## Contents:

1 Overview  
1.1 Installation .................................................. 3  
1.2 Usage .......................................................... 3  
  1.2.1 dict fromat .................................................. 3  
  1.2.2 FINI format ............................................... 4  
  1.2.3 With argparse ............................................ 5  
1.3 API Overview .................................................. 6  
1.4 Value Selection ............................................... 6  
  1.4.1 Section ..................................................... 6  
  1.4.2 Option ..................................................... 6  
  1.4.3 Nonstring ................................................ 7  
  1.4.4 ArgumentParser Details .................................. 7  
  1.4.5 Conversion ................................................. 8  
  1.4.6 Function .................................................. 8  
  1.4.7 Concatenation ............................................. 8  
1.5 Builtln Functions ............................................. 8  
1.6 User Functions ............................................... 10  
1.7 FINI Syntax ................................................... 11  
1.8 Configuring Arguments ....................................... 11  
  1.8.1 For FINI format (FiniOptionBuilder) .................... 12  
  1.8.2 For dictionary (DictOptionBuilder) ..................... 13  
1.9 API ........................................................... 13  
  1.9.1 ConfigFetch ............................................... 13  
  1.9.2 fetch ..................................................... 14  
  1.9.3 Double .................................................... 14  
  1.9.4 minusadapter ............................................ 15  
  1.9.5 ConfigPrinter ............................................ 16  

2 Indices and tables .............................................. 17

Index ....................................................................... 19
Helper to get values from `configparser` and `argparse`. 
This library helps to build and access configuration data. It reads specially formatted configuration data, creates `configparser.ConfigParser` object, and keeps corresponding metadata. The metadata are used for automatic config value conversion, and `argparse.ArgumentParser` building. All value accesses are done by `dot access or .get()` method. Commandline arguments and Environment Variables precedes config option values, in that order.

## 1.1 Installation

It is a single file Python module, with no other external library dependency. **Python 3.6 and above** are supported.

```
pip install configfetch
```

## 1.2 Usage

### 1.2.1 dict fromat

```python
>>> data = {
    'section1': {
        'log': {
            'func': ['bool'],
            'value': 'no',
        },
        'users': {
```

(continues on next page)
The library needs special dictionaries, which must have sub-sub-dictionaries as `configparser`'s option value counterparts.

The keys are 'argparse', 'func' and 'value'. 'value' key is required, others are optional.

'value' values are always string, since they become `configparser`'s option values. (INI format values are always string).

'func' values are always list. Each members are, again, string, some built-in function names or ones you created and registered.

'argparse' values are dictionaries. Each key-values can be passed to `argparse.ArgumentParser.add_argument()`. But it is not done automatically. So they are doing nothing for now.

**1.2.2 FINI format**

Or you can do the same thing, from a kind of INI format file or string.

```ini
# myapp.ini
[section1]
log= :: f: bool
   no

users= :: f: comma
   Alice, Bob, Charlie

output= : output format
   : when saving data
   : to a file.
   :: names: o
   :: choices: html, csv, text
   :: default: html
```
>>> import configfetch
>>> conf = configfetch.fetch('myapp.ini')
>>> conf.section1.log
False
>>> conf.section1.users
['Alice', 'Bob', 'Charlie']
>>> conf.section1.output
''

': <something>': is the same as argparse['help'] key value. For maximum readability it is specially treated.

': :f <something>': is the same as 'func' key value.

': <key>: <value>': is the same as other 'argparse' key-value pairs.

Let’s call this customized format, as FINI (Fetch-INI) format.

1.2.3 With argparse

1. Create ConfigFetch object, providing config files.
2. Create argparse.ArgumentParser.
3. ConfigFetch.build_arguments (populate ArgumentParser with argument definitions).
4. ArgumentParser.parse_args etc. (actually parse commandline).
5. ConfigFetch.set_arguments, with the new parsed commandline args.

Note: Commandline options may specify where and how config files are loaded, like '--userdir' or '--nouserdir'. In this case, you have to initialize ConfigFetch in two-pass.

In (1) above, just read the canonical (default) config file.
And after (5), read other config files.

# myapp.ini
[section1]
log= : log the program
  :: f: bool
  no

users= : assign users
  :: f: comma
  Alice, Bob, Charlie

output= : output format
  : when saving data
  : to a file.
  :: names: o
  :: choices: html, csv, text
  :: default: html

>>> import configfetch
>>> conf = configfetch.fetch('myapp.ini')
>>> import argparse

(continues on next page)
1.3 API Overview

The main constructs of this module are:

- **class ConfigFetch** Read config files, and behave as a conf data object.
  
