configfetch Documentation

Open Close

Feb 04, 2019

Contents:

Helper to get values from *configparser* and *argparse*.

CHAPTER 1

Overview

To read and prepare miscellaneous configuration data is the common process that most commandline scripts have to handle, and there are many libraries which try to facilitate that, extending python argparse or configparser somewhat.

This is one of them.

The characteristics are:

- I chose `configparser` extension path. So it can only handle `INI` format.
- But it also parses a customized `INI` format with one extension, to register value conversion functions for each option fields.
- So that the using scripts can provide, in a static file, actual data (application defaults) and data definitions at the same time. (this is the point, how humble it is.)
- All data accesses are done by `dot access` or `.get()` method.
- Config file data are automatically overridden by environment variables, which are overridden by commandline arguments.

—

Let's call the customized `INI` format as `fetch-INI` or `FINI` format. And let's differentiate the resulting custom config object, using the word `conf`, as opposed to the ordinary ConfigParser-like object `config`.

1.1 Installation

It is a single file Python module, with no other library dependency.

Only **Python 3.5 and above** are supported.

```
pip install configfetch
```

1.2 Usage

```
## myapp.ini
[main]
gui=         [=BOOL] no
users=       [=COMMA] Alice, Bob, Charlie
filetype=    html

# terminal
>>> import configfetch
>>> conf = configfetch.fetch('myapp.ini')
>>> conf.main.gui
False
>>> conf.main.users
['Alice', 'Bob', 'Charlie']
>>> conf.filetype
'html'
```

As you see, the `FINI` file has optional `[=SOMETHING]` notations before each option value, which registers `_something()` function for this particular option.

When you parse `FINI` definition files, you can use `.fetch()` convenience function like here.

After that, following config files should be written in ordinary `INI` format (user configs).

Only `.read()` is exposed. But `conf` keeps ordinary `INI ConfigParser` object as `._config` (stripped of `[=SOMETHING]` parts), so you can use other read methods if you want.

```
>>> conf.read('user.ini')
>>> conf._config.read_string('[main]\ngui=yes')
```

1.3 Limitations

Manual Arguments Building

While the script can *parse* commandline arguments automatically, it is the same before, in that you have to manually register each argument with `ArgumentParser`, e.g.:

```
parser.add_argument('--filetype')
```

ConfigParser.converters

ConfigParser already has a value conversion mechanism (.converters). In this, it is the caller who has to designate appropriate functions, `.getint()`, `.getsomething()`, etc. Maybe this is good enough for many cases.

Function Arguments

Users can customize their own functions. But precisely because our only interfaces are `dot access` and `.get()`, functions can't accept arguments other than miscellaneous internal config values. So they need some workarounds in that case. (e.g. create custom functions with arguments already internalized.)

1.4 Value Selection

`ArgumentParser` options and environment variable keys (`args` and `envs`) are always global, searched for in every section lookup.

1.4.1 Section

In the first access on `conf`, a section representation object (`SectionFetch`) is returned. `args` and `envs` are not involved.

`dot access (.__getattr__(section))` raise NoSectionError, when the section is not found.

`.get(section)` return `None`, when the section is not found.

1.4.2 Option

In the option access (the first access on `SectionFetch`), `args`, `envs`, and the section `section` are searched in order, and the first valid one is *selected* (but not yet *returned*).

If `args` has the key, and the value is not `None`, it is selected (`arg`).

(Note other non-values (`''`, `[]` or `False`) are selected.)

If `envs` has the key, and the value is not `''`, it is selected (`env`).

If `section` (or `Default section`) has the key, the value is selected (`opt`).

Otherwise:

`dot access (.__getattr__(option))` raise NoOptionError

`.get(option, fallback=_UNSET)` raise NoOptionError, when `fallback` is not provided (`_UNSET`). Otherwise, `fallback` is selected.

1.4.3 Nonstring

If the selected value is `arg`, and it is not a string, the value is *returned* as is. (`env` and `opt` are always a string.)

So `ArgumentParser` arguments that convert the value type are just passed through.

