funconf Documentation

Release 0.3.0

Michael Dorman

August 04, 2014

Contents

funconf simplifies the management of function argument default values by seamlessly integrating configuration files into mapping objects which can decorate a function to set or override its default values.

Project support:

- source code hosted at github.com.
- distributed through PyPI.
- documentation hosted at readthedocs.org.

[mjdorma+funconf@gmail.com]

CHAPTER 1

Learning

1.1 Function Configuration module - `funconf`

1.1.1 Overview.

To simplify the management of function default keyword argument values `funconf` introduces two new decorators. The first decorator `wraps_parameters()` makes it trivial to dynamically define the default values of parameters to a function. The second decorator `lazy_string_cast()` automatically casts *basestring* values to the type of the keyword default values found in the function it is wrapping and the type of the values found in the *key:value* object passed into its constructor.

For configuration, `funconf` borrows from concepts discussed in Python's core library *ConfigParser*. A configuration consists of sections made up of *option:value* entries, or two levels of mappings that take on the form *section:option:value*.

The file format YAML has been chosen to allow for option values to exist as different types instead of being restricted to string type values.

The configuration file

Example of a simple YAML configuration file:

```
$ cat my.conf
#
# Foo
#
foo:
  bar: 4
  moo:
  - how
  - are
  - you

#
# Bread
#
bread:
  butter: win
  milk: fail
```

This file contains two sections foo and bread. foo has two options bar and moo. bar is an integer value of 4 and moo is a list of strings containing `['how', 'are', 'you']`.

The above configuration could be generated through a process of adding each option: value into a `Config` object then casting it to a string. For example:

```
config = funconf.Config()
config.set('foo', 'bar', 4)
config.set('foo', 'bar', ['how', 'are', 'you'])
config.set('bread', 'butter', 'milk')
config.set('bread', 'milk', 'fail')
print(config)
```

A configuration object

The `Config` class is the root class container for all of the configuration sections. It implements the *MutableMapping* type which allows it to seamlessly integrate into the `wraps_parameters()` function.

As a dictionary `dict(config)` the `Config` object looks like:

```
{'bread_butter': 'win',
 'bread_milk': 'fail',
 'foo_bar': 4,
 'foo_moo': ['how', 'are', 'you']}
```

Notice how the configuration *section:options* have been concatenated. This facilitates the simple wrapping of an entire configuration file into the kwargs of a function. The following example will print an equivalent dictionary to the printout in the previous example.

```
@config
def myfunc(**kwargs):
    print(kwargs)
```

A configuration section

To access sections and option values the following pattern holds true:

```
assert config.bread.milk == 'fail'
assert config.bread['milk'] == 'fail'
assert config['bread_milk'] == 'fail'
```

A section is represented by the `ConfigSection` object. This object implements the *MutableMapping* type in the same way the `Config` object is implemented and is compatible with the `wraps_parameters()` function. The `ConfigSection` represented `str(config.bread)` looks like:

```
bread:
  butter: win
  milk: fail
```

Just like the `Config` class, the `ConfigSection` can be used as a decorator to a function too. Here is an example:

```
@config.bread
def myfunc(**kwargs):
    print(kwargs)
```

1.1.2 Code reference guide

funconf.**wraps_parameters**(*default_kwargs*, *hide_var_keyword=False*, *hide_var_positional=False*)

Decorate a function to define and extend its positional and keyword variables.

The following example will redefine myfunc to have defaults of a=4 and b=3:

```
mydict = {a=4, b=3}
@wraps_parameters(mydict)
def myfunc(a, b=2):
    pass
```

The *default_kwargs* object needs to satisfy the *MutableMapping* interface definition. When a wrapped function is called the following transforms occur over *kwargs* before it is passed into the wrapped function:

1. *default_kwargs* is updated with new input parameters.
2. If the wrapped function has a variable keyword argument defined (i.e `**k`) then the keywords defined by *default_kwargs* that are not defined in *kwargs* will be copied into *kwargs*.
3. If the wrapped function had no variable keyword argument defined, then the keyword input parameters that don't belong to the wrapped function's parameters list will be discarded.

Parameters

- **default_kwargs** (*mutable mapping*) – kwargs to be fix into the wrapped function.
- **hide_var_keyword** (*Boolean value default True.*) – hide the variable keyword parameter.
- **hide_var_arguments** (*Boolean value default True.*) – hide the variable keyword parameter.

