stripped. This function should be used to remove a specific extension which might, or might not, be there.

Examples
iex> Path.rootname("/foo/bar.erl", ".erl")
"/foo/bar"
iex> Path.rootname("/foo/bar.erl", ".ex")
"/foo/bar.erl"

split(path)(function)
Specs:
  • split(t) :: [binary]

Returns a list with the path split by the path separator. If an empty string is given, returns the root path.

Examples
iex> Path.split(""")
[]
iex> Path.split("foo")
["foo"]
iex> Path.split("/foo/bar")
["/", "foo", "bar"]

type(name)(function)
Specs:
  • type(t) :: :absolute | :relative | :volumerelative

Returns the path type.

Unix examples
Elixir Documentation, Release

Path.type("/usr/local/bin") #=> :absolute
Path.type("usr/local/bin") #=> :relative
Path.type("../usr/local/bin") #=> :relative
Path.type("~/file") #=> :relative

Windows examples

Path.type("D:/usr/local/bin") #=> :absolute
Path.type("usr/local/bin") #=> :relative
Path.type("D:bar.ex") #=> :volumerelative
Path.type("/bar/foo.ex") #=> :volumerelative

wildcard(glob)(function)

Specs:

• wildcard(t) :: [binary]
Traverses paths according to the given glob expression.

The wildcard looks like an ordinary path, except that certain “wildcard characters” are interpreted in a special way. The following characters are special:

• ? - Matches one character.
• * - Matches any number of characters up to the end of the filename, the next dot, or the next slash.
• ** - Two adjacent *’s used as a single pattern will match all files and zero or more directories and subdirectories.
• [char1,char2,...] - Matches any of the characters listed. Two characters separated by a hyphen will match a range of characters.
• {item1,item2,...} - Matches one of the alternatives.

Other characters represent themselves. Only paths that have exactly the same character in the same position will match. Note that matching is case-sensitive; i.e. “a” will not match “A”.

Examples

Imagine you have a directory called projects with three Elixir projects inside of it: elixir, ex_doc and dynamo. You can find all .beam files inside the ebin directory of each project as follows:

Path.wildcard("projects/*/ebin/**/*.beam")

If you want to search for both .beam and .app files, you could do:

Path.wildcard("projects/*/ebin/**/*.beam,app")

Port

Overview  Functions related to Erlang ports.
### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/3</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#port_call-3">http://www.erlang.org/doc/man/erlang.html#port_call-3</a></td>
</tr>
<tr>
<td>close/1</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#port_close-1">http://www.erlang.org/doc/man/erlang.html#port_close-1</a></td>
</tr>
<tr>
<td>command/3</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#port_command-2">http://www.erlang.org/doc/man/erlang.html#port_command-2</a></td>
</tr>
<tr>
<td>connect/2</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#port_connect-2">http://www.erlang.org/doc/man/erlang.html#port_connect-2</a></td>
</tr>
<tr>
<td>control/3</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#port_control-3">http://www.erlang.org/doc/man/erlang.html#port_control-3</a></td>
</tr>
<tr>
<td>info/1</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#port_info-1">http://www.erlang.org/doc/man/erlang.html#port_info-1</a></td>
</tr>
<tr>
<td>info/2</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#port_info-2">http://www.erlang.org/doc/man/erlang.html#port_info-2</a></td>
</tr>
<tr>
<td>list/0</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#ports-0">http://www.erlang.org/doc/man/erlang.html#ports-0</a></td>
</tr>
<tr>
<td>open/2</td>
<td>See <a href="http://www.erlang.org/doc/man/erlang.html#open_port-2">http://www.erlang.org/doc/man/erlang.html#open_port-2</a></td>
</tr>
</tbody>
</table>

### Functions

**call(port, operation, data)(function)**

**close(port)(function)**

**command(port, data, options \ [])(function)**

**connect(port, pid)(function)**

**control(port, operation, data)(function)**

**info(port)(function)**

**info(port, item)(function)**

**list()(function)**

**open(name, settings)(function)**

### Process

**Overview**  Conveniences for working with processes and the process dictionary.

Besides the functions available in this module, the Kernel module exposes and auto-imports some basic functionality related to processes available through the functions:

- Kernel.spawn/1 and Kernel.spawn/3
- Kernel.spawn_link/1 and Kernel.spawn_link/3
- Kernel.spawn_monitor/1 and Kernel.spawn_monitor/3
- Kernel.self/0
- Kernel.send/2
### Summary

- **alive?/1**
  - Returns true if the process exists and is alive, that is, is not exiting and has not exited. Otherwise, returns false.

- **delete/1**
  - Deletes the given key from the dictionary.

- **demonitor/2**
  - If monitor_ref is a reference which the calling process obtained by calling monitor/1, this monitoring is turned off. If the monitoring is already turned off, nothing happens.

- **exit/2**
  - Sends an exit signal with the given reason to the pid.

- **flag/2**
  - Sets certain flags for the process which calls this function. Returns the old value of the flag.

- **flag/3**
  - Sets certain flags for the process Pid, in the same manner as flag/2. Returns the old value of the flag. The allowed values for Flag are only a subset of those allowed in flag/2, namely: save_calls.

- **get/0**
  - Returns all key-values in the dictionary.

- **get/2**
  - Returns the value for the given key.

- **get_keys/1**
  - Returns all keys that have the given value.

- **group_leader/0**
  - Returns the pid of the group leader for the process which evaluates the function.

- **group_leader/2**
  - Sets the group leader of pid to leader. Typically, this is used when a processes started from a certain shell should have another group leader than :init.

- **info/1**
  - Returns information about the process identified by pid or nil if the process is not alive. Use this only for debugging information.

- **info/2**
  - Returns information about the process identified by pid or nil if the process is not alive.

- **link/1**
  - Creates a link between the calling process and another process (or port) pid, if there is not such a link already.

- **list/0**
  - Returns a list of process identifiers corresponding to all the processes currently existing on the local node.

- **monitor/1**
  - The calling process starts monitoring the item given. It returns the monitor reference.

- **put/2**
  - Stores the given key-value in the process dictionary.

- **register/2**
  - Associates the name with a pid or a port identifier. name, which must be an atom, can be used instead of the pid / port identifier with the Kernel.send/2 function.

- **registered/0**
  - Returns a list of names which have been registered using register/2.

- **send/3**
  - Sends a message to the given process.

- **send_after/3**
  - Sends msg to dest after time milliseconds.

- **spawn/2**
  - Spawns the given module and function passing the given args according to the given options.

- **spawn/4**
  - Spawns the given module and function passing the given args according to the given options.

- **unlink/1**
  - Removes the link, if there is one, between the calling process and the process or port referred to by pid. Returns true and does not fail, even if there is no link or id does not exist.

- **unregister/1**
  - Removes the registered name, associated with a pid or a port identifier.

- **whereis/1**
  - Returns the pid or port identifier with the registered name. Returns nil if the name is not registered.

### Types

- **spawn_opt**
  - `:link | :monitor | {:priority, :low | :normal | :high} | {:fullsweep_after, non_neg_integer} | {:min_heap_size, non_neg_integer} | {:min_bin_vheap_size, non_neg_integer}`

- **spawn_opts**
  - `[spawn_opt]`

### Functions

- **alive? (pid) (function)**
  - **Specs:**
    ```
    • alive?(pid) :: boolean
    ```
  - Returns true if the process exists and is alive, that is, is not exiting and has not exited. Otherwise, returns false. pid must refer to a process at the local node.

- **delete (key) (function)**
  - **Specs:**
    ```
    ```

---

310  Chapter 3. API reference
•delete(term) :: term | nil

Deletes the given key from the dictionary.

demonitor(monitor_ref, options \ [\ ])(function)

Specs:

•demonitor(reference, options :: [flush | info]) :: boolean

If monitor_ref is a reference which the calling process obtained by calling monitor/1, this monitoring is turned off. If the monitoring is already turned off, nothing happens.

See http://www.erlang.org/doc/man/erlang.html#demonitor-2 for more info.

Inlined by the compiler.

exit(pid, reason)(function)

Specs:

•exit(pid, term) :: true

Sends an exit signal with the given reason to the pid.

The following behaviour applies if reason is any term except :normal or :kill:

1. If pid is not trapping exits, pid will exit with the given reason;
2. If pid is trapping exits, the exit signal is transformed into a message {:EXIT, from, reason} and delivered to the message queue of pid;
3. If reason is the atom :normal, pid will not exit. If it is trapping exits, the exit signal is transformed into a message {:EXIT, from, :normal} and delivered to its message queue;
4. If reason is the atom :kill, that is if exit(pid, :kill) is called, an untrappable exit signal is sent to pid which will unconditionally exit with exit reason :killed.

Inlined by the compiler.

Examples

Process.exit(pid, :kill)

flag(flag, value)(function)

Specs:

•flag(process_flag, term) :: term

Sets certain flags for the process which calls this function. Returns the old value of the flag.

See http://www.erlang.org/doc/man/erlang.html#process_flag-2 for more info.

flag(pid, flag, value)(function)

Specs:

•flag(pid, process_flag, term) :: term

Sets certain flags for the process Pid, in the same manner as flag/2. Returns the old value of the flag. The allowed values for Flag are only a subset of those allowed in flag/2, namely: save_calls.

See http://www.erlang.org/doc/man/erlang.html#process_flag-3 for more info.

get() (function)

Specs:

•get :: [{term, term}]

Returns all key-values in the dictionary.
get(key, default \ nil)(function)
Specs:
  • get(term, default :: term) :: term
    Returns the value for the given key.

get_keys(value)(function)
Specs:
  • get_keys(term) :: [term]
    Returns all keys that have the given value.

group_leader()(function)
Specs:
  • group_leader :: pid
    Returns the pid of the group leader for the process which evaluates the function.

group_leader(pid, leader)(function)
Specs:
  • group_leader(pid, leader :: pid) :: true
    Sets the group leader of pid to leader. Typically, this is used when a processes started from a certain shell
    should have another group leader than :init.

info(pid)(function)
Specs:
  • info(pid) :: Keyword.t
    Returns information about the process identified by pid or nil if the process is not alive. Use this only for
debugging information.

    See http://www.erlang.org/doc/man/erlang.html#process_info-1 for more info.

info(pid, spec)(function)
Specs:
  • info(pid, atom) :: {atom, term}
    Returns information about the process identified by pid or nil if the process is not alive.

    See http://www.erlang.org/doc/man/erlang.html#process_info-2 for more info.

link(pid)(function)
Specs:
  • link(pid | port) :: true
    Creates a link between the calling process and another process (or port) pid, if there is not such a link already.

    See http://www.erlang.org/doc/man/erlang.html#link-1 for more info.

Inlined by the compiler.

list()(function)
Specs:
  • list :: [pid]
    Returns a list of process identifiers corresponding to all the processes currently existing on the local node.

    Note that a process that is exiting, exists but is not alive, i.e., alive?/1 will return false for a process that is exiting,
    but its process identifier will be part of the result returned.
See http://www.erlang.org/doc/man/erlang.html#processes-0 for more info.

monitor(item) (function)
Specs:

•monitor(pid | {reg_name :: atom, node :: atom} | reg_name :: atom) :: reference

The calling process starts monitoring the item given. It returns the monitor reference.
See http://www.erlang.org/doc/man/erlang.html#monitor-2 for more info.

Inlined by the compiler.

put(key, value) (function)
Specs:

•put(term, term) :: term | nil

Stores the given key-value in the process dictionary.

register(pid, name) (function)
Specs:

•register(pid | port, atom) :: true

Associates the name with a pid or a port identifier. name, which must be an atom, can be used instead of the pid / port identifier with the Kernel.send/2 function.

Process.register/2 will fail with ArgumentError if the pid supplied is no longer alive, (check with alive?/1) or if the name is already registered (check with registered?/1).

registered() (function)
Specs:

•registered :: [atom]

Returns a list of names which have been registered using register/2.

send(dest, msg, options) (function)
Specs:


Sends a message to the given process.

If the option :noconnect is used and sending the message would require an auto-connection to another node the message is not sent and :noconnect is returned.

If the option :nosuspend is used and sending the message would cause the sender to be suspended the message is not sent and :nosuspend is returned.

Otherwise the message is sent and :ok is returned.

Examples

iex> Process.send({:name, :node_does_not_exist}, :hi, [:noconnect])
:noconnect

send_after(dest, msg, time) (function)
Specs:

•send_after(pid | atom, term, non_neg_integer) :: reference

Sends msg to dest after time milliseconds.
If dest is a pid, it has to be a pid of a local process, dead or alive. If dest is an atom, it is supposed to be the name of a registered process which is looked up at the time of delivery. No error is given if the name does not refer to a process.

This function returns a timer reference, which can be read or canceled with \[ :erlang.read_timer/1 \]('http://www.erlang.org/doc/man/erlang.html#read_timer-1'), \[ :erlang.start_timer/3 \]('http://www.erlang.org/doc/man/erlang.html#start_timer-3) and \[ :erlang.cancel_timer/1 \]('http://www.erlang.org/doc/man/erlang.html#cancel_timer-1). Note time cannot be greater than 4294967295.

Finally, the timer will be automatically canceled if the given dest is a pid which is not alive or when the given pid exits. Note that timers will not be automatically canceled when dest is an atom (as the atom resolution is done on delivery).

\[ \text{spawn (fun, opts)} \] (function)

Specs:

\* \text{spawn}((\() \rightarrow \text{any}), \text{spawn_opts}) :: \text{pid | \{pid, reference}}

Spawns the given module and function passing the given args according to the given options.

The result depends on the given options. In particular, if :monitor is given as an option, it will return a tuple containing the pid and the monitoring reference, otherwise just the spawned process pid.

It also accepts extra options, for the list of available options check http://www.erlang.org/doc/man/erlang.html#spawn_opt-4

Inlining by the compiler.

\[ \text{spawn (mod, fun, args, opts)} \] (function)

Specs:

\* \text{spawn}(module, atom, [], \text{spawn_opts}) :: \text{pid | \{pid, reference}}

Spawns the given module and function passing the given args according to the given options.

The result depends on the given options. In particular, if :monitor is given as an option, it will return a tuple containing the pid and the monitoring reference, otherwise just the spawned process pid.

It also accepts extra options, for the list of available options check http://www.erlang.org/doc/man/erlang.html#spawn_opt-4

Inlining by the compiler.

\[ \text{unlink (pid)} \] (function)

Specs:

\* \text{unlink(pid | port)} :: true

Removes the link, if there is one, between the calling process and the process or port referred to by pid. Returns true and does not fail, even if there is no link or id does not exist.

See http://www.erlang.org/doc/man/erlang.html#unlink-1 for more info.

Inlining by the compiler.

\[ \text{unregister (name)} \] (function)

Specs:

\* \text{unregister}(atom) :: true

Removes the registered name, associated with a pid or a port identifier.

See http://www.erlang.org/doc/man/erlang.html#unregister-1 for more info.
whereis(name)(function)
Specs:
• whereis(atom) :: pid | port | nil

Returns the pid or port identifier with the registered name. Returns nil if the name is not registered.
See http://www.erlang.org/doc/man/erlang.html#whereis-1 for more info.

Protocol
Overview  Functions for working with protocols.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert_impl!/2</td>
<td>Checks if the given module is loaded and is an implementation of the given</td>
</tr>
<tr>
<td></td>
<td>protocol</td>
</tr>
<tr>
<td>assert_protocol!/1</td>
<td>Checks if the given module is loaded and is protocol</td>
</tr>
<tr>
<td>consolidate/2</td>
<td>Receives a protocol and a list of implementations and consolidates the given</td>
</tr>
<tr>
<td></td>
<td>protocol</td>
</tr>
<tr>
<td>consolidated?&gt;/1</td>
<td>Returns true if the protocol was consolidated</td>
</tr>
<tr>
<td>def/1</td>
<td>Defines a new protocol function</td>
</tr>
<tr>
<td>extract_impls/2</td>
<td>Extract all types implemented for the given protocol from the given paths</td>
</tr>
<tr>
<td>extract_protocols/1</td>
<td>Extract all protocols from the given paths</td>
</tr>
</tbody>
</table>

Functions
assert_impl!(protocol, impl)(function)
Specs:
• assert_impl!(module, module) :: :ok | no_return

Checks if the given module is loaded and is an implementation of the given protocol.
Returns :ok if so, otherwise raises ArgumentError.

assert_protocol!(module)(function)
Specs:
• assert_protocol!(module) :: :ok | no_return

Checks if the given module is loaded and is protocol.
Returns :ok if so, otherwise raises ArgumentError.

consolidate(protocol, types)(function)
Specs:
• consolidate(module, [module]) :: {:ok, binary} | {:error, :not_a_protocol} |

Receives a protocol and a list of implementations and consolidates the given protocol.

Consolidation happens by changing the protocol impl_for in the abstract format to have fast lookup rules. Usually the list of implementations to use during consolidation are retrieved with the help of extract_impls/2.

It returns the updated version of the protocol bytecode. A given bytecode or protocol implementation can be checked to be consolidated or not by analyzing the protocol attribute:
Protocol.consolidated?({Enumerable})
If the first element of the tuple is true, it means the protocol was consolidated.

This function does not load the protocol at any point nor loads the new bytecode for the compiled module. However each implementation must be available and it will be loaded.

```elixir
consolidated?(protocol)(function)
```

Specs:

- `consolidated?(module) :: boolean`

Returns true if the protocol was consolidated.

```elixir
extract_impls(protocol, paths)(function)
```

Specs:

- `extract_impls(module, [char_list | String.t]) :: [atom]`

Extract all types implemented for the given protocol from the given paths.

The paths can be either a char list or a string. Internally they are worked on as char lists, so passing them as lists avoid extra conversion.

Does not load any of the implementations.

**Examples**

```
# Get Elixir’s ebin and retrieve all protocols
iex> path = :code.lib_dir(:elixir, :ebin)
iex> mods = Protocol.extract_impls(Enumerable, [path])
iex> List in mods
true
```

```elixir
extract_protocols(paths)(function)
```

Specs:

- `extract_protocols([char_list | String.t]) :: [atom]`

Extract all protocols from the given paths.

The paths can be either a char list or a string. Internally they are worked on as char lists, so passing them as lists avoid extra conversion.

Does not load any of the protocols.

**Examples**

```
# Get Elixir’s ebin and retrieve all protocols
iex> path = :code.lib_dir(:elixir, :ebin)
iex> mods = Protocol.extract_protocols([path])
iex> Enumerable in mods
true
```

**Macros**

```elixir
def(arg1)(macro)
```

Defines a new protocol function.

Protocols do not allow functions to be defined directly, instead, the regular `Kernel.def/`* macros are replaced by this macro which defines the protocol functions with the appropriate callbacks.
Range

Overview  Defines a Range.

A Range are represented internally as a struct. However, the most common form of creating and matching on ranges is via the ../2 macro, auto-imported from Kernel:

```iex
erange = 1..3
1..3

first .. last = range

first
1

last
3
```

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new/2</td>
<td>Creates a new range</td>
</tr>
<tr>
<td>range?/1</td>
<td>Returns true if the given argument is a range</td>
</tr>
</tbody>
</table>

Types

t  
  t :: %Range{first: term, last: term}

Functions

new(first, last) (function)
  Creates a new range.

range? (range) (function)
  Returns true if the given argument is a range.

Examples

```iex
Range.range?(1..3)
true

Range.range?(0)
false
```

Record

Overview  Module to work, define and import records.

Records are simply tuples where the first element is an atom:

```iex
Record.record? {User, "jose", 27}
true
```

This module provides conveniences for working with records at compilation time, where compile-time field names are used to manipulate the tuples, providing fast operations on top of the tuples compact structure.

In Elixir, records are used mostly in two situations:

1. To work with short, internal data;
2. To interface with Erlang records;
The macros `defrecord/3` and `defrecordp/3` can be used to create records while `extract/2` can be used to extract records from Erlang files.

| `defrecord/3` | Defines a set of macros to create and access a record |
| `defrecordp/3` | Same as `defrecord/3` but generates private macros |
| `extract/2` | Extracts record information from an Erlang file |
| `record?/1` | Checks if the given data is a record |
| `record?/2` | Checks if the given data is a record of kind |

### Macros

**defrecord(name, tag \ nil, kv)(macro)**

Defines a set of macros to create and access a record.

The macros are going to have `name`, a tag (which defaults) to the name if none is given, and a set of fields given by `kv`.

**Examples**

```elixir
defmodule User do
  defrecord :user, [name: "José", age: "25"]
end
```

In the example above, a set of macros named `user` but with different arities will be defined to manipulate the underlying record:

```
# To create records
user() #=> {:user, "José", 25}
user(age: 26) #=> {:user, "José", 26}
```

```
# To get a field from the record
user(record, :name) #=> "José"
```

```
# To update the record
user(record, age: 26) #=> {:user, "José", 26}
```

By default, Elixir uses the record name as the first element of the tuple (the tag). But it can be changed to something else:

```elixir
defmodule User do
  defrecord :user, User, name: nil
end
```

```elixir
require User
User.user() #=> {User, nil}
```

**defrecordp(name, tag \ nil, kv)(macro)**

Same as `defrecord/3` but generates private macros.

**extract(name, opts)(macro)**

Extracts record information from an Erlang file.

Returns a quoted expression containing the fields as a list of tuples. It expects the record name to be an atom and the library path to be a string at expansion time.

**Examples**

```elixir`
iex> Record.extract(:file_info, from_lib: "kernel/include/file.hrl")
record? (data) (macro)
Checks if the given data is a record.
This is implemented as a macro so it can be used in guard clauses.

Examples
iex> record = {User, "jose", 27}
iex> Record.record?(record)
true
iex> tuple = {}
iex> Record.record?(tuple)
false

record? (data, kind) (macro)
Checks if the given data is a record of kind.
This is implemented as a macro so it can be used in guard clauses.

Examples
iex> record = {User, "jose", 27}
iex> Record.record?(record, User)
true

Regex
Overview
Regular expressions for Elixir built on top of Erlang’s re module.
As the re module, Regex is based on PCRE (Perl Compatible Regular Expressions). More information can be found in the ‘re documentation <http://www.erlang.org/doc/man/re.html>’.
Regular expressions in Elixir can be created using `Regex.compile!/2` or using the special form with `~r <Kernel.html#sigil_r/2>`:

# A simple regular expressions that matches foo anywhere in the string
~r/foo/

# A regular expression with case insensitive and unicode options
~r/foo/iu

A Regex is represented internally as the `Regex` struct. Therefore, `%Regex{}` can be used whenever there is a need to match on them.

Modifiers
The modifiers available when creating a Regex are:

- **unicode (u)** - enables unicode specific patterns like . it expects valid unicode strings to be given on match
- **caseless (i)** - add case insensitivity
- **dotall (s)** - causes dot to match newlines and also set newline to anycrlf. The new line setting can be overridden by setting (*CR) or (*LF) or (*CRLF) or (*ANY) according to re documentation
- **multiline (m)** - causes ^ and $ to mark the beginning and end of each line. Use \A and \z to match the end or beginning of the string
- **extended (x)** - whitespace characters are ignored except when escaped and allow # to delimit comments
• **firstline** (f) - forces the unanchored pattern to match before or at the first newline, though the matched text may continue over the newline

• **ungreedy** (r) - inverts the “greediness” of the regexp

The options not available are:

• **anchored** - not available, use ^ or \A instead

• **dollar_endonly** - not available, use \z instead

• **no_auto_capture** - not available, use ?: instead

• **newline** - not available, use (*CR) or (*LF) or (*CRLF) or (*ANYCRLF) or (*ANY) at the beginning of the regexp according to the re documentation

Captures Many functions in this module allows what to capture in a regex match via the :capture option. The supported values are:

• **:all** - all captured subpatterns including the complete matching string. This is the default;

• **:first** - only the first captured subpattern, which is always the complete matching part of the string. All explicitly captured subpatterns are discarded;

• **:all_but_first** - all but the first matching subpattern, i.e. all explicitly captured subpatterns, but not the complete matching part of the string;

• **:none** - do not return matching subpatterns at all;

• **:all_names** - captures all names in the Regex;

• **list(binary)** - a list of named captures to capture;

---

### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile!/2</td>
<td>Compiles the regular expression according to the given options. Fails with</td>
</tr>
<tr>
<td>compile/2</td>
<td>Compiles the regular expression</td>
</tr>
<tr>
<td>escape/1</td>
<td>Escapes a string to be literally matched in a regex</td>
</tr>
<tr>
<td>match?/2</td>
<td>Returns a boolean indicating whether there was a match or not</td>
</tr>
<tr>
<td>named_captures</td>
<td>Returns the given captures as a map or nil if no captures are found. The option :return can be set to :index to get indexes back</td>
</tr>
<tr>
<td>names/1</td>
<td>Returns a list of names in the regex</td>
</tr>
<tr>
<td>opts/1</td>
<td>Returns the regex options as a string</td>
</tr>
<tr>
<td>re_pattern/1</td>
<td>Returns the underlying re_pattern in the regular expression</td>
</tr>
<tr>
<td>regex?/1</td>
<td>Returns true if the given argument is a regex</td>
</tr>
<tr>
<td>replace/4</td>
<td>Receives a regex, a binary and a replacement, returns a new binary where the all matches are replaced by replacement</td>
</tr>
<tr>
<td>run/3</td>
<td>Runs the regular expression against the given string until the first match. It returns a list with all captures or nil if no match occurred</td>
</tr>
<tr>
<td>scan/3</td>
<td>Same as run/3, but scans the target several times collecting all matches of the regular expression. A list of lists is returned, where each entry in the primary list represents a match and each entry in the secondary list represents the captured contents</td>
</tr>
<tr>
<td>source/1</td>
<td>Returns the regex source as a binary</td>
</tr>
<tr>
<td>split/3</td>
<td>Splits the given target into the number of parts specified</td>
</tr>
</tbody>
</table>

### Types

\[
t :: %Regexp{re_pattern: term, source: binary, opts: binary}
\]
**Functions**

`compile(source, options \ "") (function)`

Specs:

  * `compile(binary, binary | [term]) :: {:ok, t} | {:error, any}`

Compiles the regular expression.

The given options can either be a binary with the characters representing the same regex options given to the `~r` sigil, or a list of options, as expected by the *Erlang ‘re’ docs* [http://www.erlang.org/doc/man/re.html](http://www.erlang.org/doc/man/re.html).

It returns `{:ok, regex}` in case of success, `{:error, reason}` otherwise.

**Examples**

```elixir
iex> Regex.compile("foo")
{:ok, ~r"foo"}

iex> Regex.compile("*foo")
{:error, {'nothing to repeat', 0}}
```

`compile!(source, options \ "") (function)`

Compiles the regular expression according to the given options. Fails with `Regex.CompileError` if the regex cannot be compiled.

`escape(string) (function)`

Specs:

  * `escape(String.t) :: String.t`

Escapes a string to be literally matched in a regex.

**Examples**

```elixir
iex> Regex.escape("\."
"\\.

iex> Regex.escape("\\\what \ if")
"\\\\\what\\ \ if"
```

`match?(regex, string) (function)`

Returns a boolean indicating whether there was a match or not.

**Examples**

```elixir
iex> Regex.match?(~r/foo/, "foo")
true

iex> Regex.match?(~r/foo/, "bar")
false
```

`named_captures(regex, string, options \ []) (function)`

Returns the given captures as a map or `nil` if no captures are found. The option `:return` can be set to `:index` to get indexes back.

**Examples**

```elixir
iex> Regex.named_captures(~r/c(?<foo>d)/, "abcd")
{% "foo" => "d"}

iex> Regex.named_captures(~r/a(?<foo>b)c(?<bar>d)/, "abcd")
{% "bar" => "d", "foo" => "b"}
```
iex> Regex.named_captures(~r/a(?<foo>b)c(?<bar>d)/, "efgh")
nil

names(regex) (function)
Returns a list of names in the regex.

Examples
iex> Regex.names(~r/(?<foo>bar)/) 
["foo"]

opts(regex) (function)
Returns the regex options as a string.

Examples
iex> Regex.opts(~r(foo)m) 
"m"

re_pattern(regex) (function)
Returns the underlying re_pattern in the regular expression.

regex?(regex) (function)
Returns true if the given argument is a regex.

Examples
iex> Regex.regex?(~r/foo/) 
true
iex> Regex.regex?(0) 
false

replace(regex, string, replacement, options \ [ ])(function)
Receives a regex, a binary and a replacement, returns a new binary where the all matches are replaced by replacement.

The replacement can be either a string or a function. The string is used as a replacement for every match and it allows specific captures to be accessed via \N, where N is the capture. In case \0 is used, the whole match is inserted.

When the replacement is a function, the function may have arity N where each argument maps to a capture, with the first argument being the whole match. If the function expects more arguments than captures found, the remaining arguments will receive "."

Options
  •:global - when false, replaces only the first occurrence (defaults to true)

Examples
iex> Regex.replace(~r/d/ , "abc", "d") 
"abc"

iex> Regex.replace(~r/b/ , "abc", "d") 
"adc"

iex> Regex.replace(~r/b/ , "abc", "[\0]"
"a[bc]"

iex> Regex.replace(~r/a(b|d)c/ , "abcadc", "[\1]"
"[b][d]"
run(regex, string, options \ \ [])(function)
Runs the regular expression against the given string until the first match. It returns a list with all captures or nil if no match occurred.

Options
- :return - Set to :index to return indexes. Defaults to :binary;
- :capture - What to capture in the result. Check the moduledoc for Regex to see the possible capture values;

Examples
iex> Regex.run(~r/c(d)/, "abcd")
["cd", "d"]
iex> Regex.run(~r/e/, "abcd")
nil
iex> Regex.run(~r/c(d)/, "abcd", return: :index)
[[2,2],[3,1]]

scan(regex, string, options \ \ [])(function)
Same as run/3, but scans the target several times collecting all matches of the regular expression. A list of lists is returned, where each entry in the primary list represents a match and each entry in the secondary list represents the captured contents.

Options
- :return - Set to :index to return indexes. Defaults to :binary;
- :capture - What to capture in the result. Check the moduledoc for Regex to see the possible capture values;

Examples
iex> Regex.scan(~r/c(d|e)/, "abcd abce")
[["cd", "d"], ["ce", "e"]]
iex> Regex.scan(~r/c(?:d|e)/, "abcd abce")
[["cd"], ["ce"]]
iex> Regex.scan(~r/e/, "abcd")
[]

source(regex)(function)
Returns the regex source as a binary.

Examples
iex> Regex.source(~r/foo/)"foo"

split(regex, string, options \ \ [])(function)
Splits the given target into the number of parts specified.

Options
•:parts - when specified, splits the string into the given number of parts. If not specified, :parts is defaulted to :infinity, which will split the string into the maximum number of parts possible based on the given pattern.

•:trim - when true, remove blank strings from the result:

Examples

iex> Regex.split(~r/-/,, "a-b-c")
["a","b","c"]

iex> Regex.split(~r/-/,, "a-b-c", [parts: 2])
["a","b-c"]

iex> Regex.split(~r/-/,, "abc")
["abc"]

iex> Regex.split(~r//,, "abc")
["a", "b", "c", "]

iex> Regex.split(~r//,, "abc", trim: true)
["a", "b", "c"]

Set

Overview This module specifies the Set API expected to be implemented by different representations.

It also provides functions that redirect to the underlying Set, allowing a developer to work with different Set implementations using one API.

To create a new set, use the new functions defined by each set type:

HashSet.new #=> creates an empty HashSet

In the examples below, set_impl means a specific Set implementation, for example HashSet.

Protocols Sets are required to implement both Enumerable and Collectable protocols.

Match Sets are required to implement all operations using the match (===) operator.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete/2</td>
<td>Deletes value from set</td>
</tr>
<tr>
<td>difference/2</td>
<td>Returns a set that is set1 without the members of set2</td>
</tr>
<tr>
<td>disjoint?/2</td>
<td>Checks if set1 and set2 have no members in common</td>
</tr>
<tr>
<td>equal?/2</td>
<td>Check if two sets are equal using ===</td>
</tr>
<tr>
<td>intersection/2</td>
<td>Returns a set containing only members in common between set1 and set2</td>
</tr>
<tr>
<td>member?/2</td>
<td>Checks if set contains value</td>
</tr>
<tr>
<td>put/2</td>
<td>Inserts value into set if it does not already contain it</td>
</tr>
<tr>
<td>size/1</td>
<td>Returns the number of elements in set</td>
</tr>
<tr>
<td>subset?/2</td>
<td>Checks if set1's members are all contained in set2</td>
</tr>
<tr>
<td>to_list/1</td>
<td>Converts set to a list</td>
</tr>
<tr>
<td>union/2</td>
<td>Returns a set containing all members of set1 and set2</td>
</tr>
</tbody>
</table>

Summary
Types

value
  value :: any
values
  values :: [value]
t
  t :: %{}

Functions

delete(set, value)(function)
  Specs:
    • delete(t, value) :: t
  Deletes value from set.

Examples
  iex> s = Enum.into([1, 2, 3], set_impl.new)
iex> Set.delete(s, 4) |> Enum.sort
  [1, 2, 3]
iex> s = Enum.into([1, 2, 3], set_impl.new)
iex> Set.delete(s, 2) |> Enum.sort
  [1, 3]

difference(set1, set2)(function)
  Specs:
    • difference(t, t) :: t
  Returns a set that is set1 without the members of set2.
  Notice this function is polymorphic as it calculates the difference for of any type. Each set implementation also provides a difference function, but they can only work with sets of the same type.

Examples
  iex> Set.difference(Enum.into([1,2], set_impl.new), Enum.into([2,3,4], set_impl.new)) |> Enum.sort
  [1]

disjoint?(set1, set2)(function)
  Specs:
    • disjoint?(t, t) :: boolean
  Checks if set1 and set2 have no members in common.
  Notice this function is polymorphic as it checks for disjoint sets of any type. Each set implementation also provides a disjoint? function, but they can only work with sets of the same type.

Examples
  iex> Set.disjoint?(Enum.into([1, 2], set_impl.new), Enum.into([3, 4], set_impl.new))
  true
  iex> Set.disjoint?(Enum.into([1, 2], set_impl.new), Enum.into([2, 3], set_impl.new))
  false

equal?(set1, set2)(function)
  Specs:
•equal?(t, t) :: boolean

Check if two sets are equal using ===.

Notice this function is polymorphic as it compares sets of any type. Each set implementation also provides an equal? function, but they can only work with sets of the same type.

**Examples**

iex> Set.equal?(Enum.into([1, 2], set_impl.new), Enum.into([2, 1, 1], set_impl.new))
true

iex> Set.equal?(Enum.into([1, 2], set_impl.new), Enum.into([3, 4], set_impl.new))
false

`intersection(set1, set2)` (function)

Specs:

•intersection(t, t) :: t

Returns a set containing only members in common between `set1` and `set2`.

Notice this function is polymorphic as it calculates the intersection of any type. Each set implementation also provides a intersection function, but they can only work with sets of the same type.

**Examples**

iex> Set.intersection(Enum.into([1,2], set_impl.new), Enum.into([2,3,4], set_impl.new)) |> Enum.sort
[2]

iex> Set.intersection(Enum.into([1,2], set_impl.new), Enum.into([3,4], set_impl.new)) |> Enum.sort
[]

`member?(set, value)` (function)

Specs:

•member?(t, value) :: boolean

Checks if `set` contains `value`.

**Examples**

iex> Set.member?(Enum.into([1, 2, 3], set_impl.new), 2)
true

iex> Set.member?(Enum.into([1, 2, 3], set_impl.new), 4)
false

`put(set, value)` (function)

Specs:

•put(t, value) :: t

Inserts `value` into `set` if it does not already contain it.

**Examples**

iex> Set.put(Enum.into([1, 2, 3], set_impl.new), 3) |> Enum.sort
[1, 2, 3]

iex> Set.put(Enum.into([1, 2, 3], set_impl.new), 4) |> Enum.sort
[1, 2, 3, 4]
**size(set)**

Specs:

- `size(t) :: non_neg_integer`

Returns the number of elements in `set`.

**Examples**

```iex
iex> Set.size(Enum.into([1, 2, 3], set_impl.new))
3
```

**subset?(set1, set2)**

Specs:

- `subset?(t, t) :: boolean`

Checks if `set1`'s members are all contained in `set2`.

Notice this function is polymorphic as it checks the subset for any type. Each set implementation also provides a `subset?` function, but they can only work with sets of the same type.

**Examples**

```iex
iex> Set.subset?(Enum.into([1, 2], set_impl.new), Enum.into([1, 2, 3], set_impl.new))
true

iex> Set.subset?(Enum.into([1, 2, 3], set_impl.new), Enum.into([1, 2], set_impl.new))
false
```

**to_list(set)**

Specs:

- `to_list(t) :: []`

Converts `set` to a list.

**Examples**

```iex
iex> set_impl.to_list(Enum.into([1, 2, 3], set_impl.new)) |> Enum.sort
[1, 2, 3]
```

**union(set1, set2)**

Specs:

- `union(t, t) :: t`

Returns a set containing all members of `set1` and `set2`.

Notice this function is polymorphic as it calculates the union of any type. Each set implementation also provides a `union` function, but they can only work with sets of the same type.

**Examples**

```iex
iex> Set.union(Enum.into([1, 2], set_impl.new), Enum.into([2, 3, 4], set_impl.new)) |> Enum.sort
[1, 2, 3, 4]
```

**Callbacks**

**delete/2**

Specs:

- `delete(t, value) :: t`

3.2. API Reference (v0.14.0-dev)
difference/2(callback)
   Specs:
   • difference(t, t) :: t

disjoint?/2(callback)
   Specs:
   • disjoint?(t, t) :: boolean

equal?/2(callback)
   Specs:
   • equal?(t, t) :: boolean

intersection/2(callback)
   Specs:
   • intersection(t, t) :: t

member?/2(callback)
   Specs:
   • member?(t, value) :: boolean

new/0(callback)
   Specs:
   • new :: t

put/2(callback)
   Specs:
   • put(t, value) :: t

size/1(callback)
   Specs:
   • size(t) :: non_neg_integer

subset?/2(callback)
   Specs:
   • subset?(t, t) :: boolean

to_list/1(callback)
   Specs:
   • to_list(t) :: []

union/2(callback)
   Specs:
   • union(t, t) :: t

Stream

Overview  Module for creating and composing streams.

Streams are composable, lazy enumerables. Any enumerable that generates items one by one during enumeration is called a stream. For example, Elixir’s Range is a stream:
In the example above, as we mapped over the range, the elements being enumerated were created one by one, during enumeration. The `Stream` module allows us to map the range, without triggering its enumeration:

```iex
range = 1..3
stream = Stream.map(range, &(&1 * 2))
Enum.map(stream, &(&1 + 1))
#=> [3, 5, 7]
```

Notice we started with a range and then we created a stream that is meant to multiply each item in the range by 2. At this point, no computation was done yet. Just when `Enum.map/2` is called we enumerate over each item in the range, multiplying it by 2 and adding 1. We say the functions in `Stream` are lazy and the functions in `Enum` are eager.

Due to their laziness, streams are useful when working with large (or even infinite) collections. When chaining many operations with `Enum`, intermediate lists are created, while `Stream` creates a recipe of computations that are executed at a later moment. Let’s see another example:

```iex
1..3 |> Enum.map(&IO.inspect(&1)) |> Enum.map(&(&1 * 2)) |> Enum.map(&IO.inspect(&1))
#=> [2, 4, 6]
```

Notice that we first printed each item in the list, then multiplied each element by 2 and finally printed each new value. In this example, the list was enumerated three times. Let’s see an example with streams:

```iex
stream = 1..3 |> Stream.map(&IO.inspect(&1)) |> Stream.map(&(&1 * 2)) |> Stream.map(&IO.inspect(&1))
Enum.to_list(stream)
#=> [2, 4, 6]
```

Although the end result is the same, the order in which the items were printed changed! With streams, we print the first item and then print its double. In this example, the list was enumerated just once!

That’s what we meant when we first said that streams are composable, lazy enumerables. Notice we could call `Stream.map/2` multiple times, effectively composing the streams and they are lazy. The computations are performed only when you call a function from the `Enum` module.

### Creating Streams

There are many functions in Elixir’s standard library that return streams, some examples are:

- `IO.stream/2` - Streams input lines, one by one;
• `URI.query_decoder/1` - Decodes a query string, pair by pair;

This module also provides many convenience functions for creating streams, like `Stream.cycle/1`, `Stream.unfold/2`, `Stream.resource/3` and more.

Note the functions in this module are guaranteed to return enumerables. Since enumerables can have different shapes (structs, anonymous functions, and so on), the functions in this module may return any of those shapes and that it may change at any time. For example, a function that today returns an anonymous function may return a struct in future releases.