1.4.4 ArgumentParser Details

Normally is is better not to supply `default` argument of `ArgumentParser.add_argument()`. If it is supplied, `arg` is always selected. Either the value in the commandline, or the default value.

Also take note that `store_true` and `store_false` actions default to `False` and `True` respectively. They are always selected, and in their case, always returned. (above Nonstring rule).

If this is not desirable, use `store_const` instead. E.g.:

```
parser.add_argument('--gui', action='store_const', const='true')
```

(Cf. Paul Jacobson (hpaulj) discourages `store_true` and `store_false` in a different context. See Python argparse –toggle –no-toggle flag.)

In most cases, you can delegate conversion to `conf`, by conforming to the designated `FINI` format. E.g.

```
# myapp.ini
file=   [COMMA] a.txt, b.txt, c.txt
```

```
parser.add_argument('--file', action='store')
```

```
$ myapp.py --file 'a.txt, b.txt, c.txt'
```

instead of:

```
parser.add_argument('--file', action='store', nargs='+')
```

```
$ myapp.py --file a.txt b.txt c.txt
```

or:

```
parser.add_argument('--file', action='append')
```

```
$ myapp.py --file a.txt --file b.txt --file c.txt
```

1.4.5 Conversion

The selected value is passed to the function conversion check.

If no function is registered, the value is *returned*.

If functions are registered, the value is applied to each function, left to right in order, then the resultant value is *returned*.

1.4.6 Function

Function names must start with `'_'`.

The matching string is made from stripping this `'_'`, and converting to uppercase. E.g. `_something()` to `[=SOMETHING]`.

Functions always have one argument `value`, that is a *selected* value. And they return one value. It either *returns* to the caller as the end result, or is used as the `value` of the next function, if any.

Functions can also access `values`, the original three elements list before selection (`[arg, env, opt]`). Use `Func.values` or `self.values` attribute.

1.4.7 Concatenation

The first function must accept raw string value (initial `value`) as its `value` argument.

The second function and after may define any value type for its `value` argument.

But what actually comes as `value` is, of course, dependent on the previous function.

So in general users should follow the concatenation rules each function expects.

1.5 Structure

The main constructs of this module are:

class `ConfigLoad` load `FINI` format file, and create ordinary `INI` data object and corresponding context (option-function map) object. Both are actually `config` objects.

class `ConfigFetch` from above two objects, create actual `conf` interface object.

class `Func` keep conversion functions and apply them to values.

When user want to create new functions, use this class.

function `fetch()` shortcut. Using `ConfigLoad` and `ConfigFetch`, create actual `conf` object.

1.5.1 ConfigLoad

```
class configfetch.ConfigLoad(
    *args, cfile=None, parser=configparser.ConfigParser,
    use_dash=True, use_uppercase=True, **kwargs)
```

It accepts (hopefully) all `configparser.ConfigParser.__init__()` arguments. And some keyword arguments are added.

cfile the name of `FINI` format file, or literal string to read (required).

parser `ConfigParser` like object to actually generate `INI` format object. Default is `configparser.ConfigParser`.

use_dash Default is `True`. This module uses `dot access` for all section and option lookup, so you have to choose their names as valid identifiers (`[a-zA-Z_][a-zA-Z0-9_]+`). Additionally, dash (`'-'`) can be used for options, by converting it to underline (`'_'`) internally, if this argument is `True`.

Note `argparse` does this for its own arguments, E.g. `--user-agent` in commandline is already converted in parsed object (`args.user_agent`). So `configfetch` doesn't have to do anything for this. And if `use_dash` is `True`, you can use option name `user-agent` in addition.

use_uppercase Default is `True`. `INI` format is derived from Windows, and by `ConfigParser` default, option names are not case sensitive. If this argument is `True`, make them case sensitive. (We are integrating it to commandline arguments, which has some use cases for capital letters.)

`__call__``()`
return `config` data object and context object.