Return type decorated function.

funconf.**lazy_string_cast**(*model_parameters={}*, *provide_defaults=True*)

Type cast string input values if they differ from the type of the default value found in *model_parameters*.

The following list details how each type is handled:

int, bool, float: If the input value string can not be cast into an int, bool, or float, the exception generated during the attempted conversion will be raised.

list: The input value string will be split into a list shlex.split. If the default list value in model contains items, the first item is sampled an attempt to cast the entire list is made for that type.

other: An attempt to convert other types will be made. If this fails, the input value will be passed through in its original string form.

This example demonstrates how `lazy_string_cast()` can be applied:

```
@lazy_string_cast
def main(a=4, b=[4, 2, 55]):
    pass
```

Or using `lazy_string_cast()` with `wraps_parameters()` to define new keyword defaults:

```
config = dict(a=6, b=[4])

@lazy_string_cast(config)
@wraps_parameters()
def main(a=4, b=[4, 2, 55]):
    pass
```

Parameters

- **model_parameters** (*mutable mapping*) – kwargs to model default type values and keys from.
- **provide_defaults** (*Boolean value default True.*) – If true, use model_parameters to default arguments which are empty.

Return type decorated function.

class funconf.**Config**(*filenames=[]*, *strict=False*)

The Config class is the root container that represents configuration state set programmatically or read in from YAML config files.

The following lists the main features of this class:

1. Aggregates `ConfigSection` objects that are accessible through attributes.
2. Exposes a translation of the configuration into section_option:value through a standard implementation of the *MutableMapping* abstract type.
3. When cast to a string it outputs YAML.
4. As a decorator it utilises the `wraps_parameters()` to change the defaults of a variable kwargs function.

__call__(*func=None*, *lazy=True*, *hide_var_positional=False*, *hide_var_keyword=True*)

The `Config` object can be used as a function decorator.

Applying this decorator to a function which takes variable kwargs will change its signature to a set of kwargs with default values defined using the values in this `Config` object. The input kwargs go through the `lazy_string_cast()` function and are passed into this objects update routine. The wrapped function is finally called with the updated values of this object.

For example:

```
myconfig = Config('my.conf')

@myconfig
def func(**k):
    pass
```

Parameters

- **func** (*function or method*) – Decorator parameter. The function or method to be wrapped.
- **lazy** (*Boolean value default True*) – Factory parameter. Turns lazy_string_cast on or off.

Return type As a factory returns decorator function. As a decorator function returns a decorated function.

__dir__()

Return a list of section names and the Base class attributes.

__getattribute__(*y*)

Return a section where y is the *section* name. Else, return the value of a reserved word.

__getitem__(*y*)

Return the option value for y where y is *section_option*.

__init__(*filenames=[]*, *strict=False*)

Construct a new Config object.

This is the root object for a function configuration set. It is the container for the configuration sections.

Parameters

- **filenames** (*list of filepaths*) – YAML configuration files.
- **strict** (*False*) – If True, raise `ConfigAttributeError` if a `ConfigSection` doesn't exist.

__iter__()
Iterate all of the *section_option* keys.

__len__()
Return the number of options defined in this `Config` object

__setattr__(*x*, *y*)
Only attributes that are a reserved words can be set in this object.

__setitem__(*x*, *y*)
Set the option value of y for x where x is *section_option*.

__str__()
Return a YAML formated string object that represents this object.

load(*stream*)
Parse the first YAML document from stream then load the *section:option:value* elements into this `Config` object.

Parameters **stream** (*stream object*) – the configuration to be loaded using `yaml.load`.

read(*filenames*)
Read and parse a filename or a list of filenames.

Files that cannot be opened are silently ignored; this is designed so that you can specify a list of potential configuration file locations (e.g. current directory, user's home directory, system wide directory), and all existing configuration files in the list will be read. A single filename may also be given.

Parameters **filenames** (*list of filepaths*) – YAML configuration files.

Return type list of successfully read files.

set(*section*, *option*, *value*)
Set an option.

In the event of setting an option name or section name to a reserved word a ValueError will be raised. A complete set of reserved words for both section and option can be seen by:

```
print(dir(funconf.Config))
print(dir(funconf.ConfigOption))
```

Parameters

- **section** (*str*) – Name of the section to add the option into.
- **option** (*str*) – Name of the option.
- **value** – Value assigned to this option.

class `funconf.`**ConfigSection**(*section*, *options*)
The `ConfigSection` class is a mutable mapping object that represents the *option:value* items for a configuration section.