```plaintext
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunk/2</td>
<td>Shortcut to <code>chunk(enum, n, n)</code></td>
</tr>
<tr>
<td>chunk/4</td>
<td>Streams the enumerable in chunks, containing <code>n</code> items each, where each new</td>
</tr>
<tr>
<td></td>
<td>chunk starts <code>step</code> elements into the enumerable</td>
</tr>
<tr>
<td>chunk_by/2</td>
<td>Chunks the <code>enum</code> by buffering elements for which <code>fun</code> returns the same</td>
</tr>
<tr>
<td></td>
<td>value and only emit them when <code>fun</code> returns a new value or the <code>enum</code></td>
</tr>
<tr>
<td></td>
<td>finishes</td>
</tr>
<tr>
<td>concat/1</td>
<td>Creates a stream that enumerates each enumerable in an enumerable</td>
</tr>
<tr>
<td>concat/2</td>
<td>Creates a stream that enumerates the first argument, followed by the second</td>
</tr>
<tr>
<td>cycle/1</td>
<td>Creates a stream that cycles through the given enumerable, infinitely</td>
</tr>
<tr>
<td>drop/2</td>
<td>Lazily drops the next <code>n</code> items from the enumerable</td>
</tr>
<tr>
<td>drop_while/2</td>
<td>Lazily drops elements of the enumerable while the given function returns <code>true</code></td>
</tr>
<tr>
<td>each/2</td>
<td>Execute the given function for each item</td>
</tr>
<tr>
<td>filter/2</td>
<td>Creates a stream that filters elements according to the given function on</td>
</tr>
<tr>
<td></td>
<td>enumeration</td>
</tr>
<tr>
<td>filter_map/2</td>
<td>Creates a stream that filters and then maps elements according to given</td>
</tr>
<tr>
<td></td>
<td>functions</td>
</tr>
<tr>
<td>flat_map/2</td>
<td>Creates a stream that will apply the given function on enumeration and</td>
</tr>
<tr>
<td></td>
<td>flatten the result</td>
</tr>
<tr>
<td>into/3</td>
<td>Injects the stream values into the given collectable as a side-effect</td>
</tr>
<tr>
<td>iterate/2</td>
<td>Emit a sequence of values, starting with <code>start_value</code>. Successive values</td>
</tr>
<tr>
<td></td>
<td>are generated by calling <code>next_fun</code> on the previous value</td>
</tr>
<tr>
<td>map/2</td>
<td>Creates a stream that will apply the given function on enumeration</td>
</tr>
<tr>
<td>reject/2</td>
<td>Creates a stream that will reject elements according to the given function</td>
</tr>
<tr>
<td></td>
<td>on enumeration</td>
</tr>
<tr>
<td>repeatedly/1</td>
<td>Returns a stream generated by calling <code>generator_fun</code> repeatedly</td>
</tr>
<tr>
<td>resource/3</td>
<td>Emits a sequence of values for the given resource</td>
</tr>
<tr>
<td>run/1</td>
<td>Runs the given stream</td>
</tr>
<tr>
<td>scan/2</td>
<td>Creates a stream that applies the given function to each element, emits the</td>
</tr>
<tr>
<td></td>
<td>result and uses the same result as the accumulator for the next computation</td>
</tr>
<tr>
<td>scan/3</td>
<td>Creates a stream that applies the given function to each element, emits the</td>
</tr>
<tr>
<td></td>
<td>result and uses the same result as the accumulator for the next computation.</td>
</tr>
<tr>
<td></td>
<td>Uses the given <code>acc</code> as starting value</td>
</tr>
<tr>
<td>take/2</td>
<td>Lazily takes the next <code>n</code> items from the enumerable and stops enumeration</td>
</tr>
<tr>
<td>take_every/2</td>
<td>Creates a stream that takes every <code>n</code> item from the enumerable</td>
</tr>
<tr>
<td>take_while/2</td>
<td>Lazily takes elements of the enumerable while the given function returns <code>true</code></td>
</tr>
<tr>
<td>transform/1</td>
<td>Transforms an existing stream</td>
</tr>
<tr>
<td>unfold/2</td>
<td>Emits a sequence of values for the given accumulator</td>
</tr>
<tr>
<td>uniq/2</td>
<td>Creates a stream that only emits elements if they are unique</td>
</tr>
<tr>
<td>with_index/1</td>
<td>Creates a stream where each item in the enumerable will be wrapped in a</td>
</tr>
<tr>
<td></td>
<td>tuple alongside its index</td>
</tr>
<tr>
<td>zip/2</td>
<td>Zips two collections together, lazily</td>
</tr>
</tbody>
</table>
```

**Types**

```plaintext
t :: %Stream{enum: term, funs: term, accs: term, done: term}
acc
  acc :: any

element
  element :: any
```

330 Chapter 3. API reference
Functions

chunk (enum, n) (function)

specs:

- \( \text{chunk(Enumerable.t, non\_neg\_integer)} :: \text{Enumerable.t} \)

Shortcut to \( \text{chunk(enum, n, n)} \).

chunk (enum, n, step, pad \( \\null \) nil) (function)

specs:

- \( \text{chunk(Enumerable.t, non\_neg\_integer, non\_neg\_integer, Enumerable.t \| \text{nil}) :: Enumerable.t} \)

Streams the enumerable in chunks, containing \( n \) items each, where each new chunk starts \( \text{step} \) elements into the enumerable.

\( \text{step} \) is optional and, if not passed, defaults to \( n \), i.e. chunks do not overlap. If the final chunk does not have \( n \) elements to fill the chunk, elements are taken as necessary from \( \text{pad} \) if it was passed. If \( \text{pad} \) is passed and does not have enough elements to fill the chunk, then the chunk is returned anyway with less than \( n \) elements. If \( \text{pad} \) is not passed at all or is \( \text{nil} \), then the partial chunk is discarded from the result.

Examples

```iex
iex> Stream.chunk([1, 2, 3, 4, 5, 6], 2) |> Enum.to_list
[[1, 2], [3, 4], [5, 6]]

iex> Stream.chunk([1, 2, 3, 4, 5, 6], 3, 2) |> Enum.to_list
[[1, 2, 3], [3, 4, 5]]

iex> Stream.chunk([1, 2, 3, 4, 5, 6], 3, 2, [7]) |> Enum.to_list
[[1, 2, 3], [3, 4, 5], [5, 6, 7]]

iex> Stream.chunk([1, 2, 3, 4, 5, 6], 3, 3, []) |> Enum.to_list
[[1, 2, 3], [4, 5, 6]]
```

chunk_by (enum, fun) (function)

specs:

- \( \text{chunk\_by(Enumerable.t, (element -> any)) :: Enumerable.t} \)

Chunks the \( \text{enum} \) by buffering elements for which \( \text{fun} \) returns the same value and only emit them when \( \text{fun} \) returns a new value or the \( \text{enum} \) finishes.

Examples

```iex
iex> stream = Stream.chunk_by([1, 2, 2, 3, 4, 4, 6, 7, 7], &({rem(&1, 2) == 1}))
iex> Enum.to_list(stream)
[[1], [2, 2], [3], [4, 4, 6], [7, 7]]
```

concat (enumerables) (function)

specs:

- \( \text{concat(Enumerable.t)} :: \text{Enumerable.t} \)

Creates a stream that enumerates each enumerable in an enumerable.

Examples
Elixir Documentation, Release

iex> stream = Stream.concat([1..3, 4..6, 7..9])
iex> Enum.to_list(stream)  
[1,2,3,4,5,6,7,8,9]

concat(first, second)(function)
Specs:
  *concat(Enumerable.t,Enumerable.t):: Enumerable.t

Creates a stream that enumerates the first argument, followed by the second.

Examples
iex> stream = Stream.concat(1..3, 4..6)
iex> Enum.to_list(stream)  
[1,2,3,4,5,6]
iex> stream1 = Stream.cycle([1, 2, 3])
iex> stream2 = Stream.cycle([4, 5, 6])
iex> stream = Stream.concat(stream1, stream2)
iex> Enum.take(stream, 6)  
[1,2,3,1,2,3]

cycle(enumerable)(function)
Specs:
  *cycle(Enumerable.t):: Enumerable.t

Creates a stream that cycles through the given enumerable, infinitely.

Examples
iex> stream = Stream.cycle([1,2,3])
iex> Enum.take(stream, 5)  
[1,2,3,1,2]

drop(enum, n)(function)
Specs:
  *drop(Enumerable.t,non_neg_integer):: Enumerable.t

Lazily drops the next n items from the enumerable.

If a negative n is given, it will drop the last n items from the collection. Note that the mechanism by which this is implemented will delay the emission of any item until n additional items have been emitted by the enum.

Examples
iex> stream = Stream.drop(1..10, 5)
iex> Enum.to_list(stream)  
[6,7,8,9,10]
iex> stream = Stream.drop(1..10, -5)
iex> Enum.to_list(stream)  
[1,2,3,4,5]

drop_while(enum, fun)(function)
Specs:
  *drop_while(Enumerable.t,(element -> as_boolean(term))):: Enumerable.t

Lazily drops elements of the enumerable while the given function returns true.

Examples
Elixir Documentation, Release

```elixir
iex> stream = Stream.drop_while(1..10, &(&1 <= 5))
iex> Enum.to_list(stream)
[6,7,8,9,10]
```

**each**(enum, fun)(function)

Specs:

- `each(Enumerable.t, (element -> term)) :: Enumerable.t`

Execute the given function for each item.

Useful for adding side effects (like printing) to a stream.

**Examples**

```elixir
iex> stream = Stream.each([1, 2, 3], fn(x) -> send self, x end)
iex> Enum.to_list(stream)
[1, 2, 3]
```

**filter**(enum, fun)(function)

Specs:

- `filter(Enumerable.t, (element -> as_boolean(term))) :: Enumerable.t`

Creates a stream that filters elements according to the given function on enumeration.

**Examples**

```elixir
iex> stream = Stream.filter([1, 2, 3], fn(x) -> rem(x, 2) == 0 end)
iex> Enum.to_list(stream)
[2]
```

**filter_map**(enum, filter, mapper)(function)

Specs:

- `filter_map(Enumerable.t, (element -> as_boolean(term)), (element -> any)) :: Enumerable.t`

Creates a stream that filters and then maps elements according to given functions.

Exists for symmetry with `Enum.filter_map/3`.

**Examples**

```elixir
iex> stream = Stream.filter_map(1..6, fn(x) -> rem(x, 2) == 0 end, &(&1 * 2))
iex> Enum.to_list(stream)
[4,8,12]
```

**flat_map**(enum, mapper)(function)

Specs:

- `flat_map(Enumerable.t, (element -> Enumerable.t)) :: Enumerable.t`

Creates a stream that will apply the given function on enumeration and flatten the result.

**Examples**

```elixir
iex> stream = Stream.flat_map([1, 2, 3], fn(x) -> [x, x * 2] end)
iex> Enum.to_list(stream)
[1, 2, 4, 3, 6]
```
### into(enum, collectable, transform \ fn x -> x end)(function)
Injections the stream values into the given collectable as a side-effect.

This function is often used with \run/1\ since any evaluation is delayed until the stream is executed. See \run/1\ for an example.

### iterate(start_value, next_fun)(function)
**Specs:**
- \iterate(element,(element -> element)):: Enumerable.t\n
Emit a sequence of values, starting with \start_value\. Successive values are generated by calling \next_fun\ on the previous value.

**Examples**
```iex
iex> Stream.iterate(0, &(&1+1)) |> Enum.take(5)
[0,1,2,3,4]
```

### map(enum, fun)(function)
**Specs:**
- \map(Enumerable.t,(element -> any)):: Enumerable.t\n
Creates a stream that will apply the given function on enumeration.

**Examples**
```iex
iex> stream = Stream.map([1, 2, 3], fn(x) -> x * 2 end)
iex> Enum.to_list(stream)
[2,4,6]
```

### reject(enum, fun)(function)
**Specs:**
- \reject(Enumerable.t,(element -> as_boolean(term))):: Enumerable.t\n
Creates a stream that will reject elements according to the given function on enumeration.

**Examples**
```iex
iex> stream = Stream.reject([1, 2, 3], fn(x) -> rem(x, 2) == 0 end)
iex> Enum.to_list(stream)
[1,3]
```

### repeatedly(generator_fun)(function)
**Specs:**
- \repeatedly((() -> element)):: Enumerable.t\n
Returns a stream generated by calling \generator_fun\ repeatedly.

**Examples**
```iex
iex> Stream.repeatedly(&:random.uniform/0) |> Enum.take(3)
[0.4435846174457203, 0.7230402056221108, 0.94581636451987]
```

### resource(start_fun, next_fun, after_fun)(function)
**Specs:**
- \resource((() -> acc),(acc -> \{element,acc\} | nil),(acc -> term)):: Enumerable.t\n
Emits a sequence of values for the given resource.
Similar to `unfold/2` but the initial value is computed lazily via `start_fun` and executes an `after_fun` at the end of enumeration (both in cases of success and failure).

Successive values are generated by calling `next_fun` with the previous accumulator (the initial value being the result returned by `start_fun`) and it must return a tuple with the current and next accumulator. The enumeration finishes if it returns `nil`.

As the name says, this function is useful to stream values from resources.

**Examples**

```elixir
Stream.resource(fn -> File.open!("sample") end,
  fn file ->
    case IO.read(file, :line) do
      data when is_binary(data) -> {data, file}
      _ -> nil
    end
  end,
  fn file -> File.close(file) end)
```

`run(stream)` (function)

**Specs:**

- `run(Enumerable.t) :: :ok`

Runs the given stream.

This is useful when a stream needs to be run, for side effects, and there is no interest in its return result.

**Examples**

Open up a file, replace all # by % and stream to another file without loading the whole file in memory:

```elixir
stream = File.stream!("code")
|> Stream.map(&String.replace(&1, ",#", ",%")
|> Stream.into(File.stream!("new"))
```

No computation will be done until we call one of the `Enum` functions or `Stream.run/1`.

`scan(enum, fun)` (function)

**Specs:**

- `scan(Enumerable.t, (element, acc -> any)) :: Enumerable.t`

Creates a stream that applies the given function to each element, emits the result and uses the same result as the accumulator for the next computation.

**Examples**

```iex
iex> stream = Stream.scan(1..5, &(&1 + &2))
iex> Enum.to_list(stream)
[1,3,6,10,15]
```

`scan(enum, acc, fun)` (function)

**Specs:**

- `scan(Enumerable.t, acc, (element, acc -> any)) :: Enumerable.t`

Creates a stream that applies the given function to each element, emits the result and uses the same result as the accumulator for the next computation. Uses the given `acc` as the starting value.

**Examples**
take (enum, n) (function)
Specs:

  • take(Enumerable.t, non_neg_integer) :: Enumerable.t

Lazily takes the next n items from the enumerable and stops enumeration.

If a negative n is given, the last n values will be taken. For such, the collection is fully enumerated keeping up to 2 * n elements in memory. Once the end of the collection is reached, the last count elements will be executed. Therefore, using a negative n on an infinite collection will never return.

Examples

  iex> stream = Stream.take(1..100, 5)
  iex> Enum.to_list(stream)
[1,2,3,4,5]

  iex> stream = Stream.take(1..100, -5)
  iex> Enum.to_list(stream)
[96,97,98,99,100]

  iex> stream = Stream.cycle([1, 2, 3]) |> Stream.take(5)
  iex> Enum.to_list(stream)
[1,2,3,1,2]

take_every (enum, n) (function)
Specs:

  • take_every(Enumerable.t, non_neg_integer) :: Enumerable.t

Creates a stream that takes every n item from the enumerable.

The first item is always included, unless n is 0.

Examples

  iex> stream = Stream.take_every(1..10, 2)
  iex> Enum.to_list(stream)
[1,3,5,7,9]

take_while (enum, fun) (function)
Specs:

  • take_while(Enumerable.t, (element -> as_boolean(term))) :: Enumerable.t

Lazily takes elements of the enumerable while the given function returns true.

Examples

  iex> stream = Stream.take_while(1..100, &(&1 <= 5))
  iex> Enum.to_list(stream)
[1,2,3,4,5]

transform (enum, acc, reducer) (function)
Specs:

  • (transform(Enumerable.t, acc, fun) :: Enumerable.t) when fun: (element, acc -> {Enumerable.t, acc} | {:halt, acc}), acc: any
Transforms an existing stream.

It expects an accumulator and a function that receives each stream item and an accumulator, and must return a tuple containing a new stream (often a list) with the new accumulator or a tuple with `:halt` as first element and the accumulator as second.

Note: this function is similar to `Enum.flat_map_reduce/3` except the latter returns both the flat list and accumulator, while this one returns only the stream.

**Examples**

`Stream.transform/3` is a useful as it can be used as basis to implement many of the functions defined in this module. For example, we can implement `Stream.take(enum, n)` as follows:

```iex
iex> enum = 1..100
iex> n = 3
iex> stream = Stream.transform(enum, 0,
                      fn
                        i, acc ->
                        if acc < n, do: ([i], acc + 1),
                        else: (:halt, acc)
                      end)
iex> Enum.to_list(stream)
```

`with_index(enum)` is a function that creates a stream where each item in the enumerable will be wrapped in a tuple alongside its index.

```iex
iex> Stream.with_index([1, 2, 3]) |> Enum.to_list
[[1, 0], [2, 1], [3, 2]]
```
iex> stream = Stream.with_index([1, 2, 3])
iex> Enum.to_list(stream)
[[1,0],[2,1],[3,2]]

\textbf{zip(left, right)(function)}

\begin{itemize}
  \item \textbf{zip(Enumerable.t, Enumerable.t) :: Enumerable.t}
\end{itemize}

Zips two collections together, lazily.

The zipping finishes as soon as any enumerable completes.

\textbf{Examples}

iex> concat = Stream.concat(1..3, 4..6)
iex> cycle = Stream.cycle([:a, :b, :c])
iex> Stream.zip(concat, cycle) \|> Enum.to_list
[[1,:a],[2,:b],[3,:c],[4,:a],[5,:b],[6,:c]]

\section*{String}

\textbf{Overview}  \quad A String in Elixir is a UTF-8 encoded binary.

\textbf{String and binary operations}  \quad The functions in this module act according to the Unicode Standard, version 6.3.0. For example, \texttt{capitalize/1}, \texttt{downcase/1}, \texttt{strip/1} are provided by this module.

In addition to this module, Elixir provides more low-level operations that work directly with binaries. Some of those can be found in the \texttt{Kernel} module, as:

\begin{itemize}
  \item \texttt{Kernel.binary_part/3} - retrieves part of the binary
  \item \texttt{Kernel.bit_size/1} and \texttt{Kernel.byte_size/1} - size related functions
  \item \texttt{Kernel.is_bitstring/1} and \texttt{Kernel.is_binary/1} - type checking function
  \item Plus a number of functions for working with binaries (bytes) in the \texttt{':binary'} module <http://erlang.org/doc/man/binary.html>
\end{itemize}

\textbf{Codepoints and graphemes}  \quad As per the Unicode Standard, a codepoint is an Unicode Character, which may be represented by one or more bytes. For example, the character “é” is represented with two bytes:

\begin{verbatim}
iex> byte_size("é")
2
\end{verbatim}

However, this module returns the proper length:

\begin{verbatim}
iex> String.length("é")
1
\end{verbatim}

Furthermore, this module also presents the concept of graphemes, which are multiple characters that may be “perceived as a single character” by readers. For example, the same “é” character written above could be represented by the letter “e” followed by the accent:

\begin{verbatim}
iex> string = "\x{0065}\x{0301}"
iex> byte_size(string)
3
iex> String.length(string)
1
\end{verbatim}
Although the example above is made of two characters, it is perceived by users as one.

Graphemes can also be two characters that are interpreted as one by some languages. For example, some languages may consider “ch” as a grapheme. However, since this information depends on the locale, it is not taken into account by this module.

In general, the functions in this module rely on the Unicode Standard, but does not contain any of the locale specific behaviour.

More information about graphemes can be found in the Unicode Standard Annex #29. This current Elixir version implements Extended Grapheme Cluster algorithm.

**Integer codepoints**  Although codepoints could be represented as integers, this module represents all codepoints as strings. For example:

```iex
iex> String.codepoints("josé")
["j", "o", "s", "é"]
```

There are a couple of ways to retrieve a character integer codepoint. One may use the ? special macro:

```iex
iex> ?j
106
iex> ?é
233
```

Or also via pattern matching:

```iex
iex> << eacute :: utf8 >> = "é"
iex> eacute
233
```

As we have seen above, codepoints can be inserted into a string by their hexadecimal code:

"jos\x{0065}\x{0301}"

#=>
"josé"

**Self-synchronization**  The UTF-8 encoding is self-synchronizing. This means that if malformed data (i.e., data that is not possible according to the definition of the encoding) is encountered, only one codepoint needs to be rejected.

This module relies on this behaviour to ignore such invalid characters. For example, `length/1` is going to return a correct result even if an invalid codepoint is fed into it.

In other words, this module expects invalid data to be detected when retrieving data from the external source. For example, a driver that reads strings from a database will be the one responsible to check the validity of the encoding.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>at/2</td>
<td>Returns the grapheme in the position of the given utf8 string. If position is greater than string length, return nil.</td>
</tr>
<tr>
<td>capitalize/1</td>
<td>Converts the first character in the given string to uppercase and the remaining to lowercase.</td>
</tr>
<tr>
<td>chunk/2</td>
<td>Splits the string into chunks of characters that share a common trait.</td>
</tr>
<tr>
<td>codepoints/1</td>
<td>Returns all codepoints in the string.</td>
</tr>
<tr>
<td>contains?/2</td>
<td>Check if string contains any of the given contents.</td>
</tr>
<tr>
<td>downcase/1</td>
<td>Convert all characters on the given string to lowercase.</td>
</tr>
<tr>
<td>duplicate/2</td>
<td>Returns a binary subject duplicated n times.</td>
</tr>
<tr>
<td>ends_with?/2</td>
<td>Returns true if string ends with any of the suffixes given, otherwise false. suffixes can be either a single suffix or a list of suffixes.</td>
</tr>
<tr>
<td>first/1</td>
<td>Returns the first grapheme from an utf8 string, nil if the string is empty.</td>
</tr>
<tr>
<td>graphemes/1</td>
<td>Returns unicode graphemes in the string as per Extended Grapheme Cluster algorithm outlined in the Unicode Standard.</td>
</tr>
</tbody>
</table>
Table 3.6 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>last/1</td>
<td>Returns the last grapheme from an utf8 string, nil if the string is empty</td>
</tr>
<tr>
<td>length/1</td>
<td>Returns the number of unicode graphemes in an utf8 string</td>
</tr>
<tr>
<td>ljust/2</td>
<td>Returns a new string of length len with subject left justified and padded with padding. If padding is not present, it defaults to whitespace. When len is less than the length of subject, subject is returned.</td>
</tr>
<tr>
<td>ljust/3</td>
<td>Reverses the given string. Works on graphemes</td>
</tr>
<tr>
<td>lstrip/1</td>
<td>Returns a string where leading Unicode whitespace has been removed</td>
</tr>
<tr>
<td>lstrip/2</td>
<td>Returns a string where leading char have been removed</td>
</tr>
<tr>
<td>match/?2</td>
<td>Check if string matches the given regular expression</td>
</tr>
<tr>
<td>next_codepoint/1</td>
<td>Returns the next codepoint in a String</td>
</tr>
<tr>
<td>next_grapheme/1</td>
<td>Returns the next grapheme in a String</td>
</tr>
<tr>
<td>printable?/1</td>
<td>Checks if a string is printable considering it is encoded as UTF-8. Returns true if so, false otherwise.</td>
</tr>
<tr>
<td>replace/4</td>
<td>Returns a new binary based on subject by replacing the parts matching pattern by replacement.</td>
</tr>
<tr>
<td>reverse/1</td>
<td>Reverses the given string. Works on graphemes</td>
</tr>
<tr>
<td>rjust/2</td>
<td>Returns a new string of length len with subject right justified and padded with padding. If padding is not present, it defaults to whitespace. When len is less than the length of subject, subject is returned.</td>
</tr>
<tr>
<td>rjust/3</td>
<td>Returns a string where trailing Unicode whitespace has been removed</td>
</tr>
<tr>
<td>rstrip/1</td>
<td>Returns a string where trailing char have been removed</td>
</tr>
<tr>
<td>rstrip/2</td>
<td>Returns a substring from the offset given by the start of the range to the offset given by the end of the range.</td>
</tr>
<tr>
<td>slice/2</td>
<td>Returns a substring starting at the offset given by the first, and a length given by the second. If the offset is negative, location is counted from the start of the string.</td>
</tr>
<tr>
<td>slice/3</td>
<td>Divides a string into substrings based on a pattern</td>
</tr>
<tr>
<td>split/1</td>
<td>Divides a string into substrings at each Unicode whitespace occurrence with leading and trailing whitespace removed.</td>
</tr>
<tr>
<td>split/3</td>
<td>Splits a string into two at the specified offset. When the offset given is negative, location is counted from the end of the string.</td>
</tr>
<tr>
<td>split_at/2</td>
<td>Returns a string where leading/trailing Unicode whitespace has been removed</td>
</tr>
<tr>
<td>starts_with/?2</td>
<td>Returns a string where leading/trailing char have been removed</td>
</tr>
<tr>
<td>strip/1</td>
<td>Converts a string to an atom</td>
</tr>
<tr>
<td>strip/2</td>
<td>Converts a string to a char list</td>
</tr>
<tr>
<td>to_atom/1</td>
<td>Converts a string to an existing atom</td>
</tr>
<tr>
<td>to_char_list/1</td>
<td>Returns a float whose text representation is string</td>
</tr>
<tr>
<td>to_existing_atom/1</td>
<td>Returns a integer whose text representation is string</td>
</tr>
<tr>
<td>to_float/1</td>
<td>Converts all characters on the given string to uppercase</td>
</tr>
<tr>
<td>to_integer/1</td>
<td>Checks whether str contains only valid characters</td>
</tr>
<tr>
<td>to_integer/2</td>
<td>Checks whether str is a valid character</td>
</tr>
<tr>
<td>upcase/1</td>
<td></td>
</tr>
<tr>
<td>valid?/1</td>
<td></td>
</tr>
<tr>
<td>valid_character?/1</td>
<td></td>
</tr>
</tbody>
</table>

Summary

Types

- `t` :: binary
- `codepoint` :: `t`
- `grapheme` :: `t`

Functions

- `at(string, position)` (function)

  Specs:

  • `at(t, integer) :: grapheme | nil`

  Returns the grapheme in the position of the given utf8 string. If position is greater than string length, than it returns nil.
Examples

iex> String.at("elixir", 0)
"e"

iex> String.at("elixir", 1)
"l"

iex> String.at("elixir", 10)
nil

iex> String.at("elixir", -1)
"r"

iex> String.at("elixir", -10)
nil

capitalize(string) (function)
Specs:

- capitalize(t) :: t

Converts the first character in the given string to uppercase and the remaining to lowercase.
This relies on the titlecase information provided by the Unicode Standard. Note this function makes no attempt to capitalize all words in the string (usually known as titlecase).

Examples

iex> String.capitalize("abcd")
"Abcd"

iex> String.capitalize("n")
"Fin"

iex> String.capitalize("jose")
"José"

chunk(string, trait) (function)
Specs:

- chunk(t, :valid | :printable) :: [t]

Splits the string into chunks of characters that share a common trait.
The trait can be one of two options:

- :valid – the string is split into chunks of valid and invalid character sequences
- :printable – the string is split into chunks of printable and non-printable character sequences

Returns a list of binaries each of which contains only one kind of characters.
If the given string is empty, an empty list is returned.

Examples

iex> String.chunk(<<?a, ?b, ?c, 0>>, :valid)
["abc\000"]

iex> String.chunk(<<?a, ?b, ?c, 0, 0x0fff::utf8>>, :valid)
["abc\000", <<0x0fff::utf8>>]
codepoints(string)(function)
Specs:
  • codepoints(t):: [codepoint]

Returns all codepoints in the string.

Examples
iex> String.codepoints("josé")
["j", "o", "s", "é"]
iex> String.codepoints(" ")
["\\", ",", ",", ",", ",", ",", ",", ",", ",", ",", ",", ",", ",", ",", ",", ",", ","]
iex> String.codepoints(""")
["", ",", ","]

contains?(string, contents)(function)
Specs:
  • contains?(t, t | [t]):: boolean

Check if string contains any of the given contents. matches can be either a single string or a list of strings.

Examples
iex> String.contains? "elixir of life", "of"
true
iex> String.contains? "elixir of life", ["life", "death"]
true
iex> String.contains? "elixir of life", ["death", "mercury"]
false

downcase(binary)(function)
Specs:
  • downcase(t):: t

Convert all characters on the given string to lowercase.

Examples
iex> String.downcase("ABCD")
"abcd"
iex> String.downcase("AB 123 XPTO")
"ab 123 xpto"
iex> String.downcase("JOSÉ")
"josé"
Returns a binary subject duplicated n times.

Examples

```iex
iex> String.duplicate("abc", 0)
"

iex> String.duplicate("abc", 1)
"abc"

iex> String.duplicate("abc", 2)
"abcabc"
```

`ends_with?` function

Specs:

- `ends_with?(t, t | [...]) :: boolean`

Returns true if string ends with any of the suffixes given, otherwise false. suffixes can be either a single suffix or a list of suffixes.

Examples

```iex
iex> String.ends_with? "language", "age"
true

iex> String.ends_with? "language", ["youth", "age"]
true

iex> String.ends_with? "language", ["youth", "elixir"]
false
```

`first` function

Specs:

- `first(t) :: grapheme | nil`

Returns the first grapheme from an utf8 string, nil if the string is empty.

Examples

```iex
iex> String.first("elixir")
"e"

iex> String.first(""")
"
```

`graphemes` function

Specs:

- `graphemes(t) :: [grapheme]`

Returns unicode graphemes in the string as per Extended Grapheme Cluster algorithm outlined in the Unicode Standard Annex #29, Unicode Text Segmentation.

Examples

```iex
iex> String.graphemes("stute")
["", "s", "t", "u", "t", "e"]
```

`last` function

Specs:

- `last(t) :: grapheme | nil`
Returns the last grapheme from an utf8 string, nil if the string is empty.

Examples

iex> String.last("elixir")
"r"

iex> String.last("")
"

length(string) (function)

Specs:

- `length(t) :: non_neg_integer`

Returns the number of unicode graphemes in an utf8 string.

Examples

iex> String.length("elixir")
6

iex> String.length("")
5

ljust(subject, len) (function)

Specs:

- `ljust(t, pos_integer) :: t`

Returns a new string of length `len` with `subject` left justified and padded with `padding`. If `padding` is not present, it defaults to whitespace. When `len` is less than the length of `subject`, `subject` is returned.

Examples

iex> String.ljust("abc", 5)
"abc  

iex> String.ljust("abc", 5, ?-)
"abc--"  

ljust(subject, len, padding) (function)

Specs:

- `ljust(t, pos_integer, char) :: t`

lstrip(binary) (function)

Returns a string where leading Unicode whitespace has been removed.

Examples

iex> String.lstrip(" abc ")
"abc  

lstrip(other, char) (function)

Specs:

- `lstrip(t, char) :: t`

Returns a string where leading `char` have been removed.

Examples
match?(string, regex) (function)
Specs:
  • match?(t, Regex.t) :: boolean

Check if string matches the given regular expression.

Examples
iex> String.match?("foo", ~r/foo/)
true
iex> String.match?("bar", ~r/foo/)
false

next_codepoint(string) (function)
Specs:
  • next_codepoint(t) :: {codepoint, t} | nil

Returns the next codepoint in a String.

The result is a tuple with the codepoint and the remaining of the string or nil in case the string reached its end.

As with other functions in the String module, this function does not check for the validity of the codepoint. That said, if an invalid codepoint is found, it will be returned by this function.

Examples
iex> String.next_codepoint("José")
{"j", "osé"}

next_grapheme(string) (function)
Specs:
  • next_grapheme(t) :: {grapheme, t} | nil

Returns the next grapheme in a String.

The result is a tuple with the grapheme and the remaining of the string or nil in case the String reached its end.

Examples
iex> String.next_grapheme("José")
{"j", "osé"}

printable?(b) (function)
Specs:
  • printable?(t) :: boolean

Checks if a string is printable considering it is encoded as UTF-8. Returns true if so, false otherwise.

Examples
iex> String.printable?("abc")
true

replace(subject, pattern, replacement, options \ []) (function)
Specs:
  • replace(t, t, t, Keyword.t) :: t
Returns a new binary based on subject by replacing the parts matching pattern by replacement. By default, it replaces all entries, except if the global option is set to false.

A pattern may be a string or a regex.

**Examples**

```iex
iex> String.replace("a,b,c", ",", "-")
"a-b-c"

iex> String.replace("a,b,c", ",", "-", global: false)
"a-b,c"
```

The pattern can also be a regex. In those cases, one can give \N in the replacement string to access a specific capture in the regex:

```iex
iex> String.replace("a,b,c", ~r/,(.)/, ",\1\1")
"a,bb,cc"
```

Notice we had to escape the escape character \. By giving \, one can inject the whole matched pattern in the replacement string.

When strings are used as a pattern, a developer can also use the replaced part inside the replacement via the :insert_replaced option:

```iex
iex> String.replace("a,b,c", "b", "[", insert_replaced: 1)
"a,[b],c"

iex> String.replace("a,b,c", ",", "]", insert_replaced: 2)
"a[[],b[[],c"

iex> String.replace("a,b,c", ",", "]", insert_replaced: [1, 1])
"a[,,]b[,,]c"
```

**reverse**(string) (function)

Specs:

```
  • reverse(t) :: t
```

Reverses the given string. Works on graphemes.

**Examples**

```iex
iex> String.reverse("abcd")
"dcba"

iex> String.reverse("hello world")
"dlrow olleh"

iex> String.reverse("hello og")
"go olleh"
```

**rjust**(subject, len) (function)

Specs:

```
  • rjust(t, pos_integer) :: t
```

Returns a new string of length len with subject right justified and padded with padding. If padding is not present, it defaults to whitespace. When len is less than the length of subject, subject is returned.

**Examples**
iex> String.rjust("abc", 5)
  " abc"

iex> String.rjust("abc", 5, ?-)
  "--abc"

**rjust(subject, len, padding)(function)**

Specs:

•rjust(t, pos_integer, char) :: t

**rstrip(binary)(function)**

Specs:

•rstrip(t) :: t

Returns a string where trailing Unicode whitespace has been removed.

**Examples**

iex> String.rstrip(" abc ")
  " abc"

**rstrip(string, char)(function)**

Specs:

•rstrip(t, char) :: t

Returns a string where trailing char have been removed.

**Examples**

iex> String.rstrip(" abc _", ?_)
  " abc "

**slice(string, range)(function)**

Specs:

•slice(t, Range.t) :: t | nil

Returns a substring from the offset given by the start of the range to the offset given by the end of the range. If the start of the range is not a valid offset for the given string or if the range is in reverse order, returns nil.

**Examples**

iex> String.slice("elixir", 1..3)
  "lix"

iex> String.slice("elixir", 1..10)
  "lixir"

iex> String.slice("elixir", 10..3)
  nil

iex> String.slice("elixir", -4..-1)
  "ixir"

iex> String.slice("elixir", 2..-1)
  "ixir"

iex> String.slice("elixir", -4..6)
  "ixir"
slice(string, start, len)(function)
Specs:
  • slice(t, integer, integer) :: grapheme | nil

Returns a substring starting at the offset given by the first, and a length given by the second. If the offset is greater than string length, than it returns nil.

Examples
iex> String.slice("elixir", 1, 3)
"lix"
iex> String.slice("elixir", 1, 10)
"lixir"
iex> String.slice("elixir", 10, 3)
nil
iex> String.slice("elixir", -4, 4)
"ixir"
iex> String.slice("elixir", -10, 3)
nil
iex> String.slice("a", 0, 1500)
"a"
iex> String.slice("a", 1, 1500)
"
iex> String.slice("a", 2, 1500)
nil

split(binary)(function)
Specs:
  • split(t) :: [t]

Divides a string into substrings at each Unicode whitespace occurrence with leading and trailing whitespace ignored.

Examples
split(binary, pattern, options \ \ [])(function)

Specs:

*split(t, t | [t] | Regex.t, Keyword.t)::[t]

Divides a string into substrings based on a pattern.

Returns a list of these substrings. The pattern can be a string, a list of strings or a regular expression.

The string is split into as many parts as possible by default, but can be controlled via the parts: num option. If you pass parts: :infinity, it will return all possible parts.

Empty strings are only removed from the result if the trim option is set to true.

Examples

Splitting with a string pattern:

iex> String.split("a,b,c", ",")
["a", "b", "c"]

iex> String.split("a,b,c", ",", parts: 2)
["a", "b,c"]

iex> String.split(" a b c ", " ", trim: true)
["a", "b", "c"]

A list of patterns:

iex> String.split("1,2 3,4", [" ", ","]) 
["1", "2", "3", "4"]

A regular expression:

iex> String.split("a,b,c", ~r{,})
["a", "b", "c"]

iex> String.split("a,b,c", ~r{,}, parts: 2)
["a", "b,c"]

iex> String.split(" a b c ", ~r{\s}, trim: true)
["a", "b", "c"]

Splitting on empty patterns returns codepoints:

iex> String.split("abc", ~r{})
["a", "b", "c", "]

iex> String.split("abc", "")
["a", "b", "c", "]

iex> String.split("abc", ",", trim: true)
["a", "b", "c"]
iex> String.split("abc", ",", parts: 2)
["a", "bc"]

(split_at(string, offset))(function)

Specs:

- split_at(t, integer) :: {t, t}

Splits a string into two at the specified offset. When the offset given is negative, location is counted from the end of the string.

The offset is capped to the length of the string.

Returns a tuple with two elements.

Examples

iex> String.split_at "sweetelixir", 5
{"sweet", "elixir"}

iex> String.split_at "sweetelixir", -6
{"sweet", "elixir"}

iex> String.split_at "abc", 0
{"", "abc"}

iex> String.split_at "abc", 1000
{"abc", ""}

iex> String.split_at "abc", -1000
{"", "abc"}

(starts_with?(string, prefixes))(function)

Specs:

- starts_with?(t, t | [t]) :: boolean

Returns true if string starts with any of the prefixes given, otherwise false. prefixes can be either a single prefix or a list of prefixes.

Examples

iex> String.starts_with? "elixir", "eli"
true

iex> String.starts_with? "elixir", ["erlang", "elixir"]
true

iex> String.starts_with? "elixir", ["erlang", "ruby"]
false

strip(string)(function)

Specs:

- strip(t) :: t

Returns a string where leading/trailing Unicode whitespace has been removed.

Examples

iex> String.strip(" abc ")
"abc"
strip(string, char)(function)
Specs:

•strip(t, char) :: t

Returns a string where leading/trailing char have been removed.

Examples

iex> String.strip("a abc a", ?a)
" abc "

\n
to_atom(string)(function)
Specs:

•to_atom(String.t) :: atom

Converts a string to an atom.

Currently Elixir does not support conversions from strings which contains Unicode codepoints greater than 0xFF.

Inlined by the compiler.

Examples

iex> String.to_atom("my_atom")
:my_atom

\n
to_char_list(string)(function)
Specs:

•to_char_list(t) :: char_list

Converts a string into a char list.

Examples

iex> String.to_char_list("æß")
'æß'

Notice that this function expect a list of integer representing UTF-8 codepoints. If you have a raw binary, you must instead use the `:binary` module <http://erlang.org/doc/man/binary.html>__.

\n
to_existing_atom(string)(function)
Specs:

•to_existing_atom(String.t) :: atom

Converts a string to an existing atom.

Currently Elixir does not support conversions from strings which contains Unicode codepoints greater than 0xFF.

Inlined by the compiler.

Examples

iex> :my_atom
iex> String.to_existing_atom("my_atom")
:my_atom

iex> String.to_existing_atom("this_atom_will_never_exist")
** (ArgumentError) argument error
to_float(string)(function)
Specs:
•to_float(String.t):: float
Returns a float whose text representation is string.
Inlined by the compiler.
Examples
iex> String.to_float("2.2017764e+0")
2.2017764

to_integer(string)(function)
Specs:
•to_integer(String.t):: integer
Returns an integer whose text representation is string.
Inlined by the compiler.
Examples
iex> String.to_integer("123")
123

to_integer(string, base)(function)
Specs:
•to_integer(String.t, pos_integer):: integer
Returns an integer whose text representation is string in base base.
Inlined by the compiler.
Examples
iex> String.to_integer("3FF", 16)
1023

upcase(binary)(function)
Specs:
•upcase(t):: t
Convert all characters on the given string to uppercase.
Examples
iex> String.upcase("abcd")
"ABCD"

iex> String.upcase("ab 123 xpto")
"AB 123 XPTO"

iex> String.upcase("José")
"JOSÉ"

valid?(arg1)(function)
Specs:
•valid?(t):: boolean
Elixir Documentation, Release

Checks whether str contains only valid characters.
Examples
iex> String.valid?("a")
true
iex> String.valid?("ø")
true
iex> String.valid?(<<0xffff :: 16>>)
false
iex> String.valid?("asd" <> <<0xffff :: 16>>)
false

valid_character?(codepoint)(function)
Specs:
•valid_character?(t) :: boolean
Checks whether str is a valid character.
All characters are codepoints, but some codepoints are not valid characters. They may be reserved, private, or
other.
Examples
iex> String.valid_character?("a")
true
iex> String.valid_character?("ø")
true
iex> String.valid_character?("\x{ffff}")
false

StringIO

Overview This module provides an IO device that wraps a string.
Examples
iex> {:ok, pid} = StringIO.open("foo")
iex> IO.read(pid, 2)
"fo"

Summary

close/1
contents/1
open/2

Stops the IO device and returns remaining buffers
Returns current buffers
Creates an IO device

Functions
close(pid)(function)
Specs:
•close(pid) :: {:ok, {binary, binary}}
3.2. API Reference (v0.14.0-dev)

353


Stops the IO device and returns remaining buffers.

**Examples**

```elixir
terms/ast/ast.ex: iex> {:ok, pid} = StringIO.open("in")
iex> IO.write(pid, "out")
iex> StringIO.close(pid)
{:ok, 
    {"in", "out"}}
```

**contents** *(pid) (function)*

**Specs:**

- `contents(pid) :: {binary, binary}`

Returns current buffers.

**Examples**

```elixir
terms/ast/ast.ex: iex> {:ok, pid} = StringIO.open("in")
iex> IO.write(pid, "out")
iex> StringIO.contents(pid)
{"in", "out"}
```

**open**(string, options \ 
\ [ ])(function)

**Specs:**

- `open(binary, Keyword.t) :: {:ok, pid}`

Creates an IO device.

If the `:capture_prompt` option is set to `true`, prompts (specified as arguments to `IO.get*` functions) are captured.

**Examples**

```elixir
terms/ast/ast.ex: iex> {:ok, pid} = StringIO.open("foo")
iex> IO.gets(pid, ">")
"foo"
iex> StringIO.contents(pid)
{"", ""}

iex> {:ok, pid} = StringIO.open("foo", capture_prompt: true)
iex> IO.gets(pid, ">")
"foo"
iex> StringIO.contents(pid)
{"", ">"}
```

**Supervisor**

**Overview**  A behaviour module for implementing super vision functionality.

A supervisor is a process which supervises other processes called child processes. Supervisors are used to build an hierarchical process structure called a supervision tree, a nice way to structure fault-tolerant applications.

A supervisor implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an supervision tree.

**Example**  In order to define a supervisor, we need to first define a child process that is going to be supervised. In order to do so, we will define a GenServer that represents a stack:
defmodule Stack do
  use GenServer

  def start_link(state) do
    GenServer.start_link(__MODULE__, state, [name: :sup_stack])
  end

  def handle_call(:pop, _from, [h|t]) do
    {:reply, h, t}
  end

  def handle_cast({:push, h}, _from, t) do
    {:noreply, [h|t]}
  end
end

We can now define our supervisor and start it as follows:

# Import helpers for defining supervisors
import Supervisor.Spec

# We are going to supervise the Stack server which will
# be started with a single argument [:hello]
children = [worker(Stack, [:hello])]

# Start the supervisor with our one child
{:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)

Notice that when starting the GenServer, we have registered it with name :sup_stack, which allows us to call it directly and get what is on the stack:

GenServer.call(:sup_stack, :pop)
#=> :hello

GenServer.cast(:sup_stack, {:push, :world})
#=> :ok

GenServer.call(:sup_stack, :pop)
#=> :world

However, there is a bug in our stack server. If we call :pop and the stack is empty, it is going to crash because no clause matches. Let’s try it:

GenServer.call(:sup_stack, :pop)
=ERROR REPORT====

Luckily, since the server is being supervised by a supervisor, the supervisor will automatically start a new one, with the default stack of [:hello] like before:

GenServer.call(:sup_stack, :pop) == :hello

Supervisors support different strategies; in the example above, we have chosen :one_for_one. Furthermore, each supervisor can have many workers and supervisors as children, each of them with their specific configuration, shutdown values, and restart strategies.

Continue reading this moduledoc to learn more about supervision strategies and then follow to the Supervisor.Spec module documentation to learn about the specification for workers and supervisors.
Module-based supervisors  In the example above, a supervisor was dynamically created by passing the supervision structure to `start_link/2`. However, supervisors can also be created by explicitly defining a supervisor module:

```elixir
defmodule MyApp.Supervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, [])
  end

  def init([]) do
    import Supervisor.Spec

    children = [
      worker(Stack, [:hello])
    ]

    supervisor(children, strategy: :one_for_one)
  end
end
```

You may want to use a module-based supervisor if:

- You need to do some particular action on supervisor initialization, like setting up an ETS table;
- You want to perform partial hot-code swapping of the tree. For example, if you add or remove a children, the module-based supervision will add and remove the new children directly, while the dynamic supervision requires the whole tree to be restarted in order to perform such swaps;

Strategies

- **:one_for_one** - If a child process terminates, only that process is restarted;
- **:one_for_all** - If a child process terminates, all other child processes are terminated and then all child processes (including the terminated one) are restarted;
- **:rest_for_one** - If a child process terminates, the “rest” of the child processes, i.e. the child processes after the terminated one in start order, are terminated. Then the terminated child process and the rest of the child processes are restarted;
- **:simple_one_for_one** - Similar to :one_for_one but suits better when dynamically attaching children. This strategy requires the supervisor specification to contain only one children. Many functions in this module behave slightly differently when this strategy is used;

Name Registration  A supervisor is bound to the same name registration rules as a GenServer. Read more about it in the GenServer docs.

<table>
<thead>
<tr>
<th>count_children/1</th>
<th>Returns a map containing count values for the supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete_child/2</td>
<td>Deletes the child specification identified by child_id</td>
</tr>
<tr>
<td>restart_child/2</td>
<td>Restarts a child process identified by child_id</td>
</tr>
<tr>
<td>start_child/2</td>
<td>Dynamically adds and starts a child specification to the supervisor</td>
</tr>
<tr>
<td>start_link/2</td>
<td>Starts a supervisor with the given children</td>
</tr>
<tr>
<td>start_link/3</td>
<td>Starts a supervisor module with the given arg</td>
</tr>
<tr>
<td>terminate_child/2</td>
<td>Terminates the given pid or child id</td>
</tr>
<tr>
<td>which_children/1</td>
<td>Returns a list with information about all children</td>
</tr>
</tbody>
</table>
Types

**on_start**

