fnc Documentation

Release 0.4.0

Derrick Gilland

Jan 23, 2019

Contents

Functional programming in Python with generators and other utilities.

CHAPTER 1

Links

- Project: https://github.com/dgilland/fnc
- Documentation: https://fnc.readthedocs.io
- PyPI: https://pypi.python.org/pypi/fnc/
- TravisCI: https://travis-ci.org/dgilland/fnc

CHAPTER 2

Features

- Functional-style methods that work with and return generators.
- Shorthand-style iteratees (callbacks) to easily filter and map data.
- String object-path support for references nested data structures.
- 100% test coverage.
- Python 3.4+

CHAPTER 3

Quickstart

Install using pip:

```
pip3 install fnc
```

Import the main module:

```
import fnc
```

Start working with data:

```
users = [{'id': 1, 'name': 'Jack', 'email': 'jack@example.org', 'active': True},
         {'id': 2, 'name': 'Max', 'email': 'max@example.com', 'active': True},
         {'id': 3, 'name': 'Allison', 'email': 'allison@example.org', 'active': False}
↪,
         {'id': 4, 'name': 'David', 'email': 'david@example.net', 'active': False}]
```

Filter active users:

```
# Uses "matches" shorthand iteratee: dictionary
active_users = fnc.filter({'active': True}, users)
# <filter object at 0x7fa85940ec88>

active_uesrs = list(active_users)
# [{'name': 'Jack', 'email': 'jack@example.org', 'active': True},
#  {'name': 'Max', 'email': 'max@example.com', 'active': True}]
```

Get a list of email addresses:

```
# Uses "pathgetter" shorthand iteratee: string
emails = fnc.map('email', users)
# <map object at 0x7fa8577d52e8>

emails = list(emails)
# ['jack@example.org', 'max@example.com', 'allison@example.org', 'david@example.net']
```

Create a `dict` of users keyed by `'id'`:

```
# Uses "pathgetter" shorthand iteratee: string
users_by_id = fnc.keyby('id', users)
# {1: {'id': 1, 'name': 'Jack', 'email': 'jack@example.org', 'active': True},
#  2: {'id': 2, 'name': 'Max', 'email': 'max@example.com', 'active': True},
#  3: {'id': 3, 'name': 'Allison', 'email': 'allison@example.org', 'active': False},
#  4: {'id': 4, 'name': 'David', 'email': 'david@example.net', 'active': False}}
```

Select only `'id'` and `'email'` fields and return as dictionaries:

```
# Uses "pickgetter" shorthand iteratee: set
user_emails = list(fnc.map({'id', 'email'}, users))
# [{'email': 'jack@example.org', 'id': 1},
#  {'email': 'max@example.com', 'id': 2},
#  {'email': 'allison@example.org', 'id': 3},
#  {'email': 'david@example.net', 'id': 4}]
```

Select only `'id'` and `'email'` fields and return as tuples:

```
# Uses "atgetter" shorthand iteratee: tuple
user_emails = list(fnc.map(('id', 'email'), users))
# [(1, 'jack@example.org'),
#  (2, 'max@example.com'),
#  (3, 'allison@example.org'),
#  (4, 'david@example.net')]
```

Access nested data structures using object-path notation:

```
fnc.get('a.b.c[1][0].d', {'a': {'b': {'c': [None, [{'d': 100}]]}}})
# 100

# Same result but using a path list instead of a string.
fnc.get(['a', 'b', 'c', 1, 0, 'd'], {'a': {'b': {'c': [None, [{'d': 100}]]}}})
# 100
```

Compose multiple functions into a generator pipeline:

```
from functools import partial

filter_active = partial(fnc.filter, {'active': True})
get_emails = partial(fnc.map, 'email')
get_email_domains = partial(fnc.map, lambda email: email.split('@')[1])

get_active_email_domains = fnc.compose(filter_active,
                                       get_emails,
                                       get_email_domains,
                                       set)

email_domains = get_active_email_domains(users)
# {'example.com', 'example.org'}
```

Or do the same thing except using a terser "partial" shorthand:

```
get_active_email_domains = fnc.compose((fnc.filter, {'active': True}),
                                       (fnc.map, 'email'),
                                       (fnc.map, lambda email: email.split('@')[1]),
                                       set)
```

(continues on next page)

(continued from previous page)

```
email_domains = get_active_email_domains(users)
# {'example.com', 'example.org'}
```

For more details and examples, please see the full documentation at https://fnc.readthedocs.io.

CHAPTER 4

Guide

4.1 Installation

fnc requires Python >= 3.4.

To install from PyPI:

```
pip install fnc
```

4.2 API Reference

The `fnc` library is a functional-style utility library with an emphasis on using generators to process sequences of data. This allows one to easily build data processing pipelines to more efficiently munge data through function composition.

All public functions are available from the main module.

```
import fnc

fnc.<function>
```

Note: It is recommended to use the above syntax or import functions from the main module instead of importing from submodules. Future versions may change/reorganize things which could break that usage.

So what makes this library different than other function libraries for Python? Some main features to highlight are:

1. Generators when possible.
2. Shorthand iteratee support.
3. Shorthand partial function composition support.

4.2.1 Generators

By using generators, large datasets can be processed more efficiently which can enable one to build data pipelines that "push" each item of a sequence all the way through to the end before being "collected" in a final data structure. This means that these data pipelines can iterate over a sequence just once while transforming the data as it goes. These pipelines can be built through function composition (e.g. `fnc.compose` + `functools.partial`) or by simply building up the final form through successive generator passing.

4.2.2 Iteratees

The other main feature is shorthand iteratee support. But what is an iteratee? From Wikipedia (https://en.wikipedia.org/wiki/Iteratee):

> ...an iteratee is a composable abstraction for incrementally processing sequentially presented chunks of input data in a purely functional fashion.

What does that mean exactly? An iteratee is a function that is applied to each item in a sequence.

Note: All functions that accept an iteratee have the iteratee as the first argument to the function. This mirrors the Python standard library for functions like `map`, `filter`, and `reduce`. It also makes it easier to use `functools.partial` to create ad-hoc functions with bound iteratees.

Functions that accept iteratees can of course use a callable as the iteratee, but they can also accept the shorthand styles below.

Note: If iteratee shorthand styles are not your thing, each shorthand style has a corresponding higher-order function that can be used to return the same callable iteratee.

Dict

A dictionary-iteratee returns a "conforms" comparator that matches source key-values to target key-values. Typically, this iteratee is used to filter a list of dictionaries by checking if the targets are a superset of the source.