In initialization, `ConfigLoad` creates a temporary `config` object, using `parser.read(cfile)` or `parser.read_string(cfile)`. And then analyzing the object, it creates ordinary `INI` `config` object and corresponding context object. E.g. approximately:

```
{'main': {'gui': '[=BOOL] no'}}
```

becomes

```
{'main': {'gui': 'no'}}  # config data object
```

and

```
{'main': {'gui': '_bool'}}  # context object
```

Example:

```
loader = ConfigLoad(cfile='myapp.ini')
config, ctxs = loader()
```

1.5.2 ConfigFetch

```
class configfetch.ConfigFetch(config, ctxs=None,
    fmts=None, args=None, envs=None, Func=Func):
```

Initialization returns a config data object with `dot access` lookup (`conf` object).

config config data object (required).

ctxs corresponding context object.

Above two arguments are supposed to be provided by `ConfigLoad`.

fmts dictionary used by conversion function `_fmt()`. See *_fmt()*.

args `argparse Namespace` object to override data in `config`.

env dictionary in which keys are config option names and values are environment variable names to override.

So no automatic retrieval mechanism is provided. You have to assign them manually. E.g.:

```
{'gui': 'MYAPP_GUI'}
```

Func Function registration object, either default one the module provides, or user customized one.

Example:

```
import argparse
parser = argparse.ArgumentParser()
[...]
args = parser.parse_args()

loader = ConfigLoad(cfile='myapp.ini')
config, ctxs = loader()
conf = ConfigFetch(config, ctxs, args=args)
```

1.5.3 SectionFetch

```
class configfetch.SectionFetch(conf, section, ctx, fmts, Func)
```

In the first `dot access` on an `ConfigFetch` object, what actually returns is a proxy object called `SectionFetch`. The mechanism is the same as `ConfigParser`, and users normally don't have to think about them.

Initialization, with appropriate arguments, is done automatically when a section is first accessed from `ConfigFetch` object.

Example:

```
>>> conf
<configfetch.ConfigFetch object at 0x1234567890ab>
>>> conf.main
<configfetch.SectionFetch object at 0x567890abcdef>
>>> conf.main.gui
False
```

1.5.4 fetch()

```
function configfetch.fetch(cfile, *,
    fmts=None, args=None, envs=None, Func=Func,
    parser=configparser.ConfigParser,
    use_dash=True, use_uppercase=True, **kwargs):
```

A convenience function, actually doing the same thing as the `ConfigFetch` example above. Return a config data object (`conf`).

The meaning of arguments are the same as `ConfigLoad` and `ConfigFetch`.

Example:

```
conf = fetch('myapp.ini', args=args)
```

1.5.5 Func

```
class configfetch.Func(ctx, fmts)
```

The meaning of arguments are the same as `ConfigFetch`. In ordinary cases, instance initialization is only done by `ConfigFetch` internally, so user doesn't have to think about these arguments.

1.5.6 Builtin Functions

All builtin functions except `_bar()`, expect initial string `value`, so they should come in first.

`_bar()` expects an list type `value`, so it should come the second or after. (usually immediately after `_comma()` or `_line()`.)

_bool (*value*)
return `True` or `False`, according to the same rule as `configparser`'s.

`'1'`, `'yes'`, `'true'`, `'on'` are `True`.
`'0'`, `'no'`, `'false'`, `'off'` are `False`.
Case insensitive.
Other values raise an error.

_comma (*value*)
return a list using comma as separaters. No comma value returns one element list. Blank value returns a blank list (`[]`).

Heading and tailing whitespaces are stripped from each element.

_line (*value*)
return a list using line break as separaters. No line break value returns one element list. Blank value returns a blank list (`[]`).

Heading and tailing whitespaces and *commas* are stripped from each element.

_bar (*value*)
receive a list as `value` and return a concatenated string with bar (`'|'`) between them. One element list returns that element (`string`). Blank list returns `''`. E.g.

```
scheme=     [=COMMA][=BAR] https?, ftp, mailto
```

```
>>> conf.main.scheme
'https?|ftp|mailto'
```

_cmd (*value*)
return a list ready to put in subprocess.

That means the end users can write strings as they type in a terminal, which, when processed by `subprocess`, run corresponding commnad. E.g.

```
command=    [=CMD] ls -l 'Live at the Apollo'
```

```
>>> conf.main.command
['ls', '-l', 'Live at the Apollo']
```

Note it uses shlex.split, with `comments='#'`.