```
on_start :: {:ok, pid} | :ignore | {:error, {:already_started, pid} | :shutdown, term} | term
```

Return values of `start_link` functions

**on_start_child**

```
on_start_child :: {:ok, child} | {:ok, child, info :: term} | {:error, {:already_started, child} | :already_present | term}
```

Return values of `start_child` functions

**child**

```
child :: pid | undefined
```

**name**

```
name :: atom | :global, term | :via, module, term
```

The Supervisor name

**options**

```
options :: [name: name, strategy: Supervisor.Spec.strategy, max_restarts: non_neg_integer, max_seconds: non_neg_integer]
```

Options used by the `start*` functions

**supervisor**

```
supervisor :: pid | name | {atom, node}
```

The supervisor reference

Functions

**count_children(supervisor)(function)**

Specs:

```
•count_children(supervisor) :: specs: non_neg_integer, active: non_neg_integer, supervisors: non_neg_integer, workers: non_neg_integer
```

Returns a map containing count values for the supervisor.

The map contains the following keys:

- `:specs` - the total count of children, dead or alive;
- `:active` - the count of all actively running child processes managed by this supervisor;
- `:supervisors` - the count of all supervisors whether or not the child process is still alive;
- `:workers` - the count of all workers, whether or not the child process is still alive;

**delete_child(supervisor, child_id)(function)**

Specs:

```
•(delete_child(supervisor, Supervisor.Spec.child_id) :: :ok | {:error, error}) when error: :not_found | :simple_one_for_one | :running | :restarting
```

Deletes the child specification identified by `child_id`.

The corresponding child process must not be running, use `terminate_child/2` to terminate it.

If successful, the function returns `:ok`. This function may error with an appropriate error tuple if the `child_id` is not found, or if the current process is running or being restarted.

This operation is not supported by `simple_one_for_one` supervisors.
**restart_child** (supervisor, child_id)(function)

Specs:

- *(restart_child(supervisor, Supervisor.Spec.child_id)): {ok, child} | {ok, child, term} | {error, error}) when error: not_found | :simple_one_for_one | :running | :restarting | term

Restarts a child process identified by child_id.

The child specification must exist and the corresponding child process must not be running.

Note that for temporary children, the child specification is automatically deleted when the child terminates, and thus it is not possible to restart such children.

If the child process start function returns {:ok, child} or {:ok, child, info}, the pid is added to the supervisor and the function returns the same value.

If the child process start function returns :ignore, the pid remains set to undefined and the function returns {:ok, undefined}.

This function may error with an appropriate error tuple if the child_id is not found, or if the current process is running or being restarted.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the function returns {:error, error}.

This operation is not supported by simple_one_for_one supervisors.

**start_child** (supervisor, child_spec_or_args)(function)

Specs:

- *start_child(supervisor, Supervisor.Spec.spec | [term])) :: on_start_child

Dynamically adds and starts a child specification to the supervisor.

child_spec should be a valid child specification (unless the supervisor is a :simple_one_for_one supervisor, see below). The child process will be started as defined in the child specification.

In the case of :simple_one_for_one, the child specification defined in the supervisor will be used and instead of a child_spec, an arbitrary list of terms is expected. The child process will then be started by appending the given list to the existing function arguments in the child specification.

If there already exists a child specification with the specified id, child_spec is discarded and the function returns an error with :already_started or :already_present if the corresponding child process is running or not.

If the child process start function returns {:ok, child} or {:ok, child, info}, the child specification and pid is added to the supervisor and the function returns the same value.

If the child process start function returns :ignore, the child specification is added to the supervisor, the pid is set to undefined and the function returns {:ok, undefined}.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the child specification is discarded and the function returns {:error, error} where error is a term containing information about the error and child specification.

**start_link** (children, options)(function)

Specs:

- *start_link([tuple], options): on_start

Starts a supervisor with the given children.

A strategy is required to be given as an option. Furthermore, the :max_restarts and :max_seconds value can be configured as described in Supervisor.Spec.supervise/2 docs.
The options can also be used to register a supervisor name. The supported values are described under the Name Registration section in the GenServer module docs.

If the supervisor and its child processes are successfully created (i.e. if the start function of all child processes returns \{:ok, child\}, \{:ok, child, info\}, or :ignore\) the function returns \{:ok, pid\}, where pid is the pid of the supervisor. If there already exists a process with the specified name, the function returns \{:error, \{already_started, pid\}\}, where pid is the pid of that process.

If any of the child process start functions fail or return an error tuple or an erroneous value, the supervisor will first terminate all already started child processes with reason :shutdown and then terminate itself and return \{:error, \{shutdown, reason\}\}.

\[\text{start_link}(\text{module}, \text{arg}, \text{options } \backslash \text{ []})\](function)

\text{Starts a supervisor module with the given arg.}

To start the supervisor, the init/1 callback will be invoked in the given module. The init/1 callback must return a supervision specification which can be created with the help of Supervisor.Spec module.

If the init/1 callback returns :ignore, this function returns :ignore as well and the supervisor terminates with reason :normal. If it fails or returns an incorrect value, this function returns \{:error, term\} where term is a term with information about the error, and the supervisor terminates with reason term.

The :name option can also be given in order to register a supervisor name, the supported values are described under the Name Registration section in the GenServer module docs.

Other failure conditions are specified in \text{start_link/2} docs.

\[\text{terminate_child}(\text{supervisor}, \text{pid_or_child_id})\](function)

\text{Terminates the given pid or child id.}

If the supervisor is not a simple_one_for_one, the child id is expected and the process, if there is one, is terminated; the child specification is kept unless the child is temporary.

In case of a simple_one_for_one supervisor, a pid is expected. If the child specification identifier is given instead of a pid, the function will return \{:error, :simple_one_for_one\}.

A non-temporary child process may later be restarted by the supervisor. The child process can also be restarted explicitly by calling \text{restart_child/2}. Use \text{delete_child/2} to remove the child specification.

If successful, the function returns :ok. If there is no child specification or pid, the function returns \{:error, :not_found\}.

\[\text{which_children}(\text{supervisor})\](function)

\text{Returns a list with information about all children.}

Note that calling this function when supervising a large number of children under low memory conditions can cause an out of memory exception.

This function returns a list of tuples containing:
Elixir Documentation, Release

• id - as defined in the child specification or :undefined in the case of a simple_one_for_one supervisor;
• child - the pid of the corresponding child process, the atom :restarting if the process is about to be restarted, or :undefined if there is no such process;
• type - :worker or :supervisor as defined in the child specification;
• modules – as defined in the child specification;

Supervisor.Spec

Overview  Convenience functions for defining a supervision specification.

Example  By using the functions in this module one can define a supervisor and start it with Supervisor.start_link/2:

```elixir
import Supervisor.Spec

children = [
  worker(MyWorker, [arg1, arg2, arg3]),
  supervisor(MySupervisor, [arg1])
]

Supervisor.start_link(children, strategy: :one_for_one)
```

In many situations, it may be handy to define supervisors backed by a module:

```elixir
defmodule MySupervisor do
  use Supervisor

  def start_link(arg) do
    Supervisor.start_link(__MODULE__, arg)
  end

  def init(arg) do
    children = [
      worker(MyWorker, [arg], restart: :temporary)
    ]
    
    supervise(children, strategy: :simple_one_for_one)
  end
end
```

Notice in this case we don’t have to explicitly import Supervisor.Spec as use Supervisor automatically does so.

Explicit supervisors as above are required when there is a need to:

1. partially change the supervision tree during hot-code swaps;
2. define supervisors inside other supervisors;
3. perform actions inside the supervision init/1 callback.

For example, you may want to start an ETS table that is linked to the supervisor (i.e. if the supervision tree needs to be restarted, the ETS table must be restarted too);
**Supervisor and worker options**  In the example above, we have defined workers and supervisors and each accepts the following options:

- **:id** - a name used to identify the child specification internally by the supervisor. Defaults to the given module name;
- **:function** - the function to invoke on the child to start it;
- **:restart** - defines when the child process should restart;
- **:shutdown** - defines how a child process should be terminated;
- **:modules** - it should be a list with one element [module], where module is the name of the callback module only if the child process is a Supervisor or GenServer If the child process is a GenEvent, modules should be :dynamic;

**Restart values**  The following restart values are supported:

- **:permanent** - the child process is always restarted;
- **:temporary** - the child process is never restarted (not even when the supervisor’s strategy is :rest_for_one or :one_for_all);
- **:transient** - the child process is restarted only if it terminates abnormally, i.e. with another exit reason than :normal, :shutdown or {:shutdown, term};

**Shutdown values**  The following shutdown values are supported:

- **:brutal_kill** - the child process is unconditionally terminated using exit(child, :kill);
- **:infinity** - if the child process is a supervisor, it is a mechanism to give the subtree enough time to shutdown. It can also be used with workers with care;
- Finally, it can also be any integer meaning that the supervisor tells the child process to terminate by calling Process.exit(child, :shutdown) and then waits for an exit signal back. If no exit signal is received within the specified time (in miliseconds), the child process is unconditionally terminated using Process.exit(child, :kill);

| **Summary** | **supervise/2** | Receives a list of children (workers or supervisors) to supervise and a set of options |
| **supervisor/3** | Defines the given module as a supervisor which will be started with the given arguments |
| **worker/3** | Defines the given module as a worker which will be started with the given arguments |

**Types**

- **strategy**

  - strategy :: :simple_one_for_one | :one_for_one | :one_for_all | :rest_for_one

  Supported strategies

- **restart**

  - restart :: :permanent | :transient | :temporary

  Supported restart values

- **shutdown**

  - shutdown :: :brutal_kill | :infinity | non_neg_integer

  Supported shutdown values
worker

worker :: worker | supervisor

Supported worker values

modules

modules :: dynamic | [module]

Supported module values

child_id

child_id :: term

Supported id values

spec

spec :: {child_id, start_fun :: [module, atom, [term]], restart, shutdown, worker, modules}

The supervisor specification

Functions

supervise(children, options)(function)

Specs:

*supervise([spec], strategy: strategy, max_restarts: non_neg_integer, max_seconds: non_neg_integer) :: {:ok, tuple}

Receives a list of children (workers or supervisors) to supervise and a set of options.

Returns a tuple containing the supervisor specification.

Examples

supervise children, strategy: :one_for_one

Options

*strategy - the restart strategy option. It can be either :one_for_one, :rest_for_one, :one_for_all, or :simple_one_for_one. You can learn more about strategies in the Supervisor module docs;

*max_restarts - the maximum amount of restarts allowed in a time frame. Defaults to 5;

*max_seconds - the time frame in which :max_restarts applies. Defaults to 5;

The :strategy option is required and by default maximum 5 restarts are allowed within 5 seconds. Please check the Supervisor module for a complete description of the available strategies.

supervisor(module, args, options \ [[]]])(function)

Specs:

*supervisor(module, [term], restart: restart, shutdown: shutdown, id: term, function: atom, modules: modules) :: spec

Defines the given module as a supervisor which will be started with the given arguments.

supervisor ExUnit.Runner, [], restart: :permanent

By default, the function start_link is invoked on the given module. Overall, the default values for the options are:

[id: module,
 function: :start_link,
 restart: :permanent,
worker(module, args, options \ [\ [])(function)

Specs:

•worker(module, [term], restart: restart, shutdown: shutdown, id: term, function: atom, modules: modules) :: spec

Defines the given module as a worker which will be started with the given arguments.

worker ExUnit.Runner, [], restart: :permanent

By default, the function start_link is invoked on the given module. Overall, the default values for the options are:

[id: module,
 function: :start_link,
 restart: :permanent,
 shutdown: 5000,
 modules: [module]]

Check Supervisor.Spec module docs for more information on the options.

System

Overview  The System module provides access to variables used or maintained by the VM and to functions that interact directly with the VM or the host system.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv/0</td>
<td>List command line arguments</td>
</tr>
<tr>
<td>argv/1</td>
<td>Modify command line arguments</td>
</tr>
<tr>
<td>at_exit/1</td>
<td>Register a program exit handler function</td>
</tr>
<tr>
<td>build_info/0</td>
<td>Elixir build information</td>
</tr>
<tr>
<td>cmd/1</td>
<td>Execute a system command</td>
</tr>
<tr>
<td>cmd!/0</td>
<td>Current working directory, exception on error</td>
</tr>
<tr>
<td>cwd/0</td>
<td>Current working directory</td>
</tr>
<tr>
<td>delete_env/1</td>
<td>Deletes an environment variable</td>
</tr>
<tr>
<td>find_executable/1</td>
<td>Locate an executable on the system</td>
</tr>
<tr>
<td>get_env/0</td>
<td>System environment variables</td>
</tr>
<tr>
<td>get_env/1</td>
<td>Environment variable value</td>
</tr>
<tr>
<td>get_PID/0</td>
<td>Erlang VM process identifier</td>
</tr>
<tr>
<td>halt/1</td>
<td>Halt the Erlang runtime system</td>
</tr>
<tr>
<td>put_env/1</td>
<td>Set multiple environment variables</td>
</tr>
<tr>
<td>put_env/2</td>
<td>Set an environment variable value</td>
</tr>
<tr>
<td>stacktrace/0</td>
<td>Last exception stacktrace</td>
</tr>
<tr>
<td>tmp_dir!/0</td>
<td>Writable temporary directory, exception on error</td>
</tr>
<tr>
<td>tmp_dir/0</td>
<td>Writable temporary directory</td>
</tr>
<tr>
<td>user_home!/0</td>
<td>User home directory, exception on error</td>
</tr>
<tr>
<td>user_home/0</td>
<td>User home directory</td>
</tr>
<tr>
<td>version/0</td>
<td>Elixir version information</td>
</tr>
</tbody>
</table>

Functions
argv() (function)
Specs:

   • argv :: [String.t]

List command line arguments.
Returns the list of command line arguments passed to the program.

argv(args) (function)
Specs:

   • argv(String.t) :: :ok

Modify command line arguments.
Changes the list of command line arguments. Use it with caution, as it destroys any previous argv information.

at_exit(fun) (function)
Register a program exit handler function.

Registers a function that will be invoked at the end of program execution. Useful for invoking a hook in “script” mode.

The function must receive the exit status code as an argument.

build_info() (function)
Specs:

   • build_info :: Keyword.t

Elixir build information.

Returns a keyword list with Elixir version, git tag info and compilation date.

cmd(command) (function)
Specs:

   • cmd(char_list) :: char_list
   • cmd(String.t) :: String.t

Execute a system command.

Executes command in a command shell of the target OS, captures the standard output of the command and returns the result as a binary.

If command is a char list, a char list is returned. Otherwise a string, correctly encoded in UTF-8, is expected.

cwd() (function)
Current working directory.

Returns the current working directory or nil if one is not available.

cwd!() (function)
Current working directory, exception on error.

Returns the current working directory or raises RuntimeError.

delete_env(varname) (function)
Specs:

   • delete_env(String.t) :: :ok

Deletes an environment variable.

Removes the variable varname from the environment.
**find_executable** (program) (function)

Specs:

- find_executable(char_list) :: char_list | nil
- find_executable(binary) :: binary | nil

Locate an executable on the system.

This function looks up an executable program given its name using the environment variable PATH on Unix and Windows. It also considers the proper executable extension for each OS, so for Windows it will try to lookup files with `.com`, `.cmd` or similar extensions.

If `program` is a char list, a char list is returned. Returns a binary otherwise.

**get_env ()** (function)

Specs:

- get_env :: %{String.t => String.t}

System environment variables.

Returns a list of all environment variables. Each variable is given as a `{name, value}` tuple where both `name` and `value` are strings.

**get_env** (varname) (function)

Specs:

- get_env(binary) :: binary | nil

Environment variable value.

Returns the value of the environment variable `varname` as a binary, or `nil` if the environment variable is undefined.

**get_pid ()** (function)

Specs:

- get_pid :: binary

Erlang VM process identifier.

Returns the process identifier of the current Erlang emulator in the format most commonly used by the operating system environment.

See http://www.erlang.org/doc/man/os.html# getpid-0 for more info.

**halt** (status \ 0) (function)

Specs:

- halt(non_neg_integer | binary | :abort) :: no_return

Halt the Erlang runtime system.

Halts the Erlang runtime system where the argument `status` must be a non-negative integer, the atom `:abort` or a binary.

- If an integer, the runtime system exits with the integer value which is returned to the operating system;
- If `:abort`, the runtime system aborts producing a core dump, if that is enabled in the operating system;
- If a string, an erlang crash dump is produced with status as slogan, and then the runtime system exits with status code 1;

Note that on many platforms, only the status codes 0-255 are supported by the operating system.

For more information, check: http://www.erlang.org/doc/man/erlang.html#halt-1
Examples

```erlang
System.halt(0)
System.halt(1)
System.halt(:abort)
```

**put_env(dict) (function)**

Specs:

```
•put_env(Dict.t) :: :ok
```

Set multiple environment variables.

Sets a new value for each environment variable corresponding to each key in `dict`.

**put_env(varname, value) (function)**

Specs:

```
•put_env(binary, binary) :: :ok
```

Set an environment variable value.

Sets a new value for the environment variable `varname`.

**stacktrace() (function)**

Last exception stacktrace.

Note that the Erlang VM (and therefore this function) does not return the current stacktrace but rather the stacktrace of the latest exception.

Inlined by the compiler into `{erlang.get_stacktrace/0}'(http://www.erlang.org/doc/man/erlang.html#get_stacktrace-0).

**tmp_dir() (function)**

Writable temporary directory.

Returns a writable temporary directory. Searches for directories in the following order:

1. The directory named by the TMPDIR environment variable
2. The directory named by the TEMP environment variable
3. The directory named by the TMP environment variable
4. C:\TMP on Windows or /tmp on Unix
5. As a last resort, the current working directory

Returns `nil` if none of the above are writable.

**tmp_dir!() (function)**

Writable temporary directory, exception on error.

Same as `tmp_dir/0` but raises `RuntimeError` instead of returning `nil` if no temp dir is set.

**user_home() (function)**

User home directory.

Returns the user home directory (platform independent). Returns `nil` if no user home is set.

**user_home!() (function)**

User home directory, exception on error.

Same as `user_home/0` but raises `RuntimeError` instead of returning `nil` if no user home is set.

**version() (function)**

Specs:
•version :: String.t

Elixir version information.

Returns Elixir’s version as binary.

Task

Overview  Conveniences for spawning and awaiting for tasks.

Tasks are processes that meant to execute one particular action throughout their life-cycle, often with little explicit communication with other processes. The most common use case for tasks is to compute a value asynchronously:

```elixir
task = Task.async(fn -> do_some_work() end)
res = do_some_other_work()
res + Task.await(task)
```

Tasks spawned with async can be awaited on by its caller process (and only its caller) as shown in the example above. They are implemented by spawning a process that sends a message to the caller once the given computation is performed.

Besides async/1 and await/1, tasks can also be used as part of supervision trees and dynamically spawned in remote nodes. We will explore all three scenarios next.

async and await  The most common way to spawn a task is with Task.async/1. A new process will be created and this process is linked and monitored by the caller. However, the processes are unlinked right before the task finishes, allowing the proper error to be triggered only on await/1.

This implies three things:

1. In case the caller crashes, the task will be killed and its computation will abort;
2. In case the task crashes due to an error, the parent will crash only on await/1;
3. In case the task crashes because a linked process caused it to crash, the parent will crash immediately;

Supervised tasks  It is also possible to spawn a task inside a supervision tree with start_link/1 and start_link/3:

```elixir
Task.start_link(fn -> IO.puts "ok" end)
```

Such can be mounted in your supervision tree as:

```elixir
import Supervisor.Spec

cchildren = [
  worker(Task, [fn -> IO.puts "ok" end])
]
```

Since such tasks are supervised and not directly linked to the caller, they cannot be awaited on. For such reason, differently from async/1, start_link/1 returns {:ok, pid} (which is the result expected by supervision trees).

Such tasks are useful as workers that run during your application life-cycle and rarely communicate with other workers. For example, a worker that pushes data to another server or a worker that consumes events from an event manager and writes it to a log file.
Supervision trees  The Task.Supervisor module allows developers to start supervisors that dynamically supervise tasks:

```elixir```
{:ok, pid} = Task.Supervisor.start_link()
Task.Supervisor.async(pid, fn -> do_work() end)
```

Task.Supervisor also makes it possible to spawn tasks in remote nodes as long as the supervisor is registered locally or globally:

```elixir```
# In the remote node
Task.Supervisor.start_link(name: :tasks_sup)

# On the client
Task.Supervisor.async({:tasks_sup, :remote@local}, fn -> do_work() end)
```

Task.Supervisor is more often started in your supervision tree as:

```elixir```
import Supervisor.Spec

children = [
  supervisor(Task.Supervisor, [[name: :tasks_sup]])
]
```

Check Task.Supervisor for other operations supported by the Task supervisor.

<table>
<thead>
<tr>
<th>Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>struct</strong>/0</td>
<td>The Task struct</td>
</tr>
<tr>
<td>async/1</td>
<td>Starts a task that can be awaited on</td>
</tr>
<tr>
<td>async/3</td>
<td>Starts a task that can be awaited on</td>
</tr>
<tr>
<td>await/2</td>
<td>Awaits for a task reply</td>
</tr>
<tr>
<td>find/2</td>
<td>Receives a group of tasks and a message and finds a task that matches the given message</td>
</tr>
<tr>
<td>start_link/1</td>
<td>Starts a task as part of a supervision tree</td>
</tr>
<tr>
<td>start_link/3</td>
<td>Starts a task as part of a supervision tree</td>
</tr>
</tbody>
</table>

Types
t
t :: %Task{pid: term, ref: term}

Functions
__struct__() (function)

Specs:

- __struct__: t

The Task struct.

It contains two fields:

- *:pid - the proces reference of the task process. It may be a pid or a tuple containing the process and node names;
- *:ref - the task monitor reference;

async(fun) (function)

Specs:

- async((... -> any)): t

Starts a task that can be awaited on.
This function spawns a process that is linked and monitored to the caller process. A Task struct is returned containing the relevant information.

**Task’s message format**

The reply sent by the task will be in the format \{ref, msg\}, where ref is the monitoring reference hold by the task.

```elixir
async(mod, fun, args)(function)
```

 Specs:

```elixir
•async(module, atom, [term]) :: t
```

Starts a task that can be awaited on.

Similar to `async/1`, but the task is specified by the given module, function and arguments.

```elixir
await(task, timeout \ 5000)(function)
```

 Specs:

```elixir
•await(t, timeout) :: term | no_return
```

Awaits for a task reply.

A timeout, in miliseconds, can be given with default value of 5000. In case the task process dies, this function will exit with the same reason as the task.

```elixir
find(tasks, msg)(function)
```

 Specs:

```elixir
•find([t], any) :: {term, t} | nil | no_return
```

Receives a group of tasks and a message and finds a task that matches the given message.

This function returns a tuple with the task and the returned value in case the message matches a task that exited with success, it raises in case the found task failed or nil if no task was found.

This function is useful in situations where multiple tasks are spawned and their results are collected just later on. For example, a GenServer can spawn tasks, store the tasks in a list and later use `Task.find/2` to see if upcoming messages are from any of the tasks.

```elixir
start_link(fun)(function)
```

 Specs:

```elixir
•start_link((... -> any)) :: {:ok, pid}
```

Starts a task as part of a supervision tree.

```elixir
start_link(mod, fun, args)(function)
```

 Specs:

```elixir
•start_link(module, atom, [term]) :: {:ok, pid}
```

Starts a task as part of a supervision tree.

**Task.Supervisor**

**Overview**  A tasks supervisor.

This module defines a supervisor which can be used to dynamically supervise tasks. Behind the scenes, this module is implemented as a `:simple_one_for_one` supervisor where the workers are temporary (i.e. they are not restarted after they die).

The functions in this module allow tasks to be spawned and awaited from a supervisor, similar to the functions defined in the `Task` module.
Name Registration  A Task.Supervisor is bound to the same name registration rules as a GenServer. Read more about it in the GenServer docs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>async/2</td>
<td>Starts a task that can be awaited on</td>
</tr>
<tr>
<td>async/4</td>
<td>Starts a task that can be awaited on</td>
</tr>
<tr>
<td>children/1</td>
<td>Returns all children pids</td>
</tr>
<tr>
<td>start_child/2</td>
<td>Starts a task as child of the given supervisor</td>
</tr>
<tr>
<td>start_child/4</td>
<td>Starts a task as child of the given supervisor</td>
</tr>
<tr>
<td>start_link/1</td>
<td>Starts a new supervisor</td>
</tr>
<tr>
<td>terminate_child/2</td>
<td>Terminates the given child at pid</td>
</tr>
</tbody>
</table>

Summary

### Functions

**async(supervisor, fun)** (function)

Specs:

- \( \text{async}(\text{Supervisor.supervisor}, (...) \to \text{any}) :: \text{Task.t} \)

Starts a task that can be awaited on.

The supervisor must be a reference as defined in Task.Supervisor. For more information on tasks, check the Task module.

**async(supervisor, module, fun, args)** (function)

Specs:

- \( \text{async}(\text{Supervisor.supervisor}, \text{module}, \text{atom}, \text{[term]}) :: \text{Task.t} \)

Starts a task that can be awaited on.

The supervisor must be a reference as defined in Task.Supervisor. For more information on tasks, check the Task module.

**children(supervisor)** (function)

Specs:

- \( \text{children}(\text{Supervisor.supervisor}) :: \text{[pid]} \)

Returns all children pids.

**start_child(supervisor, fun)** (function)

Specs:

- \( \text{start_child}(\text{Supervisor.supervisor}, (...) \to \text{any}) :: \{\text{ok, pid}\} \)

Starts a task as child of the given supervisor.

Note the spawned process is not linked to the caller but only to the supervisor. This command is useful in case the task needs to emit side-effects (like I/O) and does not need to report back to the caller.

**start_child(supervisor, module, fun, args)** (function)

Specs:

- \( \text{start_child}(\text{Supervisor.supervisor}, \text{module}, \text{atom}, \text{[term]}) :: \{\text{ok, pid}\} \)

Starts a task as child of the given supervisor.

Similar to start_child/2 except the task is specified by the given module, fun and args.

**start_link**(opts \[ \]) (function)

Specs:

- \( \text{start_link}(\text{Supervisor.options}) :: \text{Supervisor.on_start} \)
Elixir Documentation, Release

Starts a new supervisor.

The supported options are:

• :name - used to register a supervisor name, the supported values are described under the Name Registration section in the GenServer module docs;
• :shutdown - :brutal_kill if the tasks must be killed directly on shutdown or an integer indicating the timeout value, defaults to 5000 miliseconds;

terminate_child(supervisor, pid)(function)
Specs:

• terminate_child(Supervisor.supervisor, pid) :: :ok

Terminates the given child at pid.

Tuple

Overview  Functions for working with tuples.

<table>
<thead>
<tr>
<th>Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>delete_at/2</td>
<td>Removes an element from a tuple</td>
</tr>
<tr>
<td>duplicate/2</td>
<td>Creates a new tuple</td>
</tr>
<tr>
<td>insert_at/3</td>
<td>Inserts an element into a tuple</td>
</tr>
<tr>
<td>to_list/1</td>
<td>Converts a tuple to a list</td>
</tr>
</tbody>
</table>

Functions
delete_at(tuple, index)(function)
Specs:

• delete_at(tuple, non_neg_integer) :: tuple

Removes an element from a tuple.

Deletes the element at the zero-based index from tuple. Raises an ArgumentError if index is greater than or equal to the length of tuple.

Inlined by the compiler.

Examples

iex> tuple = {:foo, :bar, :baz}
iex> Tuple.delete_at(tuple, 0)
{:bar, :baz}

duplicate(data, size)(function)
Specs:

• duplicate(term, non_neg_integer) :: tuple

Creates a new tuple.

Creates a tuple of size size containing the given data at every position.

Inlined by the compiler.

Examples

iex> Tuple.duplicate(:hello, 3)
{:hello, :hello, :hello}
**insert_at(tuple, index, term)** (function)

Specs:

- `insert_at(tuple, non_neg_integer, term) :: tuple`

Inserts an element into a tuple. Inserts `value` into `tuple` at the given zero-based `index`. Raises an ArgumentError if `index` is greater than the length of `tuple`.

Inlined by the compiler.

**Examples**

```iex
tuple = {:bar, :baz}
Tuple.insert_at(tuple, 0, :foo)
{:foo, :bar, :baz}
```

**to_list(tuple)** (function)

Specs:

- `to_list(tuple) :: []`

Converts a tuple to a list.

Inlined by the compiler.

**URI**

**Overview** Utilities for working with and creating URIs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decode/1</code></td>
<td>Percent-unescape a URI</td>
</tr>
<tr>
<td><code>decode/2</code></td>
<td>Decodes a query string into a dictionary (by default uses a map)</td>
</tr>
<tr>
<td><code>decode_query/2</code></td>
<td>Returns the default port for a given scheme</td>
</tr>
<tr>
<td><code>default_port/1</code></td>
<td>Registers a scheme with a default port</td>
</tr>
<tr>
<td><code>default_port/2</code></td>
<td>Percent-escape a URI</td>
</tr>
<tr>
<td><code>encode/1</code></td>
<td>Encodes an enumerable into a query string</td>
</tr>
<tr>
<td><code>encode_query/1</code></td>
<td>Normalizes the scheme according to the spec by downcasing it</td>
</tr>
<tr>
<td><code>normalize_scheme/1</code></td>
<td>Parses a URI into components</td>
</tr>
<tr>
<td><code>parse/1</code></td>
<td>Returns an iterator function over the query string that decodes the query string in steps</td>
</tr>
</tbody>
</table>

**Types**

- `t :: %URI{scheme: term, path: term, query: term, fragment: term, authority: term, userinfo: term, host: term, port: term}`

**Functions**

**decode(uri)** (function)

Percent-unescape a URI.

**Examples**

```iex
URI.decode("http%3A%2F%2Felixir-lang.org")
"http://elixir-lang.org"
```

**decode(arg1, uri)** (function)
decode_query(q, dict \ %{})\(\text{(function)}\)
Decodes a query string into a dictionary (by default uses a map).

Given a query string of the form “key1=value1&key2=value2...”, produces a map with one entry for each key-

value pair. Each key and value will be a binary. Keys and values will be percent-unescaped.

Use query_decoder/1 if you want to iterate over each value manually.

Examples

\text{ex> URI.decode_query("foo=1&bar=2")}
%{"bar" => "2", "foo" => "1"}

default_port(scheme)\(\text{(function)}\)
Returns the default port for a given scheme.

If the scheme is unknown to URI, returns nil. Any scheme may be registered via default_port/2.

Examples

\text{ex> URI.default_port("ftp")}
21

\text{ex> URI.default_port("ponzi")}
nil

default_port(scheme, port)\(\text{(function)}\)
Registers a scheme with a default port.

It is recommended for this function to be invoked in your application start callback in case you want to register

new URIs.

code(s)\(\text{(function)}\)
Percent-escape a URI.

Example

\text{ex> URI.encode("http://elixir-lang.org/getting_started/2.html")}
"http%3A%2F%2Felixir-lang.org%2Fgetting_started%2F2.html"

code_query(l)\(\text{(function)}\)
Encodes an enumerable into a query string.

Takes an enumerable (containing a sequence of two-item tuples) and returns a string of the form “key1=value1&key2=value2...” where keys and values are URL encoded as per encode/1.

Keys and values can be any term that implements the String.Chars protocol, except lists which are explicitly forbidden.

Examples

\text{ex> hd = %{"foo" => 1, "bar" => 2}}
\text{ex> URI.encode_query(hd)}
"bar=2&foo=1"

normalize_scheme(scheme)\(\text{(function)}\)
Normalizes the scheme according to the spec by downcasing it.

parse(s)\(\text{(function)}\)
 Parses a URI into components.

URIs have portions that are handled specially for the particular scheme of the URI. For example, http and https have different default ports. Such values can be accessed and registered via URI.default_port/1 and URI.default_port/2.
Examples

```elixir
iex> URI.parse("http://elixir-lang.org/")
%URI{scheme: "http", path: "/", query: nil, fragment: nil,
   authority: "elixir-lang.org", userinfo: nil,
   host: "elixir-lang.org", port: 80}
```

**query_decoder(q)(function)**

Returns an iterator function over the query string that decodes the query string in steps.

Examples

```elixir
iex> URI.query_decoder("foo=1&bar=2") |> Enum.map &(&1)
[{"foo", "1"}, {"bar", "2"}]
```

Version

**Overview** Functions for parsing and matching versions against requirements.

A version is a string in a specific format or a Version generated after parsing via Version.parse/1. Version parsing and requirements follow SemVer 2.0 schema.

**Versions** In a nutshell, a version is given by three numbers:

**MAJOR**.**MINOR**.**PATCH**

Pre-releases are supported by appending **-[0-9A-Za-z-\.]**: 

"1.0.0-alpha.3"

Build information can be added by appending **+[0-9A-Za-z-\.]**: 

"1.0.0-alpha.3+20130417140000"

**Struct** The version is represented by the Version struct and it has its fields named according to Semver: :major, :minor, :patch, :pre and :build.

**Requirements** Requirements allow you to specify which versions of a given dependency you are willing to work against. It supports common operators like >=, <=, >, == and friends that work as one would expect:

# Only version 2.0.0
"== 2.0.0"

# Anything later than 2.0.0
"> 2.0.0"

Requirements also support and and or for complex conditions:

# 2.0.0 and later until 2.1.0
">= 2.0.0 and < 2.1.0"

Since the example above is such a common requirement, it can be expressed as:

"-> 2.0.0"
### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare/2</td>
<td>Compares two versions. Returns :gt if first version is greater than the second and :lt for vice versa. If the two versions are equal :eq is returned.</td>
</tr>
<tr>
<td>match?/2</td>
<td>Check if the given version matches the specification.</td>
</tr>
<tr>
<td>parse/1</td>
<td>Parse a version string into a Version.</td>
</tr>
<tr>
<td>parse_requirement</td>
<td>Parse a version requirement string into a Version.Requirement.</td>
</tr>
</tbody>
</table>

### Types

- `t` :: %Version{major: term, minor: term, patch: term, pre: term, build: term}
- `version` :: String.t | t
- `requirement` :: String.t | Version.Requirement.t/0
- `matchable` :: {major :: String.t | non_neg_integer, minor :: non_neg_integer | nil, patch :: non_neg_integer | nil, pre :: [String.t]}

### Functions

#### compare(vsn1, vsn2)(function)

**Specs:**

- `compare(version, version) :: :gt | :eq | :lt`

- Compares two versions. Returns :gt if first version is greater than the second and :lt for vice versa. If the two versions are equal :eq is returned.
- Raises a `Version.InvalidVersionError` exception if `version` is not parseable. If given an already parsed version this function won’t raise.

**Examples**

```iex
iex> Version.compare("2.0.1-alpha1", "2.0.0")
:gt
iex> Version.compare("2.0.1+build0", "2.0.1")
:eq
iex> Version.compare("invalid", "2.0.1")
** (Version.InvalidVersionError) invalid
```

#### match?(vsn, req)(function)

**Specs:**

- `match?(version, requirement) :: boolean`

- Check if the given version matches the specification.
- Returns true if version satisfies requirement, false otherwise. Raises a `Version.InvalidRequirementError` exception if requirement is not parseable, or `Version.InvalidVersionError` if version is not parseable. If given an already parsed version and requirement this function won’t raise.

**Examples**

```iex
iex> Version.match?("2.0.0", ">1.0.0")
true
iex> Version.match?("2.0.0", "==1.0.0")
```
false

iex> Version.match?("foo", "==1.0.0")
** (Version.InvalidVersionError) foo

iex> Version.match?("2.0.0", "== ==1.0.0")
** (Version.InvalidRequirementError) == ==1.0.0

parse(string) (function)
Specs:
  • parse(String.t):: {:ok, t} | :error

Parse a version string into a Version.

Examples
iex> Version.parse("2.0.1-alpha1") |> elem(1)
#Version<2.0.1-alpha1>

iex> Version.parse("2.0-alpha1")
:error

parse_requirement(string) (function)
Specs:
  • parse_requirement(String.t):: {:ok, Version.Requirement.t/0} | :error

Parse a version requirement string into a Version.Requirement.

Examples
iex> Version.parse_requirement("== 2.0.1") |> elem(1)
#Version.Requirement<== 2.0.1>

iex> Version.parse_requirement("== == 2.0.1")
:error
Exceptions

<table>
<thead>
<tr>
<th>ArgumentError</th>
<th>ArithmeticError</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadArityError</td>
<td>BadFunctionError</td>
</tr>
<tr>
<td>BadStructError</td>
<td>CaseClauseError</td>
</tr>
<tr>
<td>Code.LoadError</td>
<td>Enum.EmptyError</td>
</tr>
<tr>
<td>Enum.OutOfBoundsError</td>
<td>ErlangError</td>
</tr>
<tr>
<td>File.CopyError</td>
<td>File.Error</td>
</tr>
<tr>
<td>FunctionClauseError</td>
<td>IO.StreamError</td>
</tr>
<tr>
<td>KeyError</td>
<td>MatchError</td>
</tr>
<tr>
<td>Protocol.UndefinedError</td>
<td>Regex.CompileError</td>
</tr>
<tr>
<td>RuntimeError</td>
<td>SyntaxError</td>
</tr>
<tr>
<td>SystemLimitError</td>
<td>TokenMissingError</td>
</tr>
<tr>
<td>TryClauseError</td>
<td>UndefinedFunctionError</td>
</tr>
<tr>
<td>UnicodeConversionError</td>
<td>Version.InvalidRequirementError</td>
</tr>
<tr>
<td>Version.InvalidVersionError</td>
<td></td>
</tr>
</tbody>
</table>

ArgumentError

Overview

<table>
<thead>
<tr>
<th>Summary</th>
<th>exception/1</th>
<th>message/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback implementation of Exception.exception/1</td>
<td>Callback implementation of Exception.message/1</td>
<td></td>
</tr>
</tbody>
</table>

Types

t :: %ArgumentError{__exception__: term, message: term}

Functions

exception(args) (function)

Specs:

• exception(term) :: t

Callback implementation of Exception.exception/1.

message(exception) (function)

Specs:
•message(t)::String.t

Callback implementation of Exception.message/1.

ArithmeticError

Overview

Summary

<table>
<thead>
<tr>
<th>Exception</th>
<th>Callback implementation of Exception.exception/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>Callback implementation of Exception.message/1</td>
</tr>
</tbody>
</table>

Types

t :: %ArithmeticError{__exception__: term}

Functions

exception (args) (function)

Specs:

•exception(term) :: t

Callback implementation of Exception.exception/1.

message () (function)

Callback implementation of Exception.message/1.

BadArityError

Overview

Summary

<table>
<thead>
<tr>
<th>Exception</th>
<th>Callback implementation of Exception.exception/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>Callback implementation of Exception.message/1</td>
</tr>
</tbody>
</table>

Types

t :: %BadArityError{__exception__: term, function: term, args: term}

Functions

exception (args) (function)

Specs:

•exception(term) :: t

Callback implementation of Exception.exception/1.

message (exception) (function)

Callback implementation of Exception.message/1.

BadFunctionError

Overview
**BadFunctionError**

**Overview**

**Types**

\[ t :: \%\text{BadFunctionError}\{\_\_\text{exception}_\_\_\_\_\_\: \text{term}, \text{term}: \text{term}\} \]

**Functions**

- **exception (args) (function)**
  
  Specs:
  
  \[ \text{exception} (\text{term}) :: t \]
  
  Callback implementation of Exception.exception/1.

- **message (exception) (function)**
  
  Callback implementation of Exception.message/1.

**CaseClauseError**

**Overview**

**Types**

\[ t :: \%\text{CaseClauseError}\{\_\_\text{exception}_\_\_\_\_\_\: \text{term}, \text{struct}: \text{term}, \text{term}: \text{term}\} \]

**Functions**

- **exception (args) (function)**
  
  Specs:
  
  \[ \text{exception} (\text{term}) :: t \]
  
  Callback implementation of Exception.exception/1.

- **message (exception) (function)**
  
  Callback implementation of Exception.message/1.
Functions

**exception(args)(function)**

Specs:

- `exception(term) :: t`

Callback implementation of `Exception.exception/1`.

**message(exception)(function)**

Specs:

- `message(t) :: String.t`

Callback implementation of `Exception.message/1`.

---

**Code.LoadError**

Overview

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exception/1</code></td>
<td>Callback implementation of <code>Exception.exception/1</code></td>
</tr>
<tr>
<td><code>message/1</code></td>
<td>Callback implementation of <code>Exception.message/1</code></td>
</tr>
</tbody>
</table>

Types

`t`:

`t :: %Code.LoadError{__exception__: term, file: term, message: term}`

Functions

**exception(args)(function)**

Specs:

- `exception(term) :: t`

Callback implementation of `Exception.exception/1`.

**message(exception)(function)**

Specs:

- `message(t) :: String.t`

Callback implementation of `Exception.message/1`.

---

**CompileError**

Overview

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exception/1</code></td>
<td>Callback implementation of <code>Exception.exception/1</code></td>
</tr>
<tr>
<td><code>message/1</code></td>
<td>Callback implementation of <code>Exception.message/1</code></td>
</tr>
</tbody>
</table>

Types

`t`:

`t :: %CompileError{__exception__: term, file: term, line: term, description: term}`

Functions

**exception(args)(function)**

Specs:

- `exception(term) :: t`

Callback implementation of `Exception.exception/1`. 
**message(exception)(function)**

Callback implementation of \texttt{Exception.message/1}.

---

**Enum.EmptyError**

**Overview**

**Summary**

| exception/1 | Callback implementation of \texttt{Exception.exception/1} |
| message/1   | Callback implementation of \texttt{Exception.message/1} |

**Types**

t

t :: \%Enum.EmptyError\{\_exception\_: term\}

**Functions**

- **exception(args)(function)**

  Specs:

  - \texttt{exception(term) :: t}

  Callback implementation of \texttt{Exception.exception/1}.

- **message()(function)**

  Callback implementation of \texttt{Exception.message/1}.

---

**Enum.OutOfBoundsError**

**Overview**

**Summary**

| exception/1 | Callback implementation of \texttt{Exception.exception/1} |
| message/1   | Callback implementation of \texttt{Exception.message/1} |

**Types**

t

t :: \%Enum.OutOfBoundsError\{\_exception\_: term\}

**Functions**

- **exception(args)(function)**

  Specs:

  - \texttt{exception(term) :: t}

  Callback implementation of \texttt{Exception.exception/1}.

- **message()(function)**

  Callback implementation of \texttt{Exception.message/1}.

---

**ErlangError**

**Overview**
Summary

**exception/1**  Callback implementation of `Exception.exception/1`  
**message/1**  Callback implementation of `Exception.message/1`  

Types

\[ t :: \%ErlangError\{__exception__: term, original: term\} \]

Functions

**exception** *(args)* *(function)*  
Specs:

\[ *\text{exception}(\text{term}) :: t \]

Callback implementation of `Exception.exception/1`.  

**message** *(exception)* *(function)*  
Callback implementation of `Exception.message/1`.  

File.CopyError

Overview

Summary

**exception/1**  Callback implementation of `Exception.exception/1`  
**message/1**  Callback implementation of `Exception.message/1`  

Types

\[ t :: \%File.CopyError\{__exception__: term, reason: term, action: term, source: term, destination: term, on: term\} \]

Functions

**exception** *(args)* *(function)*  
Specs:

\[ *\text{exception}(\text{term}) :: t \]

Callback implementation of `Exception.exception/1`.  

**message** *(exception)* *(function)*  
Callback implementation of `Exception.message/1`.  

File.Error

Overview

Summary

**exception/1**  Callback implementation of `Exception.exception/1`  
**message/1**  Callback implementation of `Exception.message/1`  

Types

\[ t :: \%File.Error\{__exception__: term, reason: term, action: term, path: term\} \]
Functions

```elixir
exception(args)(function)
```  
Specs:

```elixir
•exception(term) :: t
```

Callback implementation of `Exception.exception/1`.