For example:

```
x = [*fnc.filter({'a': 1, 'b': 2}, [{'a': 1}, {'a': 1, 'b': 2, 'c': 3}])]
x == [{'a': 1, 'b': 2, 'c': 3}]
```

which is the same as:

```
x = list(fnc.filter(fnc.conformance({'a': 1, 'b': 2}), ...))
```

Note: When values in the dictionary-iteratee are callables, they will be treated as predicate functions that will be called with the corresponding value in the comparison target.

Set

A set-iteratee applies a "pickgetter" function to select a subset of fields from an object.

For example:

```
x = [*fnc.map({'a', 'b'}, [{'a': 1, 'b': 2, 'c': 3}, {'b': 4, 'd': 5}, {'a': 1}])]
x == [{'a': 1, 'b': 2}, {'a': None, 'b': 4}, {'a': 1, 'b': None}]
```

which is the same as:

```
x = [*fnc.map(fnc.pickgetter(['a', 'b']), ...)]

# or
from functools import partial
x = [*fnc.map(partial(fnc.pick, ['a', 'b']), ...)]
```

Tuple

A tuple-iteratee applies an "atgetter" function to return a tuple of values at the given paths.

For example:

```
x = [
    *fnc.map(
        ('a', 'b'),
        [{'a': 1, 'b': 2, 'c': 3}, {'b': 4, 'd': 5}, {'a': 1}]
    )
]
x == [(1, 2), (None, 4), (1, None)]
```

which is the same as:

```
x = [*fnc.map(fnc.atgetter(['a', 'b']), ...)]

# or
x = [*fnc.map(partial(fnc.at, ['a', 'b']), ...)]
```

List

A list-iteratee applies a "pathgetter" function to return the value at the given object path.

For example:

```
x = [
    *fnc.map(
        ['a', 'aa', 0, 'aaa'],
        [{'a': {'aa': [{'aaa': 1}]}}, {'a': {'aa': [{'aaa': 2}]}}]
    )
]
x == [1, 2]
```

which is the same as:

```
x = [*fnc.map(fnc.pathgetter(['a', 'aa', 0, 'aaa']), ...)]

# or
x = [*fnc.map(partial(fnc.get, ['a', 'aa', 0, 'aaa']), ...)]
```

String

A string-iteratee is like a list-iteratee except that an object path is represented in object-path notiation like `'a.aa[0].aaa'`.

For example:

```
x = [
    *fnc.map(
        'a.aa[0].aaa',
        [{'a': {'aa': [{'aaa': 1}]}}, {'a': {'aa': [{'aaa': 2}]}}]
    )
]
x == [1, 2]
```

which is the same as:

```
x = [*fnc.map(fnc.pathgetter('a.aa[0].aaa'), ...)]

# or
x = [*fnc.map(partial(fnc.get, 'a.aa[0].aaa'), ...)]
```

Other Values

All other non-callable values will be used in a "pathgetter" iteratee as a top-level "key" to return the object value from. Callable values will be used directly as iteratees.

Note: To reference a mapping that has a `tuple` key (e.g. {(1, 2): 'value}), use the list-iteratee like `fnc.map([(1, 2)], ...)`.

4.2.3 Function Composition

The primary method for function composition is `fnc.compose` combined with "partial" shorthand as needed.

What is "partial" shorthand? Instead of passing callables to `fnc.compose`, one can pass a tuple with the same arguments to `functools.partial`.

```
count_by_age_over21 = fnc.compose(
    (fnc.filter, {'age': lambda age: age >= 21}),
    (fnc.countby, 'age')
)

# is equivalent to...
# count_by_age_over21 = fnc.compose(
#     partial(fnc.filter, {'age': lambda age: age >= 21}),
#     partial(fnc.countby, 'age')
# )

x = count_by_age_over21(
    [
        {'age': 20},
        {'age': 21},
        {'age': 30},
```

(continues on next page)

(continued from previous page)

```
        {'age': 22},
        {'age': 21},
        {'age': 22}
    ]
)
x == {21: 2, 30: 1, 22: 2}
```

Note: The "partial" shorthand only supports invoking `functools.partial` using positional arguments. If keyword argument partials are needed, then use `functools.partial` directly.

4.2.4 Sequences

Functions that operate on sequences.

Most of these functions return generators so that they will be more efficient at processing large datasets. All generator functions will have a `Yields` section in their docstring to easily identify them as generators. Otherwise, functions that return concrete values with have a `Returns` section instead.

`fnc.sequences.`**`chunk`**(*size*, *seq*)
Split elements of *seq* into chunks with length *size* and yield each chunk.

Examples

```
>>> list(chunk(2, [1, 2, 3, 4, 5]))
[[1, 2], [3, 4], [5]]
```

Parameters

- **`seq`** (*`Iterable`*) – Iterable to chunk.
- **`size`** (*`int, optional`*) – Chunk size. Defaults to `1`.

Yields *list* – Chunked groups.

`fnc.sequences.`**`compact`**(*seq*)
Exclude elements from *seq* that are falsey.

Examples

```
>>> list(compact(['', 1, 0, True, False, None]))
[1, True]
```

Parameters **`seq`** (*`Iterable`*) – Iterable to compact.

Yields Elements that are truthy.

`fnc.sequences.`**`concat`**(**seqs*)
Concatenates zero or more iterables into a single iterable.

Examples

```
>>> list(concat([1, 2], [3, 4], [[5], [6]]))
[1, 2, 3, 4, [5], [6]]
```

Parameters ***seqs** (*Iterable*) – Iterables to concatenate.

Yields Each element from all iterables.

fnc.sequences.**countby**(*iteratee*, *seq*)

Return a `dict` composed of keys generated from the results of running each element of *seq* through the *iteratee*.