The following lists the main features of this class:

1. Exposes a configuration *option:value* through a standard implementation of the *MutableMapping* abstract type.

2. When cast to a string it outputs YAML.

3. As a decorator it utilises the `wraps_parameters()` to change the defaults of a variable kwargs function.

__call__ (*func=None*, *lazy=True*, *hide_var_positional=False*, *hide_var_keyword=True*)
The `ConfigSection` object can be used as a function decorator.

Applying this decorator to a function which takes variable kwargs will change its signature to a set of kwargs with default values defined using the values in this `ConfigSection` object. The input kwargs go through the `lazy_string_cast()` function and are passed into this objects update routine. The wrapped function is finally called with the updated values of this object.

For example:

```
myconfig = Config('my.conf')

@myconfig.mysection
def func(**k):
    pass
```

Parameters

- **func** (*function or method*) – Decorator parameter. The function or method to be wrapped.
- **lazy** (*Boolean value default True*) – Factory parameter. Turns lazy_string_cast on or off.

Return type As a factory returns decorator function. As a decorator function returns a decorated function.

__dir__ ()
Return a list of option names and the Base class attributes.

__getattribute__ (*y*)
Return a option value where y is the *option* name. Else, return the value of a reserved word.

__getitem__ (*y*)
Return the option value for y where y is *option*.

__init__ (*section*, *options*)
Construct a new `ConfigSection` object.

This object represents a section which contains the mappings between the section's options and their respective values.

Parameters

- **section** (*str*) – defines the name for this section.
- **options** (*mutable mapping*) – kwargs to initialise this `ConfigSection`'s *option:value*

__iter__ ()
Iterate all of the 'option' keys.

__len__ ()
Return the number of options defined in this `ConfigSection` object

__setattr__ (*x*, *y*)
Only attributes that are a reserved words can be set in this object.

__setitem__ (*x*, *y*)
Set the option value of y for x where x is *option*.

__str__()
Return a YAML formated string object that represents this object.

dirty
The dirty property is a convenience property which is set when a change has occurred to one of the options under this *section*. Once this property is read, it is reset to False.

This property is particularly useful if you're software system has the ability to change the configuration state during runtime. It means that you no longer need to remember the state of the options, you just need to know that when the dirty property is set, there has been a change in the configuration for this section.

Return type boolean value of dirty.

1.2 funconf with begins

The motivation for funconf was to build a simple file configuration management capability that integrates with the command line interface tool begins.

1.2.1 Walk through

Taking this example YAML configuration file demo.conf:

```
#
# Foo
#
foo:
  bar: 4
  moo:
  - how
  - are
  - you

#
# Bread
#
bread:
  butter: win
  milk: fail
```

Applying it to this simple program demo.py:

```
import begin
import funconf

config = funconf.Config('demo.conf')

@begin.subcommand
@config.foo
def foo(*a, **k):
    "This is the foo code"
    print("Foo got a=%s k=%s" % (a, k))
    print("Config is:")
    print(config)

@begin.subcommand
@config.bread
```

```
def bread(*a, **k):
    "This is the bread code"
    print("Bread got a=%s k=%s" % (a, k))
    print("Config is:")
    print(config)

@begin.subcommand
@config
def run(*a, **k):
    "This is the run command that controls all"
    print("Run got a=%s k=%s" % (a, k))
    print("Config is:")
    print(config)

@begin.start
def entry():
    "This is a super dooper program..."
    pass
```

You will end up with the following help from the main:

```
$ python demo.py -h
usage: demo.py [-h] {bread,foo,run} ...

This is a super dooper program...

optional arguments:
  -h, --help       show this help message and exit

Available subcommands:
  {bread,foo,run}
    bread          This is the bread code
    foo            This is the foo code
    run            This is the run command that controls all
```

Now you can see how the `funconf.Config` object has been bound to the `run()` function:

```
$ python demo.py run -h
usage: demo.py run [-h] [--foo_bar FOO_BAR] [--bread_butter BREAD_BUTTER]
                   [--foo_moo FOO_MOO] [--bread_milk BREAD_MILK]

This is the run command that controls all

optional arguments:
  -h, --help            show this help message and exit
  --foo_bar FOO_BAR     (default: 4)
  --bread_butter BREAD_BUTTER
                        (default: win)
  --foo_moo FOO_MOO, -f FOO_MOO
                        (default: ['how', 'are', 'you'])
  --bread_milk BREAD_MILK, -b BREAD_MILK
                        (default: fail)
```

Finally, to see how the `funconf.ConfigSection` objects foo and bread have bound to their respective functions:

```
$ python demo.py foo --help
usage: demo.py foo [-h] [--moo MOO] [--bar BAR]

This is the foo code
```

```
optional arguments:
  -h, --help         show this help message and exit
  --moo MOO, -m MOO  (default: ['how', 'are', 'you'])
  --bar BAR, -b BAR  (default: 4)
```

1.2.2 Conclusion

The default values read into the `funconf.Config` object from demo.conf will be overridden by begins when it passes in user defined option values from the command line. As soon as your program entry has executed, you will now have a simple up to date global object which represents the programs configuration state.