```elixir
message(exception)(function)
```  
Callback implementation of `Exception.message/1`.

### FunctionClauseError

#### Overview

#### Summary

| exception/1 | Callback implementation of `Exception.exception/1` |
| message/1 | Callback implementation of `Exception.message/1` |

#### Types

```elixir
t
```

```elixir
  t :: %FunctionClauseError{__exception__: term, module: term, function: term, arity: term}
```

### IO.StreamError

#### Overview

#### Summary

| exception/1 | Callback implementation of `Exception.exception/1` |
| message/1 | Callback implementation of `Exception.message/1` |

#### Types

```elixir
t
```

```elixir
  t :: %IO.StreamError{__exception__: term, reason: term, message: term}
```

### Functions

```elixir
exception(args)(function)
```  
Specs:

```elixir
•exception(term) :: t
```

Callback implementation of `Exception.exception/1`.

```elixir
message(exception)(function)
```  
Callback implementation of `Exception.message/1`.
• message(t):: String.t  
  Callback implementation of Exception.message/1.

KeyError

Overview

Summary

| exception/1 | Callback implementation of Exception.exception/1 |
| message/1   | Callback implementation of Exception.message/1 |

Types

  t :: %KeyError{__exception__: term, key: term, term: term}

Functions

  exception(args)(function)
  Specs:
    • exception(term) :: t
      Callback implementation of Exception.exception/1.
  message(exception)(function)
      Callback implementation of Exception.message/1.

MatchError

Overview

Summary

| exception/1 | Callback implementation of Exception.exception/1 |
| message/1   | Callback implementation of Exception.message/1 |

Types

  t :: %MatchError{__exception__: term, term: term}

Functions

  exception(args)(function)
  Specs:
    • exception(term) :: t
      Callback implementation of Exception.exception/1.
  message(exception)(function)
      Callback implementation of Exception.message/1.

Protocol.UndefinedError

Overview
Types

\(t\) :: %Protocol.UndefinedError{\_exception\_: term, protocol: term, value: term, description: term}

Functions

**exception** (args) (function)

Specs:

\* exception(term) :: t

Callback implementation of Exception.exception/1.

**message** (exception) (function)

Callback implementation of Exception.message/1.

---

**Regex.CompileError**

Overview

Types

\(t\) :: %Regex.CompileError{\_exception\_: term, message: term}

Functions

**exception** (args) (function)

Specs:

\* exception(term) :: t

Callback implementation of Exception.exception/1.

**message** (exception) (function)

Specs:

\* message(t) :: String.t

Callback implementation of Exception.message/1.

---

**RuntimeError**

Overview

Types

\(t\) :: %RuntimeError{\_exception\_: term, message: term}

Functions

**exception** (args) (function)

Specs:

\* exception(term) :: t

Callback implementation of Exception.exception/1.

**message** (exception) (function)

Specs:

\* message(t) :: String.t

Callback implementation of Exception.message/1.
Types

\[ t :: \%\text{RuntimeError}\{\_\_\text{exception\_\_}: \text{term}, \text{message}: \text{term}\} \]

Functions

\text{exception}(\text{args})(\text{function})

Specs:

\[ \cdot \text{exception(\text{term})} :: t \]

Callback implementation of \text{Exception}.\text{exception}/1.

\text{message}(\text{exception})(\text{function})

Specs:

\[ \cdot \text{message}(t) :: \text{String.t} \]

Callback implementation of \text{Exception}.\text{message}/1.

\textbf{SyntaxError}

Overview

| \text{Summary} | \begin{tabular}{|c|c|}
| exception/1 & \text{Callback implementation of Exception}.\text{exception}/1 \\
| message/1 & \text{Callback implementation of Exception}.\text{message}/1 \\
\end{tabular} |

Types

\[ t :: \%\text{SyntaxError}\{\_\_\text{exception\_\_}: \text{term}, \text{file}: \text{term}, \text{line}: \text{term}, \text{description}: \text{term}\} \]

Functions

\text{exception}(\text{args})(\text{function})

Specs:

\[ \cdot \text{exception(\text{term})} :: t \]

Callback implementation of \text{Exception}.\text{exception}/1.

\text{message}(\text{exception})(\text{function})

Callback implementation of \text{Exception}.\text{message}/1.

\textbf{SystemLimitError}

Overview

| \text{Summary} | \begin{tabular}{|c|c|}
| exception/1 & \text{Callback implementation of Exception}.\text{exception}/1 \\
| message/1 & \text{Callback implementation of Exception}.\text{message}/1 \\
\end{tabular} |

Types

\[ t :: \%\text{SystemLimitError}\{\_\_\text{exception\_\_}: \text{term}\} \]
### Elixir Documentation, Release

**Functions**

#### exception (args) (function)

Specs:

* exception(term) :: t

Callback implementation of Exception.exception/1.

#### message (exception) (function)

Callback implementation of Exception.message/1.

---

**TokenMissingError**

#### Overview

**Summary**

| exception/1 | Callback implementation of Exception.exception/1 |
| message/1   | Callback implementation of Exception.message/1 |

#### Types

t :: %TokenMissingError{__exception__: term, file: term, line: term, description: term}

**Functions**

#### exception (args) (function)

Specs:

* exception(term) :: t

Callback implementation of Exception.exception/1.

#### message (exception) (function)

Callback implementation of Exception.message/1.

---

**TryClauseError**

#### Overview

**Summary**

| exception/1 | Callback implementation of Exception.exception/1 |
| message/1   | Callback implementation of Exception.message/1 |

#### Types

t :: %TryClauseError{__exception__: term, term: term}

**Functions**

#### exception (args) (function)

Specs:

* exception(term) :: t

Callback implementation of Exception.exception/1.

#### message (exception) (function)

Callback implementation of Exception.message/1.
UndefinedFunctionError

Overview

Summary

<table>
<thead>
<tr>
<th></th>
<th>Callback implementation of Exception.exception/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception/1</td>
<td>Callback implementation of Exception.message/1</td>
</tr>
<tr>
<td>message/1</td>
<td></td>
</tr>
</tbody>
</table>

Types

\[ t \:: \%\text{UndefinedFunctionError}\{__exception__: \text{term}, \text{module}: \text{term}, \text{function}: \text{term}, \text{arity}: \text{term}\} \]

Functions

exception (args) (function)

Specs:

\[ \cdot \text{exception}(\text{term}) :: t \]

Callback implementation of Exception.exception/1.

message(exception) (function)

Specs:

\[ \cdot \text{message}(t) :: \text{String.t} \]

Callback implementation of Exception.message/1.

UnicodeConversionError

Overview

Summary

<table>
<thead>
<tr>
<th></th>
<th>Callback implementation of Exception.exception/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception/1</td>
<td>Callback implementation of Exception.message/1</td>
</tr>
<tr>
<td>message/1</td>
<td></td>
</tr>
</tbody>
</table>

Types

\[ t \:: \%\text{UnicodeConversionError}\{__exception__: \text{term}, \text{encoded}: \text{term}, \text{message}: \text{term}\} \]

Functions

exception (args) (function)

Specs:

\[ \cdot \text{exception}(\text{term}) :: t \]

Callback implementation of Exception.exception/1.

message(exception) (function)

Specs:

\[ \cdot \text{message}(t) :: \text{String.t} \]

Callback implementation of Exception.message/1.

Version.InvalidRequirementError

Overview
Summary

| exception/1 | Callback implementation of Exception.exception/1 |
| message/1   | Callback implementation of Exception.message/1 |

Types

t :: %Version.InvalidRequirementError{__exception__: term, message: term}

Functions

exception(args)(function)

Specs:

• exception(term) :: t

Callback implementation of Exception.exception/1.

message(exception)(function)

Specs:

• message(t) :: String.t

Callback implementation of Exception.message/1.

Version.InvalidVersionError

Overview

Summary

| exception/1 | Callback implementation of Exception.exception/1 |
| message/1   | Callback implementation of Exception.message/1 |

Types

t :: %Version.InvalidVersionError{__exception__: term, message: term}

Functions

exception(args)(function)

Specs:

• exception(term) :: t

Callback implementation of Exception.exception/1.

message(exception)(function)

Specs:

• message(t) :: String.t

Callback implementation of Exception.message/1.
Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>The Access protocol is the underlying protocol invoked when the brackets syntax is used. For instance, foo[bar] is translated to access foo, bar which, by default, invokes the Access.access protocol.</td>
</tr>
<tr>
<td>Collectable</td>
<td>A protocol to traverse data structures.</td>
</tr>
<tr>
<td>Enumerable</td>
<td>Enumerable protocol used by Enum and Stream modules.</td>
</tr>
<tr>
<td>Inspect</td>
<td>The Inspect protocol is responsible for converting any Elixir data structure into an algebra document. This document is then formatted, either in pretty printing format or a regular one. The List.Chars protocol is responsible for converting a structure to a list (only if applicable). The only function required to be implemented is to_char_list which does the conversion.</td>
</tr>
<tr>
<td>List.Chars</td>
<td>A protocol used for iterating range elements.</td>
</tr>
<tr>
<td>Range.iterator</td>
<td>The Range iterator protocol is responsible for converting a structure to a Binary (only if applicable). The only function required to be implemented is to_string which does the conversion.</td>
</tr>
<tr>
<td>String.Chars</td>
<td>The String Chars protocol is responsible for converting a structure to a Binary (only if applicable). The only function required to be implemented is to_string which does the conversion.</td>
</tr>
</tbody>
</table>

### Access

**Overview**  The Access protocol is the underlying protocol invoked when the brackets syntax is used. For instance, foo[bar] is translated to access foo, bar which, by default, invokes the Access.access protocol.

This protocol is implemented by default for Lists, Maps and dictionary like types:

```iex
iex> keywords = [a: 1, b: 2]
iex> keywords[:a]
1

iex> map = %{a: 1, b: 2}
iex> map[:a]
1

iex> star_ratings = %{1.0 => "", 1.5 => "", 2.0 => "}
iex> star_ratings[1.5]
""
```

The key access must be implemented using the `===` operator. This protocol is limited and is implemented only for the following built-in types: keywords, maps and functions.

**Summary**  get/2  Receives the element being accessed and the access item

**Types**

t
t :: term

**Functions**

get(container, key)(function)

Receives the element being accessed and the access item.

### Collectable

**Overview**  A protocol to traverse data structures.

The `Enum.into/2` function uses this protocol to insert an enumerable into a collection:
If a collection implements both `Enumerable` and `Collectable`, both operations can be combined with `Enum.traverse/2`:

```iex
iex> Enum.traverse(%{a: 1, b: 2}, fn {k, v} -> {k, v * 2} end)
%{a: 2, b: 4}
```

**Why Collectable?** The `Enumerable` protocol is useful to take values out of a collection. In order to support a wide range of values, the functions provided by the `Enumerable` protocol do not keep shape. For example, passing a dictionary to `Enum.map/2` always returns a list.

This design is intentional. `Enumerable` was designed to support infinite collections, resources and other structures with fixed shape. For example, it doesn’t make sense to insert values into a range, as it has a fixed shape where just the range limits are stored.

The `Collectable` module was designed to fill the gap left by the `Enumerable` protocol. It provides two functions: `into/1` and `empty/1`.

`into/1` can be seen as the opposite of `Enumerable.reduce/3`. If `Enumerable` is about taking values out, `Collectable.into/1` is about collecting those values into a structure.

`empty/1` receives a collectable and returns an empty version of the same collectable. By combining the enumerable functionality with `into/1` and `empty/1`, one can, for example, implement a traversal mechanism.

<table>
<thead>
<tr>
<th>Summary</th>
<th>empty/1</th>
<th>Receives a collectable structure and returns an empty one</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>into/1</td>
<td>Returns a function that collects values alongside the initial accumulation value</td>
</tr>
</tbody>
</table>

**Types**

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>command ::= {:cont, term}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ::= term</td>
</tr>
</tbody>
</table>

**Functions**

- **empty(collectable)(function)**
  - Specs:
    - •`empty(t) ::= t`
  - Receives a collectable structure and returns an empty one.

- **into(collectable)(function)**
  - Specs:
    - •`into(t) ::= {term, (term, command -> t | term)}`
  - Returns a function that collects values alongside the initial accumulation value.

  The returned function receives a collectable and injects a given value into it for every `{:cont, term}` instruction.

  :`done` is passed when no further values will be injected, useful for closing resources and normalizing values. A collectable must be returned on :`done`.

  If injection is suddenly interrupted, :`halt` is passed and it can return any value, as it won’t be used.
Enumerable

Overview  Enumerable protocol used by Enum and Stream modules.

When you invoke a function in the Enum module, the first argument is usually a collection that must implement this protocol. For example, the expression

Enum.map([1, 2, 3], &(&1 * 2))

invokes underneath Enumerable.reduce/3 to perform the reducing operation that builds a mapped list by calling the mapping function &(&1 * 2) on every element in the collection and cons’ing the element with an accumulated list.

Internally, Enum.map/2 is implemented as follows:

def map(enum, fun) do
  reducer = fn x, acc -> {:cont, [fun.(x)|acc]} end
  Enumerable.reduce(enum, {:cont, []}, reducer) |> elem(1) |> :lists.reverse()
end

Notice the user given function is wrapped into a reducer function. The reducer function must return a tagged tuple after each step, as described in the acc/0 type.

The reason the accumulator requires a tagged tuple is to allow the reducer function to communicate to the underlying enumerable the end of enumeration, allowing any open resource to be properly closed. It also allows suspension of the enumeration, which is useful when interleaving between many enumerables is required (as in zip).

Finally, Enumerable.reduce/3 will return another tagged tuple, as represented by the result/0 type.

<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
</table>
| count/1  | Retrieves the collection’s size  
| member?/2| Checks if a value exists within the collection  
| reduce/3 | Reduces the collection into a value  

Types

acc  

acc:: {:cont, term} | {:halt, term} | {:suspend, term}

The accumulator value for each step.

It must be a tagged tuple with one of the following “tags”:

• :cont - the enumeration should continue

• :halt - the enumeration should halt immediately

• :suspend - the enumeration should be suspended immediately

Depending on the accumulator value, the result returned by Enumerable.reduce/3 will change. Please check the result type docs for more information.

In case a reducer function returns a :suspend accumulator, it must be explicitly handled by the caller and never leak.

reducer  

reducer :: (term, term -> acc)

The reducer function.

Should be called with the collection element and the accumulator contents. Returns the accumulator for the next enumeration step.
**result**

\[
\text{result :: \{\text{done, term}\} \mid \{\text{halted, term}\} \mid \{\text{suspended, term, continuation}\}}
\]

The result of the reduce operation.

It may be *done* when the enumeration is finished by reaching its end, or *halted/suspended* when the enumeration was halted or suspended by the reducer function.

In case a reducer function returns the :suspend accumulator, the :suspended tuple must be explicitly handled by the caller and never leak. In practice, this means regular enumeration functions just need to be concerned about :done and :halted results.

Furthermore, a :suspend call must always be followed by another call, eventually halting or continuing until the end.

**continuation**

\[
\text{continuation :: (acc -> result)}
\]

A partially applied reduce function.

The continuation is the closure returned as a result when the enumeration is suspended. When invoked, it expects a new accumulator and it returns the result.

A continuation is easily implemented as long as the reduce function is defined in a tail recursive fashion. If the function is tail recursive, all the state is passed as arguments, so the continuation would simply be the reducing function partially applied.

**t**

\[
\text{t :: term}
\]

**Functions**

**count (collection) (function)**

Specs:

\[
\ast \text{count(t) :: \{ok, non_neg_integer\} \mid \{error, module\}}
\]

Retrieves the collection’s size.

It should return \{ok, size\}.

If \{error, __MODULE__\} is returned a default algorithm using reduce and the match (===) operator is used. This algorithm runs in linear time.

Please force use of the default algorithm unless you can implement an algorithm that is significantly faster.

**member? (collection, value) (function)**

Specs:

\[
\ast \text{member?(t, term) :: \{ok, boolean\} \mid \{error, module\}}
\]

Checks if a value exists within the collection.

It should return \{ok, boolean\}.

If \{error, __MODULE__\} is returned a default algorithm using reduce and the match (===) operator is used. This algorithm runs in linear time.

Please force use of the default algorithm unless you can implement an algorithm that is significantly faster.

**reduce (collection, acc, fun) (function)**

Specs:

\[
\ast \text{reduce(t, acc, reducer) :: result}
\]
Reduces the collection into a value.

Most of the operations in Enum are implemented in terms of reduce. This function should apply the given reducer function to each item in the collection and proceed as expected by the returned accumulator.

As an example, here is the implementation of reduce for lists:

```elixir
def reduce(_, {:halt, acc}, _fun), do: {:halted, acc}
def reduce(list, {:suspend, acc}, fun), do: {:suspended, acc, &reduce(list, &1, fun)}
def reduce([], {:cont, acc}, _fun), do: {:done, acc}
def reduce([h|t], {:cont, acc}, fun), do: reduce(t, fun.(h, acc), fun)
```

**Inspect**

**Overview** The Inspect protocol is responsible for converting any Elixir data structure into an algebra document. This document is then formatted, either in pretty printing format or a regular one.

The inspect/2 function receives the entity to be inspected followed by the inspecting options, represented by the struct Inspect.Opts.

Inspection is done using the functions available in Inspect.Algebra.

**Examples** Many times, inspecting a structure can be implemented in function of existing entities. For example, here is HashSet’s inspect implementation:

```elixir
defimpl Inspect, for: HashSet do
  import Inspect.Algebra

  def inspect(dict, opts) do
    concat ["#HashSet<", to_doc(HashSet.to_list(dict), opts), "]"]
  end
end
```

The concat function comes from Inspect.Algebra and it concatenates algebra documents together. In the example above, it is concatenating the string "HashSet<" (all strings are valid algebra documents that keep their formatting when pretty printed), the document returned by Inspect.Algebra.to_doc/2 and the other string "]".

Since regular strings are valid entities in an algebra document, an implementation of inspect may simply return a string, although that will devoid it of any pretty-printing.

**Error handling** In case there is an error while your structure is being inspected, Elixir will automatically fall back to a raw representation.

You can however access the underlying error by invoking the Inspect implementation directly. For example, to test Inspect.HashSet above, you can invoke it as:

```elixir
Inspect.HashSet.inspect(HashSet.new, Inspect.Opts.new)
```

**Summary**

**Types**

```elixir
t :: term
```

394 Chapter 3. API reference
Functions
inspect(thing, opts)(function)

List.Chars

Overview The List.Chars protocol is responsible for converting a structure to a list (only if applicable). The only function required to be implemented is to_char_list which does the conversion.

The to_char_list function automatically imported by Kernel invokes this protocol.

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_char_list</td>
<td>Converts the structure to a list (only if applicable)</td>
</tr>
</tbody>
</table>

Types
t

t :: term

Functions
to_char_list(thing)(function)

Range.Iterator

Overview A protocol used for iterating range elements.

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Count how many items are in the range</td>
</tr>
<tr>
<td>next</td>
<td>Returns the function that calculates the next item</td>
</tr>
</tbody>
</table>

Types
t

t :: term

Functions
count(first, range)(function)

next(first, range)(function)

String.Chars

Overview The String.Chars protocol is responsible for converting a structure to a Binary (only if applicable). The only function required to be implemented is to_string which does the conversion.

The to_string function automatically imported by Kernel invokes this protocol. String interpolation also invokes to_string in its arguments. For example, "foo#{bar}" is the same as "foo" <> to_string(bar).

Summary
to_string/1
Types

t

t :: term

Functions

to_string(thing)(function)

3.2.2 ExUnit v0.14.0-dev

• Modules

• Exceptions

Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExUnit</td>
<td>Basic unit testing framework for Elixir</td>
</tr>
<tr>
<td>ExUnit.Assertions</td>
<td>This module contains a set of assertion functions that are imported by default into your test cases</td>
</tr>
<tr>
<td>ExUnit.Callbacks</td>
<td>Defines ExUnit Callbacks</td>
</tr>
<tr>
<td>ExUnit.CaptureIO</td>
<td>Functionality to capture IO for testing</td>
</tr>
<tr>
<td>ExUnit.Case</td>
<td>Sets up an ExUnit test case</td>
</tr>
<tr>
<td>ExUnit.TestCase</td>
<td>This module allows a developer to define a test case template to be used throughout his tests. This is useful when there are a set of functions that should be shared between tests or a set of setup/teardown callbacks</td>
</tr>
<tr>
<td>ExUnit.DocTest</td>
<td>ExUnit.DocTest implements functionality similar to Python's doctest</td>
</tr>
<tr>
<td>ExUnit.Filters</td>
<td>Conveniences for parsing and evaluating filters</td>
</tr>
<tr>
<td>ExUnit.Formatter</td>
<td>This module holds helper functions related to formatting and contains documentation about the formatting protocol</td>
</tr>
<tr>
<td>ExUnit.Test</td>
<td>A struct that keeps information about the test</td>
</tr>
<tr>
<td>ExUnit.TestCase</td>
<td>A struct that keeps information about the test case</td>
</tr>
</tbody>
</table>

ExUnit

Overview

Basic unit testing framework for Elixir.

Example

A basic setup for ExUnit is shown below:

```elixir
# File: assertion_test.exs

# 1) Start ExUnit.
ExUnit.start

# 2) Create a new test module (test case) and use 'ExUnit.Case'.
defmodule AssertionTest do
  # 3) Notice we pass 'async: true', this runs the test case concurrently with other test cases
  use ExUnit.Case, async: true

  # 4) Use the 'test' macro instead of 'def' for clarity.
  test "the truth" do
    assert true
```
To run the tests above, run the file using `elixir` from the command line. Assuming you named the file `assertion_test.exs`, you can run it as:

```
bin/elixir assertion_test.exs
```

### Case, Callbacks and Assertions

See `ExUnit.Case` and `ExUnit.Callbacks` for more information about defining test cases.

The `ExUnit.Assertions` module contains a set of macros to easily generate assertions with appropriate error messages.

### Integration with Mix

Mix is the project management and build tool for Elixir. Invoking `mix test` from the command line will run the tests in each file matching the pattern `*_test.exs` found in the `test` directory of your project.

You must create a `test_helper.exs` file inside the `test` directory and put the code common to all tests there.

The minimum example of a `test_helper.exs` file would be:

```elixir
# test/test_helper.exs
ExUnit.start
```

Mix will load the `test_helper.exs` file before executing the tests. It is not necessary to `require` the `test_helper.exs` file in your test files. See `Mix.Tasks.Test` for more information.

---

## Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>configuration()</code></td>
<td>Returns ExUnit configuration</td>
</tr>
<tr>
<td><code>configure(options)</code></td>
<td>Configures ExUnit</td>
</tr>
<tr>
<td><code>run/0</code></td>
<td>API used to run the tests. It is invoked automatically if ExUnit is started via <code>ExUnit.start/1</code></td>
</tr>
<tr>
<td><code>start/1</code></td>
<td>Starts ExUnit and automatically runs tests right before the VM terminates. It accepts a set of options to configure ExUnit (the same ones accepted by <code>configure/1</code>)</td>
</tr>
</tbody>
</table>

## Types

### state

```
state :: nil | {:failed, failed} | {:skip, binary} | {:invalid, module}
```

The state returned by `ExUnit.Test` and `ExUnit.TestCase`

### failed

```
failed :: {Exception.kind, reason :: term, stacktrace :: [tuple]}
```

## Functions

### configuration() (function)

Returns ExUnit configuration.

### configure(options) (function)

Configures ExUnit.

## Options

ExUnit supports the following options:

- `:color` - When color should be used by specific formatters. Defaults to the result of `IO.ANSI.terminal?/1`;
- `:formatters` - The formatters that will print results. Defaults to `[ExUnit.CLIFormatter]`;
**Max Cases**
Maximum number of cases to run in parallel. Defaults to `:erlang.system_info(:schedulers_online)`.

**Trace**
Set ExUnit into trace mode, this sets `:max_cases` to 1 and prints each test case and test while running.

**Autorun**
If ExUnit should run by default on exit, defaults to `true`.

**Include**
Specify which tests are run by skipping tests that do not match the filter.

**Exclude**
Specify which tests are run by skipping tests that match the filter.

**Seed**
An integer seed value to randomize the test suite.

---

**ExUnit.Assertions**

**Overview**
This module contains a set of assertion functions that are imported by default into your test cases.

In general, a developer will want to use the general `assert` macro in tests. This macro tries to be smart and provide good reporting whenever there is a failure. For example, `assert some_fun() == 10` will fail (assuming `some_fun()` returns 13):

Comparison (using `==`) failed in: code: some_fun() == 10 lhs: 13 rhs: 10

This module also provides other convenience functions like `assert_in_delta` and `assert_raise` to easily handle other common cases such as checking a floating point number or handling exceptions.
Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert/1</td>
<td>Asserts its argument is true</td>
</tr>
<tr>
<td>assert/2</td>
<td>Asserts value is true, displaying the given message otherwise</td>
</tr>
<tr>
<td>assert/4</td>
<td>Asserts value is true. If it fails, it raises an expectation error using the given left and right values</td>
</tr>
<tr>
<td>assert_in_delta/4</td>
<td>Asserts that val1 and val2 differ by no more than delta</td>
</tr>
<tr>
<td>assert_raise/2</td>
<td>Asserts the exception is raised during function execution. Returns the rescued exception, fails otherwise</td>
</tr>
<tr>
<td>assert_raise/4</td>
<td>Asserts the exception is raised during function execution with the expected_message. Returns the rescued exception, fails otherwise</td>
</tr>
<tr>
<td>assert_receive/4</td>
<td>Asserts a message was or is going to be received. Unlike assert_received, it has a default timeout of 100 milliseconds</td>
</tr>
<tr>
<td>assert_received/2</td>
<td>Asserts a message was received and is in the current process’ mailbox. Timeout is set to 0, so there is no waiting time</td>
</tr>
<tr>
<td>catch_error/1</td>
<td>Asserts expression will cause an error. Returns the error or fails otherwise</td>
</tr>
<tr>
<td>catch_exit/1</td>
<td>Asserts expression will exit. Returns the exit status/message or fails otherwise</td>
</tr>
<tr>
<td>catch_throw/1</td>
<td>Asserts expression will throw a value. Returns the thrown value or fails otherwise</td>
</tr>
<tr>
<td>flunk/1</td>
<td>Fails with a message</td>
</tr>
<tr>
<td>refute/1</td>
<td>This is a negative assertion, failing if its parameter is truthy</td>
</tr>
<tr>
<td>refute/2</td>
<td>Asserts value is nil or false (that is, value is not truthy)</td>
</tr>
<tr>
<td>refute_in_delta/4</td>
<td>Asserts val1 and val2 are not within delta</td>
</tr>
<tr>
<td>refute_receive/3</td>
<td>refute_receive message, timeout Â 100, message Â nil</td>
</tr>
<tr>
<td>refute_received/2</td>
<td>Asserts a message was not received (i.e. it is not in the current process mailbox). The not_expected argument must be a match pattern</td>
</tr>
</tbody>
</table>

Functions

assert(value, message)(function)

Asserts value is true, displaying the given message otherwise.

Examples

assert false, "it will never be true"

assert(value, left, right, message)(function)

Asserts value is true. If it fails, it raises an expectation error using the given left and right values.

You probably don’t need to use this—the regular assert function handles this for you.

Examples

assert this > that, this, that, "more than"

assert_in_delta(val1, val2, delta, message \ nil)(function)

Asserts that val1 and val2 differ by no more than delta.

Examples

assert_in_delta 1.1, 1.5, 0.2
assert_in_delta 10, 15, 4

assert_raise(exception, function)(function)

Asserts the exception is raised during function execution. Returns the rescued exception, fails otherwise.

Examples

assert_raise ArithmeticError, fn ->
  1 + "test"
  end

3.2. API Reference (v0.14.0-dev) 399
assert_raise(exception, message, function)  

Asserts the exception is raised during function execution with the expected_message. Returns the rescued exception, fails otherwise.

Examples

```elixir
assert_raise ArithmeticError, "bad argument in arithmetic expression", fn ->
  1 + "test"
end
```

flunk(message, "Flunked!")  

Specs:

- flunk(String.t) :: no_return

Fails with a message.

Examples

```elixir
flunk "This should raise an error"
```

refute(value, message)  

Asserts value is nil or false (that is, value is not truthy).

Examples

```elixir
refute true, "This will obviously fail"
```

refute_in_delta(val1, val2, delta, message, nil)  

Asserts val1 and val2 are not within delta.

If you supply message, information about the values will automatically be appended to it.

Examples

```elixir
refute_in_delta 1.1, 1.2, 0.2
refute_in_delta 10, 11, 2
```

Macros

assert(assertion)  

Asserts its argument is true.

assert tries to be smart and provide good reporting whenever there is a failure. In particular, if given a match expression, it will report any failure in terms of that match. Given

```elixir
assert [one] = [two]
```

you’ll see:

```elixir
match (=) failed
code: [one] = [two]
rhs: [2]
```

If the expression is a comparison operator, the message will show the values of the two sides. The assertion

```elixir
assert 1+2+3+4 > 15
```

will fail with the message:

```elixir
Assertion with > failed
code: 1+2+3+4 > 15
lhs:  10
rhs:  15
```
assert_receive(expected, timeout \ 100, message \ nil)(macro)
  Asserts a message was or is going to be received. Unlike assert_received, it has a default timeout of 100 milliseconds.

  The expected argument is a pattern.

  Examples
  
  assert_receive :hello

  Asserts against a larger timeout:
  
  assert_receive :hello, 20_000

  You can also match against specific patterns:
  
  assert_receive {:hello, _}
  
  \[\]
x = 5
  assert_receive {:count, ^x}

assert_received(expected, message \ nil)(macro)
  Asserts a message was received and is in the current process’ mailbox. Timeout is set to 0, so there is no waiting time.

  The expected argument is a pattern.

  Examples
  
  send self, :hello
  assert_received :hello

  You can also match against specific patterns:
  
  send self, {:hello, "world"}
  assert_received {:hello, _}

catch_error(expression)(macro)
  Asserts expression will cause an error. Returns the error or fails otherwise.

  Examples
  
  assert catch_error(error 1) == 1

catch_exit(expression)(macro)
  Asserts expression will exit. Returns the exit status/message or fails otherwise.

  Examples
  
  assert catch_exit(exit 1) == 1

catch_throw(expression)(macro)
  Asserts expression will throw a value. Returns the thrown value or fails otherwise.

  Examples
  
  assert catch_throw(throw 1) == 1

refute(assertion)(macro)
  This is a negative assertion, failing if its parameter is truthy.

  Examples
refute age < 0

**refute_receive**(not_expected, timeout \ 100, message \ nil)(macro)

refute_receive message, timeout \ 100, message \ nil

Asserts message was not received (and won’t be received) within the timeout period.
The not_expected argument is a match pattern.

**Examples**

refute_receive :bye

Refute received with a explicit timeout:

refute_receive :bye, 1000

**refute_received**(not_expected, message \ nil)(macro)

Asserts a message was not received (i.e. it is not in the current process mailbox). The not_expected argument must be a match pattern.

Timeout is set to 0, so there is no waiting time.

**Examples**

send self, :hello
refute_received :bye

**ExUnit.Callbacks**

**Overview**  Defines ExUnit Callbacks.

This module defines four callbacks: setup_all, teardown_all, setup and teardown.

These callbacks are defined via macros and each one can optionally receive a map with metadata, usually referred to as context. The callback may optionally put extra data into context to be used in the tests.

If you return {:ok, <dict>} from setup or teardown, the keyword list will be merged into the context that will be available in all subsequent setup, test or teardown calls.

Similarly, returning {:ok, <dict>} from setup_all or teardown_all will merge the keyword list into the context that will be available in all subsequent setup_all or teardown_all calls.

Returning :ok leaves the context unchanged in both cases.

Returning anything else from setup or teardown will force the current test to fail, and subsequent setup, test and teardown callbacks won’t be called for it.

Returning anything else from setup_all or teardown_all will force the whole case to fail, and no other callback will be called.

It is possible to define multiple setup and teardown callbacks and they will be called sequentially. In the case of setup_all and teardown_all callbacks, each setup_all will be called only once before the first test’s setup and each teardown_all will be called once after the last test. No callback runs if the test case has no tests or all tests were filtered out via include/exclude.
Examples

defmodule AssertionTest do
  use ExUnit.Case, async: true

  # 'setup' is called before each test is run
  setup do
    IO.puts "This is a setup callback"
  end

  # Return extra metadata, it must be a keyword list / map
  {:ok, hello: "world"}
end

# Same as 'setup', but receives the context for the current test
setup context do
  # We can access the current test in the context
  IO.puts "Setting up: #\{context[:test]\}"

  # We can also access the data returned from 'setup/0'
  assert context[:hello] == "world"

  # No metadata
  :ok
end

# This is called after each test finishes
 teardown context do
  assert context[:hello] == "world"
  :ok
end

test "always pass" do
  assert true
end

test "another one", context do
  assert context[:hello] == "world"
end
end

<table>
<thead>
<tr>
<th>Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup/2</td>
<td>Called before the start of each test</td>
</tr>
<tr>
<td>setup_all/2</td>
<td>Called before the start of a case, i.e. called once before the first test in the current module and before any setup callbacks</td>
</tr>
<tr>
<td>teardown/2</td>
<td>Called after the completion of each test</td>
</tr>
<tr>
<td>teardown_all</td>
<td>Called once after the last test finishes without emitting an :exit message</td>
</tr>
</tbody>
</table>

Macros

setup(var \ {:_, [], ExUnit.Callbacks}, block)(macro)
Called before the start of each test.

setup_all(var \ {:_, [], ExUnit.Callbacks}, block)(macro)
Called before the start of a case, i.e. called once before the first test in the current module and before any setup callbacks.

tear down(var \ {:_, [], ExUnit.Callbacks}, block)(macro)
Called after the completion of each test.

Note that if the test crashed with an :exit message, teardown will not be run.
teardown_all(var \ {:_, [], ExUnit.Callbacks}, block)(macro)
 Called once after the last test finishes without emitting an :exit message.

ExUnit.CaptureIO

Overview   Functionality to capture IO for testing.

Examples

defmodule AssertionTest do
  use ExUnit.Case

  import ExUnit.CaptureIO

  test :example do
    assert capture_io(fn ->
                       IO.puts "a"
    end) == "a
end
end

Summary

capture_io/1
  captures IO generated when evaluating fun

capture_io/2
  captures IO generated when evaluating fun

Functions

capture_io(fun)(function)
 Captures IO generated when evaluating fun.

Returns the binary which is the captured output.

By default, capture_io replaces the group_leader (:stdio) for the current process. However, the capturing of any other named device, such as :stderr, is also possible globally by giving the registered device name explicitly as an argument.

Note that when capturing something other than :stdio, the test should run with async false.

When capturing :stdio, if the :capture_prompt option is false, prompts (specified as arguments to IO.get* functions) are not captured.

A developer can set a string as an input. The default input is :eof.

Examples

iex> capture_io(fn -> IO.write "josé" end) == "josé"
true

iex> capture_io(:stderr, fn -> IO.write(:stderr, "josé") end) == "josé"
true

iex> capture_io("this is input", fn -> ...
  ...> input = IO.gets ">
  ...> 10.write input
  ...> end) == ">this is input"
true

iex> capture_io([input: "this is input", capture_prompt: false], fn ->
...> input = IO.gets ">
...> IO.write input
...> end) == "this is input"
true

capture_io(device, fun)(function)
capture_io(device, input, fun)(function)

ExUnit.Case

Overview  Sets up an ExUnit test case.
This module must be used in other modules as a way to configure and prepare them for testing.

When used, it accepts the following options:

• :async - configure Elixir to run that specific test case in parallel with others. Must be used for performance when
your test cases do not change any global state;

This module automatically includes all callbacks defined in ExUnit.Callbacks. See that module’s documentation
for more information.

Examples
defmodule AssertionTest do
  # Use the module
  use ExUnit.Case, async: true

  # The 'test' macro is imported by ExUnit.Case
  test "always pass" do
    assert true
  end
end

Context  All tests receive a context as an argument. The context is particularly useful for sharing information between
callbacks and tests:
defmodule KVTest do
  use ExUnit.Case

  setup do
    {:ok, pid} = KV.start_link
    {:ok, [pid: pid]}
  end

  test "stores key-values", context do
    assert KV.put(context[:pid], :hello, :world) == :ok
    assert KV.get(context[:pid], :hello) == :world
  end
end

As the context is a map, it can be pattern matched on to extract information:
test "stores key-values", %{pid: pid} do
  assert KV.put(pid, :hello, :world) == :ok
  assert KV.get(pid, :hello) == :world
end
Tags  The context is used to pass information from the callbacks to the test. In order to pass information from the test to the callback, ExUnit provides tags.

By tagging a test, the tag value can be accessed in the context, allowing the developer to customize the test. Let’s see an example:

```elixir
defmodule FileTest do
  # Changing directory cannot be async
  use ExUnit.Case, async: false

  setup context do
    # Read the :cd tag value
    if cd = context[:cd] do
      prev_cd = File.cwd!
      File.cd!(cd)
      {:ok, [prev_cd: prev_cd]}
    else
      :ok
    end
  end

  teardown context do
    # Revert to the previous working directory
    if cd = context[:prev_cd] do
      File.cd!(cd)
    end
  end

  @tag cd: "fixtures"
  test "reads utf-8 fixtures" do
    File.read("hello")
  end
end
```

In the example above, we have defined a tag called :cd that is read in the setup callback to configure the working directory the test is going to run on. We then use the same context to store the previous working directory that is reverted to after the test in the teardown callback.

Tags are also very effective when used with case templates (ExUnit.CaseTemplate) allowing callbacks in the case template to customize the test behaviour.

Note a tag can be set in two different ways:

```elixir
@tag key: value
@tag :key                # equivalent to setting @tag key: true
```

If a tag is given more than once, the last value wins.

Module tags A tag can be set for all tests in a module by setting @moduletag:

```elixir
@moduletag :external
```

If the same key is set via @tag, the @tag value has higher precedence.

Reserved tags The following tags are set automatically by ExUnit and are therefore reserved:

- :case - the test case module
- :test - the test name
Filters  Tags can also be used to identify specific tests, which can then be included or excluded using filters. The most common functionality is to exclude some particular tests from running, which can be done via `ExUnit.configure/1`:

```elixir
# Exclude all external tests from running
ExUnit.configure(exclude: [external: true])
```

From now on, ExUnit will not run any test that has the `external` flag set to true. This behaviour can be reversed with the `:include` option which is usually passed through the command line:

```bash
mix test --include external:true
```

Run `mix help test` for more information on how to run filters via Mix.

Another use case for tags and filters is to exclude all tests that have a particular tag by default, regardless of its value, and include only a certain subset:

```elixir
ExUnit.configure(exclude: :os, include: [os: :unix])
```

Keep in mind that all tests are included by default, so unless they are excluded first, the `include` option has no effect.

**Summary**  test/3  Define a test with a string

**Macros**

`test(message, var \ {_, [], ExUnit.Case}, contents)(macro)`  Define a test with a string.

Provides a convenient macro that allows a test to be defined with a string. This macro automatically inserts the atom `:ok` as the last line of the test. That said, a passing test always returns `:ok`, but, more importantly, it forces Elixir to not tail call optimize the test and therefore avoids hiding lines from the backtrace.

**Examples**

```elixir
test "true is equal to true" do
  assert true == true
end
```

**ExUnit.CaseTemplate**

**Overview**  This module allows a developer to define a test case template to be used throughout his tests. This is useful when there are a set of functions that should be shared between tests or a set of setup/teardown callbacks.

By using this module, the callbacks and assertions available for regular test cases will also be available.

**Example**

```elixir
defmodule MyCase do
  use ExUnit.CaseTemplate

  setup do
    IO.puts "This will run before each test that uses this case"
  end
end
```
defmodule MyTest do
  use MyCase, async: true

  test "truth" do
    assert true
  end
end

Summary

using/2 | Allows a developer to customize the using block when the case template is used

Macros

using(var \ \ {:_, \[], ExUnit.CaseTemplate}, list2)(macro)
  Allows a developer to customize the using block when the case template is used.

ExUnit.DocTest

Overview

ExUnit.DocTest implements functionality similar to Python’s doctest.

In a nutshell, it allows us to generate tests from the code examples existing in a module/function/macro’s documentation. In order to do that, one needs to invoke the doctest/1 macro from their test case and write their examples according to some guidelines.

The syntax for examples is as follows. Every new test starts on a new line, with an iex> prefix. Multiline expressions can be employed if the following lines start with either ...> (recommended) or iex> prefix.

The expected result should start at the next line after iex> or ...> line(s) and is terminated either by a newline, new iex> prefix or end of the string literal.

Examples

Currently, the only way to run doctests is to include them into an ExUnit case with a doctest macro:

defmodule MyModule.Test do
  use ExUnit.Case, async: true
  doctest MyModule
end

The doctest macro is going to loop through all functions and macros defined in MyModule, parsing their documentation in search of code examples.

A very basic example is:

iex> 1+1
2

Expressions on multiple lines are also supported:

iex> Enum.map [1, 2, 3], fn(x) ->
...> x * 2
...> end
[2,4,6]

Multiple results can be checked within the same test:
If you want to keep any two tests separate, add an empty line between them:

```iex
iex> a = 1
1
iex> a + 1 # will fail with a "function a/0 undefined" error
2
```

Similarly to iex you can use numbers in your “prompts”:

```iex
iex(1)> [1+2,
...(1)> 3]
[3,3]
```

This is useful in two use cases:

- Being able to refer to specific numbered scenarios
- Copy-pasting examples from an actual iex session

We also allow you to select or skip some functions when calling `doctest`. See the documentation for more info.

**Opaque types** Some types internal structure are kept hidden and instead show a user-friendly structure when inspecting the value. The idiom in Elixir is to print those data types as `#Name<...>`. Doctest will test these values by doing a string compare.