Examples

```
>>> result = countby(None, [1, 2, 1, 2, 3, 4])
>>> result == {1: 2, 2: 2, 3: 1, 4: 1}
True
>>> result = countby(lambda x: x.lower(), ['a', 'A', 'B', 'b'])
>>> result == {'a': 2, 'b': 2}
True
>>> result = countby('a', [{'a': 'x'}, {'a': 'x'}, {'a': 'y'}])
>>> result == {'x': 2, 'y': 1}
True
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to iterate over.

Returns dict

fnc.sequences.**difference**(*seq*, **seqs*)

Yields elements from *seq* that are not in *seqs*.

Note: This function is like `set.difference()` except it works with both hashable and unhashable values and preserves the ordering of the original iterables.

Examples

```
>>> list(difference([1, 2, 3], [1], [2]))
[3]
>>> list(difference([1, 4, 2, 3, 5, 0], [1], [2, 0]))
[4, 3, 5]
>>> list(difference([1, 3, 4, 1, 2, 4], [1, 4]))
[3, 2]
```

Parameters

- **seq** (*Iterable*) – Iterable to compute difference against.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Each element in *seq* that doesn't appear in *seqs*.

fnc.sequences.**differenceby**(*iteratee*, *seq*, **seqs*)

Like *difference()* except that an *iteratee* is used to modify each element in the sequences. The modified values are then used for comparison.

Note: This function is like `set.difference()` except it works with both hashable and unhashable values and preserves the ordering of the original iterables.

Examples

```
>>> list(differenceby('a',
...      [{'a': 1}, {'a': 2}, {'a': 3}], [{'a': 1}], [{'a': 2}])
... )
[{'a': 3}]
>>> list(differenceby(lambda x: x % 4, [1, 4, 2, 3, 5, 0], [1], [2, 0]))
[3]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to compute difference against.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Each element in *seq* that doesn't appear in *seqs*.

fnc.sequences.**duplicates**(*seq*, **seqs*)

Yields unique elements from *seq* that are repeated one or more times.

Note: The order of yielded elements depends on when the second duplicated element is found and not when the element first appeared.

Examples

```
>>> list(duplicates([0, 1, 3, 2, 3, 1]))
[3, 1]
>>> list(duplicates([0, 1], [3, 2], [3, 1]))
[3, 1]
```

Parameters

- **seq** (*Iterable*) – Iterable to check for duplicates.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Duplicated elements.

fnc.sequences.**duplicatesby**(*iteratee*, *seq*, **seqs*)

Like *duplicates()* except that an *iteratee* is used to modify each element in the sequences. The modified values are then used for comparison.

Examples

```
>>> list(duplicatesby('a', [{'a':1}, {'a':3}, {'a':2}, {'a':3}, {'a':1}]))
[{'a': 3}, {'a': 1}]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to check for duplicates
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Each element in *seq* that doesn't appear in *seqs*.

fnc.sequences.**filter**(*iteratee*, *seq*)

Filter *seq* by *iteratee*, yielding only the elements that the iteratee returns truthy for.

Note: This function is like the builtin `filter` except it converts *iteratee* into a fnc-style predicate.

Examples

```
>>> result = filter({'a': 1}, [{'a': 1}, {'b': 2}, {'a': 1, 'b': 3}])
>>> list(result) == [{'a': 1}, {'a': 1, 'b': 3}]
True
>>> list(filter(lambda x: x >= 3, [1, 2, 3, 4]))
[3, 4]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to filter.

Yields Filtered elements.

fnc.sequences.**find**(*iteratee*, *seq*)

Iterates over elements of *seq*, returning the first element that the iteratee returns truthy for.

Examples

```
>>> find(lambda x: x >= 3, [1, 2, 3, 4])
3
>>> find(lambda x: x >= 5, [1, 2, 3, 4]) is None
True
>>> find({'a': 1}, [{'a': 1}, {'b': 2}, {'a': 1, 'b': 2}])
{'a': 1}
>>> result = find({'a': 1}, [{'b': 2}, {'a': 1, 'b': 2}, {'a': 1}])
>>> result == {'a': 1, 'b': 2}
True
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to iterate over.

Returns First element found or `None`.

`fnc.sequences.`**findindex**(*iteratee*, *seq*)

Return the index of the element in *seq* that returns `True` for *iteratee*. If no match is found, `-1` is returned.

Examples

```
>>> findindex(lambda x: x >= 3, [1, 2, 3, 4])
2
>>> findindex(lambda x: x > 4, [1, 2, 3, 4])
-1
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to process.

Returns Index of found item or `-1` if not found.

Return type int

`fnc.sequences.`**findlast**(*iteratee*, *seq*)

This function is like *find()* except it iterates over elements of *seq* from right to left.

Examples

```
>>> findlast(lambda x: x >= 3, [1, 2, 3, 4])
4
>>> findlast(lambda x: x >= 5, [1, 2, 3, 4]) is None
True
>>> result = findlast({'a': 1}, [{'a': 1}, {'b': 2}, {'a': 1, 'b': 2}])
>>> result == {'a': 1, 'b': 2}
True
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to iterate over.

Returns Last element found or `None`.

`fnc.sequences.`**findlastindex**(*iteratee*, *seq*)

Return the index of the element in *seq* that returns `True` for *iteratee*. If no match is found, `-1` is returned.

Examples

```
>>> findlastindex(lambda x: x >= 3, [1, 2, 3, 4])
3
>>> findlastindex(lambda x: x > 4, [1, 2, 3, 4])
-1
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to process.

Returns Index of found item or `-1` if not found.

Return type int

fnc.sequences.**flatten**(**seqs*)

Flatten iterables a single level deep.

Examples

```
>>> list(flatten([[1], [2, [3]], [[4]]]))
[1, 2, [3], [4]]
>>> list(flatten([[1], [2, [3]], [[4]]], [5, [6, 7]]))
[1, 2, [3], [4], 5, 6, 7]
```

Parameters ***seqs** (*Iterables*) – Iterables to flatten.

Yields Eelements from the flattened iterable.

fnc.sequences.**flattendeep**(**seqs*)

Recursively flatten iterables.

Examples

```
>>> list(flattendeep([[1], [2, [3]], [[4]]]))
[1, 2, 3, 4]
>>> list(flattendeep([[1], [2, [3]], [[4]]], [5, [6, 7]]))
[1, 2, 3, 4, 5, 6, 7]
>>> list(flattendeep([[1], [2, [3]], [[4]]], [5, [[[[6, [[[7]]]]]]]]))
[1, 2, 3, 4, 5, 6, 7]
```

Parameters ***seqs** (*Iterables*) – Iterables to flatten.

Yields Flattened elements.

fnc.sequences.**groupall**(*iteratees*, *seq*)

This function is like *groupby()* except it supports nested grouping by multiple iteratees. If only a single iteratee is given, it is like calling *groupby()*.

Examples

```
>>> result = groupall(
...     ['shape', 'qty'],
...     [{'shape': 'square', 'color': 'red', 'qty': 5},
...      {'shape': 'square', 'color': 'blue', 'qty': 10},
...      {'shape': 'square', 'color': 'orange', 'qty': 5},
...      {'shape': 'circle', 'color': 'yellow', 'qty': 5},
...      {'shape': 'circle', 'color': 'pink', 'qty': 10},
...      {'shape': 'oval', 'color': 'purple', 'qty': 5}]
... )
>>> expected = {
...     'square': {5: [{'shape': 'square', 'color': 'red', 'qty': 5},
...                    {'shape': 'square', 'color': 'orange', 'qty': 5}],
...                10: [{'shape': 'square', 'color': 'blue', 'qty': 10}]},
...     'circle': {5: [{'shape': 'circle', 'color': 'yellow', 'qty': 5}],
...                10: [{'shape': 'circle', 'color': 'pink', 'qty': 10}]},
...     'oval': {5: [{'shape': 'oval', 'color': 'purple', 'qty': 5}]}
... }
>>> result == expected
True
```

Parameters

- **iteratees** (*Iterable*) – Iteratees to group by.
- **seq** (*Iterable*) – Iterable to iterate over.