  See API for details.

- **class Func** Keep conversion functions and apply them to values.
  
  See Built-in Function for included functions.

  See User Functions for customization example.

- **function fetch** Shortcut. Using ConfigFetch, return conf object.
  
  See API for details.

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 (SectionProxy) 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 SectionProxy), 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 \texttt{section} (or \texttt{Default section}) has the key, the value is selected \((\texttt{opt})\).

Otherwise:

\begin{verbatim}
   dot access (.\texttt{__getattr__}(\texttt{option})) raise NoOptionError
   .get(\texttt{option}, fallback=_UNSET) raise NoOptionError, when fallback is not provided
   (_UNSET). Otherwise, fallback is selected.
\end{verbatim}

### 1.4.3 Nonstring

If the selected value is \texttt{arg}, and it is not a string, the value is \textit{returned} as is. \((\texttt{env} \text{ and } \texttt{opt} \text{ are always a string.})\)

So \texttt{ArgumentParser} arguments that convert the value type are just passed through.

### 1.4.4 ArgumentParser Details

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

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

If this is not desirable, use \texttt{store_const} instead. E.g.:

\begin{verbatim}
parser.add_argument('--log', action='store_const', const='true')
\end{verbatim}

\textbf{Note:} Paul Jacobson (hpaulj), active in \texttt{argparse} development, discourages \texttt{store_true} and \texttt{store_false} in a different context. See a stackoverflow.)

In most cases, you can delegate conversion to \texttt{conf}, by conforming to the designated \texttt{FINI} format. E.g.

\begin{verbatim}
# myapp.ini
file= :: f: comma
   a.txt, b.txt, c.txt

# myapp.py
parser.add_argument('--file', action='store')

# terminal
$ myapp.py --file 'a.txt, b.txt, c.txt'
\end{verbatim}

instead of:

\begin{verbatim}
parser.add_argument('--file', action='store', nargs='+')
$ myapp.py --file a.txt b.txt c.txt
\end{verbatim}

or:

\begin{verbatim}
parser.add_argument('--file', action='append')
$ myapp.py --file a.txt --file b.txt --file c.txt
\end{verbatim}
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 are searched in Func or a subclass methods.
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 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 Builtin Functions

All builtin functions except bar, expect a string as value.
bar expects a list of strings as value.

**bool**(value)
return True, False or None.

"1", "yes", "true", "on" are True.
"0", "no", "false", "off" are False.
Case insensitive.

As a special case blank string (" ") returns None.
Other values raise an error.

**int**(value)
return integer from integer number string.
blank string (" ") returns None.

**float**(value)
return float number from float number string.
blank string (" ") returns None.
**comma** (*value*)

Return a list using commas as separators. No comma value returns one element list. Blank value returns a blank list ([]).

Leading and tailing whitespaces are stripped from each element.

If the previous character is '\'', ' throws ' is a literal comma, not a separator. This '\' is discarded.

Any other '\' is kept as is.

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'aa, bb'</td>
<td>['aa', 'bb']</td>
</tr>
<tr>
<td>r'aa, bb'</td>
<td>['aa, bb']</td>
</tr>
<tr>
<td>r'aa, bb'</td>
<td>[r'aa, bb']</td>
</tr>
<tr>
<td>r'a\a'</td>
<td>[r'a\a']</td>
</tr>
<tr>
<td>r'a\a'</td>
<td>[r'a\a']</td>
</tr>
</tbody>
</table>

**line** (*value*)

Return a list using line breaks as separators. No line break value returns one element list. Blank value returns a blank list ([]).

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

The escaping behavior with '\' is the same as **comma**.

**bar** (*value*)

Concatenate with bar ('|').

Receive a list of strings as value, return a string.

One element list returns that element. Blank list returns ''. 

```
scheme= :: f: comma, bar
    https?, ftp, mailto
```

```
>>> conf.section1.scheme
'https?|ftp|mailto'
```

**cmd** (*value*)

Return a list of strings ready to put in subprocess.

Users have to write strings as in a terminal (quotes and escapes).

Note '##' and after are comments, they are discarded.

Example:

```
command= :: f: cmd
    echo -e "'I have a dream.'", he said.\n"
```

```
>>> conf.section1.command
[('echo', '-e', '"'I have a dream."', he said.\n')]
```

**cmds** (*value*)

Return a list of list of strings.

List version of **cmd**. The input value is a list of strings, with each item made into a list by **cmd**.