```iex
iex> Enum.into([a: 10, b: 20], HashDict.new)
#HashDict<[b: 20, a: 10]>
```

The above example will be tested with the following match: `"#HashDict<[b: 20, a: 10]>" = inspect(Enum.into([a: 10, b: 20], HashDict.new)).`

**Exceptions** You can also showcase expressions raising an exception, for example:

```iex
iex(1)> String.to_atom(fn() -> 1 end).()
** (ArgumentError) argument error
```

What DocTest will be looking for is a line starting with `**` (`and it will parse it accordingly to extract the exception name and message. At this moment, the exception parser would make the parser treat the next line as a start of a completely new expression (if it is prefixed with `iex>` or a no-op line with documentation. Thus, multiline messages are not supported.

**When not to use doctest** In general, doctests are not recommended when your code examples contain side effects. For example, if a doctest prints to standard output, doctest will not try to capture the output.

Similarly, doctests do not run in any kind of sandbox. So any module defined in a code example is going to linger throughout the whole test suite run.

**Summary**  `doctest/2` This macro is used to generate ExUnit test cases for doctests
Macros

`doctest(mod, opts \ [])(macro)`

This macro is used to generate ExUnit test cases for doctests.

Calling `doctest(Module)` will generate tests for all doctests found in the module `Module`.

Options can also be supplied:

- `:except` — generate tests for all functions except those listed (list of `{function, arity}` tuples)
- `:only` — generate tests only for functions listed (list of `{function, arity}` tuples)
- `:import` — when true, one can test a function defined in the module without referring to the module name. However, this is not feasible when there is a clash with a module like Kernel. In these cases, `import` should be set to `false` and a full `M.f` construct should be used.

**Examples**

`doctest MyModule, except: [trick_fun: 1]`

This macro is auto-imported with every `ExUnit.Case`.

**ExUnit.Filters**

**Overview**

Conveniences for parsing and evaluating filters.

**Summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eval/3</code></td>
<td>Evaluates the include and exclude filters against the given tags</td>
</tr>
<tr>
<td><code>normalize/2</code></td>
<td>Normalizes include and excludes to remove duplicates and keep precedence</td>
</tr>
<tr>
<td><code>parse/1</code></td>
<td>Parses the given filters, as one would receive from the command line</td>
</tr>
<tr>
<td><code>parse_path/1</code></td>
<td>Parses filters out of a path</td>
</tr>
</tbody>
</table>

**Types**

`t ::= [{atom, any} | atom]`

**Functions**

`eval(include, exclude, tags)(function)`

Specs:

- `eval(t, t, %{}) :: ok | {:error, atom}`

Evaluates the include and exclude filters against the given tags.

Examples

```elixir`
 iex> ExUnit.Filters.eval([foo: "bar"], [:foo], %{foo: "bar"})
 :ok
 iex> ExUnit.Filters.eval([foo: "bar"], [:foo], %{foo: "baz"})
 {:error, :foo}
```

`normalize(include, exclude)(function)`

Specs:

- `normalize(t \ nil, t \ nil) :: [t, t]`

```elixir```
Normalizes include and excludes to remove duplicates and keep precedence.

Examples

```iex
iex> ExUnit.Filters.normalize(nil, nil)
([], [])

iex> ExUnit.Filters.normalize([:foo, :bar, :bar], [:foo, :baz])
([:foo, :bar], [:baz])
```

**parse(filters)** (function)

Specs:

- `parse(String.t) :: t`

Parses the given filters, as one would receive from the command line.

Examples

```iex
iex> ExUnit.Filters.parse(["foo:bar", "baz", "line:9", "bool:true"])
[{:foo, "bar"}, :baz, {:line, "9"}, {:bool, "true"}]
```

**parse_path(file)** (function)

Specs:

- `parse_path(String.t) :: [String.t, any]`

Parses filters out of a path.

Determines whether a given file path (supplied to ExUnit/Mix as arguments on the command line) includes a line number filter, and if so returns the appropriate ExUnit configuration options.

**ExUnit.Formatter**

**Overview**
This module holds helper functions related to formatting and contains documentation about the formatting protocol.

Formatters are registered at the `ExUnit.EventManager` event manager and will be send events by the runner.

The following events are possible:

- `{:suite_started, opts}` - The suite has started with the specified options to the runner.
- `{:suite_finished, run_us, load_us}` - The suite has finished. `run_us` and `load_us` are the run and load times in microseconds respectively.
- `{:case_started, test_case}` - A test case has started. See `ExUnit.TestCase` for details.
- `{:case_finished, test_case}` - A test case has finished. See `ExUnit.TestCase` for details.
- `{:test_started, test_case}` - A test case has started. See `ExUnit.Test` for details.
- `{:test_finished, test_case}` - A test case has finished. See `ExUnit.Test` for details.

**Summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format_filters/2</td>
<td>Formats filters used to constrain cases to be run</td>
</tr>
<tr>
<td>format_test_case_failure/5</td>
<td>Receives a test case and formats its failure</td>
</tr>
<tr>
<td>format_test_failure/5</td>
<td>Receives a test and formats its failure</td>
</tr>
<tr>
<td>format_time/2</td>
<td>Formats time taken running the test suite</td>
</tr>
</tbody>
</table>
Types

id
  id :: term

test_case
  test_case :: ExUnit.TestCase.t

test
  test :: ExUnit.Test.t

run_us
  run_us :: pos_integer

load_us
  load_us :: pos_integer | nil

Functions

format_filters(filters, type)(function)
  Specs:
    • format_filters(Keyword.t, atom) :: String.t

  Formats filters used to constrain cases to be run.

Examples

iex> format_filters([run: true, slow: false, :include]) "Including tags: [run: true, slow: false]"

format_test_case_failure(test_case, arg2, counter, width, formatter)(function)
  Receives a test case and formats its failure.

format_test_failure(test, arg2, counter, width, formatter)(function)
  Receives a test and formats its failure.

format_time(run_us, load_us)(function)
  Specs:
    • format_time(run_us, load_us) :: String.t

  Formats time taken running the test suite.

  It receives the time spent running the tests and optionally the time spent loading the test suite.

Examples

iex> format_time(10000, nil)
"Finished in 0.01 seconds"

iex> format_time(10000, 20000)
"Finished in 0.03 seconds (0.02s on load, 0.01s on tests)"

iex> format_time(10000, 200000)
"Finished in 0.2 seconds (0.2s on load, 0.01s on tests)"

ExUnit.Test

Overview
  A struct that keeps information about the test.

  It is received by formatters and contains the following fields:

  • :name - the test name
  • :case - the test case
• :state - the test state (see ExUnit.state)
• :time - the time to run the test
• :tags - the test tags

Types
t
  t :: %ExUnit.Test{name: atom, case: module, state: ExUnit.state, time: non_neg_integer, tags: %{}}

ExUnit.TestCase

Overview  A struct that keeps information about the test case.
It is received by formatters and contains the following fields:
  • :name - the test case name
  • :state - the test state (see ExUnit.state)
  • :tests - all tests for this case

Types
t
  t :: %ExUnit.TestCase{name: module, state: ExUnit.state, tests: [ExUnit.Test.t]}

Exceptions

ExUnit.AssertionError

Overview

Summary

| exception/1 | Callback implementation of Exception.exception/1 |
| message/1   | Callback implementation of Exception.message/1 |
| no_value/0  | Indicates no meaningful value for a field        |

Types
t
  t :: %ExUnit.AssertionError{|exception__: term, left: term, right: term, message: term, expr: term}

Functions

exception (args) (function)

Specs:
  • exception(term) :: t

  Callback implementation of Exception.exception/1.

message (exception) (function)

Specs:
• `message(t): String.t`
  Callback implementation of `Exception.message/1`.

`no_value()` (function)
Indicates no meaningful value for a field.

### ExUnit.DocTest.Error

#### Overview

<table>
<thead>
<tr>
<th>Summary</th>
<th>Exception/1</th>
<th>Callback implementation of <code>Exception.exception/1</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Message/1</td>
<td>Callback implementation of <code>Exception.message/1</code></td>
</tr>
</tbody>
</table>

#### Types

`t`: `%ExUnit.DocTest.Error{__exception__: term, message: term}`

#### Functions

**exception** *(args)* (function)
Specs:

• `exception(term): t`
  Callback implementation of `Exception.exception/1`.

**message**(exception) (function)
Specs:

• `message(t): String.t`
  Callback implementation of `Exception.message/1`.

### 3.2.3 EEx v0.14.0-dev

- Modules
- Exceptions

#### Modules

<table>
<thead>
<tr>
<th>EEx</th>
<th>EEx stands for Embedded Elixir. It allows you to embed Elixir code inside a string in a robust way:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEx.AssignsEngine</td>
<td>An abstract engine that, when used with the TransformerEngine, allows a developer to access assigns using @ as syntax</td>
</tr>
<tr>
<td>EEx.Engine</td>
<td>This is the basic EEx engine that ships with Elixir. An engine needs to implement three functions:</td>
</tr>
<tr>
<td>EEx.SmartEngine</td>
<td>An engine meant for end-user usage that includes EEx.AssignsEngine and other conveniences. Read EEx.AssignsEngine for examples</td>
</tr>
<tr>
<td>EEx.Transformer</td>
<td>An abstract engine that is meant to be used and built upon in other modules. This engine implements the EEx.Engine behaviour and provides a transform overridable directive that allows a developer to customize the expression returned by the engine</td>
</tr>
</tbody>
</table>
EEx

Overview  EEx stands for Embedded Elixir. It allows you to embed Elixir code inside a string in a robust way:

```elixir
eix> EEx.eval_string "foo <%= bar %>", [bar: "baz"]
"foo baz"
```

API  This module provides 3 main APIs for you to use:

1. Evaluate a string (eval_string) or a file (eval_file) directly. This is the simplest API to use but also the slowest, since the code is evaluated and not compiled before;
2. Define a function from a string (function_from_string) or a file (function_from_file). This allows you to embed the template as a function inside a module which will then be compiled. This is the preferred API if you have access to the template at compilation time;
3. Compile a string (compile_string) or a file (compile_file) into Elixir syntax tree. This is the API used by both functions above and is available to you if you want to provide your own ways of handling the compiled template.

Options  All functions in this module accepts EEx-related options. They are:

- :line - the line to be used as the template start. Defaults to 1;
- :file - the file to be used in the template. Defaults to the given file the template is read from or to “nofile” when compiling from a string;
- :engine - the EEx engine to be used for compilation.

Engine  EEx has the concept of engines which allows you to modify or transform the code extracted from the given string or file.

By default, EEx uses the EEx.SmartEngine that provides some conveniences on top of the simple EEx.Engine.

Tags  EEx.SmartEngine supports the following tags:

- `<% Elixir expression - inline with output %>`
- `<%= Elixir expression - replace with result %>`
- `<%% EEx quotation - returns the contents inside %>`
- `<%# Comments - they are discarded from source %>`

All expressions that output something to the template must use the equals sign (=). Since everything in Elixir is a macro, there are no exceptions for this rule. For example, while some template languages would special-case if clauses, they are treated the same in EEx and also require = in order to have their result printed:

```elixir
<%= if true do %>
  It is obviously true
<%= else %>
  This will never appear
<%= end %>
```

Notice that different engines may have different rules for each tag. Other tags may be added in future versions.
Macros  EEx.SmartEngine also adds some macros to your template. An example is the @ macro which allows easy data access in a template:

```iex
 iex> EEx.eval_string "<%= @foo %>", assigns: [foo: 1] 
 "1"
```

In other words, <%= @foo %> is simply translated to:

<%= Dict.get assigns, :foo %>

The assigns extension is useful when the number of variables required by the template is not specified at compilation time.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compile_file</code></td>
<td>Get a filename and generate a quoted expression that can be evaluated by Elixir or compiled to a function.</td>
</tr>
<tr>
<td><code>compile_string</code></td>
<td>Get a string source and generate a quoted expression that can be evaluated by Elixir or compiled to a function.</td>
</tr>
<tr>
<td><code>eval_file</code></td>
<td>Get a filename and evaluate the values using the bindings.</td>
</tr>
<tr>
<td><code>eval_string</code></td>
<td>Get a string source and evaluate the values using the bindings.</td>
</tr>
<tr>
<td><code>function_from_file</code></td>
<td>Generates a function definition from the file contents. The kind (:def or :defp) must be given, the function name, its arguments and the compilation options.</td>
</tr>
<tr>
<td><code>function_from_string</code></td>
<td>Generates a function definition from the string. The kind (:def or :defp) must be given, the function name, its arguments and the compilation options.</td>
</tr>
</tbody>
</table>

### Functions

**compile_file(filename, options \ [])(function)**

Get a filename and generate a quoted expression that can be evaluated by Elixir or compiled to a function.

```iex
 iex> EEx.compile_file "sample.ex", [bar: "baz"]  #=> "foo baz"
```

**compile_string(source, options \ [])(function)**

Get a string source and generate a quoted expression that can be evaluated by Elixir or compiled to a function.

**eval_file(filename, bindings \ [], options \ [])(function)**

Get a filename and evaluate the values using the bindings.

```iex
 iex> EEx.eval_file "sample.ex", [bar: "baz"]  #=> "foo baz"
```

**eval_string(source, bindings \ [], options \ [])(function)**

Get a string source and evaluate the values using the bindings.

```iex
 iex> EEx.eval_string "foo <%= bar %>", [bar: "baz"]  #=> "foo baz"
```

### Macros

**function_from_file(kind, name, file, args \ [], options \ [])(macro)**

Generates a function definition from the file contents. The kind (:def or :defp) must be given, the function name, its arguments and the compilation options.

This function is useful in case you have templates but you want to precompile inside a module for speed.

```iex
 iex> EEx.eval_string "foo <%= bar %>", [bar: "baz"]  #=> "foo baz"
```
## samples.eex

```eex
<% a + b %>
```

## sample.ex

```eex
defmodule Sample do
  require EEx
  EEx.function_from_file :def, :sample, "sample.eex", [:a, :b]
end
```

```iex
Sample.sample(1, 2) #=> "3"
```

### function_from_string(kind, name, source, args \ [], options \ [])(macro)

Generates a function definition from the string. The kind (:def or :defp) must be given, the function name, its arguments and the compilation options.

### Examples

```iex> defmodule Sample do
  ...
  require EEx
  EEx.function_from_string :def, :sample, "<%= a + b %>", [:a, :b]
  ...
  end
iex> Sample.sample(1, 2)
"3"
```

### EEx.AssignsEngine

#### Overview

An abstract engine that, when used with the TransformerEngine, allows a developer to access assigns using `@` as syntax.

This engine is included by default on the SmartEngine.

#### Examples

```eex
defmodule MyEngine do
  use EEx.TransformerEngine
  use EEx.AssignsEngine
end
```

```iex> EEx.eval_string("<%= @foo %>", assigns: [foo: 1])
"1"
```

In the example above, we can access the value `foo` under the binding `assigns` using `@foo`. This is useful when a template, after compiled, may receive different assigns and the developer don’t want to recompile it for each variable set.

Assigns can also be used when compiled to a function:

```eex
<%= @a + @b %>
```

```eex
defmodule Sample do
  require EEx
  EEx.function_from_file :def, :sample, "sample.eex", [:assigns]
end
```

```iex
Sample.sample(a: 1, b: 2) #=> "3"
```
EEx.Engine

Overview  This is the basic EEx engine that ships with Elixir. An engine needs to implement three functions:

- handle_body(quoted) - receives the final built quoted expression, should do final post-processing and return a quoted expression;
- handle_text(buffer, text) - it receives the buffer, the text and must return a new quoted expression;
- handle_expr(buffer, marker, expr) - it receives the buffer, the marker, the expr and must return a new quoted expression;

The marker is what follows exactly after <%. For example, <%= foo %> has "=" as marker. The allowed markers so far are:

- ""
- "="

Read handle_expr/3 below for more information about the markers implemented by default by this engine.

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_body/1</td>
<td>The default implementation simply returns the given expression</td>
</tr>
<tr>
<td>handle_expr/3</td>
<td>Implements expressions according to the markers</td>
</tr>
<tr>
<td>handle_text/2</td>
<td>The default implementation simply concatenates text to the buffer</td>
</tr>
</tbody>
</table>

Functions

**handle_body (quoted) (function)**

The default implementation implementation simply returns the given expression.

**handle_expr (buffer, binary2, expr) (function)**

Implements expressions according to the markers.

<% Elixir expression - inline with output %>
<%== Elixir expression - replace with result %>

All other markers are not implemented by this engine.

**handle_text (buffer, text) (function)**

The default implementation simply concatenates text to the buffer.

Callbacks

**handle_body/1(callback)**

Specs:

- handle_body(Macro.t)::Macro.t

**handle_expr/3(callback)**

Specs:

- handle_expr(Macro.t, binary, Macro.t)::Macro.t

**handle_text/2(callback)**

Specs:

- handle_text(Macro.t, binary)::Macro.t
**EEx.SmartEngine**

**Overview**  An engine meant for end-user usage that includes EEx.AssignsEngine and other conveniences. Read EEx.AssignsEngine for examples.

<table>
<thead>
<tr>
<th>Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_body/1</td>
<td>Callback implementation of EEx.Engine.handle_body/1</td>
</tr>
<tr>
<td>handle_expr/3</td>
<td>Callback implementation of EEx.Engine.handle_expr/3</td>
</tr>
<tr>
<td>handle_text/2</td>
<td>Callback implementation of EEx.Engine.handle_text/2</td>
</tr>
</tbody>
</table>

**Functions**

- **handle_body (body) (function)**  
  Callback implementation of EEx.Engine.handle_body/1.
- **handle_expr (buffer, mark, expr) (function)**  
  Callback implementation of EEx.Engine.handle_expr/3.
- **handle_text (buffer, text) (function)**  
  Callback implementation of EEx.Engine.handle_text/2.

**EEx.TransformerEngine**

**Overview**  An abstract engine that is meant to be used and built upon in other modules. This engine implements the EEx.Engine behaviour and provides a transform overridable directive that allows a developer to customize the expression returned by the engine.

Check EEx.AssignsEngine and EEx.SmartEngine for examples of using this module.

**Exceptions**

**EEx.SyntaxError**

**Overview**

<table>
<thead>
<tr>
<th>Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>exception/1</td>
<td>Callback implementation of Exception.exception/1</td>
</tr>
<tr>
<td>message/1</td>
<td>Callback implementation of Exception.message/1</td>
</tr>
</tbody>
</table>

**Types**

- t :: %EEx.SyntaxError{__exception__: term, message: term}

**Functions**

- **exception (args) (function)**  
  Specs:
  
  ```
  *exception(term) :: t
  ```  

  Callback implementation of Exception.exception/1.

- **message (exception) (function)**  
  Specs:
**message(t)** :: String.t

Callback implementation of Exception.message/1.

### 3.2.4 Mix v0.14.0-dev

- Modules
- Exceptions

## Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix</td>
<td>Mix is a build tool that provides tasks for creating, compiling and testing Elixir projects. Mix is inspired by the Leiningen build tool for Clojure and was written by one of its contributors</td>
</tr>
<tr>
<td>Mix.Archive</td>
<td>Module responsible for managing archives</td>
</tr>
<tr>
<td>Mix.Config</td>
<td>Module for reading and merging app configurations</td>
</tr>
<tr>
<td>Mix.Generator</td>
<td>Conveniences for working with paths and generating content</td>
</tr>
<tr>
<td>Mix.Project</td>
<td>Defines and manipulate Mix projects</td>
</tr>
<tr>
<td>Mix.SCM</td>
<td>This module provides helper functions and defines the behaviour required by any SCM used with Mix. Defines Mix.Shell contract</td>
</tr>
<tr>
<td>Mix.Shell</td>
<td>This is Mix’s default shell. It simply prints messages to stdout and stderr</td>
</tr>
<tr>
<td>Mix.Shell.IO</td>
<td>This is a Mix shell that uses the current process mailbox for communication instead of IO</td>
</tr>
<tr>
<td>Mix.Shell.Process</td>
<td>A simple module that provides conveniences for creating, loading and manipulating tasks</td>
</tr>
<tr>
<td>Mix.Task</td>
<td>Starts all registered apps. If no apps key exists, it starts the current application</td>
</tr>
<tr>
<td>Mix.Tasks.App.Start</td>
<td>Packages the current project (though not its dependencies) into a zip file according to the specification of the Erlang Archive Format</td>
</tr>
<tr>
<td>Mix.Tasks.Archive</td>
<td>Clean generated application files</td>
</tr>
<tr>
<td>Mix.Tasks.Cmd</td>
<td>Executes the given command</td>
</tr>
<tr>
<td>Mix.Tasks.Compile</td>
<td>A meta task that compiles source files</td>
</tr>
<tr>
<td>Mix.Tasks.Compile.Elixir</td>
<td>Compiles Elixir source files</td>
</tr>
<tr>
<td>Mix.Tasks.Compile.Erlang</td>
<td>Compile Erlang source files</td>
</tr>
<tr>
<td>Mix.Tasks.Compile.Leex</td>
<td>Compile Leex source files</td>
</tr>
<tr>
<td>Mix.Tasks.Compile.Protocols</td>
<td>Consolidates all protocols in all paths</td>
</tr>
<tr>
<td>Mix.Tasks.Compile.Yecc</td>
<td>Compile Yecc source files</td>
</tr>
<tr>
<td>Mix.Tasks.Deps</td>
<td>List all dependencies and their status</td>
</tr>
<tr>
<td>Mix.Tasks.Deps.Check</td>
<td>Checks if all dependencies are valid and if not, abort. Prints the invalid dependencies’ status before aborting</td>
</tr>
<tr>
<td>Mix.Tasks.Deps.Clean</td>
<td>Remove the given dependencies’ files</td>
</tr>
<tr>
<td>Mix.Tasks.Deps.Compile</td>
<td>Compile dependencies</td>
</tr>
<tr>
<td>Mix.Tasks.Deps.Get</td>
<td>Get all out of date dependencies, i.e. dependencies that are not available or have an invalid lock, and loads them</td>
</tr>
<tr>
<td>Mix.Tasks.Deps.Loadpaths</td>
<td>Loads all dependencies for the current build. This is invoked directly by loadpaths when unlocking dependencies</td>
</tr>
<tr>
<td>Mix.Tasks.Deps.Unlock</td>
<td>Unlock the given dependencies</td>
</tr>
<tr>
<td>Mix.Tasks.Deps.Update</td>
<td>Update the given dependencies</td>
</tr>
<tr>
<td>Mix.Tasks.Do</td>
<td>Executes the tasks separated by comma</td>
</tr>
<tr>
<td>Mix.Tasks.Escriptize</td>
<td>Generates an escript for the project</td>
</tr>
<tr>
<td>Mix.Tasks.Help</td>
<td>Lists all tasks or prints the documentation for a given task</td>
</tr>
<tr>
<td>Mix.Tasks.Tex</td>
<td>A task that is simply meant to redirect users to iex -S mix</td>
</tr>
<tr>
<td>Mix.Tasks.Loadconfig</td>
<td>Loads and persists the project configuration</td>
</tr>
<tr>
<td>Mix.Tasks.Loadpaths</td>
<td>Loads the application and its dependencies paths</td>
</tr>
<tr>
<td>Mix.Tasks.Local</td>
<td>List local tasks</td>
</tr>
<tr>
<td>Mix.Tasks.Local.Hex</td>
<td>Install hex locally from <a href="https://hex.pm/installs/hex.cz">https://hex.pm/installs/hex.cz</a></td>
</tr>
<tr>
<td>Mix.Tasks.Local.Install</td>
<td>Install an archive locally</td>
</tr>
<tr>
<td>Mix.Tasks.Local.Rebar</td>
<td>Fetch a copy of rebar from the given path or url. It defaults to a rebar copy that ships with Elixir.</td>
</tr>
<tr>
<td>Mix.Tasks.Local.Uninstall</td>
<td>Uninstall local tasks</td>
</tr>
</tbody>
</table>
Mix.Tasks.New
Mix.Tasks.Run
Mix.Tasks.Test
Mix.Utils

Mix

Overview Mix is a build tool that provides tasks for creating, compiling and testing Elixir projects. Mix is inspired by the Leiningen build tool for Clojure and was written by one of its contributors.

This module works as a facade for accessing the most common functionality in Elixir, such as the shell and the current project configuration.

For getting started with Elixir, checkout out the guide available on Elixir’s website.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env/0</td>
<td>Returns the mix environment.</td>
</tr>
<tr>
<td>env/1</td>
<td>Changes the current mix env.</td>
</tr>
<tr>
<td>shell/0</td>
<td>The shell is a wrapper for doing IO.</td>
</tr>
<tr>
<td>shell/1</td>
<td>Sets the current shell.</td>
</tr>
</tbody>
</table>

Mix.Archive

Overview Module responsible for managing archives.

An archive is a zip file containing the app and beam files. A valid archive must be named with the name of the application and it should contain the relative paths beginning with the application name, e.g. the root of the zip file should be my_app/ebin/Elixir.My.App.beam.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create/2</td>
<td>Creates an application archive.</td>
</tr>
<tr>
<td>dir/1</td>
<td>Returns the archive internal directory from its path.</td>
</tr>
<tr>
<td>ebin/1</td>
<td>Returns the ebin directory inside the given archive path.</td>
</tr>
<tr>
<td>name/2</td>
<td>Returns the archive name based on app and version.</td>
</tr>
</tbody>
</table>
Functions

create(source, target)(function)

Creates an application archive.

It receives the archive file in the format path/to/archive/app-vsn.ez and the path to the root of the project to be archived. Everything in the ebin and priv directories is archived. Dependencies are not archived.

dir(path)(function)

Returns the archive internal directory from its path.

Examples

iex> Mix.Archive.dir("foo/bar/baz-0.1.0.ez")
"baz-0.1.0"

ebin(path)(function)

Returns the ebin directory inside the given archive path.

Examples

iex> Mix.Archive.ebin("foo/bar/baz-0.1.0.ez")
"foo/bar/baz-0.1.0.ez/baz-0.1.0/ebin"

name(app, vsn)(function)

Returns the archive name based on app and version.

Examples

iex> Mix.Archive.name("foo", nil)
"foo.ez"

iex> Mix.Archive.name("foo", "0.1.0")
"foo-0.1.0.ez"

Mix.Config

Overview

Module for reading and merging app configurations.

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge/2</td>
<td>Merges two configurations</td>
</tr>
<tr>
<td>merge/3</td>
<td>Merges two configurations</td>
</tr>
<tr>
<td>persist/1</td>
<td>Persists the given configuration by modifying the configured applications environment</td>
</tr>
<tr>
<td>read/1</td>
<td>Reads a configuration file</td>
</tr>
<tr>
<td>validate!/1</td>
<td>Validates a configuration</td>
</tr>
</tbody>
</table>

Functions

merge(config1, config2)(function)

Merges two configurations.

The configuration of each application is merged together with the values in the second one having higher preference than the first in case of conflicts.

Examples

iex> Mix.Config.merge([app: [k: :v1]], [app: [k: :v2]])
[app: [k: :v2]]

iex> Mix.Config.merge([app1: []], [app2: []])
[app1: [], app2: []]
merge(config1, config2, callback)(function)
  Merges two configurations.
  The configuration of each application is merged together and a callback is invoked in case of conflicts receiving
  the app, the conflicting key and both values. It must return a value that will be used as part of the conflict
  resolution.

  Examples
  iex> Mix.Config.merge([app: [k: :v1]], [app: [k: :v2]],
  ...>   fn app, k, v1, v2 -> {app, k, v1, v2} end)
  [app: [k: {:app, :k, :v1, :v2}]]

persist(config)(function)
  Persists the given configuration by modifying the configured applications environment.

read(file)(function)
  Reads a configuration file.
  It returns the read configuration and a list of dependencies this configuration may have on.

validate!(config)(function)
  Validates a configuration.

Mix.Generator

Overview
  Conveniences for working with paths and generating content.

All of those functions are verbose, in the sense they log the action to be performed via Mix.shell.

| Summary |
|-----------------|-----------------|
| create_directory(path)(function) | Creates a directory if one does not exist yet |
| create_file(path, contents)(function) | Creates a file with the given contents. If the file already exists, asks for user confirmation |
| embed_template(name, contents)(macro) | Embed a template given by contents into the current module |
| embed_text(name, contents)(macro) | Embeds a text given by contents into the current module |
| from_file(path) | Reads the content from a file relative to the current file and not relative to the cwd. Useful when used with embed macros: |

Functions

create_directory(path)(function)
  Creates a directory if one does not exist yet.

create_file(path, contents)(function)
  Creates a file with the given contents. If the file already exists, asks for user confirmation.

Macros

embed_template(name, contents)(macro)
  Embed a template given by contents into the current module.
  It will define a private function with the name followed by _template that expects assigns as arguments.
  This function must be invoked passing a keyword list. Each key in the keyword list can be accessed in the
  template using the @ macro.
  For more information, check EEx.SmartEngine.

embed_text(name, contents)(macro)
  Embeds a text given by contents into the current module.
  It will define a private function with the name followed by _text that expects no argument.
from_file(path)(macro)

Reads the content from a file relative to the current file and not relative to the cwd. Useful when used with embed macros:

```elixir
eembed_template :lib, from_file("../templates/lib.eex")
```

Mix.Project

**Overview**  Defines and manipulate Mix projects.

In order to configure Mix, a developer needs to use `Mix.Project` in a module and define a function named `project` that returns a keyword list with configuration.

```elixir
defmodule MyApp do
  use Mix.Project

  def project do
    [app: :my_app,
     vsn: "0.6.0"]
  end
end
```

After being defined, the configuration for this project can be read as `Mix.Project.config/0`. Notice that `config/0` won’t fail if a project is not defined; this allows many mix tasks to work even without a project.

In case the developer needs a project or wants to access a special function in the project, he/she can call `Mix.Project.get!/0` which fails with `Mix.NoProjectError` in case a project is not defined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_path/1</td>
<td>Returns the application path inside the build</td>
</tr>
<tr>
<td>build_path/1</td>
<td>Returns the build path for this project</td>
</tr>
<tr>
<td>build_structure/2</td>
<td>Builds the project structure for the current application</td>
</tr>
<tr>
<td>compile_path/1</td>
<td>Returns the paths this project compiles to</td>
</tr>
<tr>
<td>config/0</td>
<td>Returns the project configuration</td>
</tr>
<tr>
<td>config_files/0</td>
<td>Returns a list of project configuration files for this project</td>
</tr>
<tr>
<td>deps_path/1</td>
<td>Returns the path to store dependencies for this project</td>
</tr>
<tr>
<td>get!/0</td>
<td>Same as <code>get/0</code>, but raises an exception if there is no current project</td>
</tr>
<tr>
<td>get/0</td>
<td>Retrieves the current project if there is one</td>
</tr>
<tr>
<td>in_project/4</td>
<td>Runs the given <code>fun</code> inside the given project</td>
</tr>
<tr>
<td>load_paths/0</td>
<td>Returns all load paths for this project</td>
</tr>
<tr>
<td>manifest_path/1</td>
<td>The path to store manifests</td>
</tr>
<tr>
<td>umbrella?/0</td>
<td>Returns true if project is an umbrella project</td>
</tr>
</tbody>
</table>

**Summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_path</td>
<td>Returns the application path inside the build</td>
</tr>
<tr>
<td>build_path</td>
<td>Returns the build path for this project</td>
</tr>
<tr>
<td>build_structure</td>
<td>Builds the project structure for the current application</td>
</tr>
<tr>
<td>compile_path</td>
<td>Returns the paths this project compiles to</td>
</tr>
<tr>
<td>config</td>
<td>Returns the project configuration</td>
</tr>
<tr>
<td>config_files</td>
<td>Returns a list of project configuration files for this project</td>
</tr>
<tr>
<td>deps_path</td>
<td>Returns the path to store dependencies for this project</td>
</tr>
<tr>
<td>get</td>
<td>Same as <code>get/0</code>, but raises an exception if there is no current project</td>
</tr>
<tr>
<td>in_project</td>
<td>Runs the given <code>fun</code> inside the given project</td>
</tr>
<tr>
<td>load_paths</td>
<td>Returns all load paths for this project</td>
</tr>
<tr>
<td>manifest_path</td>
<td>The path to store manifests</td>
</tr>
<tr>
<td>umbrella?</td>
<td>Returns true if project is an umbrella project</td>
</tr>
</tbody>
</table>

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_path</td>
<td>Returns the application path inside the build</td>
</tr>
<tr>
<td>build_path</td>
<td>Returns the build path for this project</td>
</tr>
</tbody>
</table>

**Examples**

```elixir
Mix.Project.app_path
#=> "/path/to/project/_build/shared/lib/app"
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_path</td>
<td>Returns the build path for this project</td>
</tr>
<tr>
<td></td>
<td>The returned path will be expanded.</td>
</tr>
</tbody>
</table>

```elixir
Mix.Project.build_path
```
Examples

```
Mix.Project.build_path
#=> "/path/to/project/_build/shared"
```

If :build_per_environment is set to true (the default), it will create a new build per environment:

```
Mix.env
#=> :dev
Mix.Project.build_path
#=> "/path/to/project/_build/dev"
```

`build_structure(config \ config(), opts \ [])()`

Builds the project structure for the current application.

**Options**

- `:symlink_ebin` - Symlink ebin instead of copying it

`compile_path(config \ config())`

Returns the paths this project compiles to.

The returned path will be expanded.

**Examples**

```
Mix.Project.compile_path
#=> "/path/to/project/_build/shared/lib/app/priv"
```

`config()`

Returns the project configuration.

If there is no project defined, it still returns a keyword list with default values. This allows many mix tasks to work without the need for an underlying project.

Note this configuration is cached once the project is pushed into the stack. Calling it multiple times won’t cause it to be recomputed.

Do not use `Mix.Project.config` to rely on runtime configuration. Use it only to configure aspects of your project (like compilation directories) and not your application runtime.

`config_files()`

Returns a list of project configuration files for this project.

This function is usually used in compilation tasks to trigger a full recompilation whenever such configuration files change.

By default it includes the mix.exs file, the lock manifest and all config files in the `config` directory.

`deps_path(config \ config())`

Returns the path to store dependencies for this project.

The returned path will be expanded.

**Examples**

```
Mix.Project.deps_path
#=> "/path/to/project/deps"
```

`get()`

Retrieves the current project if there is one.

Otherwise `nil` is returned. It may happen in cases there is no mixfile in the current directory.

If you expect a project to be defined, i.e. it is a requirement of the current task, you should call `get!/0` instead.
**get!()** *(function)*

Same as `get/0`, but raises an exception if there is no current project.

This is usually called by tasks that need additional functions on the project to be defined. Since such tasks usually depend on a project being defined, this function raises `Mix.NoProjectError` in case no project is available.

**in_project(app, path, post_config \ [\ ], fun)** *(function)*

Runs the given `fun` inside the given project.

This function changes the current working directory and loads the project at the given directory onto the project stack.

A `post_config` can be passed that will be merged into the project configuration.

**load_paths()** *(function)*

Returns all load paths for this project.

**manifest_path(config \ \ config())** *(function)*

The path to store manifests.

By default they are stored in the app path inside the build directory but it may be changed in future releases.

The returned path will be expanded.

**Examples**

```elixir
Mix.Project.manifest_path
#=> "/path/to/project/_build/shared/lib/app"
```

**umbrella?()** *(function)*

Returns `true` if project is an umbrella project.

---

**Mix.SCM**

**Overview**  
This module provides helper functions and defines the behaviour required by any SCM used by mix.

<table>
<thead>
<tr>
<th>Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append/1</td>
<td>Append the given SCM module to the list of available SCMs</td>
</tr>
<tr>
<td>available/0</td>
<td>Returns all available SCMs. Each SCM is tried in order until a matching one is found</td>
</tr>
<tr>
<td>prepend/1</td>
<td>Prepend the given SCM module to the list of available SCMs</td>
</tr>
</tbody>
</table>

**Types**

- `opts :: Keyword.t`

**Functions**

- `append(mod)` *(function)*
  
  Append the given SCM module to the list of available SCMs.

- `available()` *(function)*
  
  Returns all available SCMs. Each SCM is tried in order until a matching one is found.

- `prepend(mod)` *(function)*
  
  Prepend the given SCM module to the list of available SCMs.
Callbacks

accepts_options/2(callback)
Specs:

- accepts_options(app :: atom, opts :: opts | nil)

This behaviour function receives a keyword list of opts and should return an updated list in case the SCM consumes the available options. For example, when a developer specifies a dependency:

```elixir
{:foo, "0.1.0", github: "foo/bar"}
```

Each registered SCM will be asked if they consume this dependency, receiving [github: "foo/bar"] as argument. Since this option makes sense for the Git SCM, it will return an update list of options while other SCMs would simply return nil.

checked_out?/1(callback)
Specs:

- checked_out?(opts) :: boolean

This behaviour function returns a boolean if the dependency is available.

checkout/1(callback)
Specs:

- checkout(opts) :: any

This behaviour function checks out dependencies.

If the dependency is locked, a lock is received in opts and the repository must be check out at the lock. Otherwise, no lock is given and the repository can be checked out to the latest version.

It must return the current lock.

equal?/2(callback)
Specs:

- equal?(opts1 :: opts, opts2 :: opts) :: boolean

Receives two options and must return true if they refer to the same repository. The options are guaranteed to belong to the same SCM.

fetchable?/0(callback)
Specs:

- fetchable? :: boolean

Returns a boolean if the dependency can be fetched or it is meant to be previously available in the filesystem.

format/1(callback)
Specs:

- format(opts) :: String.t

Returns a string representing the SCM. This is used when printing the dependency and not for inspection, so the amount of information should be concise and easy to spot.

format_lock/1(callback)
Specs:

- format_lock(opts) :: String.t | nil

Returns a string representing the SCM. This is used when printing the dependency and not for inspection, so the amount of information should be concise and easy to spot.

If nil is returned, it means no lock information is available.
**lock_status/1(callback)**

Specs:

- `lock_status(opts) :: mismatch | outdated | ok`

This behaviour function checks the status of the lock. In particular, it checks if the revision stored in the lock is the same as the repository it is currently in. It may return:

- `mismatch` - if the lock doesn’t match and we need to simply move to the latest lock
- `outdated` - the repository options are outdated in the lock and we need to trigger a full update
- `ok` - everything is fine

The lock is sent via `opts[:lock]` but it may not always be available. In such cases, if the SCM requires a lock, it must return `:lockmismatch`, otherwise simply `:ok`.

Note the lock may also belong to another SCM and as such, an structural check is required. A structural mismatch should always return `:outdated`.

**update/1(callback)**

Specs:

- `update(opts) :: any`

This behaviour function updates dependencies. It may be called by `deps.get` or `deps.update`.

In the first scenario, a lock is received in `opts` and the repository must be updated to the lock. In the second, no lock is given and the repository can be updated freely.

It must return the current lock.

**Mix.Shell**

**Overview**

Defines Mix.Shell contract.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmd/2</code></td>
<td>An implementation of the command callback that is shared across different shells.</td>
</tr>
<tr>
<td><code>output_app?/0</code></td>
<td>Returns if we should output application name to shell. Calling this function automatically toggles its value to false.</td>
</tr>
</tbody>
</table>

**Functions**

- `cmd(command, callback)` (function)
  An implementation of the command callback that is shared across different shells.

- `output_app?/0` (function)
  Returns if we should output application name to shell. Calling this function automatically toggles its value to false.

**Callbacks**

- `cmd/1(callback)`
  Specs:

  ```
  cmd(command :: String.t) :: integer
  ```

  Executes the given command and returns its exit status.

- `error/1(callback)`
  Specs:

  ```
  error(message :: String.t) :: any
  ```
Warns about the given error message.

**info/1(callback)**

Specs:

- info(message :: String.t) :: any

Informs the given message.

**prompt/1(callback)**

Specs:

- prompt(message :: String.t) :: String.t

Prompts the user for input.

**yes?/1(callback)**

Specs:

- yes?(message :: String.t) :: boolean

Asks the user for confirmation.

---

**Mix.Shell.IO**

**Overview**  This is Mix’s default shell. It simply prints messages to stdio and stderr.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd(command)</td>
<td>Executes the given command and prints its output to stdout as it comes</td>
</tr>
<tr>
<td>error(message)</td>
<td>Writes an error message to the shell followed by new line</td>
</tr>
<tr>
<td>info(message)</td>
<td>Writes a message to the shell followed by new line</td>
</tr>
<tr>
<td>prompt(message)</td>
<td>Writes a message shell followed by prompting the user for input. Input will be consumed until enter is pressed</td>
</tr>
<tr>
<td>yes?(message)</td>
<td>Receives a message and asks the user if he wants to proceed. He must press enter or type anything that matches the a “yes” regex ~r/^Y(es)?$/i</td>
</tr>
</tbody>
</table>

**Mix.Shell.Process**

**Overview**  This is a Mix shell that uses the current process mailbox for communication instead of IO.

When a developer calls info("hello"), the following message will be sent to the current process:
This is mainly useful in tests, allowing us to assert if given messages were received or not. Since we need to guarantee a clean slate between tests, there is also a `flush/1` function responsible for flushing all `:mix_shell` related messages from the process inbox.

<table>
<thead>
<tr>
<th>cmd/1</th>
<th>Executes the given command and forwards its messages to the current process</th>
</tr>
</thead>
<tbody>
<tr>
<td>error/1</td>
<td>Forwards the message to the current process</td>
</tr>
<tr>
<td>flush/1</td>
<td>Flush all <code>:mix_shell</code> and <code>:mix_shell_input</code> messages from the current process. If a callback is given, it is invoked for each received message</td>
</tr>
<tr>
<td>info/1</td>
<td>Forwards the message to the current process</td>
</tr>
<tr>
<td>prompt/1</td>
<td>Forwards the message to the current process. It also checks the inbox for an input message matching:</td>
</tr>
<tr>
<td>yes?/1</td>
<td>Forwards the message to the current process. It also checks the inbox for an input message matching:</td>
</tr>
</tbody>
</table>

**Summary**

**cmd(command)** (function)

Executes the given command and forwards its messages to the current process.

**error(message)** (function)

Forwards the message to the current process.

**flush(callback (\ fn x -> x end))** (function)

Flush all `:mix_shell` and `:mix_shell_input` messages from the current process. If a callback is given, it is invoked for each received message.

**Examples**

`flush &IO.inspect(&1)`

**info(message)** (function)

Forwards the message to the current process.

**prompt(message)** (function)

Forwards the message to the current process. It also checks the inbox for an input message matching: `{:mix_shell_input, :prompt, value}`

If one does not exist, it will abort since there was no shell process inputs given. Value must be a string.

**yes?(message)** (function)

Forwards the message to the current process. It also checks the inbox for an input message matching: `{:mix_shell_input, :yes?, value}`

If one does not exist, it will abort since there was no shell process inputs given. Value must be `true` or `false`.

**Mix.Task**

**Overview**  A simple module that provides conveniences for creating, loading and manipulating tasks.

A Mix task can be defined by simply using `Mix.Task` in a module starting with `Mix.Tasks` and defining the `run/1` function:

```elixir
defmodule Mix.Tasks.Hello do
  use Mix.Task

  def run(_) do
```

430 Chapter 3. API reference
The `run/1` function will receive all arguments passed to the command line.

**Attributes**

There are a couple attributes available in Mix tasks to configure them in Mix:

- `@shortdoc` - makes the task public with a short description that appears on `mix help`
- `@recursive` - run the task recursively in umbrella projects

### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>all_modules/0</code></td>
<td>Returns all loaded tasks</td>
</tr>
<tr>
<td><code>clear/0</code></td>
<td>Clears all invoked tasks, allowing them to be reinvoked</td>
</tr>
<tr>
<td><code>get!/1</code></td>
<td>Receives a task name and retrieves the task module</td>
</tr>
<tr>
<td><code>get/1</code></td>
<td>Receives a task name and retrieves the task module. Returns nil if the task cannot be found</td>
</tr>
<tr>
<td><code>is_task?/1</code></td>
<td>Returns <code>true</code> if given module is a task</td>
</tr>
<tr>
<td><code>load_all/0</code></td>
<td>Loads all tasks in all code paths</td>
</tr>
<tr>
<td><code>load_tasks/(paths)</code></td>
<td>Loads all tasks in the given paths</td>
</tr>
<tr>
<td><code>moduledoc/(module)</code></td>
<td>Gets the moduledoc for the given task module. Returns the moduledoc or nil</td>
</tr>
<tr>
<td><code>recursive/1</code></td>
<td>Checks if the task should be run recursively for all sub-apps in umbrella projects. Returns <code>true</code>, <code>false</code> or <code>:both</code></td>
</tr>
<tr>
<td><code>reenable/1</code></td>
<td>Reenables a given task so it can be executed again down the stack. If an umbrella project reenables a task it is reenabled for all sub projects</td>
</tr>
<tr>
<td><code>run/2</code></td>
<td>Runs a task with the given <code>args</code></td>
</tr>
<tr>
<td><code>shortdoc/(task)</code></td>
<td>Gets the shortdoc for the given task module. Returns the shortdoc or nil</td>
</tr>
<tr>
<td><code>task_name/(module)</code></td>
<td>Returns the task name for the given module</td>
</tr>
</tbody>
</table>

### Functions

- `all_modules/0` *(function)*
  - Returns all loaded tasks.

- `clear/0` *(function)*
  - Clears all invoked tasks, allowing them to be reinvoked.

- `get/(task)` *(function)*
  - Receives a task name and retrieves the task module. Returns nil if the task cannot be found.

- `get!/(task)` *(function)*
  - Receives a task name and retrieves the task module.

- `is_task?/(module)` *(function)*
  - Returns `true` if given module is a task.

- `load_all/0` *(function)*
  - Loads all tasks in all code paths.

- `load_tasks/(paths)` *(function)*
  - Loads all tasks in the given `paths`.

- `moduledoc/(module)` *(function)*
  - Gets the moduledoc for the given task module. Returns the moduledoc or `nil`.

- `recursive/1` *(function)*
  - Checks if the task should be run recursively for all sub-apps in umbrella projects. Returns `true`, `false` or `:both`.

- `reenable/1` *(function)*
  - Reenables a given task so it can be executed again down the stack. If an umbrella project reenables a task it is reenabled for all sub projects.

- `run/2` *(function)*
  - Runs a task with the given `args`.

- `shortdoc/(task)` *(function)*
  - Gets the shortdoc for the given task module. Returns the shortdoc or `nil`.

- `task_name/(module)` *(function)*
  - Returns the task name for the given module.

**Exceptions**

- `Mix.NoTaskError` - raised if the task could not be found;
- `Mix.InvalidTaskError` - raised if the task is not a valid `Mix.Task`
moduledoc(module)(function)
   Gets the moduledoc for the given task module. Returns the moduledoc or nil.

recursive(module)(function)
   Checks if the task should be run recursively for all sub-apps in umbrella projects. Returns true, false or :both.

reenable(task)(function)
   Reenables a given task so it can be executed again down the stack. If an umbrella project reenables a task it is reenabled for all sub projects.

run(task, args \ [ ])(function)
   Runs a task with the given args.
   If the task was not yet invoked, it runs the task and returns the result.
   If the task was already invoked, it does not run the task again and simply aborts with :noop.
   It may raise an exception if the task was not found or it is invalid. Check get!/1 for more information.

shortdoc(module)(function)
   Gets the shortdoc for the given task module. Returns the shortdoc or nil.

task_name(module)(function)
   Returns the task name for the given module.

Callbacks
run/1(callback)
   Specs:
   •run([binary]) :: any
   A task needs to implement run which receives a list of command line args.

Mix.Tasks.App.Start

Overview  Starts all registered apps. If no apps key exists, it starts the current application.

Command line options
   • --force - force compilation regardless of compilation times
   • --no-compile - do not compile even if files require compilation
   • --no-deps-check - do not check dependencies
   • --no-elixir-version-check - do not check elixir version
   • --no-start - do not start applications after compilation

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions
run (args)(function)
   Callback implementation of Mix.Task.run/1.
Mix.Tasks.Archive

Overview  Packages the current project (though not its dependencies) into a zip file according to the specification of the Erlang Archive Format.

Archives are meant to bundle small projects, usually installed locally.
The file will be created in the current directory (which is expected to be the project root), unless an argument -o is provided with the file name.

Command line options

• -o - specify output file name. If there is a mix.exs, defaults to app-vsn.ez
• -i - specify the input directory to archive. If there is a mix.exs, defaults to the current application build
• --no-compile - skip compilation. Only applies to projects.

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions

run (args)  (function)
  Callback implementation of Mix.Task.run/1.

Mix.Tasks.Clean

Overview  Clean generated application files.

This command delete all build artifacts for the current application accross all environments. Dependencies are only cleaned up if the --all option is given.

Command line options

• --all - Clean everything, including builds and dependencies

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions

run (args)  (function)
  Callback implementation of Mix.Task.run/1.

Mix.Tasks.Cmd

Overview  Executes the given command.

Useful in umbrella applications to execute a command on each child app:

    mix cmd echo pwd

Aborts when the first command exits with status different than zero.

Summary  run/1  Callback implementation of Mix.Task.run/1
Functions

**run (args)** (function)

Callback implementation of `Mix.Task.run/1`.

**Mix.Tasks.Compile**

**Overview**

A meta task that compiles source files.

It simply runs the compilers registered in your project. At the end of compilation it ensures load paths are set.

**Configuration**

- **:compilers** - compilers to run, defaults to:
  
  [:leex, :yeec, :erlang, :elixir, :app]

**Command line options**

- **--list** - List all enabled compilers
- **--no-deps-check** - Skips checking of dependencies
- **--force** - Forces compilation

**Summary**

<table>
<thead>
<tr>
<th>manifest/0</th>
<th>run/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns manifests for all compilers</td>
<td>Callback implementation of <code>Mix.Task.run/1</code></td>
</tr>
</tbody>
</table>

**Functions**

**manifests () (function)**

Returns manifests for all compilers.

**run (args) (function)**

Callback implementation of `Mix.Task.run/1`.

**Mix.Tasks.Compile.App**

**Overview**

Writes an `.app` file.

An `.app` file is a file containing Erlang terms that defines your application. Mix automatically generates this file based on your `mix.exs` configuration. You can learn more about OTP applications by seeing the documentation for the `Application` module.

In order to generate the `.app` file, Mix expects your application to have both :app and :version keys. Furthermore, you can configure the generated application by defining an application function in your `mix.exs` with the following options:

- **:applications** - all applications your application depends on at runtime. For example, if your application depends on Erlang’s `:crypto`, it needs to be added to this list. Most of your dependencies must be added as well (unless they’re a development or test dependency). Mix and other tools use this list in order to properly boot your application dependencies before starting the application itself;
- **:registered** - the name of all registered processes in the application. If your application defines a local GenServer with name `MyServer`, it is recommended to add `MyServer` to this list. It is mostly useful to detect conflicts in between applications that register the same names;
• \texttt{:mod} - specify a module to invoke when the application is started, it must be in the format \texttt{(Mod, args)} where \texttt{args} is often an empty list. The module specified here must implement the callbacks defined by the \texttt{Application} module;

• \texttt{:env} - default values for the application environment. The application environment is one of the most common ways to configure applications;

Let's see an example application function:

```
def application do
    [mod: {MyApp, []},
     env: [default: :value],
     applications: [:crypto]]
end
```

Besides the options above, \texttt{.app} files also expects other options like \texttt{:modules} and \texttt{:vsn}, but those are automatically filled by Mix.

**Command line options**

• \texttt{--force} - forces compilation regardless of modification times

**Summary**

<table>
<thead>
<tr>
<th>Run/1</th>
<th>Callback implementation of Mix.Task.run/1</th>
</tr>
</thead>
</table>

**Functions**

\texttt{run (args) (function)}

Callback implementation of \texttt{Mix.Task.run/1}.

**Mix.Tasks.Compile.Elixir**

**Overview**  Compiles Elixir source files.

Elixir is smart enough to recompile only files that changed and their dependencies. This means if \texttt{lib/a.ex} is invoking a function defined over \texttt{lib/b.ex}, whenever \texttt{lib/b.ex} changes, \texttt{lib/a.ex} is also recompiled.

Note it is important to recompile a file dependencies because often there are compilation time dependencies in between them.

**Command line options**

• \texttt{--force} - forces compilation regardless of modification times;

• \texttt{--no-docs} - Do not attach documentation to compiled modules;

• \texttt{--no-debug-info} - Do not attach debug info to compiled modules;

• \texttt{--ignore-module-conflict}

• \texttt{--warnings-as-errors} - Treat warnings as errors and return a non-zero exit code

**Configuration**

• \texttt{:elixirc_paths} - directories to find source files. Defaults to \texttt{["lib"]}, can be configured as:

• \texttt{:elixirc_options} - compilation options that apply to Elixir’s compiler, they are: \texttt{:ignore_module_conflict}, \texttt{:docs} and \texttt{:debug_info}. By default, uses the same behaviour as Elixir;
**Summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>files_to_path/5</code></td>
<td>Compiles stale Elixir files</td>
</tr>
<tr>
<td><code>manifests/0</code></td>
<td>Returns Elixir manifests</td>
</tr>
<tr>
<td><code>run/1</code></td>
<td>Runs this task</td>
</tr>
</tbody>
</table>

**Functions**

`files_to_path(manifest, force, all, path, on_start) (function)`

Compiles stale Elixir files.

It expects a manifest file, a flag if compilation should be forced or not, all source files available (including the ones that are not stale) and a path where compiled files will be written to. All paths are required to be relative to the current working directory.

The manifest is written down with information including dependencies in between modules, which helps it recompile only the modules that have changed at runtime.

`manifests() (function)`

Returns Elixir manifests.

`run(args) (function)`

Runs this task.

---

**Mix.Tasks.Compile.Erlang**

**Overview**

Compile Erlang source files.

When this task runs, it will first check the modification times of all files to be compiled and if they haven’t been changed since the last compilation, it will not compile them. If any of them have changed, it compiles everything.

For this reason, the task touches your `:compile_path` directory and sets the modification time to the current time and date at the end of each compilation. You can force compilation regardless of modification times by passing the `--force` option.

**Command line options**

- `--force` - forces compilation regardless of modification times

**Configuration**

- `ERL_COMPILER_OPTIONS` - can be used to give default compile options. The value must be a valid Erlang term. If the value is a list, it will be used as is. If it is not a list, it will be put into a list.
- `:erlc_paths` - directories to find source files. Defaults to `["src"]`, can be configured as:
  ```erlang
  [erlc_paths:  ["src", "other"]]
  ```
- `:erlc_include_path` - directory for adding include files. Defaults to "include", can be configured as:
  ```erlang
  [erlc_include_path:  "other"]
  ```
- `:erlc_options` - compilation options that apply to Erlang’s compiler. `:debug_info` is enabled by default.

There are many available options here: [http://www.erlang.org/doc/man/compile.html#file-2](http://www.erlang.org/doc/man/compile.html#file-2)
### Summary

- **compile_mappings/6**
  - Extracts the extensions from the mappings, automatically invoking the callback for each stale input and output pair (or for all if `force` is true) and removing files that no longer have a source, while keeping the manifest up to date.

- **manifests/0**
  - Returns Erlang manifests.

- **run/1**
  - Runs this task.

- **to_erl_file/1**
  - Converts the given file to a format accepted by the Erlang compilation tools.

### Functions

#### compile_mappings(manifest, mappings, src_ext, dest_ext, force, callback)(function)

Extracts the extensions from the mappings, automatically invoking the callback for each stale input and output pair (or for all if `force` is true) and removing files that no longer have a source, while keeping the manifest up to date.

**Examples**

For example, a simple compiler for Lisp Flavored Erlang would be implemented like:

```erlang
compile_mappings ",compile.lfe",
    [{"src", "ebin"}],
    :lfe, :beam, opts[:force], fn
    input, output ->
      :lfe_comp.file(to_erl_file(input),
                      [output_dir: Path.dirname(output)])
    end
```

The command above will:

1. Look for files ending with the `lfe` extension in `src` and their `beam` counterpart in `ebin`;
2. For each stale file (or for all if `force` is true), invoke the callback passing the calculated input and output;
3. Update the manifest with the newly compiled outputs;
4. Remove any output in the manifest that does not have an equivalent source;

The callback must return `{:ok, mod}` or `:error` in case of error. An error is raised at the end if any of the files failed to compile.

#### manifests/0(function)

Returns Erlang manifests.

#### run(args)(function)

Runs this task.

#### to_erl_file(file)(function)

Converts the given file to a format accepted by the Erlang compilation tools.

### Mix.Tasks.Compile.Leex

#### Overview

Compile Leex source files.

When this task runs, it will check the modification time of every file, and if it has changed, the file will be compiled. Files will be compiled in the same source directory with a `.erl` extension. You can force compilation regardless of modification times by passing the `--force` option.

#### Command line options

- `--force` - forces compilation regardless of modification times;
Configuration

- :erlc_paths - directories to find source files. Defaults to ["src"], can be configured as:

```
{erlc_paths: ["src", "other"]}
```

- :leex_options - compilation options that apply to Leex’s compiler. There are many available options here:

```
http://www.erlang.org/doc/man/leex.html#file-2
```

## Mix.Tasks.Compile.Protocols

**Overview**  
Consolidates all protocols in all paths.

This module consolidates all protocols in the code path and output the new binary files to the given directory (defaults to “consolidated”).

A new directory will be created with the consolidated protocol versions. Simply add it to your codepath to make use of it:

```
mix run -pa _build/dev/consolidated
```

You can verify a protocol is consolidated by checking its attributes:

```
elixir -pa _build/dev/consolidated -S mix run -e "IO.puts Protocol.consolidated?(Enumerable)"
```

## Functions

### manifests() (function)

Returns Leex manifests.

### run(args) (function)

Runs this task.

## Mix.Tasks.Compile.Yecc

**Overview**  
Compile Yecc source files.

When this task runs, it will check the modification time of every file, and if it has changed, the file will be compiled. Files will be compiled in the same source directory with a .erl extension. You can force compilation regardless of modification times by passing the --force option.

**Command line options**

- --force - forces compilation regardless of modification times;
Configuration

- **:erlc_paths** - directories to find source files. Defaults to "$src". can be configured as:

  ```erlang
  {erlc_paths, ["src", "other"]}
  ```

- **:yecc_options** - compilation options that apply to Yecc’s compiler. There are many other available options here: [http://www.erlang.org/doc/man/yecc.html#file-1](http://www.erlang.org/doc/man/yecc.html#file-1)

Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>manifests/0</td>
<td>Returns Yecc manifests</td>
</tr>
<tr>
<td>run/1</td>
<td>Runs this task</td>
</tr>
</tbody>
</table>

Functions

**manifests/0** *(function)*

Returns Yecc manifests.

**run/1** *(function)*

Runs this task.

Mix.Tasks.Deps

Overview

List all dependencies and their status.

Dependencies must be specified in the `mix.exs` file in one of the following formats:

```
{app, requirement}
{app, opts}
{app, requirement, opts}
```

Where:

- **app** is an atom
- **requirement** is a version requirement or a regular expression
- **opts** is a keyword list of options

By default, dependencies are fetched using the Hex package manager:

```
{:plug, ">= 0.4.0"}
```

By specifying such dependencies, Mix will automatically install Hex (if it wasn’t previously installed and download a package suitable to your project).

Mix also supports git and path dependencies:

```
{:foobar, git: "https://github.com/elixir-lang/foobar.git", tag: "0.1"}
```

And also umbrella applications:

```
{:myapp, in_umbrella: true}
```

The dependencies versions are expected to follow Semantic Versioning and the requirements must be specified as defined in the `Version` module.

Below we provide a more detailed look into the available options.
Mix options

- **:app** - When set to false, does not read the app file for this dependency
- **:env** - The environment to run the dependency on, defaults to :prod
- **:compile** - A command to compile the dependency, defaults to a mix, rebar or make command
- **:optional** - The dependency is optional and used only to specify requirements
- **:only** - The dependency will belong only to the given environments, useful when declaring dev- or test-only dependencies
- **:override** - If set to true the dependency will override any other definitions of itself by other dependencies

Git options (**:git**)

- **:git** - The git repository URI
- **:github** - A shortcut for specifying git repos from github, uses git:
- **:ref** - The reference to checkout (may be a branch, a commit sha or a tag)
- **:branch** - The git branch to checkout
- **:tag** - The git tag to checkout
- **:submodules** - When true, initialize submodules for the repo

Path options (**:path**)

- **:path** - The path for the dependency
- **:in_umbrella** - When true, sets a path dependency pointing to "./#{app}", sharing the same environment as the current application

mix deps task  This task lists all dependencies in the following format:

- APP VERSION (SCM) [locked at REF] STATUS

It supports the following options:

- **--all** - check all dependencies, regardless of specified environment

Summary  | run/1  | Callback implementation of Mix.Task.run/1 |

Functions

**run (args) (function)**

Callback implementation of Mix.Task.run/1.

Mix.Tasks.Deps.Check

Overview  Checks if all dependencies are valid and if not, abort. Prints the invalid dependencies’ status before aborting.

This task is not shown in mix help but it is part of the mix public API and can be depended on.
Command line options

- `--no-compile` - do not compile dependencies
- `--quiet` - do not output on compilation

Summary

run/1 Callback implementation of Mix.Task.run/1

Functions

run (args) (function)

Callback implementation of Mix.Task.run/1.

Mix.Tasks.Deps.Clean

Overview

Remove the given dependencies’ files.

Since this is a destructive action, cleaning of all dependencies can only happen by passing the `--all` command line option. It also works across all environments, unless `--only` is given.

Clean does not unlock the dependencies, unless `--unlock` is given.

Summary

run/1 Callback implementation of Mix.Task.run/1

Functions

run (args) (function)

Callback implementation of Mix.Task.run/1.

Mix.Tasks.Deps.Compile

Overview

Compile dependencies.

By default, compile all dependencies. A list of dependencies can be given to force the compilation of specific dependencies.

By default, attempt to detect if the project contains one of the following files:

- `mix.exs` - if so, invokes `mix compile`
- `rebar.config` - if so, invokes `rebar compile`
- `Makefile` - if so, invokes `make`

The compilation can be customized by passing a `compile` option in the dependency:

```
{:some_dependency, "0.1.0", git: "...", compile: "command to compile"}
```

Command line options

- `--quiet` - do not output verbose messages

Summary

run/1 Callback implementation of Mix.Task.run/1
Functions

run(args) (function)

Callback implementation of Mix.Task.run/1.

Mix.Tasks.Deps.Get

Overview  Get all out of date dependencies, i.e. dependencies that are not available or have an invalid lock.

Command line options

• --quiet - do not output verbose messages
• --only - only fetch dependencies for given environment

Summary  run/1  Callback implementation of Mix.Task.run/1

Mix.Tasks.Deps.Loadpaths

Overview  Loads all dependencies for the current build. This is invoked directly by loadpaths when the CLI boots.

Command line options

• --no-deps-check - skip dependency check

Summary  run/1  Callback implementation of Mix.Task.run/1

Mix.Tasks.Deps.Unlock

Overview  Unlock the given dependencies.

Since this is a destructive action, unlocking of all dependencies can only happen by passing the --all command line option.

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions

run(args) (function)

Callback implementation of Mix.Task.run/1.
Mix.Tasks.Deps.Update

Overview  Update the given dependencies.
Since this is a destructive action, update of all dependencies can only happen by passing the --all command line option.
All dependencies are automatically recompiled after update.

Command line options
• --all - update all dependencies
• --only - only fetch dependencies for given environment

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions
run (args) (function)
Callback implementation of Mix.Task.run/1.

Mix.Tasks.Do

Overview  Executes the tasks separated by comma.

Examples  The example below prints the available compilers and then the list of dependencies.
mix do compile --list, deps

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions
run (args) (function)
Callback implementation of Mix.Task.run/1.

Mix.Tasks.Escriptize

Overview  Generates an escript for the project.

Command line options
• --force - forces compilation regardless of modification times
• --no-compile - skips compilation to .beam files
Configuration

The following option must be specified in your `mix.exs`:

- `:escript_main_module` - the module to be invoked once the escript starts. The module must contain a function named `main/1` that will receive the command line arguments as binaries.

The remaining options can be specified to further customize the escript:

- `:escript_name` - the name of the generated escript. Defaults to app name;
- `:escript_path` - the path to write the escript to. Defaults to app name;
- `:escript_app` - the app to start with the escript. Defaults to app name. Set it to `nil` if no application should be started.
- `:escript_embed_elixir` - if `true` embed elixir in the escript file. Defaults to `true`.
- `:escript_embed_extra_apps` - embed additional Elixir applications. If `:escript_embed_elixir` is `true`. Defaults to `[]`.
- `:escript_shebang` - shebang interpreter directive used to execute the escript. Defaults to `"#!/usr/bin/env escript"`.
- `:escript_comment` - comment line to follow shebang directive in the escript. Defaults to `"%%%"`
- `:escript_emu_args` - emulator arguments to embed in the escript file. Defaults to `"%%%!"`.

Summary

| run/1 | Callback implementation of `Mix.Task.run/1` |

Functions

`run (args) (function)`

Callback implementation of `Mix.Task.run/1`.

Mix.Tasks.Help

Overview

Lists all tasks or prints the documentation for a given task.

Arguments

- `mix help` - prints all tasks and their shortdoc
- `mix help TASK` - prints full docs for the given task

Colors

When possible, `mix help` is going to use coloring for formatting guides. The formatting can be customized by configuring the Mix application either inside your project (in `config/config.exs`) or by using the local config (in `~/.mix/config.exs`).

For example, to disable, one may:

```
[mix: [colors: [enabled: false]]]
```

The available color options are:

- `:enabled` - show ANSI formatting (defaults to `IO.ANSI.terminal??`)
- `:doc_code` — the attributes for code blocks (cyan, bright)
- `:doc_inline_code` - inline code (cyan)
- `:doc_headings` - h1 and h2 (yellow, bright)
- `:doc_title` — the overall heading for the output (reverse,yellow,bright)
• :doc_bold - (bright)
• :doc_underline - (underline)

Summary

**run/1**
Callback implementation of `Mix.Task.run/1`

**Functions**

**run (arg1) (function)**
Callback implementation of `Mix.Task.run/1`.

**Mix.Tasks.lex**

**Overview**
A task that is simply meant to redirect users to `iex -S mix`.

Summary

**run/1**
Callback implementation of `Mix.Task.run/1`

**Functions**

**run () (function)**
Callback implementation of `Mix.Task.run/1`.

**Mix.Tasks.Loadconfig**

**Overview**
Loads and persists the project configuration.

In case the application is an umbrella application, the configuration for all children app will be merged together and, in case there are any conflicts, they need to be resolved in the umbrella application.

Summary

**load/0**
Loads the configuration for the current project

**run/1**
Runs this task

**Functions**

**load () (function)**
Loads the configuration for the current project.

**run () (function)**
Runs this task.

**Mix.Tasks.Loadpaths**

**Overview**
Loads the application and its dependencies paths.

**Command line options**

• `--no-elixir-version-check` - do not check elixir version

Summary

**run/1**
Callback implementation of `Mix.Task.run/1`
Elixir Documentation, Release

Functions

run(args) (function)
Callback implementation of Mix.Task.run/1.

Mix.Tasks.Local

Overview
List local tasks.

Summary
run/1 Callback implementation of Mix.Task.run/1

Functions

run(list1) (function)
Callback implementation of Mix.Task.run/1.

Mix.Tasks.Local.Hex

Overview
Install hex locally from https://hex.pm/installs/hex.ez.
mix local.hex

Command line options

• --force - forces installation without a shell prompt. Primarily intended for automation in build systems like make.

Summary
run/1 Callback implementation of Mix.Task.run/1

Functions

run(args) (function)
Callback implementation of Mix.Task.run/1.

Mix.Tasks.Local.Install

Overview
Install an archive locally.
If no argument is supplied but there is an archive in the root (created with mix archive), then the archive will be installed locally. For example:
mix do archive, local.install

The argument can be an archive located at some URL:
mix local.install http://example.com/foo.ez

After installed, the tasks in the archive are available locally:
mix some_task
Command line options

- **--force** - forces installation without a shell prompt. Primarily intended for automation in build systems like make.

Summary

| run/1 | Callback implementation of Mix.Task.run/1 |

Functions

**run (argv) (function)**

Callback implementation of Mix.Task.run/1.

Mix.Tasks.Local.Rebar

Overview

Fetch a copy of rebar from the given path or URL. It defaults to a rebar copy that ships with Elixir source if available or fetches it from http://s3.hex.pm/rebar.

The local copy is stored in your MIX_HOME (defaults to ~/.mix). This version of rebar will be used as required by `mix deps.compile`.

Summary

| run/1 | Callback implementation of Mix.Task.run/1 |

Functions

**run (argv) (function)**

Callback implementation of Mix.Task.run/1.

Mix.Tasks.Local.Uninstall

Overview

Uninstall local tasks:

```
mix local.uninstall archive
```

Summary

| run/1 | Callback implementation of Mix.Task.run/1 |

Functions

**run (argv) (function)**

Callback implementation of Mix.Task.run/1.

Mix.Tasks.New

Overview

Creates a new Elixir project. It expects the path of the project as argument.

```
mix new PATH [--bare] [--module MODULE] [--umbrella]
```

A project at the given PATH will be created. The application name and module name will be retrieved from the path, unless --module is given.

A --sup option can be given to generate an OTP application skeleton including a supervision tree. Normally an app is generated without a supervisor and without the app callback.

An --umbrella option can be given to generate an umbrella project.
Examples

mix new hello_world

Is equivalent to:

mix new hello_world --module HelloWorld

To generate an app with supervisor and application callback:

mix new hello_world --sup

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions

run (argv) (function)

Callback implementation of Mix.Task.run/1.

Mix.Tasks.Run

Overview  Runs the given file or expression in the context of the application.

Before running the code, it invokes the app.start task which compiles and loads your project.

It is the goal of this task to provide a subset of the functionality existent in the elixir executable, including setting up the System.argv:

```
mix run -e Hello.world
mix run my_script.exs arg1 arg2 arg3
```

Many command line options need to be passed to the elixir executable directly, which can be done as follows:

```
elixir --sname hello -S mix run -e "My.code"
```

Command line options

- --eval, -e  evaluate the given code
- --require, -r  require pattern before running the command
- --parallel-require, -pr  requires pattern in parallel
- --no-compile  do not compile even if files require compilation
- --no-deps-check  do not check dependencies
- --no-halt  do not halt the system after running the command
- --no-start  do not start applications after compilation

Summary  run/1  Callback implementation of Mix.Task.run/1

Functions

run (args) (function)

Callback implementation of Mix.Task.run/1.
Mix.Tasks.Test

Overview  Run the tests for a project.
This task starts the current application, loads up test/test_helper.exs and then requires all files matching the test/**/_test.exs pattern in parallel.
A list of files can be given after the task name in order to select the files to compile:
mix test test/some/particular/file_test.exs

Command line options

• --trace - run tests with detailed reporting. Automatically sets --max-cases to 1
• --max-cases - set the maximum number of cases running async
• --cover - the directory to include coverage results
• --force - forces compilation regardless of modification times
• --no-compile - do not compile, even if files require compilation
• --no-start - do not start applications after compilation
• --no-color - disable color in the output
• --include - include tests that match the filter
• --exclude - exclude tests that match the filter
• --only - run only tests that match the filter
• --seed - seeds the random number generator used to randomize tests order

Filters   ExUnit provides tags and filtering functionality that allows developers to select which tests to run. The most common functionality is to exclude some particular tests from running by default in your test helper file:

```elixir
# Exclude all external tests from running
ExUnit.configure exclude: [external: true]
```

Then, whenever desired, those tests could be included in the run via the --include flag:
mix test --include external

The example above will run all tests that have the external flag set to true. It is also possible to include all examples that have a given tag, regardless of its value:
mix test --include external

Note that all tests are included by default, so unless they are excluded first (either in the test helper or via the --exclude option), the --include flag has no effect.

For this reason, mix also provides an --only option that excludes all tests and includes only the given ones:
mix test --only external

Which is equivalent to:
mix test --include external --exclude test

When filtering tests by line number the following styles are equivalent:
mix test test/some/particular/file_test.exs:12
mix test --only line:12 test/some/particular/file_test.exs

Configuration

- :test_paths - list of paths containing test files, defaults to ["test"]. It is expected all test paths to contain a test_helper.exs file
- :test_pattern - a pattern to load test files, defaults to *_test.exs
- :test_coverage - a set of options to be passed down to the coverage mechanism

Coverage

The :test_coverage configuration accepts the following options:

- :output - the output for cover results, defaults to "cover"
- :tool - the coverage tool

By default, a very simple wrapper around OTP’s cover is used as a tool, but it can be overridden as follows:

test_coverage: [tool: CoverModule]

CoverModule can be any module that exports start/2, receiving the compilation path and the test_coverage options as arguments. It must return an anonymous function of zero arity that will be run after the test suite is done or nil.

Summary

run/1  Callback implementation of Mix.Task.run/1

Functions

run (args) (function)

Callback implementation of Mix.Task.run/1.

Mix.UTILS

Overview

Utilities used throughout Mix and tasks.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>camelize/1</td>
<td>Converts the given string to CamelCase format</td>
</tr>
<tr>
<td>command_to_module/2</td>
<td>Take a command name and attempts to load a module with the command name</td>
</tr>
<tr>
<td>command_to_module_name/2</td>
<td>converted to a module name in the given at scope</td>
</tr>
<tr>
<td>extract_files/2</td>
<td>Takes a command and converts it to the module name format</td>
</tr>
<tr>
<td>extract_stale/2</td>
<td>Extract files from a list of paths</td>
</tr>
<tr>
<td>last_modified/1</td>
<td>Extract all stale sources compared to the given targets</td>
</tr>
<tr>
<td>mix_home/0</td>
<td>Returns the date the given path was last modified</td>
</tr>
<tr>
<td>mix_paths/0</td>
<td>Get the mix home</td>
</tr>
<tr>
<td>module_name_to_command/2</td>
<td>Get all paths defined in the MIX_PATH env variable</td>
</tr>
<tr>
<td>read_manifest/1</td>
<td>Takes a module and converts it to a command</td>
</tr>
<tr>
<td>read_path!/1</td>
<td>Reads the given file as a manifest and returns each entry as a list</td>
</tr>
<tr>
<td>stale?/2</td>
<td>Opens and reads content from either a URL or a local filesystem path</td>
</tr>
<tr>
<td>symlink_or_copy/2</td>
<td>Returns true if any of the sources are stale compared to the given targets</td>
</tr>
<tr>
<td>underscore/1</td>
<td>Symlink directory source to target or copy it recursively in case symlink fails</td>
</tr>
<tr>
<td>write_manifest/2</td>
<td>Converts the given atom or binary to underscore format</td>
</tr>
<tr>
<td></td>
<td>Writes a manifest file with the given entries list</td>
</tr>
</tbody>
</table>
Functions

**camelize(arg1)** (function)
Converts the given string to CamelCase format.

Examples

```iex
iex> Mix.Utils.camelize "foo_bar"
"FooBar"
```

**command_to_module(command, at \ Elixir)** (function)
Take a command name and attempts to load a module with the command name converted to a module name in the given at scope.

Returns `{:module, module}` in case a module exists and is loaded, `{:error, reason}` otherwise.

Examples

```iex
iex> Mix.Utils.command_to_module("compile", Mix.Tasks)
{:module, Mix.Tasks.Compile}
```

**command_to_module_name(s)** (function)
Takes a command and converts it to the module name format.

Examples

```iex
iex> Mix.Utils.command_to_module_name("compile.elixir")
"Compile.Elixir"
```

**extract_files(paths, exts_or_pattern)** (function)
Extract files from a list of paths.

`exts_or_pattern` may be a list of extensions or a Path.wildcard/1 pattern.

If the path in `paths` is a file, it is included in the return result. If it is a directory, it is searched recursively for files with the given extensions or matching the given patterns.

Any file starting with "." is ignored.

**extract_stale(sources, targets)** (function)
Extract all stale sources compared to the given targets.

**last_modified(path)** (function)
Returns the date the given path was last modified.

If the path does not exist, it returns the unix epoch (1970-01-01 00:00:00).

**mix_home()** (function)
Get the mix home.

It defaults to `~/.mix` unless the MIX_HOME environment variable is set.

**mix_paths()** (function)
Get all paths defined in the MIX_PATH env variable.

`MIX_PATH` may contain multiple paths. If on Windows, those paths should be separated by `;`, if on unix systems, use `:`.

**module_name_to_command(module, nesting \ 0)** (function)
Takes a module and converts it to a command.

The nesting argument can be given in order to remove the nesting of a module.

Examples
iex> Mix.Utils.module_name_to_command(Mix.Tasks.Compile, 2)
"compile"

"compile.elixir"

**read_manifest** *(file) (function)*
Reads the given file as a manifest and returns each entry as a list.

A manifest is a tabular file where each line is a row and each entry in a row is separated by "". The first entry must always be a path to a compiled artifact.

In case there is no manifest file, returns an empty list.

**read_path!** *(path) (function)*
Opens and reads content from either a URL or a local filesystem path.

Used by tasks like `local.install` and `local.rebar` that support installation either from a URL or a local file.

Raises if the given path is not a url, nor a file or if the file or url are invalid.

**stale?** *(sources, targets) (function)*
Returns `true` if any of the `sources` are stale compared to the given `targets`.

**symlink_or_copy** *(source, target) (function)*
Symlink directory `source` to `target` or copy it recursively in case symlink fails.

Expect source and target to be absolute paths as it generates a relative symlink.

**underscore** *(atom) (function)*
Converts the given atom or binary to underscore format.

If an atom is given, it is assumed to be an Elixir module, so it is converted to a binary and then processed.

**Examples**

iex> Mix.Utils.underscore "FooBar"
"foo_bar"

iex> Mix.Utils.underscore "Foo.Bar"
"foo/bar"

iex> Mix.Utils.underscore Foo.Bar
"foo/bar"

In general, underscore can be thought of as the reverse of camelize, however, in some cases formatting may be lost:

Mix.Utils.underscore "SAPExample" #=> "sap_example"
Mix_Utils.camelize "sap_example" #=> "SapExample"

**write_manifest** *(file, entries) (function)*
Writes a manifest file with the given `entries` list.
Exceptions


Mix.ElixirVersionError

Overview

Summary

Types

t :: %Mix.ElixirVersionError{__exception__: term, mix_error: term, target: term, expected: term, actual: term, message: term}

Functions

exception(args) (function)

Specs:

•exception(term) :: t

Callback implementation of Exception.exception/1.

message(exception) (function)

Specs:

•message(t) :: String.t

Callback implementation of Exception.message/1.

Mix.Error

Overview

Summary

Types

t :: %Mix.Error{__exception__: term, mix_error: term, message: term}

Functions

exception(args) (function)

Specs:

•exception(term) :: t

Callback implementation of Exception.exception/1.
message(exception)(function)
Specs:
  *message(t):: String.t
  Callback implementation of Exception.message/1.

Mix.InvalidTaskError

Overview

Summary

Types
t
t :: %Mix.InvalidTaskError{__exception__: term, task: term, mix_error: term, message: term}

Functions
exception(args)(function)
Specs:
  *exception(term):: t
  Callback implementation of Exception.exception/1.
message(exception)(function)
Specs:
  *message(t):: String.t
  Callback implementation of Exception.message/1.

Mix.NoProjectError

Overview

Summary

Types
t
t :: %Mix.NoProjectError{__exception__: term, mix_error: term, message: term}

Functions
exception(args)(function)
Specs:
  *exception(term):: t
  Callback implementation of Exception.exception/1.
message(exception)(function)
Specs:
•message(t)::String.t

Callback implementation of Exception.message/1.

Mix.NoTaskError

Overview

<table>
<thead>
<tr>
<th>Summary</th>
<th>exception/1</th>
<th>Callback implementation of Exception.exception/1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>message/1</td>
<td>Callback implementation of Exception.message/1</td>
</tr>
</tbody>
</table>

Types

t

t::%Mix.NoTaskError{__exception__: term, task: term, mix_error: term, message: term}

Functions

exception(args)(function)

Specs:

•exception(term)::t

Callback implementation of Exception.exception/1.

message(exception)(function)

Specs:

•message(t)::String.t

Callback implementation of Exception.message/1.

3.2.5 IEx v0.14.0-dev

• Modules

Modules

<table>
<thead>
<tr>
<th>IEx</th>
<th>IEx.Helpers</th>
<th>IEx.History.Server</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elixir’s interactive shell</td>
<td>Welcome to Interactive Elixir. You are currently seeing the documentation for the module IEx.Helpers which provides many helpers to make Elixir’s shell more joyful to work with</td>
</tr>
</tbody>
</table>

IEx

Overview

Elixir’s interactive shell.

This module is the main entry point for Interactive Elixir and in this documentation we will talk a bit about how IEx works.

Notice that some of the functionality described here will not be available depending on your terminal. In particular, if you get a message saying that the smart terminal could not be run, some of the features described here won’t work.

Helpers

IEx provides a bunch of helpers. They can be accessed by typing h() into the shell or as a documentation for the IEx.Helpers module.
The Break command  Inside IEx, hitting Ctrl+C will open up the BREAK menu. In this menu you can quit the shell, see process and ets tables information and much more.

The User Switch command  Besides the break command, one can type Ctrl+G to get to the user switch command menu. When reached, you can type h to get more information.

In this menu, developers are able to start new shells and alternate between them. Let’s give it a try:

```
User switch command
--> s 'Elixir.IEx'
--> c
```

The command above will start a new shell and connect to it. Create a new variable called hello and assign some value to it:

```
hello = :world
```

Now, let’s roll back to the first shell:

```
User switch command
--> c 1
```

Now, try to access the hello variable again:

```
hello
** (UndefinedFunctionError) undefined function: hello/0
```

The command above fails because we have switched shells. Since shells are isolated from each other, you can’t access the variables defined in one shell from the other one.

The user switch command menu also allows developers to connect to remote shells using the r command. A topic which we will discuss next.

Remote shells  IEx allows you to connect to another node in two fashions. First of all, we can only connect to a shell if we give names both to the current shell and the shell we want to connect to.

Let’s give it a try. First start a new shell:

```
$ iex --sname foo
```

```
iex(foo@HOST)1>
```

The string in between parenthesis in the prompt is the name of your node. We can retrieve it by calling the node() function:

```
iex(foo@HOST)1> node()
:"foo@HOST"
iex(foo@HOST)2> Node.alive?()
true
```

For fun, let’s define a simple module in this shell too:

```
iex(foo@HOST)3> defmodule Hello do
... end
```

Now, let’s start another shell, giving it a name as well:

```
$ iex --sname bar
```

```
iex(bar@HOST)1>
```
If we try to dispatch to `Hello.world`, it won’t be available as it was defined only in the other shell:

```elixir
iex(bar@HOST)1> Hello.world
** (UndefinedFunctionError) undefined function: Hello.world/0
```

However, we can connect to the other shell remotely. Open up the User Switch prompt (Ctrl+G) and type:

```
User switch command
  --> r 'foo@HOST' 'Elixir.IEx'
  --> c
```

Now we are connected into the remote node, as the prompt shows us, and we can access the information and modules defined over there:

```elixir
rem(foo@macbook)1> Hello.world
"it works"
```

In fact, connecting to remote shells is so common that we provide a shortcut via the command line as well:

```
$ iex --sname baz --remsh foo@HOST
```

Where “remsh” means “remote shell”. In general, Elixir supports:

- remsh from an elixir node to an elixir node
- remsh from a plain erlang node to an elixir node (through the ^G menu)
- remsh from an elixir node to a plain erlang node (and get an erl shell there)

Connecting an Elixir shell to a remote node without Elixir is not supported.

**The .iex.exs file**  When starting IEx, it will look for a local `.iex.exs` file (located in the current working directory), then a global one (located at `~/.iex.exs`) and will load the first one it finds (if any). The code in the chosen `.iex` file will be evaluated in the shell’s context. So, for instance, any modules that are loaded or variables that are bound in the `.iex` file will be available in the shell after it has booted.

Sample contents of a local `.iex` file:

```elixir
# source another '.iex' file
import_file "~/.iex.exs"

# print something before the shell starts
IO.puts "hello world"

# bind a variable that’ll be accessible in the shell
value = 13
```

Running the shell in the directory where the above `.iex` file is located results in:

```
$ iex
Erlang 17 [...] hello world
```

It is possible to load another file by supplying the `--dot-iex` option to `iex`. See `iex --help`.
Configuring the shell  

There are a number of customization options provided by the shell. Take a look at the docs for the `IEx.configure/1` function by typing `h IEx.configure/1`.

Those options can be configured in your project configuration file or globally by calling `IEx.configure/1` from your `~/.iex.exs` file like this:

```
# .iex
IEx.configure(inspect: [limit: 3])
```

### now run the shell ###

```
$ iex
Erlang 17 (erts-5.10.1) [...]
Interactive Elixir - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> [1, 2, 3, 4, 5]
[1,2,3,...]
```

Expressions in IEx  

As an interactive shell, IEx evaluates expressions. This has some interesting consequences that are worth discussing.

The first one is that the code is truly evaluated and not compiled. This means that any benchmarking done in the shell is going to have skewed results. So never run any profiling nor benchmarks in the shell.

Second, IEx allows you to break an expression into many lines, since this is common in Elixir. For example:

```
iex(1)> "ab
...(1)> c"
"ab\nc"
```

In the example above, the shell will be expecting more input until it finds the closing quote. Sometimes it is not obvious which character the shell is expecting, and the user may find themselves trapped in the state of incomplete expression with no ability to terminate it other than by exiting the shell.

For such cases, there is a special break-trigger (`#iex:break`) that when encountered on a line by itself will force the shell to break out of any pending expression and return to its normal state:

```
iex(1)> ["ab
...(1)> c"
...(1)> 
...(1)> ]
...(1)> #iex:break
** (TokenMissingError) iex:1: incomplete expression
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>after_spawn/0</code></td>
<td>Returns registered <code>after_spawn</code> callbacks</td>
</tr>
<tr>
<td><code>after_spawn/1</code></td>
<td>Registers a function to be invoked after the IEx process is spawned</td>
</tr>
<tr>
<td><code>color/2</code></td>
<td>Returns string escaped using the specified color. ANSI escapes in string are not processed in any way</td>
</tr>
<tr>
<td><code>configuration/0</code></td>
<td>Returns IEx configuration</td>
</tr>
<tr>
<td><code>configure/1</code></td>
<td>Configures IEx</td>
</tr>
<tr>
<td><code>pry/1</code></td>
<td>Pries into the process environment</td>
</tr>
<tr>
<td><code>started?/0</code></td>
<td>Returns true if IEx was properly started</td>
</tr>
<tr>
<td><code>width/0</code></td>
<td>Get the width to be used on helpers with a maximum (and default) of 80 chars</td>
</tr>
</tbody>
</table>

Functions

**after_spawn/0** *(function)*

Returns registered `after_spawn` callbacks.
**after_spawn(fun)(function)**
Registers a function to be invoked after the IEx process is spawned.

**color(color_name, string)(function)**
Returns `string` escaped using the specified color. ANSI escapes in `string` are not processed in any way.

**configuration() (function)**
Returns IEx configuration.

**configure(options)(function)**
Configures IEx.

The supported options are: `:colors`, `:inspect`, `:default_prompt`, `:alive_prompt` and `:history_size`.

**Colors**
A keyword list that encapsulates all color settings used by the shell. See documentation for the `IO.ANSI` module for the list of supported colors and attributes.

The value is a keyword list. List of supported keys:

• `:enabled` - boolean value that allows for switching the coloring on and off
• `:eval_result` - color for an expression’s resulting value
• `:eval_info` - various informational messages
• `:eval_error` - error messages
• `:stack_app` - the app in stack traces
• `:stack_info` - the remaining info in stack traces
• `:ls_directory` - for directory entries (ls helper)
• `:ls_device` - device entries (ls helper)

When printing documentation, IEx will convert the markdown documentation to ANSI as well. Those can be configured via:

• `:doc_code` — the attributes for code blocks (cyan, bright)
• `:doc_inline_code` - inline code (cyan)
• `:doc_headings` - h1 and h2 (yellow, bright)
• `:doc_title` — the overall heading for the output (reverse,yellow,bright)
• `:doc_bold` - (bright)
• `:doc_underline` - (underline)

**Inspect**
A keyword list containing inspect options used by the shell when printing results of expression evaluation. Default to pretty formatting with a limit of 50 entries.

See `Inspect.Opts` for the full list of options.

**History size**
Number of expressions and their results to keep in the history. The value is an integer. When it is negative, the history is unlimited.

**Prompt**
This is an option determining the prompt displayed to the user when awaiting input.
The value is a keyword list. Two prompt types:

- :default_prompt - used when Node.alive? returns false
- :alive_prompt - used when Node.alive? returns true

The part of the listed in the following of the prompt string is replaced.