Returns Results of recursively grouping by all *iteratees*.

Return type dict

fnc.sequences.**groupby**(*iteratee*, *seq*)

Return a `dict` composed of keys generated from the results of running each element of *seq* through the *iteratee*.

Examples

```
>>> result = groupby('a', [{'a': 1, 'b': 2}, {'a': 3, 'b': 4}])
>>> result == {1: [{'a': 1, 'b': 2}], 3: [{'a': 3, 'b': 4}]}
True
>>> result = groupby({'a': 1}, [{'a': 1, 'b': 2}, {'a': 3, 'b': 4}])
>>> result == {False: [{'a': 3, 'b': 4}], True: [{'a': 1, 'b': 2}]}
True
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to iterate over.

Returns Results of grouping by *iteratee*.

Return type dict

fnc.sequences.**intercalate**(*value*, *seq*)

Insert *value* between each element in *seq* and concatenate the results.

Examples

```
>>> list(intercalate('x', [1, [2], [3], 4]))
[1, 'x', 2, 'x', 3, 'x', 4]
>>> list(intercalate(', ', ['Lorem', 'ipsum', 'dolor']))
['Lorem', ', ', 'ipsum', ', ', 'dolor']
>>> ''.join(intercalate(', ', ['Lorem', 'ipsum', 'dolor']))
'Lorem, ipsum, dolor'
>>> list(intercalate([0,0,0], [[1,2,3],[4,5,6],[7,8,9]]))
[1, 2, 3, 0, 0, 0, 4, 5, 6, 0, 0, 0, 7, 8, 9]
```

Parameters

- **value** (*object*) – Element to insert.
- **seq** (*Iterable*) – Iterable to intercalate.

Yields Elements of the intercalated iterable.

fnc.sequences.**interleave**(**seqs*)

Merge multiple iterables into a single iterable by inserting the next element from each iterable by sequential round-robin.

Examples

```
>>> list(interleave([1, 2, 3], [4, 5, 6], [7, 8, 9]))
[1, 4, 7, 2, 5, 8, 3, 6, 9]
```

Parameters ***seqs** (*Iterable*) – Iterables to interleave.

Yields Elements of the interleaved iterable.

fnc.sequences.**intersection**(*seq*, **seqs*)

Computes the intersection of all the passed-in iterables.

Note: This function is like `set.intersection()` except it works with both hashable and unhashable values and preserves the ordering of the original iterables.

Examples

```
>>> list(intersection([1, 2, 3], [1, 2, 3, 4, 5], [2, 3]))
[2, 3]
>>> list(intersection([1, 2, 3]))
[1, 2, 3]
```

Parameters

- **seq** (*Iterable*) – Iterable to compute intersection against.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Elements that itersect.

fnc.sequences.**intersectionby**(*iteratee*, *seq*, **seqs*)
Like *intersection()* except that an *iteratee* is used to modify each element in the sequences. The modified values are then used for comparison.

Note: This function is like `set.intersection()` except it works with both hashable and unhashable values and preserves the ordering of the original iterables.

Examples

```
>>> list(intersectionby('a',
...      [{'a': 1}, {'a': 2}, {'a': 3}],
...      [{'a': 1}, {'a': 2}, {'a': 3}, {'a': 4}, {'a': 5}],
...      [{'a': 2}, {'a': 3}]
... ))
[{'a': 2}, {'a': 3}]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to compute intersection against.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Elements that itersect.

fnc.sequences.**intersperse**(*value*, *seq*)
Insert a separating element between each element in *seq*.

Examples

```
>>> list(intersperse('x', [1, [2], [3], 4]))
[1, 'x', [2], 'x', [3], 'x', 4]
```

Parameters

- **value** (*object*) – Element to insert.
- **seq** (*Iterable*) – Iterable to intersperse.

Yields Elements of the interspersed iterable.

fnc.sequences.**keyby**(*iteratee*, *seq*)
Return a `dict` composed of keys generated from the results of running each element of *seq* through the *iteratee*.

Examples

```
>>> results = keyby('a', [{'a': 1, 'b': 2}, {'a': 3, 'b': 4}])
>>> results == {1: {'a': 1, 'b': 2}, 3: {'a': 3, 'b': 4}}
True
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to iterate over.

Returns Results of indexing by *iteratee*.

Return type dict

fnc.sequences.**map**(*iteratee*, **seqs*)

Map *iteratee* to each element of iterable and yield the results. If additional iterable arguments are passed, *iteratee* must take that many arguments and is applied to the items from all iterables in parallel.

Note: This function is like the builtin `map` except it converts *iteratee* into a fnc-style predicate.

Examples

```
>>> list(map(str, [1, 2, 3, 4]))
['1', '2', '3', '4']
>>> list(map('a',
...      [{'a': 1, 'b': 2}, {'a': 3, 'b': 4}, {'a': 5, 'b': 6}]
... ))
[1, 3, 5]
>>> list(map('0.1', [[[0, 1]], [[2, 3]], [[4, 5]]]))
[1, 3, 5]
>>> list(map('a.b', [{'a': {'b': 1}}, {'a': {'b': 2}}]))
[1, 2]
>>> list(map('a.b[1]', [{'a': {'b': [0, 1]}}, {'a': {'b': [2, 3]}}]))
[1, 3]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- ***seqs** (*Iterable*) – Iterables to map.

Yields Mapped elements.

fnc.sequences.**mapcat**(*iteratee*, **seq*)

Map an *iteratee* to each element of each iterable in *seqs* and concatenate the results into a single iterable.

Examples

```
>>> list(mapcat(lambda x: list(range(x)), range(4)))
[0, 0, 1, 0, 1, 2]
```

Parameters

- **iteratee** (*object*) – Iteratee to apply to each element.
- **seq** (*Iterable*) – Iterable to map and concatenate.

Yields Elements resulting from concat + map operations.

fnc.sequences.**mapflat**(*iteratee*, **seqs*)

Map an *iteratee* to each element of each iterable in *seqs* and flatten the results.