_fmt (*value*)
return a string processed by `str.format`, using `fmts` dictionary. E.g.

```
# myapp.ini
css=        [=FMT] {USER}/data/my.css
```

```
# myapp.py
fmts = {'USER': '/home/john'}
```

```
>>> conf.main.css
'/home/john/data/my.css'
```

_plus (*value*)
receive `value` as argument, but actually it doesn't use this, and use `values` (a `[arg, env, opt]` list before selection) instead.

Let's call an item starting with `'+'` as `plus item`, one starting with `'-'` as `minus item`, and others as `normal item`.

It reads each value in `values` in order, and:

1. It makes a list using the same mechanism as `_comma()`.
2. If items in the list are all `normal items`, then the list overwrites the previous list.
3. If they consist only of `plus items` and `minus items`, then it adds `plus items` to, and subtracts `minus items` from, the previous list.
4. Otherwise (mixing cases), it raises error.

Adding existing items, or subtracting nonexistant items doesn't cause errors. It just ignores them.

(Internally, the list is converted to (a kind of) ordered `set`. So duplicate items are discarded).

Example:

```
'Alice, Bob, Charlie'    -->  ['Alice', 'Bob', 'Charlie']
'-Alice, +Dave'          -->  ['Bob', 'Charlie', 'Dave']
'+Bob'                   -->  ['Bob', 'Charlie', 'Dave']
'-Xavier'                -->  ['Bob', 'Charlie', 'Dave']
'Judy, Malloy, Niaj'     -->  ['Judy', 'Malloy', 'Niaj']
```

1.5.7 User Functions

When registering user functions,

1. add them in a `Func` subclass
2. put `register()` decorator above the function
3. and call `ConfigFetch` with that subclass.

Example:

```
## myapp.ini
[main]
search=     [=GLOB] python
```

```
## myapp.py
import configfetch

class MyFunc(configfetch.Func):

    @configfetch.register
    def _glob(self, value):
        if not value.startswith('*'):
            value = '*' + value
        if not value.endswith('*'):
            value = value + '*'
        return value

conf = configfetch.fetch('myapp.ini', Func=MyFunc)
```

```
# terminal
>>> import myapp
>>> conf = myapp.conf
>>> conf.main.search
'*python*'
```

1.5.8 Double

```
class Double(sec, parent_sec)
```

sec `SectionFetch` object.

parent_sec `SectionFetch` object to fallback.

It is an accessory helper class.

Default section is a useful feature of `INI` format, but it is always global and unconditional. Sometimes more fine-tuned one is needed. For example, a section may want to look up a related section when no option is found. In that case, use this class. E.g:

```
conf.japanese = Double(conf.japanese, conf.asian)
```

When the option is not found even in the parent section, What happens is determined by the global environment (`conf`, or more exactly `conf._config`), most likely the Default section will be looked up.

1.5.9 minusadapter()

```
function minusadapter(parser, matcher=None, args=None)
```

parser ArgumentParser object, already `actions` registered.

matcher regex string to match options.

Only matched options are checked for edit. When it is `None`, All options are checked (default).

args commandline arguments list. It defaults to `sys.argv[1:]`, as `argparse` does (when it is `None`).

It is an accessory helper function.

One problem of `argparse` is when required arguments begin with `prefix_chars`. For example, if `--file` requires one argument:

```
--file -myfile.txt
```

raises error, because it always search prefixed words first, and assign them as option designating strings. (So it thinks `--file` doesn't have a required argument, and `-m` has one concatenated argument `'yfile.txt'`, if `-m` is registered.).

This is different from most traditional unix software. For the details, see:

- https://bugs.python.org/issue9334
- https://stackoverflow.com/a/21894384

It is troublesome for us because when employing *_plus* function, we use this type of arguments frequently.

In that case, one solution is to use this `minusadapter`. It parses commandline arguments, and checking `ArgumentParser` object, rewrites them suitably.