- %counter - the index of the history
- %prefix - a prefix given by IEx.Server
- %node - the name of the local node

started?() (function)
Returns true if IEx was properly started.

width() (function)
Get the width to be used on helpers with a maximum (and default) of 80 chars.

Macros

pry(timeout \ \ 1000)(macro)
Pries into the process environment.

This is useful for debugging a particular chunk of code and inspect the state of a particular process. The process is temporarily changed to trap exits (i.e. the process flag :trap_exit is set to true) and has the group_leader changed to support ANSI escape codes. Those values are reverted by calling respawn, which starts a new IEx shell, freeing up the pried one.

When a process is pried, all code runs inside IEx and, as such, it is evaluated and cannot access private functions of the module being pried. Module functions still need to be accessed via Mod.fun(args).

Status: This feature is experimental.

Examples

Let’s suppose you want to investigate what is happening with some particular function. By invoking IEx.pry from the function, IEx will allow you to access its binding (variables), verify its lexical information and access the process information. Let’s see an example:

```elixir
import Enum, only: [map: 2]

def add(a, b) do
  c = a + b
  IEx.pry
end
```

When invoking `add(1, 2)`, you will receive a message in your shell to pry the given environment. By allowing it, the shell will be reset and you gain access to all variables and the lexical scope from above:

```elixir
iex(1)> map([a,b,c], &IO.inspect(&1))
1
2
3
```

Keep in mind that IEx.pry runs in the caller process, blocking the caller during the evaluation cycle. The caller process can be freed by calling respawn, which starts a new IEx evaluation cycle, letting this one go:

```elixir
iex(2)> respawn
true
```

Interactive Elixir - press Ctrl+C to exit (type h() ENTER for help)
Setting variables or importing modules in IEx does not affect the caller the environment (hence it is called pry).

**IEx.Helpers**

**Overview** Welcome to Interactive Elixir. You are currently seeing the documentation for the module **IEx.Helpers** which provides many helpers to make Elixir’s shell more joyful to work with.

This message was triggered by invoking the helper `h()`, usually referred to as `h/0` (since it expects 0 arguments).

There are many other helpers available:

- `c/2` — compiles a file at the given path
- `cd/1` — changes the current directory
- `clear/0` — clears the screen
- `flush/0` — flushes all messages sent to the shell
- `h/0` — prints this help message
- `h/1` — prints help for the given module, function or macro
- `1s/0` — lists the contents of the current directory
- `ls/1` — lists the contents of the specified directory
- `ls/0` — lists the contents of the current directory
- `pwd/0` — prints the current working directory
- `r/1` — recompiles and reloads the given module’s source file
- `respawn/0` — respawn the current shell
- `s/1` — prints spec information
- `t/1` — prints type information
- `v/0` — prints the history of commands evaluated in the session
- `v/1` — retrieves the nth value from the history
- `import_file/1` — evaluates the given file in the shell’s context

Help for functions in this module can be consulted directly from the command line, as an example, try:

```erlang
h(c/2)
```

You can also retrieve the documentation for any module or function. Try these:

```erlang
h(Enum)
h(Enum.reverse/1)
```

To learn more about IEx as a whole, just type `h(IEx)`.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c/2</code></td>
<td>Expects a list of files to compile and a path to write their object code to. It returns the name of the compiled modules.</td>
</tr>
<tr>
<td><code>cd/1</code></td>
<td>Changes the current working directory to the given path.</td>
</tr>
<tr>
<td><code>clear/0</code></td>
<td>Clear the console screen.</td>
</tr>
<tr>
<td><code>flush/0</code></td>
<td>Flushes all messages sent to the shell and prints them out.</td>
</tr>
<tr>
<td><code>h/0</code></td>
<td>Prints the documentation for <code>IEx.Helpers</code>.</td>
</tr>
<tr>
<td><code>h/1</code></td>
<td>Prints the documentation for the given module or for the given function/arity pair.</td>
</tr>
<tr>
<td><code>import_file</code></td>
<td>Evaluates the contents of the file at <code>path</code> as if it were directly typed into the shell. <code>path</code> has to be a literal binary.</td>
</tr>
<tr>
<td><code>l/1</code></td>
<td>Load the given module’s beam code (and ensures any previous old version was properly purged before).</td>
</tr>
<tr>
<td><code>ls/1</code></td>
<td>Produces a simple list of a directory’s contents. If <code>path</code> points to a file, prints its full path.</td>
</tr>
<tr>
<td><code>pwd/0</code></td>
<td>Prints the current working directory.</td>
</tr>
<tr>
<td><code>r/1</code></td>
<td>Recompiles and reloads the specified module’s source file.</td>
</tr>
<tr>
<td><code>respawn/0</code></td>
<td>Respawns the current shell by starting a new process and a new scope. Returns true if it worked.</td>
</tr>
<tr>
<td><code>s/1</code></td>
<td>Similar to <code>t/1</code>, only for specs.</td>
</tr>
<tr>
<td><code>t/1</code></td>
<td>When given a module, prints specifications (or simply specs) for all the types defined in it.</td>
</tr>
<tr>
<td><code>v/0</code></td>
<td>Prints the history of expressions evaluated during the session along with their results.</td>
</tr>
<tr>
<td><code>v/1</code></td>
<td>Retrieves the <code>n</code>th expression’s value from the history.</td>
</tr>
</tbody>
</table>

**Summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `c(files, path \ " \\
                 .")` | Expects a list of files to compile and a path to write their object code to. It returns the name of the compiled modules.                   |
|               | When compiling one file, there is no need to wrap it in a list.                                                                             |
|               | **Examples**                                                                                                                              |
|               | `c ["foo.ex", "bar.ex"], "ebin"                                                        => [Foo,Bar]                                                                        |
|               | `c "baz.ex"                                                                  => [Baz]                                                                                                     |
| `cd(directory)` | Changes the current working directory to the given path.                                                                                   |
| `clear()`     | Clear the console screen.                                                                                                                  |
| `flush()`     | Flushes all messages sent to the shell and prints them out.                                                                                 |
| `h()`         | Prints the documentation for `IEx.Helpers`.                                                                                                 |
| `l(module)`   | Load the given module’s beam code (and ensures any previous old version was properly purged before).                                          |
| `ls(path \ " \\
                 ".")` | Produces a simple list of a directory’s contents. If `path` points to a file, prints its full path.                                          |
| `pwd()`       | Prints the current working directory.                                                                                                        |
| `r(module)`   | Recompiles and reloads the specified module’s source file.                                                                                  |
|               | Please note that all the modules defined in the same file as `module` are recompiled and reloaded.                                         |

Please note that all the modules defined in the same file as `module` are recompiled and reloaded.
**respawn() (function)**
Respawns the current shell by starting a new process and a new scope. Returns true if it worked.

**v() (function)**
Prints the history of expressions evaluated during the session along with their results.

**v(n) (function)**
Retrieves the nth expression’s value from the history.
Use negative values to lookup expression values relative to the current one. For instance, v(-1) returns the result of the last evaluated expression.

**Macros**

**h(other) (macro)**
Prints the documentation for the given module or for the given function/arity pair.

**Examples**

h(Enum)
#=> Prints documentation for Enum

It also accepts functions in the format `fun/arity` and `module.fun/arity`, for example:

h receive/1
h Enum.all?/2
h Enum.all?

**import_file(path) (macro)**
Evaluates the contents of the file at `path` as if it were directly typed into the shell. `path` has to be a literal binary.
A leading ~ in `path` is automatically expanded.

**Examples**

# ~/file.exs
value = 13

# in the shell
iex(1)> import_file "~/file.exs"
13
iex(2)> value
13

**s(other) (macro)**
Similar to `t/1`, only for specs.
When given a module, prints the list of all specs defined in the module.
When given a particular spec name (with optional arity), prints its spec.

**Examples**

s(Enum)
s(Enum.all?)
s(Enum.all?/2)
s(is_atom)
s(is_atom/1)

**t(module) (macro)**
When given a module, prints specifications (or simply specs) for all the types defined in it.
When given a particular type name, prints its spec.
Examples

t(Enum)
t(Enum.t/0)
t(Enum.t)

IEx.History.Server

Overview

3.3 Topical Reference

3.3.1 Core

• Access
• Atom
• Base
• Bitwise
• Code
• Exception
• Float
• Integer
• Kernel
  – Kernel.ParallelCompiler
  – Kernel.ParallelRequire
  – Kernel.SpecialForms
  – Kernel.Typespec
• Inspect.Algebra
• Inspect.Opts
• Macro
  – Macro.Env
• Module
• Protocol
• URI

3.3.2 Collections

• Dict
• Enum
• HashDict
3.3.3 String manipulation

- Regex
- String
- StringIO

3.3.4 Concurrency & distribution

- Node
- Process

3.3.5 OTP / Xgen

- Agent
- Application
- Behaviour
- GenEvent
- GenServer
- Supervisor
  - Supervisor.Spec
- Task
  - Task.Supervisor

3.3.6 IO, files, and OS interop

- File
  - File.Stat
  - File.Stream
- IO
• OptionParser
• Path
• Port
• System
4.1 Readme

For more about Elixir, installation and documentation, check Elixir’s website.

4.1.1 Usage

If you want to contribute to Elixir or run it from source, clone this repository to your machine, compile and test it:

```
$ git clone https://github.com/elixir-lang/elixir.git
$ cd elixir
$ make clean test
```

If Elixir fails to build (specifically when pulling in a new version via git), be sure to remove any previous build artifacts by running `make clean`, then `make test`.

If tests pass, you are ready to move on to the Getting Started guide or to try Interactive Elixir by running: `bin/iex` in your terminal.

However, if tests fail, it is likely you have an outdated Erlang version (Elixir requires Erlang 17.0 or later). You can check your Erlang version by calling `erl` in the command line. You will see some information as follows:

```
Erlang/OTP 17 [erts-6.0] [source-07b8f44] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]
```

If you have the correct version and tests still fail, feel free to open an issue.

4.1.2 Building documentation

Building the documentation requires `ex_doc` to be installed and built in the same containing folder as elixir.

```
# After cloning and compiling Elixir
$ git clone git://github.com/elixir-lang/ex_doc.git
$ cd ex_doc && ../elixir/bin/mix compile
$ cd ../elixir && make docs
```

4.1.3 Contributing

We appreciate any contribution to Elixir, so check out our CONTRIBUTING.md guide for more information. We usually keep a list of features and bugs in the issue tracker.
4.1.4 Important links

- #elixir-lang on freenode IRC
- Website
- Issue tracker
- elixir-talk Mailing list (questions)
- elixir-core Mailing list (development)

4.1.5 License

“Elixir” and the Elixir logo are copyright (c) 2012 Plataformatec.
Elixir source code is released under Apache 2 License with some parts under Erlang’s license (EPL).
Check LEGAL and LICENSE files for more information.

4.2 Contributing to Elixir

Please take a moment to review this document in order to make the contribution process easy and effective for everyone involved!

4.2.1 Using the issue tracker

Use the issues tracker for:
- bug reports
- submitting pull requests

Please **do not** use the issues tracker for personal support requests nor feature requests. Support requests should be send to:
- the elixir-talk mailing list
- Stack Overflow
- #elixir-lang

Feature requests can be discussed on the elixir-core mailing list.

We do our best to keep the issues tracker tidy and organized, making it useful for everyone. For example, we classify open issues per application and perceived difficulty of the issue, making it easier for developers to contribute to Elixir.

4.2.2 Bug reports

A bug is a *demonstrable problem* that is caused by the code in the repository. Good bug reports are extremely helpful - thank you!

Guidelines for bug reports:

1. Use the GitHub issue search — check if the issue has already been reported.
2. Check if the issue has been fixed — try to reproduce it using the `master` branch in the repository.
3. Isolate and report the problem — ideally create a reduced test case.
Please try to be as detailed as possible in your report. Include information about your Operating System, your Erlang and Elixir versions. Please provide steps to reproduce the issue as well as the outcome you were expecting! All these details will help developers to fix any potential bugs.

Example:

Short and descriptive example bug report title

A summary of the issue and the environment in which it occurs. If suitable, include the steps required to reproduce the bug.

1. This is the first step
2. This is the second step
3. Further steps, etc.

[url] - a link to the reduced test case (e.g. a GitHub Gist)

Any other information you want to share that is relevant to the issue being reported. This might include the lines of code that you have identified as causing the bug, and potential solutions (and your opinions on their merits).

4.2.3 Feature requests

Feature requests are welcome and should be discussed on the elixir-core mailing list. But take a moment to find out whether your idea fits with the scope and aims of the project. It’s up to you to make a strong case to convince the community of the merits of this feature. Please provide as much detail and context as possible.

4.2.4 Contributing

We incentivize everyone to contribute to Elixir and help us tackle existing issues! To do so, there are a few things you need to know about the code. First, Elixir code is divided in applications inside the lib folder:

- elixir - Contains Elixir’s kernel and stdlib
- eex - Template engine that allows you to embed Elixir
- ex_unit - Simple test framework that ships with Elixir
- iex — IEx, Elixir’s interactive shell
- mix — Elixir’s build tool

You can run all tests in the root directory with `make test` and you can also run tests for a specific framework `make test_{NAME}`, for example, `make test_ex_unit`.

In case you are changing a single file, you can compile and run tests only for that particular file for fast development cycles. For example, if you are changing the String module, you can compile it and run its tests as:

```
$ bin/elixirc lib/elixir/lib/string.ex -o lib/elixir/ebin
$ bin/elixir lib/elixir/test/elixir/string_test.exs
```

After your changes are done, please remember to run the full suite with `make test`.

From time to time, your tests may fail in an existing Elixir checkout and may require a clean start by running `make clean compile`. You can always check the official build status on Travis-CI.

With tests running and passing, you are ready to contribute to Elixir and send your pull requests.
Building on Windows

There are a few extra steps you’ll need to take for contributing from Windows. Basically, once you have Erlang 17, Git, and MSYS from MinGW on your system, you’re all set. Specifically, here’s what you need to do to get up and running:

1. Install Git, Erlang, and the MinGW Installation Manager.
2. Use the MinGW Installation Manager to install the msys-bash, msys-make, and msys-grep packages.
3. Add `;C:\Program Files (x86)\Git\bin;C:\Program Files\erl6.0\bin;C:\MinGW\msys\1.0\bin` to your “Path” environment variable. (This is under Control Panel > System and Security > System > Advanced system settings > Environment Variables > System variables)

You can now work in the Command Prompt similar to how you would on other OS’es, except for some things (like creating symlinks) you’ll need to run the Command Prompt as an Administrator.

4.2.5 Contributing Documentation

Code documentation (@doc, @moduledoc, @typedoc) has a special convention: the first paragraph is considered to be a short summary.

For functions, macros and callbacks say what it will do. For example write something like:

```
@doc ""
Returns only those elements for which 'fun' is true.
"
```

```
def filter(collection, fun) ...
```

For modules, protocols and types say what it is. For example write something like:

```
defmodule File.Stat do
  @moduledoc ""
  Information about a file.
"
```

```
defstruct [...] end
```

Keep in mind that the first paragraph might show up in a summary somewhere, long texts in the first paragraph create very ugly summaries. As a rule of thumb anything longer than 80 characters is too long.

Try to keep unnecessary details out of the first paragraph, it’s only there to give a user a quick idea of what the documented “thing” does/is. The rest of the documentation string can contain the details, for example when a value and when nil is returned.

If possible include examples, preferably in a form that works with doctests. For example:

```
@doc ""
Return only those elements for which 'fun' is true.
"

## Examples

```
iex> Enum.filter([1, 2, 3], fn(x) -> rem(x, 2) == 0 end)
[2]
```
This makes it easy to test the examples so that they don’t go stale and examples are often a great help in explaining what a function does.

4.2.6 Pull requests

Good pull requests - patches, improvements, new features - are a fantastic help. They should remain focused in scope and avoid containing unrelated commits.

IMPORTANT: By submitting a patch, you agree that your work will be licensed under the license used by the project.

If you have any large pull request in mind (e.g. implementing features, refactoring code, etc), please ask first otherwise you risk spending a lot of time working on something that the project’s developers might not want to merge into the project.

Please adhere to the coding conventions in the project (indentation, accurate comments, etc.) and don’t forget to add your own tests and documentation. When working with git, we recommend the following process in order to craft an excellent pull request:

1. Fork the project, clone your fork, and configure the remotes:

bash
# Clone your fork of the repo into the current directory
git clone https://github.com/<your-username>/elixir
# Navigate to the newly cloned directory
cd elixir
# Assign the original repo to a remote called "upstream"
git remote add upstream https://github.com/elixir-lang/elixir

2. If you cloned a while ago, get the latest changes from upstream:

bash
git checkout master
git pull upstream master

3. Create a new topic branch (off of master) to contain your feature, change, or fix.

bash
git checkout -b <topic-branch-name>

IMPORTANT: Making changes in master is discouraged. You should always keep your local master in sync with upstream master and make your changes in topic branches.

bash
git checkout -b <topic-branch-name>

4. Commit your changes in logical chunks. Keep your commit messages organized, with a short description in the first line and more detailed information on the following lines. Feel free to use Git’s interactive rebase feature to tidy up your commits before making them public.

5. Make sure all the tests are still passing.

bash
make test

This command will compile the code in your branch and use that version of Elixir to run the tests. This is needed to ensure your changes can pass all the tests.

6. Push your topic branch up to your fork:

bash
git push origin <topic-branch-name>

7. Open a Pull Request with a clear title and description.

8. If you haven’t updated your pull request for a while, you should consider rebasing on master and resolving any conflicts.

IMPORTANT: Never ever merge upstream master into your branches. You should always git rebase on master to bring your changes up to date when necessary.
bash git checkout master git pull upstream master git checkout
<your-topic-branch> git rebase master

We have saved some excellent pull requests we have received in the past in case you are looking for some examples:

- https://github.com/elixir-lang/elixir/pull/992
- https://github.com/elixir-lang/elixir/pull/1041
- https://github.com/elixir-lang/elixir/pull/1058
- https://github.com/elixir-lang/elixir/pull/1059

Thank you for your contributions!

4.3 Release process

This document simply outlines the release process:

1. Remove -dev extension from VERSION
2. Ensure CHANGELOG is updated and timestamp
3. Commit changes above with title “Release vVERSION” and generate new tag
4. Run make clean test to ensure all tests pass from scratch and the CI is green
5. Push master and tags
6. Release new docs with make release_docs, move docs to docs/stable
7. Release new zip with make release_zip, push new zip to GitHub Releases
8. Merge master into stable branch and push it
9. After release, bump versions, add -dev back and commit
10. make release_docs once again and push it to elixir-lang.github.com

4.3.1 Places where version is mentioned

- VERSION
- CHANGELOG
- src/elixir.app.src

4.4 License

Copyright 2012-2013 Plataformatec.

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.
4.5 Changelog

4.5.1 v0.14.0-dev

- Enhancements
  - [Kernel] Store documentation in the abstract code to avoid loading them when the module is loaded
  - [Protocol] Add `Protocol.consolidate/2`, `Protocol.consolidated?/1` and a mix `compile.protocols` task for protocol consolidation
  - [String] Add `String.chunk/2`
  - [Struct] Add support for `@derive before defstruct/2` definitions
- Bug fixes
  - [Kernel] Ensure `Mix._build` structure works on Windows when copying projects
- Soft deprecations (no warnings emitted)
- Deprecations
  - [Access] `Access.access/2` is deprecated in favor of `Access.get/2`
  - [Dict] `Dict.Behaviour` is deprecated in favor of `Dict`
  - [Kernel] `defexception/3` is deprecated in favor of `defexception/1`
  - [Kernel] `raise/3` is deprecated in favor of `reraise/2`
  - [Kernel] `set_elem/3` is deprecated in favor of `put_elem/3`
- Backwards incompatible changes
  - [Access] `Kernel.access/2` no longer exists and the `Access` protocol now uses `get/2` instead of `access/2`
  - [Kernel] Retrieving docs as `module.__info__(:docs)` is deprecated, instead `Code.get_docs/2` must be used
  - [Mix] `mix new` no longer generates a supervision tree by default, please pass `--sup` instead

4.5.2 v0.13.3 (2014-05-24)

- Enhancements
  - [OptionParser] Add `:strict` option that only parses known switches
  - [OptionParser] Add `next/2` useful for manual parsing of options
  - [Macro] Add `Macro.prewalk/2/3` and `Macro.postwalk/2/3`
  - [Kernel] `GenEvent`, `GenServer`, `Supervisor`, `Agent` and `Task` modules added
  - [Kernel] Make deprecations compiler warnings to avoid the same deprecation being printed multiple times
- Bug fixes
  - [Enum] Fix `Enum.join/2` and `Enum.map_join/3` for empty binaries at the beginning of the collection
• [ExUnit] Ensure the formatter doesn’t error when printing :EXITs

• [Kernel] Rename ELIXIR_ERL_OPTS to ELIXIR_ERL_OPTIONS for consistency with ERL_COMPILER_OPTIONS

• [OptionParser] Parse -- as a plain argument

• [OptionParser] -- is always removed from argument list on parse/2 and when it is the leading entry on parse_head/2

• [Regex] Properly escape regex (previously regex controls were double escaped)

• Soft deprecations (no warnings emitted)

• [Dict] Dict.Behaviour is deprecated in favor of Dict


• [Kernel] defexception/3 is deprecated in favor of defexception/1

• [Kernel] raise/3 is deprecated in favor of reraise/2

• [Kernel] set_elem/3 is deprecated in favor of put_elem/3

• Soft deprecations for conversions (no warnings emitted)

• [Kernel] atom_to_binary/1 and atom_to_list/1 are deprecated in favor of Atom.to_string/1 and Atom.to_char_list/1

• [Kernel] bitstring_to_list/1 and list_to_bitstring/1 are deprecated in favor of the :erlang ones

• [Kernel] binary_to_atom/1, binary_to_existing_atom/1, binary_to_float/1, binary_to_integer/1 and binary_to_integer/2 are deprecated in favor of conversion functions in String

• [Kernel] float_to_binary/* and float_to_list/* are deprecated in favor of Float.to_string/* and Float.to_char_list/*

• [Kernel] integer_to_binary/* and integer_to_list/* are deprecated in favor of Integer.to_string/* and Integer.to_char_list/*

• [Kernel] iodata_to_binary/1 and iodata_length/1 are deprecated IO.iodata_to_binary/1 and IO.iodata_length/1

• [Kernel] list_to_atom/1, list_to_existing_atom/1, list_to_float/1, list_to_integer/1, list_to_integer/2 and list_to_tuple/1 are deprecated in favor of conversion functions in List

• [Kernel] tuple_to_list/1 is deprecated in favor of Tuple.to_list/1

• [List] List.from_char_data/1 and List.from_char_data!/1 deprecated in favor of String.to_char_list/1

• [String] String.from_char_data/1 and String.from_char_data!/1 deprecated in favor of List.to_string/1

• Deprecations

• [Kernel] is_exception/1, is_record/1 and is_record/2 are deprecated in favor of Exception.exception?/1, Record.record?/1 and Record.record?/2

• [Kernel] defrecord/3 is deprecated in favor of structs
• [Kernel] :hygiene in quote is deprecated
• [Mix] Mix.project/0 is deprecated in favor of Mix.Project.config/0
• Backwards incompatible changes
• [Exception] Exceptions now generate structs instead of records
• [OptionParser] Errors on parsing returns the switch and value as binaries (unparsed)
• [String] String.to_char_list/1 (previously deprecated) no longer returns a tuple but the char list only and raises in case of failure

4.5.3 v0.13.2 (2014-05-11)

• Enhancements
• [Application] Add an Application module with common functions to work with OTP applications
• [Exception] Add Exception.message/1, Exception.format_banner/1, Exception.format_exit/1 and Exception.format/1
• [File] Add File.ln_s/1
• [Mix] mix deps.clean now works across environments
• [Mix] Support line numbers in mix test, e.g. test/some/file_test.exs:12
• [Mix] Use @file attributes to detect dependencies in between .ex and external files. This means changing an .eex file will no longer recompile the whole project only the files that depend directly on it
• [Mix] Support application configurations in config/config.exs which can be customized by specifying your own :config_path
• [Mix] Support user-wide configuration with ~/.mix/config.exs
• [Mix] mix help now uses ANSI formatting to print guides
• [Regex] Support functions in Regex.replace/4
• [String] Support :parts in String.split/3
• Bug fixes
• [Code] Ensure we don’t lose the caller stacktrace on code evaluation
• [IEx] Exit signals now exits the IEx evaluator and a new one is spawned on its place
• [IEx] Ensure we don’t prune too much stacktrace when reporting failures
• [IEx] Fix an issue where iex.bat on Windows was not passing the proper parameters forward
• [Kernel] Ensure modules defined on root respect defined aliases
• [Kernel] Do not wrap single lists in :__block__
• [Kernel] Ensure emitted beam code works nicely with dialyzer
• [Kernel] Do not allow a module named Elixir to be defined
• [Kernel] Create remote funs even if mod is a variable in capture &mod.fun/arity
• [Kernel] Improve compiler message when duplicated modules are detected
• [Mix] Generate `.gitignore` for `--umbrella` projects
• [Mix] Verify if a git dependency in `deps` has a proper git checkout and clean it automatically when it doesn’t
• [Mix] Ensure `mix test` works with `IEx.pry/0`
• [System] Convert remaining functions in `System` to rely on char data
• Soft deprecations (no warnings emitted)
  • [Exception] `exception.message` is deprecated in favor of `Exception.message/1` for retrieving exception messages
  • [Kernel] `is_exception/1`, `is_record/1` and `is_record/2` are deprecated in favor of `Exception.exception?/1`, `Record.record?/1` and `Record.record?/2`
• [Mix] `Mix.project/0` is deprecated in favor of `Mix.Project.config/0`
• Deprecations
• [IEx] `IEx.Options` is deprecated in favor of `IEx.configure/1` and `IEx.configuration/0`
• [Kernel] `lc` and `bc` comprehensions are deprecated in favor of `for`
• [Macro] `Macro.safe_terms/1` is deprecated
• [Process] `Process.delete/0` is deprecated
• [Regex] Deprecate `:global` option in `Regex.split/3` in favor of `parts: :infinity`
• [String] Deprecate `:global` option in `String.split/3` in favor of `parts: :infinity`
• Backwards incompatible changes
• [ExUnit] `ExUnit.Test` and `ExUnit.TestCase` has been converted to structs
• [ExUnit] The test and callback context has been converted to maps
• [Kernel] `File.Stat`, `HashDict`, `HashSet`, `Inspect.Opts`, `Macro.Env`, `Range`, `Regex` and `Version.Requirement` have been converted to structs. This means `is_record/2` checks will no longer work, instead, you can pattern match on them using `%Range{}` and similar
• [URI] The `URI.Info` record has now become the `URI` struct
• [Version] The `Version.Schema` record has now become the `Version` struct

### 4.5.4 v0.13.1 (2014-04-27)

• Enhancements
  • [Mix] Support `MIX_EXS` as configuration for running the current `mix.exs` file
  • [Mix] Support Hex out of the box. This means users do not need to install Hex directly, instead, Mix will prompt whenever there is a need to have Hex installed
• Bug fixes
  • [ExUnit] Ensure doctest failures are properly reported
  • [Kernel] Fix a bug where comprehensions arguments were not properly take into account in the variable scope
  • [Mix] Fix issue on rebar install when the endpoint was redirecting to a relative uri
• Soft deprecations (no warnings emitted)
  • [Kernel] iolist_size and iolist_to_binary are deprecated in favor of iodata_length and iodata_to_binary
  • [String] String.to_char_list/1 is deprecated in favor of List.from_char_data/1
  • [String] String.from_char_list/1 is deprecated in favor of String.from_char_data/1

• Deprecations
  • [Mix] :env key in project configuration is deprecated
  • [Regex] Regex.groups/1 is deprecated in favor of Regex.names/1

• Backwards incompatible changes
  • [Macro] Macro.unpipe/1 now returns tuples and Macro.pipe/2 was removed in favor of Macro.pipe/3 which explicitly expects the second element of the tuple returned by the new Macro.unpipe/1
  • [Path] The functions in Path now only emit strings as result, regardless if the input was a char list or a string
  • [Path] Atoms are no longer supported in Path functions
  • [Regex] Regexes are no longer unicode by default. Instead, they must be explicitly marked with the u option

4.5.5 v0.13.0 (2014-04-20)

• Enhancements
  • [Base] Add Base module which does conversions to bases 16, 32, hex32, 64 and url64
  • [Code] Add Code.eval_file/2
  • [Collectable] Add the Collectable protocol that empowers Enum.into/2 and Stream.into/2 and the :into option in comprehensions
  • [Collectable] Implement Collectable for lists, dicts, bitstrings, functions and provide both File.Stream and IO.Stream
  • [EEx] Add handle_body/1 callback to EEx.Engine
  • [Enum] Add Enum.group_by/2, Enum.into/2, Enum.into/3, Enum.traverse/2 and Enum.sum/2
  • [ExUnit] Randomize cases and tests suite runs, allow seed configuration and the --seed flag via mix test
  • [ExUnit] Support --only for filtering when running tests with mix test
  • [ExUnit] Raise an error if another capture_io process already captured the device
  • [ExUnit] Improve formatter to show source code and rely on lhs and rhs (instead of expected and actual)
  • [IEx] Allow prompt configuration with the :prompt option
  • [IEx] Use werl on Windows
  • [Kernel] Support ERL_PATH in bin/elixir
  • [Kernel] Support interpolation in keyword syntax
  • [Map] Add a Map module and support 17.0 maps and structs
  • [Mix] Add dependency option :only to specify the dependency environment. mix deps.get and mix deps.update works accross all environment unless --only is specified
• [Mix] Add `Mix.Shell.prompt/1`
• [Mix] Ensure the project is compiled in case Mix’ CLI cannot find a task
• [Node] Add `Node.ping/1`
• [Process] Include `Process.send/3` and support the `--gen-debug` option
• [Regex] Regexes no longer need the “g” option when there is a need to use named captures
• [Stream] Add `Stream.into/2` and `Stream.into/3`
• [StringIO] Add a `StringIO` module that allows a String to be used as IO device
• [System] Add `System.delete_env/1` to remove a variable from the environment
• Bug fixes
  • [CLI] Ensure `--app` is handled as an atom before processing
  • [ExUnit] Ensure `ExUnit.Assertions` does not emit compiler warnings for `assert_receive`
  • [Kernel] Ensure the same pid is not queued twice in the parallel compiler
  • [Macro] `Macro.to_string/2` considers proper precedence when translating `(foo > bar)` into a string
  • [Mix] Automatically recompile on outdated Elixir version and show proper error messages
  • [Mix] Ensure generated `.app` file includes core dependencies
  • [Mix] Allow a dependency with no SCM to be overridden
  • [Mix] Allow queries in `mix local.install` URL
  • [OptionParser] Do not recognize undefined aliases as switches
  • Soft deprecations (no warnings emitted)
    • [Kernel] `lc` and `bc` comprehensions are deprecated in favor of `for`
    • [ListDict] `ListDict` is deprecated in favor of `Map`
    • [Record] `defrecord/2`, `defrecordp/3`, `is_record/1` and `is_record/2` macros in Kernel are deprecated. Instead, use the new macros and API defined in the `Record` module
• Deprecations
  • [Dict] `Dict.empty/1, Dict.new/1` and `Dict.new/2` are deprecated
  • [Exception] `Exception.normalize/1` is deprecated in favor of `Exception.normalize/2`
• Backwards incompatible changes
  • [ExUnit] Formatters are now required to be a GenEvent and `ExUnit.run/2` returns a map with results

### 4.5.6 v0.12.5 (2014-03-09)

• Bug fixes
  • [Kernel] Ensure `try` does not generate an after clause. Generating an after clause forbade clauses in the `else` part from being tail recursive. This should improve performance and memory consumption of `Stream` functions
  • [Mix] Automatically recompile on outdated Elixir version and show proper error messages
• Deprecations
  • [File] `File.stream_to!/3` is deprecated
- [GenFSM] GenFSM is deprecated
- [Kernel] % for sigils is deprecated in favor of ~
- [Kernel] is_range/1 and is_regex/1 are deprecated in favor of Range.range?/1 and Regex.regex?/1
- [Stream] Stream.after/1 is deprecated
- [URI] URI.decode_query/1 is deprecated in favor of URI.decode_query/2 with explicit dict argument
- [URI] Passing lists as key or values in URI.encode_query/1 is deprecated
- Backwards incompatible changes
- [Mix] Remove MIX_GIT_FORCE_HTTPS as Git itself already provides mechanisms for doing so

4.5.7 v0.12.4 (2014-02-12)

- Enhancements
- [Mix] mix deps.get and mix deps.update no longer compile dependencies afterwards. Instead, they mark the dependencies which are going to be automatically compiled next time deps.check is invoked (which is done automatically by most mix tasks). This means users should have a better workflow when migrating between environments
- Deprecations
- [Kernel] // for default arguments is deprecated in favor of \ 
- [Kernel] Using % for sigils is deprecated in favor of ~. This is a soft deprecation, no warnings will be emitted for it in this release
- [Kernel] Using ^ inside function clause heads is deprecated, please use a guard instead
- Backwards incompatible changes
- [ExUnit] CaptureIO returns an empty string instead of nil when there is no capture
- [Version] The Version module now only works with SemVer. The functions Version.parse/1 and Version.parse_requirement/1 now return {:ok, res} | :error for the cases you want to handle non SemVer cases manually. All other functions will trigger errors on non semantics versions

4.5.8 v0.12.3 (2014-02-02)

- Enhancements
- [Kernel] Warnings now are explicitly tagged with “warning:” in messages
- [Kernel] Explicit functions inlined by the compiler, including operators. This means that Kernel.+/2 will now expand to :erlang.+/2 and so on
- [Mix] Do not fail if a Mix dependency relies on an outdated Elixir version
- [Version] Add Version.compare/2
- Bug fixes
- [Atom] Inspect :... and :foo@bar without quoting
- [Keyword] The list [1, 2, three: :four] now correctly expands to [1, 2, {:three, :four}]

4.5. Changelog 479
- [Kernel] Ensure undefined `@attributes` shows proper stacktrace in warnings
- [Kernel] Guarantee nullary funs/macros are allowed in guards
- [Process] Ensure monitoring functions are inlined by the compiler

### Deprecations
- [IEx] The helper `m/0` has been deprecated. The goal is to group all runtime statistic related helpers into a single module
- [Kernel] `binary_to_term/1`, `binary_to_term/2`, `term_to_binary/1` and `term_to_binary/2` are deprecated in favor of their counterparts in the `:erlang` module
- [Kernel] `//` for default arguments is deprecated in favor of `\`. This is a soft deprecation, no warnings will be emitted for it in this release
- [Kernel] Deprecated `@behavior` in favor of `@behaviour`
- [Record] `to_keywords`, `getter` and `list getter` functionalities in `defrecordp` are deprecated
- [Record] `Record.import/2` is deprecated

### Backwards incompatible changes
- [Dict] Implementations of `equal?/2` and `merge/2` in `HashDict` and `ListDict` are no longer polymorphic. To get polymorphism, use the functions in `Dict` instead
- [File] `File.cp/3` and `File.cp_r/3` no longer carry Unix semantics where the function behaves differently if the destination is an existing previous directory or not. It now always copies source to destination, doing it recursively in the latter
- [IEx] `IEx` now loads the `.iex.exs` file instead of `.iex`
- [Kernel] Remove `**` from the list of allowed operators
- [Kernel] Limit sigils delimiters to one of the following: `<>`, `{}`, `[]`, `()`, `||`, `//`, `"` and `'`
- [Range] `Range` is no longer a record, instead use `first .. last` if you need pattern matching
- [Set] Implementations of `difference/2`, `disjoint?/2`, `equal?/2`, `intersection/2`, `subset?/2` and `union/2` in `HashSet` are no longer polymorphic. To get polymorphism, use the functions in `Set` instead

### 4.5.9 v0.12.2 (2014-01-15)

- Enhancements
  - [EEx] Allow `EEx.AssignsEngine` to accept any `Dict`
  - [Enum] Add `Enum.flat_map_reduce/3`
  - [ExUnit] Support `@moduletag` in `ExUnit` cases
- [Kernel] Improve stacktraces to be relative to the compilation path and include the related application
- [Stream] Add `Stream.transform/3`

- Bug fixes
  - [ExUnit] `:include` in `ExUnit` only has effect if a test was previously excluded with `:exclude`
  - [ExUnit] Only run `setup_all` and `teardown_all` if there are tests in the case
  - [Kernel] Ensure bitstring modifier arguments are expanded
  - [Kernel] Ensure compiler does not block on missing modules
• [Kernel] Ensure <>/2 works only with binaries
• [Kernel] Fix usage of string literals inside <<<> when utf8/utf16/utf32 is used as specifier
• [Mix] Ensure mix properly copies _build dependencies on Windows

Deprecations
• [Enum] Deprecate Enum.first/1 in favor of Enum.at/2 and List.first/1
• [Kernel] Deprecate continuable heredocs. In previous versions, Elixir would continue parsing on the same line the heredoc started, this behaviour has been deprecated
• [Kernel] is_alive/0 is deprecated in favor of Node.alive?
• [Kernel] Kernel.inspect/2 with Inspect.Opts[] is deprecated in favor of Inspect.Algebra.to_doc/2
• [Kernel] Kernel.inspect/2 with :raw option is deprecated, use :records option instead
• [Kernel] Deprecate <-/2 in favor of send/2

Backwards incompatible changes
• [String] Change String.next_grapheme/1 and String.next_codepoint/1 to return nil on string end

4.5.10 v0.12.1 (2014-01-04)

Enhancements
• [ExUnit] Support :include and :exclude configuration options to filter which tests should run based on their tags. Those options are also supported via mix test as --include and --exclude
• [ExUnit] Allow doctests to match against #MyModule<> Bug fixes
• [CLI] Abort when a pattern given to elixirc does not match any file
• [Float] Fix Float.parse/1 to handle numbers of the form “-0.x”
• [IEx] Improve error message for IExHelpers.r when module does not exist
• [Mix] Ensure deps.get updates origin if lock origin and dep origin do not match
• [Mix] Use relative symlinks in _build
• [Typespec] Fix conversion of unary ops from typespec format to ast
• [Typespec] Fix handling of tuple() and {}

Deprecations
• [Kernel] Do not leak clause heads. Previously, a variable defined in a case/receive head clauses would leak to the outer scope. This behaviour is deprecated and will be removed in the next release.
• [Kernel] Deprecate __FILE__ in favor of __DIR__ or __ENV__.file

Backwards incompatible changes
• [GenFSM] GenServer now stops on unknown event/sync_event requests
• [GenServer] GenServer now stops on unknown call/cast requests
• [Kernel] Change how -> is represented in AST. Now each clause is represented by its own AST node which makes composition easier. See commit 51ae5f55 for more information.
4.5.11 v0.12.0 (2013-12-15)

- **Enhancements**
  - [Exception] Allow `exception/1` to be overridden and promote it as the main mechanism to customize exceptions
  - [File] Add `File.stream_to!/3`
  - [Float] Add `Float.floor/1`, `Float.ceil/1` and `Float.round/3`
  - [Kernel] Add `List.delete_at/2` and `List.updated_at/3`
  - [Kernel] Add `Enum.reverse/2`
  - [Kernel] Implement `defmodule/2`, `@/1`, `def/2` and friends in Elixir itself. `case/2`, `try/2` and `receive/1` have been made special forms. `var!/1`, `var!/2` and `alias!/1` have also been implemented in Elixir and demoted from special forms
  - [Record] Support dynamic fields in `defrecordp`
  - [Stream] Add `Stream.resource/3`
  - [Stream] Add `Stream.zip/2`, `Stream.filter_map/3`, `Stream.each/2`, `Stream.take_every/2`, `Stream.chunks/2`, `Stream.chunk/3`, `Stream.chunks/4`, `Stream.chunks_by/2`, `Stream.scan/2`, `Stream.scan/3`, `Stream.uniq/2`, `Stream.after/2` and `Stream.run/1`
  - [Stream] Support `Stream.take/2` and `Stream.drop/2` with negative counts

- **Bug fixes**
  - [HashDict] Ensure a `HashDict` stored in an attribute can be accessed via the attribute
  - [Enum] Fix bug in `Enum.chunks/4` where you’d get an extra element when the enumerable was a multiple of the counter and a pad was given
  - [IEx] Ensure `c/2` helper works with full paths
  - [Kernel] `quote location: :keep` now only affects definitions in order to keep the proper trace in definition exceptions
  - [Mix] Also symlink `include` directories in `_build` dependencies
  - [Version] Fix `Version.match?/2` with `~>` and versions with alphanumeric build info (like `-dev`)

- **Deprecations**
  - [Enum] `Enumerable.count/1` and `Enumerable.member?/2` should now return tagged tuples. Please see `Enumerable` docs for more info
  - [Enum] Deprecate `Enum.chunks/2`, `Enum.chunks/4` and `Enum.chunks_by/2` in favor of `Enum.chunks/2`, `Enum.chunks/4` and `Enum.chunks_by/2`
  - [File] `File.bistream!/3` is deprecated. Simply use `File.stream!/3` which is able to figure out if `stream` or `bistream` operations should be used
  - [Macro] `Macro.extract_args/1` is deprecated in favor of `Macro.decompose_call/1`

- **Backwards incompatible changes**
  - [Enum] Behaviour of `Enum.drop/2` and `Enum.take/2` has been switched when given negative counts
  - [Enum] Behaviour of `Enum.zip/2` has been changed to stop as soon as the first enumerable finishes
  - [Enum] `Enumerable.reduce/3` protocol has changed to support suspension. Please see `Enumerable` docs for more info
• [Mix] Require :escript_main_module to be set before generating escripts
• [Range] Range.Iterator protocol has changed in order to work with the new Enumerable.reduce/3. Please see Range.Iterator docs for more info
• [Stream] The Stream.Lazy structure has changed to accumulate functions and accumulators as we go (its inspected representation has also changed)
• [Typespec] when clauses were moved to the outer part of the spec and should be in the keywords format. So add(a, b) when is_subtype(a, integer) and is_subtype(b, integer) :: integer should now be written as add(a, b) :: integer when a: integer, b: integer

4.5.12 v0.11.2 (2013-11-14)

• Enhancements
  • [Mix] Add mix iex that redirects users to the proper iex -S mix command
  • [Mix] Support build_per_environment: true in project configuration that manages a separate build per environment, useful when you have per-environment behaviour/compilation
• Backwards incompatible changes
  • [Mix] Mix now compiles files to _build. Projects should update just fine, however documentation and books may want to update to the latest information

4.5.13 v0.11.1 (2013-11-07)

• Enhancements
  • [Mix] Improve dependency convergence by explicitly checking each requirement instead of expecting all requirements to be equal
  • [Mix] Support optional dependencies with optional: true. Optional dependencies are downloaded for the current project but they are automatically skipped when such project is used as a dependency
• Bug fixes
  • [Kernel] Set compilation status per ParallelCompiler and not globally
  • [Mix] Ensure Mix does not load previous dependencies versions before deps.get/deps.update
  • [Mix] Ensure umbrella apps are sorted before running recursive commands
  • [Mix] Ensure umbrella apps run in the same environment as the parent project
  • [Mix] Ensure dependency tree is topsorted before compiling
  • [Mix] Raise error when duplicated projects are pushed into the stack
  • [URI] Allow lowercase escapes in URI
• Backwards incompatible changes
  • [Mix] Setting :load_paths in your project configuration is deprecated
4.5.14 v0.11.0 (2013-11-02)

- Enhancements
  - [Code] Eval now returns variables from other contexts
  - [Dict] Document and enforce all dicts use the match operator (==) when checking for keys
  - [Enum] Add Enum.slice/2 with a range
  - [Enum] Document and enforce Enum.member?/2 to use the match operator (==)
  - [IEx] Split IEx.Evaluator from IEx.Server to allow custom evaluators
  - [IEx] Add support for IEx.pry which halts a given process for inspection
  - [IO] Add specs and allow some IO APIs to receive any data that implements String.Chars
  - [Kernel] Improve stacktraces on command line interfaces
  - [Kernel] Sigils can now handle balanced tokens as in %s(f(o)o)
  - [Kernel] Emit warnings when an alias is not used
  - [Macro] Add Macro.pipe/3 and Macro.unpipe/1 for building pipelines
  - [Mix] Allow umbrella children to share dependencies between them
  - [Mix] Allow mix to be escriptize’ed
  - [Mix] Speed mix projects compilation by relying on more manifests information
  - [Protocol] Protocols now provide impl_for/1 and impl_for!/1 functions which receive a structure and returns its respective implementation, otherwise returns nil or an error
  - [Set] Document and enforce all sets use the match operator (==) when checking for keys
  - [String] Update to Unicode 6.3.0
  - [String] Add String.slice/2 with a range
- Bug fixes
  - [Exception] Ensure defexception fields can be set dynamically
  - [Kernel] Guarantee aliases hygiene is respected when the current module name is not known upfront
  - [Kernel] Kernel.access/2 no longer flattens lists
  - [Mix] Ensure cyclic dependencies are properly handled
  - [String] Implement the extended grapheme cluster algorithm for String operations
- Deprecations
  - [Kernel] pid_to_list/1, list_to_pid/1, binary_to_atom/2, binary_to_existing_atom/2 and atom_to_binary/2 are deprecated in favor of their counterparts in the :erlang module
  - [Kernel] insert_elem/3 and delete_elem/2 are deprecated in favor of Tuple.insert_at/3 and Tuple.delete_at/2
  - [Kernel] Use of in inside matches (as in x in [1,2,3] -> x) is deprecated in favor of the guard syntax (x when x in [1,2,3])
  - [Macro] Macro.expand_all/2 is deprecated
  - [Protocol] @only and @except in protocols are now deprecated
• [Protocol] Protocols no longer fallback to Any out of the box (this functionality needs to be explicitly enabled by setting `@fallback_to_any` to `true`)

• [String] `String.to_integer/1` and `String.to_float/1` are deprecated in favor of `Integer.parse/1` and `Float.parse/1`

• Backwards incompatible changes
  
  • [CLI] Reading `.elixirrc` has been dropped in favor of setting env vars
  
  • [Kernel] `Kernel.access/2` now expects the second argument to be a compile time list
  
  • [Kernel] `fn -> end` quoted expression is no longer wrapped in a `do` keyword

  • [Kernel] Quoted variables from the same module must be explicitly shared. Previously, if a function returned `quote do: a = 1`, another function from the same module could access it as `quote do: a`. This has been fixed and the variables must be explicitly shared with `var!(a, __MODULE__)`

  • [Mix] Umbrella apps now treat children apps as dependencies. This means all dependencies will be checked out in the umbrella `deps` directory. On upgrade, child apps need to point to the umbrella project by setting `deps_path: "..../deps_path", lockfile: "..../mix.lock"` in their project config

  • [Process] `Process.group_leader/2` args have been reversed so the “subject” comes first

  • [Protocol] Protocol no longer dispatches to `Number`, but to `Integer` and `Float`

### 4.5.15 v0.10.3 (2013-10-02)

• Enhancements
  
  • [Enum] Add `Enum.take_every/2`
  
  • [IEx] IEx now respects signals sent from the Ctrl+G menu
  
  • [Kernel] Allow documentation for types with `@typedoc`
  
  • [Mix] Allow apps to be selected in umbrella projects
  
  • [Record] Generated record functions `new` and `update` also take options with strings as keys
  
  • [Stream] Add `Stream.unfold/1`

• Bug fixes
  
  • [Dict] Fix a bug when a HashDict was marked as equal when one was actually a subset of the other
  
  • [IEx] Solve issue where `do` blocks inside templates were not properly aligned
  
  • [ExUnit] Improve checks and have better error reports on poorly aligned doctests
  
  • [Kernel] Fix handling of multiple heredocs on the same line
  
  • [Kernel] Provide better error messages for match, guard and quoting errors
  
  • [Kernel] Make `Kernel.raise/2` a macro to avoid messing up stacktraces
  
  • [Kernel] Ensure `&()` works on quoted blocks with only one expression
  
  • [Mix] Address an issue where a dependency was not compiled in the proper order when specified in different projects

  • [Mix] Ensure `compile: false` is a valid mechanism for disabling the compilation of dependencies

  • [Regex] Fix bug on `Regex.scan/3` when capturing groups and the regex has no groups

  • [String] Fix a bug with `String.split/2` when given an empty pattern
• [Typespec] Guarantee typespecs error reports point to the proper line

• Deprecations
  • [Kernel] The previous partial application syntax (without the & operator) has now been deprecated
  • [Regex] Regex.captures/3 is deprecated in favor of Regex.named_captures/3
  • [String] String.valid_codepoint?/1 is deprecated in favor of pattern matching with \<<_ :: utf8 >>

• Backwards incompatible changes
  • [IEx] The r/0 helper has been removed as it caused surprising behaviour when many modules with dependencies were accumulated
  • [Mix] Mix.Version was renamed to Version
  • [Mix] File_iteratorError was renamed to IO.StreamError
  • [Mix] mix new now defaults to the --sup option, use --bare to get the previous behaviour

4.5.16 v0.10.2 (2013-09-03)

• Enhancements
  • [CLI] Add --verbose to elixirc, which now is non-verbose by default
  • [Dict] Add Dict.Behaviour as a convenience to create your own dictionaries
  • [Enum] Add Enum.split/2, Enum.reduce/2, Enum.flat_map/2, Enum.chunk/2, Enum.chunk/4, Enum.chunk_by/2, Enum.concat/1 and Enum.concat/2
  • [Enum] Support negative indices in Enum.at/fetch/fetch!
  • [ExUnit] Show failures on CLIFormatter as soon as they pop up
  • [IEx] Allow for strings in h helper
  • [IEx] Helpers r and c can handle erlang sources
  • [Integer] Add odd?/1 and even?/1
  • [IO] Added support to specifying a number of bytes to stream to IO.stream, IO.binstream, File.stream! and File.binstream!
  • [Kernel] Include file and line on error report for overriding an existing function/macro
  • [Kernel] Convert external functions into quoted expressions. This allows record fields to contain functions as long as they point to an &Mod.fun/arity
  • [Kernel] Allow foo? and bar! as valid variable names
  • [List] Add List.replace_at/3
  • [Macro] Improve printing of the access protocol on Macro.to_string/1
  • [Macro] Add Macro.to_string/2 to support annotations on the converted string
  • [Mix] Automatically recompile a project if the Elixir version changes
  • [Path] Add Path.relative_to_cwd/2
  • [Regex] Allow erlang re options when compiling Elixir regexes
  • [Stream] Add Stream.concat/1, Stream.concat/2 and Stream.flat_map/2
  • [String] Add regex pattern support to String.replace/3
- [String] Add String.ljust/2, String.rjust/2, String.ljust/3 and String.rjust/3
- [URI] URI.parse/1 supports IPv6 addresses
- Bug fixes
  - [Behaviour] Do not compile behaviour docs if docs are disabled on compilation
  - [ExUnit] Doctests no longer eat too much space and provides detailed reports for poorly indented lines
  - [File] Fix a bug where File.touch(file, datetime) was not setting the proper datetime when the file did not exist
  - [Kernel] Limit inspect results to 50 items by default to avoid printing too much data
  - [Kernel] Return a readable error on oversized atoms
  - [Kernel] Allow functions ending with ? or ! to be captured
  - [Kernel] Fix default shutdown of child supervisors to :infinity
  - [Kernel] Fix regression when calling a function/macro ending with bang, followed by do/end blocks
  - [List] Fix bug on List.insert_at/3 that added the item at the wrong position for negative indexes
  - [Macro] Macro.escape/2 can now escape improper lists
  - [Mix] Fix Mix.Version matching on pre-release info
  - [Mix] Ensure watch_exts trigger full recompilation on change with mix compile
  - [Mix] Fix regression on mix clean --all
  - [String] String.strip/2 now supports removing unicode characters
  - [String] String.slice/3 still returns the proper result when there is no length to be extracted
  - [System] System.get_env/0 now returns a list of tuples as previously advertised
- Deprecations
  - [Dict] Dict.update/3 is deprecated in favor of Dict.update!/3
  - [Enum] Enum.min/2 and Enum.max/2 are deprecated in favor of Enum.min_by/2 and Enum.max_by/2
  - [Enum] Enum.join/2 and Enum.map_join/3 with a char list are deprecated
  - [IO] IO.stream(device) and IO.binstream(device) are deprecated in favor of IO.stream(device, :line) and IO.binstream(device, :line)
  - [Kernel] list_to_binary/1, binary_to_list/1 and binary_to_list/3 are deprecated in favor of String.from_char_list!/1 and String.to_char_list!/1 for characters and binary.list_to_bin/1, binary.bin_to_list/1 and binary.bin_to_list/3 for bytes
  - [Kernel] to_binary/1 is deprecated in favor of to_string/1
  - [Kernel] Deprecate def/4 and friends in favor of def/2 with unquote and friends
  - [Kernel] Deprecate %b and %B in favor of %s and %S
  - [List] List.concat/2 is deprecated in favor of Enum.concat/2
  - [Macro] Macro.unescape_binary/1 and Macro.unescape_binary/2 are deprecated in favor of Macro.unescape_string/1 and Macro.unescape_string/2
  - [Mix] :umbrella option for umbrella paths has been deprecated in favor of :in_umbrella
- Backwards incompatible changes
• [IO] IO functions now only accept iolists as arguments
• [Kernel] Binary.Chars was renamed to String.Char.
• [Kernel] The previous ambiguous import syntax import :functions, Foo was removed in favor of import Foo, only: :functions
• [OptionParser] parse and parse_head now returns a tuple with three elements instead of two

4.5.17 v0.10.1 (2013-08-03)

• Enhancements
  • [Behaviour] Add support for defmacrocallback/1
  • [Enum] Add Enum.shuffle/1
  • [ExUnit] The :trace option now also reports run time for each test
  • [ExUnit] Add support for :color to enable/disable ANSI coloring
  • [IEx] Add the clear helper to clear the screen.
  • [Kernel] Add the capture operator &
  • [Kernel] Add support for GenFSM.Behaviour
  • [Kernel] Functions now points to the module and function they were defined when inspected
  • [Kernel] A documentation attached to a function that is never defined now prints warnings
  • [List] Add List.keysort/2
  • [Mix] :test_helper project configuration did not affect mix test and was therefore removed. A test/test_helper.exs file is still necessary albeit it doesn’t need to be automatically required in each test file
  • [Mix] Add manifests for yecc, leex and Erlang compilers, making it easier to detect dependencies in between compilers and providing a more useful clean behaviour
  • [Mix] mix help now outputs information about the default mix task
  • [Mix] Add --no-deps-check option to mix run, mix compile and friends to not check dependency status
  • [Mix] Add support for MIX_GIT_FORCE_HTTPS system environment that forces HTTPS for known providers, useful when the regular git port is blocked. This configuration does not affect the mix.lock results
  • [Mix] Allow coverage tool to be pluggable via the :test_coverage configuration
  • [Mix] Add mix cmd as a convenience to run a command recursively in child apps in an umbrella application
  • [Mix] Support umbrella: true in dependencies as a convenience for setting up umbrella path deps
  • [Mix] mix run now behaves closer to the elixir command and properly mangles the ARGV
  • [String] Add Regex.scan/3 now supports capturing groups
  • [String] Add String.reverse/1
• Bug fixes
  • [Behaviour] Ensure callbacks are stored in the definition order
  • [CLI] Speed up boot time on Elixir .bat files
• [IEx] Reduce cases where IEx parser can get stuck
• [Kernel] Improve error messages when the use of an operator has no effect
• [Kernel] Fix a bug where warnings were not being generated when imported macros conflicted with local functions or macros
• [Kernel] Document that on_definition can only be a function as it is evaluated inside the function context
• [Kernel] Ensure $w sigils with no interpolation are fully expanded at compile time
• [Mix] mix deps.update, mix deps.clean and mix deps.unlock no longer change all dependencies unless --all is given
• [Mix] Always run mix loadpaths on mix app.start, even if --no-compile is given
• [OptionParser] Do not add boolean flags to the end result if they were not given
• [OptionParser] Do not parse non-boolean flags as booleans when true or false are given
• [OptionParser] Ensure :keep and :integer|:float can be given together as options
• [OptionParser] Ensure --no-flag sets :flag to false when :flag is a registered boolean switch

4.5.18 v0.10.0 (2013-07-15)

• Enhancements
• [ExUnit] Support trace: true option which gives detailed reporting on test runs
• [HashDict] Optimize HashDict to store pairs in a cons cell reducing storage per key by half
• [Kernel] Add pretty printing support for inspect
• [Kernel] Add document algebra library used as the foundation for pretty printing
• [Kernel] Add defrecordp/3 that enables specifying the first element of the tuple
• [Kernel] Add the Set API and a hash based implementation via HashSet
• [Kernel] Add Stream as composable, lazy-enumerables
• [Mix] mix archive now includes the version of the generated archive
• [Mix] Mix now requires explicit dependency overriding to be given with override: true
• [Mix] Projects can now define an :elixir key to outline supported Elixir versions
• [Typespec] Improve error messages to contain file, line and the typespec itself
• Bug fixes
  • [CLI] Elixir can now run on Unix directories with \ in its path
  • [Kernel] `match?/2` does not leak variables to outer scope
  • [Kernel] Keep `head|tail` format when splicing at the tail
  • [Kernel] Ensure variables defined in the module body are not passed to callbacks
  • [Mix] On dependencies conflict, show from where each source is coming from
  • [Mix] Empty projects no longer leave emptyebin files on `mix compile`
  • [Module] Calling `Module.register_attribute/3` no longer automatically changes it to persisted or accumulated

• Deprecations
  • [Enum] Receiving the index of iteration in `Enum.map/2` and `Enum.each/2` is deprecated in favor of `Stream.with_index/1`
  • [File] `File.iterator/1` and `File.biniterator/1` are deprecated in favor of `IO.stream/1` and `IO.binstream/1`
  • [File] `File.iterator!/2` and `File.biniterator!/2` are deprecated in favor of `File.stream!/2` and `File.binstream!/2`
  • [Kernel] Deprecate recently added `quote binding: ...` in favor of the clearer `quote bind_quoted: ...`
  • [Kernel] Deprecate `Kernel.float/1` in favor of a explicit conversion
  • [Mix] Deprecate `mix run EXPR` in favor of `mix run -e EXPR`
  • [Record] `Record.__index__/2` deprecated in favor of `Record.__record__(:index, key)`

• Backwards incompatible changes
  • [Kernel] The `Binary.Inspect` protocol has been renamed to `Inspect`
  • [Kernel] Tighten up the grammar rules regarding parentheses omission, previously the examples below would compile but now they raise an error message:

```elixir
do_something 1, is_list [], 3
[1, is_atom :foo, 3]
```

  • [Module] Calling `Module.register_attribute/3` no longer automatically changes it to persisted or accumulated
  • [Record] First element of a record via `defrecordp` is now the `defrecordp` name and no longer the current atom
  • [URI] Remove custom URI parsers in favor of `URI.default_port/2`

4.5.19 v0.9.3 (2013-06-23)

• Enhancements
  • [File] Add `File.chgrp`, `File.chmod` and `File.chown`
  • [Kernel] Add `--warnings-as-errors` to Elixir’s compiler options
  • [Kernel] Print warnings to stderr
  • [Kernel] Warn on undefined module attributes
• [Kernel] Emit warning for \texttt{x \;in \;[]} in guards
• [Kernel] Add \texttt{binding/0} and \texttt{binding/1} for retrieving bindings
• [Kernel] \texttt{quote} now allows a binding as an option
• [Macro] Add \texttt{Macro.expand_once/2} and \texttt{Macro.expand_all/2}
• [Mix] Implement \texttt{Mix.Version} for basic versioning semantics
• [Mix] Support creation and installation of archives (.ez files)
• [Mix] \texttt{github: ...} shortcut now uses the faster \texttt{git} schema instead of \texttt{https}
• [Record] Allow types to be given to \texttt{defrecordp}
• Bug fixes
• [Kernel] The elixir executable on Windows now supports the same options as the UNIX one
• [Kernel] Improve error messages on default clauses clash
• [Kernel] \texttt{__MODULE__.Foo} now returns \texttt{Foo} when outside of a Module
• [Kernel] Improve error messages when default clauses from different definitions collide
• [Kernel] \^x variables should always refer to the value before the expression
• [Kernel] Allow \texttt{(x, y) when z} in function clauses and try expressions
• [Mix] \texttt{Mix} now properly evaluates rebar scripts
• Deprecations
• [Code] \texttt{Code.string_to_ast/1} has been deprecated in favor of \texttt{Code.string_to_quoted/1}
• [Macro] \texttt{Macro.to_binary/1} has been deprecated in favor of \texttt{Macro.to_string/1}
• [Typespec] Deprecate \texttt{(fun(...) -> ...)} in favor of \texttt{(... -> ...)}
• Backwards incompatible changes
• [Bitwise] Precedence of operators used by the Bitwise module were changed, check \texttt{elixir_parser.yrl} for more information
• [File] \texttt{rm_rf} and \texttt{cp_r} now returns a tuple with three elements on failures
• [Kernel] The quoted representation for \texttt{->} clauses changed from a tuple with two elements to a tuple with three elements to support metadata
• [Kernel] Sigils now dispatch to \texttt{sigil_$_} instead of \texttt{__$$_} where \$ is the sigil character
• [Macro] \texttt{Macro.expand/2} now expands until final form. Although this is backwards incompatible, it is very likely you do not need to change your code, since expansion until its final form is recommended, particularly if you are expecting an atom out of it
• [Mix] No longer support beam files on \texttt{mix local}

4.5.20 v0.9.2 (2013-06-13)

• Enhancements
  • [ExUnit] \texttt{capture_io} now captures prompt by default
  • [Mix] Automatically import git dependencies from Rebar
  • [Mix] Support for dependencies directly from the umbrella application
• [Regex] Add Regex.escape
• [String] Add String.contains?
• [URI] Implement Binary.Chars (aka to_binary) for URI.Info
• Bug fixes
  • [HashDict] Ensure HashDict uses exact match throughout its implementation
  • [IEx] Do not interpret ANSI codes in IEx results
  • [IEx] Ensure --cookie is set before accessing remote shell
  • [Kernel] Do not ignore nil when dispatching protocols to avoid infinite loops
  • [Mix] Fix usage of shell expressions in Mix.Shell.cmd
  • [Mix] Start the application by default on escripts
• Deprecations
  • [Regex] Regex.index/2 is deprecated in favor Regex.run/3
  • [Kernel] super no longer supports implicit arguments
• Backwards incompatible changes
  • [Kernel] The =~ operator now returns true or false instead of an index

**4.5.21 v0.9.1 (2013-05-30)**

- Enhancements
  • [IEx] Limit the number of entries kept in history and allow it to be configured
  • [Kernel] Add String.start_with? and String.end_with?
  • [Typespec] Allow keywords, e.g. [foo: integer, bar: boolean | module], in typespecs
- Bug fixes
  • [Dict] Enum.to_list and Dict.to_list now return the same results for dicts
  • [IEx] Enable shell customization via the IEx.Options module
  • [Kernel] Fix a bug where unquote_splicing did not work on the left side of a stab op
  • [Kernel] Unused functions with cyclic dependencies are now also warned as unused
  • [Mix] Fix a bug where mix deps.get was not retrieving nested dependencies
  • [Record] Fix a bug where nested records cannot be defined
  • [Record] Fix a bug where a record named Record cannot be defined

**4.5.22 v0.9.0 (2013-05-23)**

- Enhancements
  • [ExUnit] ExUnit.CaptureIO now accepts an input to be used during capture
  • [IEx] Add support for .iex files that are loaded during shell’s boot process
  • [IEx] Add import_file/1 helper
- Backwards incompatible changes
- [Enum] `Enum.Iterator` was replaced by the more composable and functional `Enumerable` protocol which supports reductions
- [File] `File.iterator/1` and `File.biniterator/1` have been removed in favor of the safe `File.iterator!/1` and `File.biniterator!/1` ones
- [Kernel] Erlang R15 is no longer supported
- [Kernel] Elixir modules are now represented as `Elixir.ModuleName` (using . instead of - as separator)

### 4.5.23 v0.8.3 (2013-05-22)

- Enhancements
- [CLI] Flags `-p` and `-pr` fails if pattern match no files
- [CLI] Support `--hidden` and `--cookie` flags for distributed Erlang
- [Enum] Add `Enum.to_list/1`, `Enum.member?/2`, `Enum.uniq/2`, `Enum.max/1`, `Enum.max/2`, `Enum.min/1` and `Enum.min/2`
- [ExUnit] Add `ExUnit.CaptureIO` for IO capturing during tests
- [ExUnit] Consider load time on ExUnit time reports
- [IEx] Support `ls` with colored output
- [IEx] Add `#iex:break` to break incomplete expressions
- [Kernel] Add `Enum.at`, `Enum.fetch` and `Enum.fetch!`
- [Kernel] Add `String.to_integer` and `String.to_float`
- [Kernel] Add `Dict.take`, `Dict.drop`, `Dict.split`, `Dict.pop` and `Dict.fetch!`
- [Kernel] Many optimizations for code compilation
- [Kernel] `in` can be used with right side expression outside guards
- [Kernel] Add `Node.get_cookie/0` and `Node.set_cookie/2`
- [Kernel] Add `__DIR__`
- [Kernel] Expand macros and attributes on quote, import, alias and require
- [Kernel] Improve warnings related to default arguments
- [Keyword] Add `Keyword.delete_first/2`
- [Mix] Add `local.rebar` to download a local copy of rebar, and change `deps.compile` to use it if needed
- [Mix] Support umbrella applications
- [Mix] Load beam files available at `MIX_PATH` on CLI usage
- [String] Add `String.valid?` and `String.valid_character?`

### Bug fixes

- [ExUnit] Handle exit messages from in ExUnit
- [ExUnit] Failures on ExUnit’s setup_all now invalidates all tests
- [Kernel] Ensure we don’t splice keyword args unnecessarily
- [Kernel] Private functions used by private macros no longer emit an unused warning
- [Kernel] Ensure Elixir won’t trip on empty receive blocks
• [Kernel] `String.slice` now returns an empty string when out of range by 1
• [Mix] Generate manifest files after compilation to avoid depending on directory timestamps and to remove unused .beam files
• [Path] `Path.expand/2` now correctly expands ~ in the second argument
• [Regex] Fix badmatch with `Regex.captures(%r/(.)/g, "cat")`
• [URI] Downcase host and scheme and URIs

Deprecations
• [Code] `Code.eval` is deprecated in favor of `Code.eval_string`
• [Exception] `Exception.format_entry` is deprecated in favor of `Exception.format_stacktrace_entry`
• [ExUnit] `assert left inlist right` is deprecated in favor of `assert left in right`
• [IO] `IO.getb` is deprecated in favor of `IO.getn`
• [List] `List.member?/2` is deprecated in favor of `Enum.member?/2`
• [Kernel] `var_context` in quote was deprecated in favor of `context`
• [Kernel] `Enum.at!` and `Dict.get!` is deprecated in favor of `Enum.fetch!` and `Dict.fetch!`

Backwards incompatible changes
• [Dict] `List.Dict` was moved to `ListDict`
• [IO] `IO.gets`, `IO.getn` and friends now return binaries when reading from stdio
• [Kernel] Precedence of `|>` has changed to lower to support constructs like `1..5 |> Enum.to_list`
• [Mix] `mix escriptize` now receives arguments as binaries

4.5.24 v0.8.2 (2013-04-20)

• Enhancements
• [ExUnit] Use ANSI escape codes in CLI output
• [ExUnit] Include suite run time on CLI results
• [ExUnit] Add support to doctests, allowing test cases to be generated from code samples
• [File] Add `File.ls` and `File.ls!`
• [IEx] Support `pwd` and `cd` helpers
• [Kernel] Better error reporting for invalid bitstring generators
• [Kernel] Improve meta-programming by allowing `unquote` on `def/2`, `defp/2`, `defmacro/2` and `defmacrop/2`
• [Kernel] Add support to R16B new functions: `insert_elem/3` and `delete_elem/2`
• [Kernel] Import conflicts are now lazily handled. If two modules import the same functions, it will fail only if the function is invoked
• [Mix] Support `--cover` on mix test and `test_coverage` on Mixfiles
• [Record] Each record now provides `Record.options` with the options supported by its `new` and `update` functions
• Bug fixes
• [Binary] inspect no longer escapes standalone hash #
• [IEx] The h helper can now retrieve docs for special forms
• [Kernel] Record optimizations were not being triggered in functions inside the record module
• [Kernel] Aliases defined inside macros should be carried over
• [Kernel] Fix a bug where nested records could not use the Record[] syntax
• [Path] Fix a bug on Path.expand when expanding paths starting with ~

• Deprecations
• [Kernel] setelem/3 is deprecated in favor of set_elem/3
• [Kernel] function(:is_atom, 1) is deprecated in favor of function(is_atom/1)

• Backwards incompatible changes
• [Kernel] unquote now only applies to the closest quote. If your code contains a quote that contains another quote that calls unquote, it will no longer work. Use Macro.escape instead and pass your quoted contents up in steps, for example:

```elixir
quote do
  quote do: unquote(x)
end
```

should become:

```elixir
quote do
  unquote(Macro.escape(x))
end
```

### 4.5.25 v0.8.1 (2013-02-17)

• Enhancements
• [ExUnit] Tests can now receive metadata set on setup/teardown callbacks
• [ExUnit] Add support to ExUnit.CaseTemplate to share callbacks in between test cases
• [IO] Add IO.ANSI to make it easy to write ANSI escape codes
• [Kernel] Better support for Unicode lists
• [Kernel] Reduce variables footprint in case/receive clauses
• [Kernel] Disable native compilation when on_load attributes is present to work around an Erlang bug
• [Macro] Macro.expand also considers macros from the current __ENV__ module
• [Mix] Improve support for compilation of .erl files
• [Mix] Add support for compilation of .yrl and .xrl files
• [OptionParser] Switches are now overridden by default but can be kept in order if chosen
• [Typespec] Better error reporting for invalid typespecs

• Bug fixes
• [Mix] Allow Mix projects to be generated with just one letter
• Backwards incompatible changes
• [Kernel] before_compile and after_compile callbacks now receive the environment as first argument instead of the module

• Deprecations
  • [ExUnit] Explicitly defined test/setup/teardown functions are deprecated
  • [Kernel] Tidy up and clean quote API
  • [Kernel] Old :local.(args) syntax is deprecated
  • [Process] Process.self is deprecated in favor Kernel.self

4.5.26 v0.8.0 (2013-01-28)

• Enhancements
  • [Binary] Support << "string" :: utf8 ::>> as in Erlang
  • [Binary] Support \a escape character in binaries
  • [Binary] Support syntax shortcut for specifying size in bit syntax
  • [CLI] Support --app option to start an application and its dependencies
  • [Dict] Support put_new in Dict and Keyword
  • [Dict] Add ListDict and a faster HashDict implementation
  • [ExUnit] ExUnit now supports multiple runs in the same process
  • [ExUnit] Failures in ExUnit now shows a tailored stacktrace
  • [ExUnit] Introduce ExUnit.ExpectationError to provide better error messages
  • [Kernel] Introduce Application.Behaviour to define application module callbacks
  • [Kernel] Introduce Supervisor.Behaviour to define supervisors callbacks
  • [Kernel] More optimizations were added to Record handling
  • [Kernel] ?\x and ?\ are now supported ways to retrieve a codepoint
  • [Kernel] Octal numbers can now be defined as 0777
  • [Kernel] Improve macros hygiene regarding variables, aliases and imports
  • [Mix] Mix now starts the current application before run, iex, test and friends
  • [Mix] Mix now provides basic support for compiling .erl files
  • [Mix] mix escriptize only generates escript if necessary and accept --force and --no-compile as options
  • [Path] Introduce Path module to hold filesystem paths related functions
  • [String] Add String.capitalize and String.slice
  • [System] Add System.tmp_dir, System.cwd and System.user_home

• Bug fixes
  • [Kernel] import with only accepts functions starting with underscore
  • [String] String.first and String.last return nil for empty binaries
  • [String] String.rstrip and String.lstrip now verify if argument is a binary
  • [Typespec] Support ... inside typespec’s lists
Elixir Documentation, Release

- Backwards incompatible changes
  - [Kernel] The AST now allows metadata to be attached to each node. This means the second item in the AST is no longer an integer (representing the line), but a keywords list. Code that relies on the line information from AST or that manually generate AST nodes need to be properly updated
- Deprecations
  - [Dict] Deprecate `Binary.Dict` and `OrdDict` in favor of `HashDict` and `ListDict`
  - [File] Deprecate path related functions in favor of the module `Path`
  - [Kernel] The `/>` operator has been deprecated in favor of `|>`
  - [Mix] `Mix.Project.sources` is deprecated in favor of `Mix.Project.config_files`
  - [Mix] `mix iex` is no longer functional, please use `iex -S mix`
  - [OptionParser] `:flags` option was deprecated in favor of `:switches` to support many types

4.5.27 v0.7.2 (2012-12-04)

- Enhancements
  - [CLI] `--debug-info` is now true by default
  - [ExUnit] Make ExUnit exit happen in two steps allowing developers to add custom `at_exit` hooks
  - [IEx] Many improvements to helpers functions `h/1, s/1` and others
  - [Kernel] Functions defined with `fn` can now handle many clauses
  - [Kernel] Raise an error if clauses with different arities are defined in the same function
  - [Kernel] `function macro` now accepts arguments in `M.f/a` and `f/a` formats
  - [Macro] Improvements to `Macro.to_binary`
  - [Mix] Mix now echoes the output as it comes when executing external commands such as git or rebar
  - [Mix] Mix now validates application callback’s values
  - [Record] Record accessors are now optimized and can be up to 6x faster in some cases
  - [String] Support `\xXX` and `\x{HEX}` escape sequences in strings, char lists and regexes
- Bug fixes
  - [Bootstrap] Compiling Elixir source no longer fails if environment variables contain utf-8 entries
  - [IEx] IEx will now wait for all command line options to be processed before starting
  - [Kernel] Ensure proper stacktraces when showing deprecations
- Deprecations
  - [Enum] `Enum.qsort` is deprecated in favor of `Enum.sort`
  - [List] `List.sort` and `List.uniq` have been deprecated in favor of their `Enum` counterparts
  - [Record] Default-based generated functions are deprecated
  - [Typespec] Enhancements and deprecations to the `@spec/@callback` and the `fun` type syntax
4.5.28 v0.7.1 (2012-11-18)

- Enhancements
- [IEx] Only show documented functions and also show docs for default generated functions
- [IO] Add IO.binread, IO.binwrite and IO.binreadline to handle raw binary file operations
- [ExUnit] Add support for user configuration at HOME/.ex_unit.exs
- [ExUnit] Add support for custom formatters via a well-defined behaviour
- [Kernel] Add support for defrecordp
- [Kernel] Improved dialyzer support
- [Kernel] Improved error messages when creating functions with aliases names
- [Mix] Improve SCM behaviour to allow more robust integration
- [Mix] Changing deps information on mix.exs forces users to fetch new dependencies
- [Mix] Support (parallel) requires on mix run
- [Mix] Support -q when running tests to compile only changed files
- [String] Support String.downcase and String.upcase according to Unicode 6.2.0
- [String] Support graphemes in String.length, String.at and others
- [Typespec] Support @opaque as attribute
- [Typespec] Define a default type t for protocols and records
- [Typespec] Add support for the access protocol in typespecs
- Bug fixes
- [Kernel] Fix an issue where variables inside clauses remained unassigned
- [Kernel] Ensure defoverridable functions can be referred in many clauses
- [Kernel] Allow keywords as function names when following a dot (useful when integrating with erlang libraries)
- [File] File is opened by default on binary mode instead of utf-8
- Deprecations
- [Behaviour] defcallback/1 is deprecated in favor of defcallback/2 which matches erlang @callbacks
- [Enum] Enum.times is deprecated in favor of using ranges
- [System] halt moved to System module

4.5.29 v0.7.0 (2012-10-20)

- Enhancements
- [Behaviour] Add Behaviour with a simple callback DSL to define callbacks
- [Binary] Add a Dict binary that converts its keys to binaries on insertion
- [Binary] Optimize Binary.Inspect and improve inspect for floats
- [CLI] Support --detached option
- [Code] Code.string_to_ast supports :existing_atoms_only as an option in order to guarantee no new atoms is generated when parsing the code
- [EEx] Support <%% and <%%# tags
- [ExUnit] Support after_spawn callbacks which are invoked after each process is spawned
- [ExUnit] Support context data in setup_all, setup, teardown and teardown_all callbacks
- [IEx] Support after_spawn callbacks which are invoked after each process is spawned
- [Kernel] Better error messages when invalid options are given to import, alias or require
- [Kernel] Allow partial application on literals, for example: {&1, &2} to build tuples or [&1|&2] to build cons cells
- [Kernel] Added integer_to_binary and binary_to_integer
- [Kernel] Added float_to_binary and binary_to_float
- [Kernel] Many improvements to unquote and unquote_splicing. For example, unquote(foo).unquote(bar)(args) is supported and no longer need to be written via apply
- [Keyword] Keyword list is no longer ordered according to Erlang terms but the order in which they are specified
- [List] Add List.keyreplace and List.keystore
- [Macro] Support Macro.safe_term which returns :ok if an expression does not execute code and is made only of raw data types
- [Mix] Add support for environments - the current environment can be set via MIX_ENV
- [Mix] Add support for handling and fetching dependencies’ dependencies
- [Module] Support module creation via Module.create
- [Range] Support decreasing ranges
- [Record] Improvements to the Record API, added Record.defmacros
- [Regex] Add :return option to Regex.run and Regex.scan
- [String] Add a String module responsible for handling UTF-8 binaries
- Bug fixes
  - [File] File.cp and File.cp_r now preserves the file’s mode
  - [IEx] Fix a bug where printing to :stdio on IEx was causing it to hang
  - [Macro] Fix a bug where quoted expressions were not behaving the same as their non-quoted counterparts
  - [Mix] mix deps.get [DEPS] now only gets the specified dependencies
  - [Mix] Mix now exits with status 1 in case of failures
  - [Protocol] Avoid false positives on protocol dispatch (a bug caused the dispatch to be triggered to an invalid protocol)
  - Backwards incompatible changes
    - [ExUnit] setup and teardown callbacks now receives the test name as second argument
    - [Kernel] Raw function definition with def/4, defp/4, defmacro/4, defmacrop/4 now evaluates all arguments. The previous behaviour was accidental and did not properly evaluate all arguments
    - [Kernel] Change tuple-related (elem and setelement), Enum functions (find_index, nth! and times) and List functions (List.key*) to zero-index
Elixir Documentation, Release

- Deprecations
  - [Code] `Code.require_file` and `Code.load_file` now expect the full name as argument
  - [Enum] `List.reverse/1` and `List.zip/2` were moved to `Enum`
  - [Kernel] Bitstring syntax now uses `::` instead of `|`
  - [Kernel] Erlang syntax is deprecated in favor of simply using atoms
  - [Module] `Module.read_attribute` and `Module.add_attribute` deprecated in favor of `Module.get_attribute` and `Module.put_attribute` which mimics `Dict` API

4.5.30 v0.6.0 (2012-08-01)

- Backwards incompatible changes
  - [Kernel] Compile files now follow `Elixir-ModuleName` convention to solve issues with Erlang embedded mode. This removes the `__MAIN__` pseudo-variable as modules are now located inside `Elixir` namespace
  - [Kernel] `__using__` callback triggered by `use` now receives just one argument. Caller information can be accessed via macros using `__CALLER__`
  - [Kernel] Comprehensions syntax changed to be more compatible with Erlang behaviour
  - [Kernel] `loop` and `recur` are removed in favor of recursion with named functions
  - [Module] Removed data functions in favor of unifying the attributes API
  - Deprecations
    - [Access] The semantics of the access protocol were reduced from a broad query API to simple data structure key-based access
    - [ExUnit] Some assertions are deprecated in favor of simply using `assert()`
    - [File] `File.read_info` is deprecated in favor of `File.stat`
    - [IO] `IO.print` is deprecated in favor of `IO.write`
    - [Kernel] Deprecate `__LINE__` and `__FUNCTION__` in favor of `__ENV__.line` and `__ENV__.function`
    - [Kernel] Deprecate `in_guard` in favor of `__CALLER__.in_guard?`
    - [Kernel] `refer` is deprecated in favor of alias
    - [Module] `Module.add_compile_callback(module, target, callback)` is deprecated in favor of `Module.put_attribute(module, :before_compile, {target, callback})`
    - [Module] `Module.function_defined?` is deprecated in favor of `Module.defines?`
    - [Module] `Module.defined_functions` is deprecated in favor of `Module.definitions_in`
  - Enhancements
    - [Enum] Enhance `Enum` protocol to support `Enum.count`
    - [Enum] Optimize functions when a list is given as collection
    - [Enum] Add `find_index`, `nth!` and others
    - [ExUnit] Support setup and teardown callbacks
    - [IEx] IEx now provides autocomplete if the OS supports tty
• [IEx] IEx now supports remsh
• [IEx] Elixir now defaults to compile with documentation and \( \text{d} \) can be used in IEx to print modules and functions documentation
• [IEx] Functions \( \text{c} \) and \( \text{m} \) are available in IEx to compile and print available module information. Functions \( \text{h} \) and \( \text{v} \) are available to show history and print previous commands values
• [IO/File] Many improvements to \text{File} and \text{IO} modules
• [Kernel] Operator \( \! \) is now allowed in guard clauses
• [Kernel] Introduce operator \( \sim \) for regular expression matches
• [Kernel] Compiled docs now include the function signature
• [Kernel] \text{defmodule} do not start a new variable scope, this improves meta-programming capabilities
• [Kernel] quote special form now supports line and unquote as options
• [Kernel] Document the macro \( \@ \) and allow attributes to be read inside functions
• [Kernel] Add support to the \( \%R \) sigil. The same as \( \%r \), but without interpolation or escaping. Both implementations were also optimized to generate the regex at compilation time
• [Kernel] Add \( \_\_\text{ENV}\_ \) which returns a \text{Macro.Env} record with information about the compilation environment
• [Kernel] Add \( \_\_\text{CALLER}\_ \) inside macros which returns a \text{Macro.Env} record with information about the calling site
• [Macro] Add \text{Macro.expand}, useful for debugging what a macro expands to
• [Mix] First Mix public release
• [Module] Add support to \@before_compile and \@after_compile callbacks. The first receives the module name while the latter receives the module name and its object code
• [OptionParser] Make OptionParser public, add support to flags and improved switch parsing
• [Range] Add a Range module with support to \( \text{x in 1..3} \) and iterators
• [Record] Allow \text{Record[\_:_ : value]} to set a default value to all records fields, as in Erlang
• [Record] Records now provide a \text{to_keywords} function
• [Regex] Back references are now properly supported
• [System] Add \text{System.find_executable}

4.5.31 v0.5.0 (2012-05-24)

• First official release
Elixir Module Index

a
Access, 390
Agent, 102
Application, 106
ArgumentError, 377
ArithmeticError, 378
Atom, 110

b
BadArityError, 378
BadFunctionError, 378
BadStructError, 379
Base, 111
Behaviour, 116
Bitwise, 116

c
CaseClauseError, 379
Code, 118
Code.LoadError, 380
Collectable, 390
CompileError, 380

d
Dict, 122

E
EEx, 415
EEx.AssignsEngine, 417
EEx.Engine, 418
EEx.SmartEngine, 419
EEx_SyntaxError, 419
EEx.TransformerEngine, 419
Enum, 132
Enum.EmptyError, 381
Enum.OutOfBoundsError, 381
Enumerable, 392
ErlangError, 381
Exception, 148
ExUnit, 396

ExUnit.AssertionError, 413
ExUnit.Assertions, 398
ExUnit.Callbacks, 402
ExUnit.CaptureIO, 404
ExUnit.Case, 405
ExUnit.TestCaseTemplate, 407
ExUnit.DocTest, 408
ExUnit.DocTest.Error, 414
ExUnit.Filters, 410
ExUnit.Formatter, 411
ExUnit.Test, 412
ExUnit.TestCase, 413

f
File, 151
File.CopyError, 382
File.Error, 382
File.Stat, 162
File.Stream, 163
Float, 163
FunctionClauseError, 383

G
GenEvent, 166
GenServer, 172

h
HashDict, 177
HashSet, 179

I
IEx, 455
IEx.Helpers, 461
IEx.History.Server, 464
Inspect, 394
Inspect.Algebra, 189
Inspect.Opts, 194
Integer, 194
IO, 180
IO.ANSI, 184
IO.Stream, 189
IO.StreamError, 383

k
Kernel, 196
Kernel.ParallelCompiler, 231
Kernel.ParallelRequire, 231
Kernel.SpecialForms, 232
Kernel.Typespec, 256
KeyError, 384
Keyword, 261

l
List, 269
List.Chars, 395

m
Macro, 277
Macro.Env, 282
Map, 284
MatcherError, 384
Mix, 421
Mix.Archive, 421
Mix.Config, 422
Mix.ElixirVersionError, 453
Mix.Error, 453
Mix.Generator, 423
Mix.InvalidTaskError, 454
Mix.NoProjectError, 454
Mix.NoTaskError, 455
Mix.Project, 424
Mix.SCM, 426
Mix.Shell, 428
Mix.Shell.IO, 429
Mix.Shell.Process, 429
Mix.Task, 430
Mix.Tasks.App.Start, 432
Mix.Tasks.Archive, 433
Mix.Tasks.Clean, 433
Mix.Tasks.Cmd, 433
Mix.Tasks.Compile, 434
Mix.Tasks.Compile.App, 434
Mix.Tasks.Compile.Elixir, 435
Mix.Tasks.Compile.Erlang, 436
Mix.Tasks.Compile.Leex, 437
Mix.Tasks.Compile.Yecc, 438
Mix.Tasks.Deps, 439
Mix.Tasks.Deps.Check, 440
Mix.Tasks.Deps.Check, 441
Mix.Tasks.Deps.Compile, 441
Mix.Tasks.Deps.Get, 442
Mix.Tasks.Deps.Loadpaths, 442
Mix.Tasks.Deps.Unlock, 442
Mix.Tasks.Deps.Update, 443
Mix.Tasks.Do, 443
Mix.Tasks.Escriptize, 443
Mix.Tasks.Help, 444
Mix.Tasks.Iex, 445
Mix.Tasks.Loadconfig, 445
Mix.Tasks.Loadpaths, 445
Mix.Tasks.Local, 446
Mix.Tasks.Local.Hex, 446
Mix.Tasks.Local.Install, 446
Mix.Tasks.Local.Rebar, 447
Mix.Tasks.Local.Uninstall, 447
Mix.Tasks.New, 447
Mix.Tasks.Run, 448
Mix.Tasks.Test, 449
Mix.UTILS, 450
Module, 286

n
Node, 296
OptionParser, 300

o
Path, 302
Port, 308
Process, 309
Protocol, 315
Protocol.UndefiError, 384

p
Range, 317
Range.Iterator, 395
Record, 317
Regex, 319
Regex.CompileError, 385
RuntimeError, 385

s
Set, 324
Stream, 328
String, 338
String.Chars, 395
StringIO, 353
Supervisor, 354
Supervisor.Spec, 360
SyntaxError, 386
System, 363
SystemLimitError, 386

u
Task, 367
Task.Supervisor, 369
TokenMissingError, 387
TryClauseError, 387
Tuple, 371

U
UndefinedFunctionError, 388
UnicodeConversionError, 388
URI, 372

V
Version, 374
Version.InvalidRequirementError, 388
Version.InvalidVersionError, 389