Examples

```
>>> list(mapflat(lambda n: [[n, n]], [1, 2]))
[[1, 1], [2, 2]]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- ***seqs** (*Iterable*) – Iterables to iterate over.

Yields Elements result from flatten + map operations.

fnc.sequences.**mapflatdeep**(*iteratee*, **seqs*)
Map an *iteratee* to each element of each iterable in *seqs* and recurisvely flatten the results.

Examples

```
>>> list(mapflatdeep(lambda n: [[n, n]], [1, 2]))
[1, 1, 2, 2]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- ***seqs** (*Iterable*) – Iterables to iterate over.

Yields Elements result from recursive flatten + map operations.

fnc.sequences.**partition**(*iteratee*, *seq*)
Return a `tuple` of 2 lists containing elements from *seq* split into two groups where the first group contains all elements the *iteratee* returned truthy for and the second group containing the falsey elements.

Examples

```
>>> partition(lambda x: x % 2, [1, 2, 3, 4])
([1, 3], [2, 4])
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to iterate over.

Returns tuple[list]

fnc.sequences.**reject**(*iteratee*, *seq*)
The opposite of *filter()* this function yields the elements of *seq* that the *iteratee* returns falsey for.

Examples

```
>>> list(reject(lambda x: x >= 3, [1, 2, 3, 4]))
[1, 2]
>>> list(reject('a', [{'a': 0}, {'a': 1}, {'a': 2}]))
[{'a': 0}]
>>> list(reject({'a': 1}, [{'a': 0}, {'a': 1}, {'a': 2}]))
[{'a': 0}, {'a': 2}]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to iterate over.

Yields Rejected elements.

fnc.sequences.**union**(*seq*, **seqs*)

Computes the union of the passed-in iterables (sometimes referred to as `unique`).

Note: This function is like `set.union()` except it works with both hashable and unhashable values and preserves the ordering of the original iterables.

Examples

```
>>> list(union([1, 2, 3, 1, 2, 3]))
[1, 2, 3]
>>> list(union([1, 2, 3], [2, 3, 4], [3, 4, 5]))
[1, 2, 3, 4, 5]
```

Parameters

- **seq** (*Iterable*) – Iterable to compute union against.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Each unique element from all iterables.

fnc.sequences.**unionby**(*iteratee*, *seq*, **seqs*)

Like *union()* except that an *iteratee* is used to modify each element in the sequences. The modified values are then used for comparison.

Note: This function is like `set.union()` except it works with both hashable and unhashable values and preserves the ordering of the original iterables.

Examples

```
>>> list(unionby('a',
...      [{'a': 1}, {'a': 2}, {'a': 3}, {'a': 1}, {'a': 2}, {'a': 3}]
... ))
[{'a': 1}, {'a': 2}, {'a': 3}]
```

Parameters

- **iteratee** (*object*) – Iteratee applied per iteration.
- **seq** (*Iterable*) – Iterable to compute union against.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Each unique element from all iterables.

fnc.sequences.**unzip**(*seq*)

The inverse of the builtin `zip` function, this method transposes groups of elements into new groups composed of elements from each group at their corresponding indexes.

Examples

```
>>> list(unzip([(1, 4, 7), (2, 5, 8), (3, 6, 9)]))
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
>>> list(unzip(unzip([(1, 4, 7), (2, 5, 8), (3, 6, 9)])))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Parameters **seq** (*Iterable*) – Iterable to unzip.

Yields *tuple* – Each transposed group.

fnc.sequences.**without**(*values*, *seq*)

Exclude elements in *seq* that are in *values*.

Examples

```
>>> list(without([2, 4], [1, 2, 3, 2, 4, 4, 3]))
[1, 3, 3]
```

Parameters

- **array** (*list*) – List to filter.
- **values** (*mixed*) – Values to remove.

Yields Elements not in *values*.

fnc.sequences.**xor**(*seq*, **seqs*)

Computes the symmetric difference of the provided iterables where the elements are only in one of the iteralbes.

Note: This function is like `set.symmetric_difference()` except it works with both hashable and unhashable values and preserves the ordering of the original iterables.

Warning: While this function returns a generator object, internally it will create intermediate non-generator iterables which may or may not be a performance concern depending on the sizes of the inputs.

Examples

```
>>> list(xor([1, 3, 4], [1, 2, 4], [2]))
[3]
```

Parameters

- **seq** (*Iterable*) – Iterable to compute symmetric difference against.
- ***seqs** (*Iterable*) – Other iterables to compare with.

Yields Elements from the symmetric difference.

4.2.5 Mappings

Functions that operate on mappings.

A mapping includes dictionaries, lists, strings, `collections.abc.Mapping` and `collections.abc.Sequence` subclasses, and other mapping-like objects that either have an `items()` method, have `keys()` and `__getitem__` methods, or have an `__iter__()` method. For functions that use *get()*, non-mapping object values can be selected from class attributes.

`fnc.mappings.`**`at`**(*paths*, *obj*)
Creates a `tuple` of elements from *obj* at the given *paths*.

Examples

```
>>> at(['a', 'c'], {'a': 1, 'b': 2, 'c': 3, 'd': 4})
(1, 3)
>>> at(['a', ['c', 'd', 'e']], {'a': 1, 'b': 2, 'c': {'d': {'e': 3}}})
(1, 3)
>>> at(['a', 'c.d.e[0]'], {'a': 1, 'b': 2, 'c': {'d': {'e': [3]}}})
(1, 3)
>>> at([0, 2], [1, 2, 3, 4])
(1, 3)
```

Parameters

- **collection** (*Iterable*) – Iterable to pick from.
- **paths** (*Iterable*) – The object paths to pick.

Returns tuple

`fnc.mappings.`**`defaults`**(**objs*)
Create a `dict` extended with the key-values from the provided dictionaries such that keys are set once and not overridden by subsequent dictionaries.

Examples

```
>>> obj = defaults(
...     {'a': 1}, {'b': 2}, {'c': 3, 'b': 5}, {'a': 4, 'c': 2}
... )
```

(continues on next page)

(continued from previous page)

```
>>> obj == {'a': 1, 'b': 2, 'c': 3}
True
```

Parameters ***objs** (*dict*) – Dictionary sources.

Returns dict

fnc.mappings.**get**(*path*, *obj*, ***, *default=None*)

Get the *path* value at any depth of an object. If path doesn't exist, *default* is returned.

Examples