**fmt** (*value*)

Return a string processed by str.format, using **fmts** dictionary. E.g.
# myapp.ini

css= : f: fmt
    {USER}/data/my.css

# myapp.py

fmts = {'USER': '/home/john'}

>>> conf.section1.css
'/home/john/data/my.css'

**plus** (*value*)

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

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** or **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 nonexistent items doesn’t cause errors. It just ignores them.

Example:

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Alice, Bob, Charlie'</td>
<td>['Alice', 'Bob', 'Charlie']</td>
</tr>
<tr>
<td>'-Alice, +Dave'</td>
<td>['Bob', 'Charlie', 'Dave']</td>
</tr>
<tr>
<td>'+Bob'</td>
<td>['Bob', 'Charlie', 'Dave']</td>
</tr>
<tr>
<td>'-Xavier'</td>
<td>['Bob', 'Charlie', 'Dave']</td>
</tr>
<tr>
<td>'Judy, Malloy, Niaj'</td>
<td>['Judy', 'Malloy', 'Niaj']</td>
</tr>
</tbody>
</table>

## 1.6 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:

```python
## myapp.ini

[section1]
search= :: f: glob
python

## myapp.py

import configfetch
```

(continues on next page)
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.section1.search
'*python*'

1.7 FINI Syntax

FINI uses ' : ' and ' :: ' as keywords, designating metadata line (a space is required). It is configurable on subclasses.

All metadata types are optional, but must follow the predetermined order.

(help)
(args)
(func)
value

args means here, arguments for argparse.ArgumentParser.add_argument. help is actually one of them, but specially treated.

help: Following string from ' : ' is help. You can repeat help on several lines. Strings are joined with newlines.

args: ' :: <name>: <value>' is parsed into a dictionary item, with some value type conversion, if <name> is not 'f'.

    <name> should be key for the argument (nargs, choices, etc.). special <name> 'names' is used for name or flags, with the option name added to the last.

    (E.g. 'output' option in the Usage example of the document top has ' :: names: o', so argument names becomes ['-o', '--output']).

func: Following string from ' :: f: ' is comma separated function names to process the option value.

value: actual option value.

1.8 Configuring Arguments

Excluding Environment Variables, there are three kinds of option types.

1. Config-only options
2. Commandline-only options
3. Common options (to commandline and config file)

1.8.1 For FINI format (FiniOptionBuilder)

1:
If you don’t provide 'help' option line to an option, it is not exposed for building process. So that makes 1. config-
only options.

2:
If you can ignore the config option (separating it in a specially chosen section, say, '[_command_only]'), it
veritably makes 2. commandline-only options.

For this, since it is unrelated to the INI format limitations, you can use any add_argument arguments.

But data types have to be guessed from INI string values none the less, only simple cases are feasible (E.g. In ': :
const: 1', is '1' int or str?).

See source code (FiniOptionBuilder._convert_arg) for details.

3:
For all common options:
  • As already said, help is required.
  • You can always add names.

They are divided in two: boolean options and non-boolean options.

  • For non-boolean options:
    They are all treated as action='store', nargs=None, which is argparse default. Optionally you can only add choices.
  • For boolean options:
    If it has bool in func, it is a boolean option, and the option is interpreted as flag (with no option_argument).
    action is always store_const, const is ‘yes’ (which will be converted to True when getting value).

| log=     : log events |
| : f: bool      |
| yes |

becomes:

```python
[...].add_argument('--log', action='store_const', const='yes')
```

If there is dest argument, it is interpreted as opposite flag, const becomes ‘no’ (converted to False).

| no_log= : do not log events |
| : dest: log |
| : f: bool   |
| no |

becomes:
1.8.2 For dictionary (DictOptionBuilder)

If you don’t provide `argparse` key to an option, it is not exposed for building process. So that makes 1. config-only options.

Otherwise, DictOptionBuilder enforces no rules, and provide no smart argument adjustments (exactly as you provided).

Although, using in the same restrictions as FINI format is generally presupposed and recommended.

1.9 API

1.9.1 ConfigFetch

class configfetch.ConfigFetch(*, fms=None, args=None, envs=None, Func=<class 'configfetch.Func'>, option_builder=<class 'configfetch.FiniOptionBuilder'>, parser=<class 'configfetch.FiniParser'>, **kwargs)

A custom Configuration object.

It keeps a ConfigParser object (_config) and a correspondent option-name-to-metadata map (_ctx).

It also has argparse.Namespace object (args), and Environment variable dictionary (envs).

If the option name counterpart is defined in args or envs, their value precedes the config value.

So most config option names must be global, since args and envs do not have section namespace.

E.g. if a config has ‘foo’ section and ‘bar’ option in it, args, and envs just check the name ‘bar’, ignoring section hierarchy.