Conditions:

- if `prefix_chars` is exactly `'-'`,
- if the argument is a registered argument,
- if it's `action` is either `store` or `append`,
- if it's `nargs` is `1` or `None`,
- and if the next argument starts with `'-'`,

Rules:

- long option is combined with the next argument with =.
- short option is simply concatenated with the next argument. E.g.:

```
['--file', '-myfile.txt']  -->  ['--file=-myfile.txt']
['-f', '-myfile.txt']      -->  ['-f-myfile.txt']
```

How to use:

```
# myapp.py
import argparse
import configfetch
parser = argparse.ArgumentParser()
parser.add_argument('--file')
args = configfetch.minusadapter(parser)
```

(continues on next page)

(continued from previous page)

```
args = parser.parse_args(args)
print(args)
```

```
$ myapp.py --file -myfile.txt
Namespace(file='-myfile.txt')
```

Note it is not a general solution for the above `argparse` problem. It just makes `_plus()` function marginally usable.

CHAPTER 2

API Reference

Helper to get values from configparser and argparse.

exception configfetch.**Error**
Base Exception class for the module.

`configparser` has 11 custom Exceptions scattered in 14 methods last time I checked. I'm not going to wrap except the most relevant ones.

exception configfetch.**NoSectionError**(*section*)
Raised when no section is found.

exception configfetch.**NoOptionError**(*option*, *section*)
Raised when no option is found.

configfetch.**register**(*meth*)
Decorate value functions to populate global value *_REGISTRY*.

class configfetch.**Func**(*ctx*, *fmts*)
Register and apply value conversions.

class configfetch.**ConfigLoad**(**args*, ***kwargs*)
A custom Configuration builder.

Read from custom config file with some additional information and launch ConfigParser.

You can pass `parser`'s keyword argumants in initializing. additional arguments are:

Parameters

- **cfile** – custom ini format filename or string
- **parser** – returned object by *__call__*, configparser.ConfigParser (default) or similar object

Furthermore, if you use this class for `ConfigFetch`, small parser customizetions are needed.

- If you use dot access (e.g. `obj.attribute`), 'dash' must be changed to 'underscore', to make the key to identifier.
- Default `configparser` converts all option names to lowercase. Disable this, for `ArgumentParser` might use uppercases.

Parameters

- **use_dash** – default True (changes dashes to underscores internally)
- **use_uppercase** – default True

class configfetch.**ConfigFetch**(*config*, *ctxs=None*, *fmts=None*, *args=None*, *envs=None*, *Func=<class 'configfetch.Func'>*)

`ConfigParser` proxy object with dot access.

Actual work is delegated to *SectionFetch*.

class configfetch.**SectionFetch**(*conf*, *section*, *ctx*, *fmts*, *Func*)

`ConfigParser` section proxy object.

Similar to `ConfigParser`'s proxy object itself. Also access `ArgumentParser` arguments and environment variables.

class configfetch.**Double**(*sec*, *parent_sec*)

A utility class to parse two `SectionFetch` objects.

To supply some secion an additional external section fallback.

configfetch.**fetch**(*cfile*, ***, *fmts=None*, *args=None*, *envs=None*, *Func=<class 'configfetch.Func'>*, *parser=<class 'configparser.ConfigParser'>*, *use_dash=True*, *use_uppercase=True*, ***kwargs*)

Fetch `ConfigFetch` object.

It is a convenience function for the basic use of the library.

configfetch.**minusadapter**(*parser*, *matcher=None*, *args=None*)

Parse and edit commandline arguments.

An accessory helper function. It unites two arguments to one, if the second argument starts with −.

e.g. `['--aa', '-somearg']` becomes `['--aa=-somearg']`.
e.g. `['-a', '-somearg']` becomes `['-a-somearg']`.

The reason is that `argparse` cannot parse this particular pattern. https://bugs.python.org/issue9334 https://stackoverflow.com/a/21894384 And *_plus* uses this type of arguments frequantly.

Parameters

- **parser** – ArgumentParser object
- **matcher** – regex string to match options
- **args** – arguments list to parse, defaults to `sys.argv[1:]`

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

C

D

E

F

M

N

R

S