```
>>> get('a.b.c', {}) is None
True
>>> get('a.b.c[1]', {'a': {'b': {'c': [1, 2, 3, 4]}}})
2
>>> get('a.b.c.1', {'a': {'b': {'c': [1, 2, 3, 4]}}})
2
>>> get('a.b.1.c[1]', {'a': {'b': [0, {'c': [1, 2]}]}})
2
>>> get(['a', 'b', 1, 'c', 1], {'a': {'b': [0, {'c': [1, 2]}]}})
2
>>> get('a.b.1.c.2', {'a': {'b': [0, {'c': [1, 2]}]}}, default=False)
False
```

Parameters

- **path** (*object*) – Path to test for. Can be a key value, list of keys, or a `.` delimited path-string.
- **obj** (*Mapping*) – Object to process.

Keyword Arguments **default** (*mixed*) – Default value to return if path doesn't exist. Defaults to `None`.

Returns Value of *obj* at path.

Return type object

fnc.mappings.**has**(*path*, *obj*)

Return whether *path* exists in *obj*.

Examples

```
>>> has(1, [1, 2, 3])
True
>>> has('b', {'a': 1, 'b': 2})
True
>>> has('c', {'a': 1, 'b': 2})
False
>>> has('a.b[1].c[1]', {'a': {'b': [0, {'c': [1, 2]}]}})
True
>>> has('a.b.1.c.2', {'a': {'b': [0, {'c': [1, 2]}]}})
False
```

Parameters

- **path** (*object*) – Path to test for. Can be a key value, list of keys, or a `.` delimited path-string.
- **obj** (*Mapping*) – Object to test.

Returns Whether *obj* has *path*.

Return type bool

fnc.mappings.**invert**(*obj*)

Return a `dict` composed of the inverted keys and values of the given dictionary.

Note: It's assumed that *obj* values are hashable as `dict` keys.

Examples

```
>>> result = invert({'a': 1, 'b': 2, 'c': 3})
>>> result == {1: 'a', 2: 'b', 3: 'c'}
True
```

Parameters **obj** (*Mapping*) – Mapping to invert.

Returns Inverted dictionary.

Return type dict

fnc.mappings.**mapkeys**(*iteratee*, *obj*)

Return a `dict` with keys from *obj* mapped with *iteratee* while containing the same values.

Examples

```
>>> result = mapkeys(lambda k: k * 2, {'a': 1, 'b': 2, 'c': 3})
>>> result == {'aa': 1, 'bb': 2, 'cc': 3}
True
```

Parameters

- **iteratee** (*object*) – Iteratee applied to each key.
- **obj** (*Mapping*) – Mapping to map.

Returns Dictionary with mapped keys.

Return type dict

fnc.mappings.**mapvalues**(*iteratee*, *obj*)

Return a `dict` with values from *obj* mapped with *iteratee* while containing the same keys.

Examples

```
>>> result = mapvalues(lambda v: v * 2, {'a': 1, 'b': 2, 'c': 3})
>>> result == {'a': 2, 'b': 4, 'c': 6}
True
>>> result = mapvalues({'d': 4}, {'a': 1, 'b': {'d': 4}, 'c': 3})
>>> result == {'a': False, 'b': True, 'c': False}
True
```

Parameters

- **iteratee** (*object*) – Iteratee applied to each key.
- **obj** (*Mapping*) – Mapping to map.

Returns Dictionary with mapped values.

Return type dict

fnc.mappings.**merge**(**objs*)

Create a `dict` merged with the key-values from the provided dictionaries such that each next dictionary extends the previous results.

Examples

```
>>> obj = merge({'a': 0}, {'b': 1}, {'b': 2, 'c': 3}, {'a': 1})
>>> obj == {'a': 1, 'b': 2, 'c': 3}
True
```

Parameters ***objs** (*dict*) – Dictionary sources.

Returns dict

fnc.mappings.**omit**(*keys*, *obj*)

The opposite of *pick()*. This method creates an object composed of the property paths of *obj* that are not omitted.

Examples

```
>>> omit(['a', 'c'], {'a': 1, 'b': 2, 'c': 3 }) == {'b': 2}
True
>>> omit([0, 3], ['a', 'b', 'c', 'd']) == {1: 'b', 2: 'c'}
True
```

Parameters

- **keys** (*Iterable*) – Keys to omit.
- **obj** (*Mapping*) – Object to process.

Returns Dictionary with *keys* omitted.

Return type dict

fnc.mappings.**pick**(*keys*, *obj*)
Create a `dict` composed of the picked *keys* from *obj*.

Examples

```
>>> pick(['a', 'b'], {'a': 1, 'b': 2, 'c': 3}) == {'a': 1, 'b': 2}
True
>>> pick(['a', 'b'], {'b': 2}) == {'b': 2}
True
```

Parameters

- **keys** (*Iterable*) – Keys to omit.
- **obj** (*Mapping*) – Object to process.

Returns Dict containg picked properties.

Return type dict

4.2.6 Utilities

General utility functions.

fnc.utilities.**after**(*method*)
Decorator that calls *method* after the decorated function is called.

Examples

```
>>> def a(): print('a')
>>> def b(): print('b')
>>> after(a)(b)()
b
a
```

Parameters **method** (*callable*) – Function to call afterwards.

fnc.utilities.**aspath**(*value*)
Converts value to an object path list.

Examples

```
>>> aspath('a.b.c')
['a', 'b', 'c']
>>> aspath('a.0.0.b.c')
['a', '0', '0', 'b', 'c']
>>> aspath('a[0].b.c')
['a', '0', 'b', 'c']
>>> aspath('a[0][1][2].b.c')
['a', '0', '1', '2', 'b', 'c']
>>> aspath('[a][0][1][2][b][c]')
```

(continues on next page)

(continued from previous page)

```
['a', '0', '1', '2', 'b', 'c']
>>> aspath('a.[]')
['a', '']
>>> aspath(0)
[0]
>>> aspath([0, 1])
[0, 1]
>>> aspath((0, 1))
[(0, 1)]
```

Parameters **value** (*object*) – Value to convert.

Returns Returns property paths.

Return type list

fnc.utilities.**atgetter**(*paths*)

Creates a function that returns the values at paths of a given object.

Examples

```
>>> get_id_name = atgetter(['data.id', 'data.name'])
>>> get_id_name({'data': {'id': 1, 'name': 'foo'}})
(1, 'foo')
```

Parameters **paths** (*Iterable*) – Path values to fetch from object.

Returns Function like `f(obj): fnc.at(paths, obj)`.

Return type function

fnc.utilities.**before**(*method*)

Decorator that calls *method* before the decorated function is called.

Examples