Find out more in the `funconf` module documentation.

1.3 Example webapp

bottle makes writing a webapp super simple. begins makes writing a CLI to a program super simple. Through this example you will see how `funconf` can make managing configuration files super simple.

Taking this example YAML configuration file webapp.conf:

```
web:
  host: 0.0.0.0
  port: 8585
```

Applying it to this simple program webapp.py:

```
import bottle
import begin
import funconf

@bottle.route('/hello')
def hello():
    return "Hello World!"

config = funconf.Config(['webapp.conf',
                         '~/.webapp.conf'])

@begin.start(env_prefix="WEBAPP_")
@config.web
def main(host='127.0.0.1', port=8080, debug=False):
    print(config)
    bottle.run(host=host, port=port, debug=debug)
```

Applying the above configuration to this application will override the host parameter default value found in the function declaration from *127.0.0.1* to *0.0.0.0* and port value from *8080* to *8585*. In this program, configuration is applied and overridden in the following order:

1. Decorated function's default keyword value.
2. Defined configuration options.
3. Environment variables.
4. Input command line interface parameters.

The port option has been included into the config file to ensure that the `funconf.Config` object contains the *port* value that has been provided by begins. Now we have a neat way of storing and accessing the application's global configuration state for both *host* and *port* under the `config.web` object.

Here is what the help printout should look like for this application:

```
$ python webapp.py --help
usage: webapp.py [--help] [--debug WEBAPP_DEBUG] [--host WEBAPP_HOST]
                 [--port WEBAPP_PORT]

optional arguments:
  --help                show this help message and exit
  --debug WEBAPP_DEBUG, -d WEBAPP_DEBUG
                        (default: False)
  --host WEBAPP_HOST, -h WEBAPP_HOST
                        (default: 0.0.0.0)
  --port WEBAPP_PORT, -p WEBAPP_PORT
                        (default: 8585)
```

The following is the output that you'd expect:

```
$ python webapp.py

#
# Web
#
web:
  host: 0.0.0.0
  port: 8585

Bottle v0.11.6 server starting up (using WSGIRefServer())...
Listening on http://0.0.0.0:8585/
Hit Ctrl-C to quit.
```

In the example below, notice how the configuration object has been updated when we specify a new value for the port through the command line interface:

```
$ python webapp.py --port 8787
#
# Web
#
web:
  host: 0.0.0.0
  port: 8787

Bottle v0.11.6 server starting up (using WSGIRefServer())...
Listening on http://0.0.0.0:8787/
Hit Ctrl-C to quit.
```

The above example also demonstrates how the configuration object has implicitly casted the input value *8787* CLI string into an integer.

CHAPTER 2

Project

2.1 Roadmap

This is a rough roadmap to the first beta release of *funconf*.

Version 0.1 - *Test primitive*

- Build the primitive test harness.
- RTD supporting documentation.
- Integrate into travis-ci.
- Work on code coverage.

Version 0.2 - *Support PyPy and Python 3x*

- Fixes and improvements for alternative platform support

Version 0.3 - Beta release

- Completed the trial by fire!

Version 1.0 - Stable release

- Tried and trued.

2.2 Change log

- 0.3.0 - Bump to BETA. Vul fix by @JoeyJoJoJrShabadu.
- 0.2.5 - Lazy list conversion hides \ from shlex.split (issue #2).
- 0.2.4 - Contribution by Trampas solved more bugs.
- 0.2.3 - lazy_string_cast can now optionally provide default values.
- 0.2.2 - Fixed bugs, removed magic hidden vars.
- 0.2.1 - Wrapper functions now supporting var positional and keyword.
- 0.2 - Py3x support.
- 0.1.1 - Wrap existing kwargs, convert strings, less bugs.
- 0.1 - Test, doc, coverage, PyPy and 2x support.

CHAPTER 3

Indices and tables

- *genindex*
- *search*

f