The metadata includes function list specific to the option name. Option access gets value from arg, envs or config, and returns a functions-applied-value.

The class __init__ should accept all ConfigParser.__init__ keyword arguments.

Additional arguments are:

Parameters

- `fms` – dictionary Func._fmt uses
- `args` – argparse.Namespace object
- `envs` – dictionary with option name and Environment Variable name as key and value
- `Func` – Func or subclasses, worker to keep and look-up functions
- `option_builder` – DictOptionBuilder or FiniOptionBuilder, worker to build value and metadata from data input
- `parser` – ConfigParser or a subclass, keep actual config values
**fetch** (*input_*)
Read input and build config data and metadata.
Note type of input entirely depends on option_builder. DictOptionBuilder accepts only python dictionary object. FiniOptionBuilder accepts only opened file object or string.

**build_arguments** (*argument_parser, sections=None*)
Run argument_parser.add_argument according to config metadata.

**Parameters**
- *argument_parser* – argparse.ArgumentParser or a subclass, either blank or with some arguments already defined
- *sections* – a section name (string) or section list to filter sections, default (None) is for all sections

**Returns** argument_parser

**set_arguments** (*namespace*)
Set _args attribute.

**Parameters** namespace – argparse.Namespace object
It manually sets _args again, after initialization.

**1.9.2 fetch**

```python
cfgfetch.fetch(*input_, *, encoding=None, fnts=None, args=None, envs=None, Func=cfgfetch.Func, parser=cfgparser.ConfigParser, option_builder=cfgfetch.FiniOptionBuilder, **kwargs)
```

Fetch ConfigFetch object.
It is a convenience function for the basic use of the library. Most arguments are the same as ConfigFetch.__init__.

the specific arguments are:

**Parameters**
- *input* – dict, file obj or string according to option_builder. Additionally, if the input is string and in system path, it tries to open to make file object
- *encoding* – encoding to use when opening the input

**1.9.3 Double**

```python
class cfgfetch.Double(sec, parent_sec)
```
Supply a parent section fallback, before ‘DEFAULT’.
An accessory helper class, not so related to this module’s main concern.
Default section is a useful feature of INI format, but it is always global and unconditional. Sometimes more fine-tuned one is needed.

**Parameters**
- *sec* – SectionProxy object
- *parent_sec* – SectionProxy object to fallback

Example:
conf.japanese = Double(conf.japanese, conf.asian)

When the option is not found even in the parent section, DEFAULT section lookup is performed, or NoOptionError, according to the uniderlined ConfigParser object (conf._config).

1.9.4 minusadapter

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

Edit option_arguments with leading dashes.

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

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 frequently.

Parameters

- **parser** – ArgumentParser object, already actions registered
- **matcher** – regex string to match options, to narrow the targets (None means to process all arguments)
- **args** – arguments list to parse, defaults to sys.argv[1:] (the same as argparse default)

Conditions:

- prefix_chars is exactly '—'
- The argument is a registered argument
- It's action is either store or append
- It's nargs is 1 or None
- The next argument starts with '—'

Process:

- long option is combined with the next argument with =
- short option is concatenated with the next argument

Example:

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

(continues on next page)
args = parser.parse_args(args)
print(args)

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

1.9.5 ConfigPrinter

class configfetch.ConfigPrinter(conf, sections=None, width=4, print=<built-in function print>):
    Print dictionary or INI format strings from configuration.

    Parameters
    • conf – ConfigFetch object, with _config and _ctx attributes
    • sections – list of section names to print, all sections if None
    • width – indent unit width
    • print – any function with one string argument, to customize printout behavior

    print_dict()
    Print dictionary string.

    print_ini()
    Print INI format string.
Indices and tables

• genindex
• modindex
• search
B
bar(), 9
bool(), 8
build_arguments() \hspace{1em} (configfetch.ConfigFetch method), 14

C
cmd(), 9
cmds(), 9
comma(), 8
ConfigFetch \hspace{1em} (class in configfetch), 13
ConfigPrinter \hspace{1em} (class in configfetch), 16

D
Double \hspace{1em} (class in configfetch), 14

F
fetch() \hspace{1em} (configfetch.ConfigFetch method), 13
fetch() \hspace{1em} (in module configfetch), 14
float(), 8
fmt(), 9

I
int(), 8

L
line(), 9

M
minusadapter() \hspace{1em} (in module configfetch), 15

P
plus(), 10
print_dict() \hspace{1em} (configfetch.ConfigPrinter method), 16
print_ini() \hspace{1em} (configfetch.ConfigPrinter method), 16

S
set_arguments() \hspace{1em} (configfetch.ConfigFetch method), 14