```
>>> def a(): print('a')
>>> def b(): print('b')
>>> before(a)(b)()
a
b
```

Parameters **method** (*callable*) – Function to call afterwards.

fnc.utilities.**compose**(**funcs*)

Create a function that is the composition of the provided functions, where each successive invocation is supplied the return value of the previous. For example, composing the functions `f()`, `g()`, and `h()` produces `h(g(f()))`.

Note: Each element in *funcs* can either be a callable or a `tuple` where the first element is a callable and the remaining elements are partial arguments. The tuples will be converted to a callable using `functools.partial(*func)`.

Note: The "partial" shorthand only supports invoking `functools.partial` using positional arguments. If keyword argument partials are needed, then use `functools.partial` directly.

Examples

```
>>> mult_5 = lambda x: x * 5
>>> div_10 = lambda x: x / 10.0
>>> pow_2 = lambda x: x ** 2
>>> mult_div_pow = compose(sum, mult_5, div_10, pow_2)
>>> mult_div_pow([1, 2, 3, 4])
25.0
>>> sum_positive_evens = compose(
...     (filter, lambda x: x > 0),
...     (filter, lambda x: x % 2 == 0),
...     sum
... )
>>> sum_positive_evens([-1, 1, 2, 3, -5, 0, 6])
8
```

Parameters ***funcs** (*function*) – Function(s) to compose. If *func* is a tuple, then it will be converted into a partial using `functools.partial(*func)`.

Returns Composed function.

Return type function

`fnc.utilities.`**`conformance`**(*source*)

Creates a function that does a shallow comparison between a given object and the *source* dictionary using *conforms()*.

Examples

```
>>> conformance({'a': 1})({'b': 2, 'a': 1})
True
>>> conformance({'a': 1})({'b': 2, 'a': 2})
False
```

Parameters **source** (*dict*) – Source object used for comparision.

Returns function

`fnc.utilities.`**`conforms`**(*source*, *target*)

Return whether the *target* object conforms to *source* where *source* is a dictionary that contains key-value pairs which are compared against the same key-values in *target*. If a key-value in *source* is a callable, then that callable is used as a predicate against the corresponding key-value in *target*.

Examples

```
>>> conforms({'b': 2}, {'a': 1, 'b': 2})
True
>>> conforms({'b': 3}, {'a': 1, 'b': 2})
False
>>> conforms({'b': 2, 'a': lambda a: a > 0}, {'a': 1, 'b': 2})
True
>>> conforms({'b': 2, 'a': lambda a: a > 0}, {'a': -1, 'b': 2})
False
```

Parameters

- **source** (*object*) – Object of path values to match.
- **target** (*object*) – Object to compare.

Returns Whether *target* is a match or not.

Return type bool

fnc.utilities.**constant**(*value*)

Creates a function that returns a constant *value*.

Examples

```
>>> pi = constant(3.14)
>>> pi()
3.14
```

Parameters **value** (*object*) – Constant value to return.

Returns Function that always returns *value*.

Return type function

fnc.utilities.**identity**(*value=None*, **args*, ***kwargs*)

Return the first argument provided.

Examples

```
>>> identity(1)
1
>>> identity(1, 2, 3)
1
>>> identity(1, 2, 3, a=4)
1
>>> identity() is None
True
```

Parameters **value** (*object, optional*) – Value to return. Defaults to `None`.

Returns First argument or `None`.

Return type object

fnc.utilities.**iteratee**(*obj*)

Return iteratee function based on the type of *obj*.

The iteratee object can be one of the following:

- `callable`: Return as-is.
- `None`: Return *`identity()`* function.
- `dict`: Return `conformance(obj)()` function.
- `set`: Return `pickgetter(obj)()` function.
- `tuple`: Return `atgetter(obj)`()` function.
- otherwise: Return `pathgetter(obj)`()` function.

Note: In most cases, this function won't need to be called directly since other functions that accept an iteratee will call this function internally.

Examples

```
>>> iteratee(lambda a, b: a + b)(1, 2)
3
>>> iteratee(None)(1, 2, 3)
1
>>> is_active = iteratee({'active': True})
>>> is_active({'active': True})
True
>>> is_active({'active': 0})
False
>>> iteratee({'a': 5, 'b.c': 1})({'a': 5, 'b': {'c': 1}})
True
>>> iteratee({'a', 'b'})({'a': 1, 'b': 2, 'c': 3}) == {'a': 1, 'b': 2}
True
>>> iteratee(('a', ['c', 'd', 'e']))({'a': 1, 'c': {'d': {'e': 3}}})
(1, 3)
>>> iteratee(['c', 'd', 'e'])({'a': 1, 'c': {'d': {'e': 3}}})
3
>>> get_data = iteratee('data')
>>> get_data({'data': [1, 2, 3]})
[1, 2, 3]
>>> iteratee(['a.b'])({'a.b': 5})
5
>>> iteratee('a.b')({'a': {'b': 5}})
5
```

Parameters **obj** (*object*) – Object to convert into an iteratee.

Returns Iteratee function.

Return type function

fnc.utilities.**negate**(*func*)

Creates a function that negates the result of the predicate *func*.

Examples

```
>>> not_number = negate(lambda x: isinstance(x, (int, float)))
>>> not_number(1)
False
>>> not_number('1')
True
```

Parameters **func** (*callabe*) – Function to negate.

Returns function

fnc.utilities.**noop**(**args*, ***kwargs*)
A no-operation function.

Examples

```
>>> noop(1, 2, 3) is None
True
```

fnc.utilities.**over**(**funcs*)
Creates a function that calls each function with the provided arguments and returns the results as a `tuple`.

Example

```
>>> minmax = over(min, max)
>>> minmax([1, 2, 3, 4])
(1, 4)
```

Parameters ***funcs** (*callable*) – Functions to call.

Returns Results from each function call

Return type tuple

fnc.utilities.**overall**(**funcs*)
Creates a function that returns `True` when all of the given functions return true for the provided arguments.

Example

```
>>> is_bool = overall(
...     lambda v: isinstance(v, bool),
...     lambda v: v is not 0 and v is not 1
... )
>>> is_bool(False)
True
>>> is_bool(0)
False
```

Parameters ***funcs** (*callable*) – Functions to call.

Returns Whether call functions evaulate to true.

Return type bool

fnc.utilities.**overany**(**funcs*)

Creates a function that returns `True` when any of the given functions return true for the provided arguments.

Example

```
>>> is_bool_like = overany(
...     lambda v: isinstance(v, bool),
...     lambda v: v in [0, 1]
... )
>>> is_bool_like(False)
True
>>> is_bool_like(0)
True
```

Parameters ***funcs** (*callable*) – Functions to call.

Returns Whether call functions evaulate to true.

Return type bool

fnc.utilities.**pathgetter**(*path*, *default=None*)

Creates a function that returns the value at path of a given object.

Examples

```
>>> get_data = pathgetter('data')
>>> get_data({'data': 1})
1
>>> get_data({}) is None
True
>>> get_first = pathgetter(0)
>>> get_first([1, 2, 3])
1
>>> get_nested = pathgetter('data.items')
>>> get_nested({'data': {'items': [1, 2]}})
[1, 2]
```

Parameters **path** (*str|list*) – Path value to fetch from object.

Returns Function like `f(obj): fnc.get(path, obj)`.

Return type function

fnc.utilities.**pickgetter**(*keys*)

Creates a function that returns the value at path of a given object.

Examples

```
>>> pick_ab = pickgetter(['a', 'b'])
>>> pick_ab({'a': 1, 'b': 2, 'c': 4}) == {'a': 1, 'b': 2}
True
```

Parameters **keys** (*Iterable*) – Keys to fetch from object.

Returns Function like `f(obj): fnc.pick(keys, obj)`.

Return type function

fnc.utilities.**random**(*start=0*, *stop=1*, *floating=False*)

Produces a random number between *start* and *stop* (inclusive). If only one argument is provided a number between 0 and the given number will be returned. If floating is truthy or either *start* or *stop* are floats a floating-point number will be returned instead of an integer.

Parameters

- **start** (*int*) – Minimum value.
- **stop** (*int*) – Maximum value.
- **floating** (*bool, optional*) – Whether to force random value to `float`. Default is `False`.

Returns Random value.

Return type int|float

Example

```
>>> 0 <= random() <= 1
True
>>> 5 <= random(5, 10) <= 10
True
>>> isinstance(random(floating=True), float)
True
```

fnc.utilities.**retry**(*attempts=3*, ***, *delay=0.5*, *max_delay=150.0*, *scale=2.0*, *jitter=0*, *exceptions=(<class 'Exception'>,)*, *on_exception=None*)

Decorator that retries a function multiple times if it raises an exception with an optional delay between each attempt. When a *delay* is supplied, there will be a sleep period in between retry attempts. The first delay time will always be equal to *delay*. After subsequent retries, the delay time will be scaled by *scale* up to *max_delay*. If *max_delay* is `0`, then *delay* can increase unbounded.

Parameters

- **attempts** (*int, optional*) – Number of retry attempts. Defaults to `3`.
- **delay** (*int|float, optional*) – Base amount of seconds to sleep between retry attempts. Defaults to `0.5`.
- **max_delay** (*int|float, optional*) – Maximum number of seconds to sleep between retries. Is ignored when equal to `0`. Defaults to `150.0` (2.5 minutes).
- **scale** (*int|float, optional*) – Scale factor to increase *delay* after first retry fails. Defaults to `2.0`.

- **jitter** (*int|float|tuple, optional*) – Random jitter to add to *delay* time. Can be a positive number or 2-item tuple of numbers representing the random range to choose from. When a number is given, the random range will be from `[0, jitter]`. When jitter is a float or contains a float, then a random float will be chosen; otherwise, a random integer will be selected. Defaults to `0` which disables jitter.
- **exceptions** (*tuple, optional*) – Tuple of exceptions that trigger a retry attempt. Exceptions not in the tuple will be ignored. Defaults to `(Exception,)` (all exceptions).
- **on_exception** (*function, optional*) – Function that is called when a retryable exception is caught. It is invoked with `on_exception(exc, attempt)` where `exc` is the caught exception and `attempt` is the attempt count. All arguments are optional. Defaults to `None`.

Example

```
>>> @retry(attempts=3, delay=0)
... def do_something():
...     print('something')
...     raise Exception('something went wrong')
>>> try: do_something()
... except Exception: print('caught something')
something
something
something
caught something
```

CHAPTER 5

Project Info

5.1 License

5.2 Versioning

This project follows Semantic Versioning with the following caveats:

- Only the public API (i.e. the objects imported into the fnc module) will maintain backwards compatibility between MINOR version bumps.
- Objects within any other parts of the library are not guaranteed to not break between MINOR version bumps.

With that in mind, it is recommended to only use or import objects from the main module, fnc.

5.3 Changelog

5.3.1 v0.4.0 (2019-01-23)

- Add functions:
  - `differenceby`
  - `duplicatesby`
  - `intersectionby`
  - `unionby`

5.3.2 v0.3.0 (2018-08-31)

- compose: Introduce new "partial" shorthand where instead of passing a callable, a `tuple` can be given which will then be converted to a callable using `functools.partial`. For example, instead of `fnc.compose(partial(fnc.filter, {'active': True}), partial(fnc.map, 'email'))`, one can do `fnc.compose((fnc.filter, {'active': True}), (fnc.map, 'email'))`.

5.3.3 v0.2.0 (2018-08-24)

- Add functions:
  - `negate`
  - `over`
  - `overall`
  - `overany`
- Rename functions: (**breaking change**)
  - `ismatch -> conforms`
  - `matches -> conformance`
- Make `conforms/conformance` (formerly `ismatch/matches`) accept callable dictionary values that act as predicates against comparison target. (**breaking change**)

5.3.4 v0.1.1 (2018-08-17)

- pick: Don't return `None` for keys that don't exist in source object. Instead of `fnc.pick(['a'], {}) == {'a': None}`, it's now `fnc.pick(['a'], {}) == {}`.

5.3.5 v0.1.0 (2018-08-15)

- First release.

5.4 Authors

5.4.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

5.4.2 Contributors

None

5.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/dgilland/fnc.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" or "help wanted" is open to whoever wants to implement it.

Write Documentation

fnc could always use more documentation, whether as part of the official fnc docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dgilland/fnc.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.5.2 Get Started!

Ready to contribute? Here's how to set up `fnc` for local development.

1. Fork the `fnc` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_username_here/fnc.git
```

3. Install Python dependencies into a virtualenv:

```
$ cd fnc
$ pip install -r requirements.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. Autoformat code using black:

```
$ tox -e black
```

6. When you're done making changes, check that your changes pass linting and all unit tests by testing with tox across all supported Python versions:

```
$ tox
```

7. Add yourself to `AUTHORS.rst`.
8. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

9. Submit a pull request through GitHub.

5.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.

2. The pull request should work for all versions Python that this project supports. Check https://travis-ci.org/dgilland/fnc/pull_requests and make sure that the all environments pass.

CHAPTER 6

Indices and Tables

- genindex
- modindex
- search

Python Module Index

f

Index

P

R

U

W

X