configman Documentation

Release 1.0

K Lars Lohn, Peter Bengtsson

July 07, 2016

Contents

Contents:

CHAPTER 1

Introduction (start here)

`configman` is a package that paves over the differences between various configuration methods to achieve a smooth road of cooperation between them.

We use it here at Mozilla to tie together all the different scripts and programs in Socorro.

The modules typically used for configuration in Python applications have inconsistent APIs. You cannot simply swap `getopt` for `argparse` and neither of them will do anything at all with configuration files like `ini` or `json`. And if applications do work with some configuration file of choice it usually doesn't support rich types such as classes, functions and Python types that aren't built in.

For example, it is possible with `configman` to define configuration in `json` and then automatically have `ini` file and command line support. Further, configman enables configuration values to be dynamically loaded Python objects, functions, classes or modules. These dynamically loaded values can, in turn, pull in more configuration definitions and more dynamic loading. This enables configman to offer configurable plugins for nearly any aspect of a Python application.

CHAPTER 2

Getting started

Once you've *understood what configman is*, all you need to do is to install it:

```
$ pip install configman
```

The code is available on github: https://github.com/mozilla/configman To clone it all you need to do is:

```
$ git clone git://github.com/mozilla/configman.git
```

Once you have it installed, you usually start by importing `configman` in your scripts and programs, define the options in Python and then start exploring how you can *use config files* and more advanced *type conversions*.

CHAPTER 3

Tutorial

We're going to go from a simple application without `configman`, to a simple application with `configman`.

3.1 Basics

Suppose we write an app similar to the `echo` command line program in Unix and Linux. Our app, though reverses the lettering of each word:

```
def backwards(x):
    return x[::-1]

if __name__ == '__main__':
    import sys
    output_string = ' '.join(sys.argv[1:])
    print backwards(output_string)
```

running examples

each example in this tutorial is available as file in the `configman/demo` folder.

When run it could look something like this:

```
$ ./tutorial01.py Peter was here
ereh saw reteP
```

Now, suppose you want to add some more options that can be selected at run time. For example, it could remove all vowels from reversed string. So you can do this:

```
$ ./tutorial02.py --devowel Lars was here
rh sw srL
```

First, let's improve our business logic to add this new feature:

```
def backwards(x):
    return x[::-1]

import re
vowels_regex = re.compile('[AEIOUY]', re.IGNORECASE)

def devowel(x):
    return vowels_regex.sub('', x)
```

Now we need `configman` to enable the run time option for removing the vowels from the input string. It will take the form of a command line switch. If, at run time, the switch is present, we'll use the `devowel` function. If the switch is not present, we won't use the function.

To add this option we create an option container. These containers are called *namespaces* and have a method that allows us to define our command line options.

Here's a function that sets up a namespace with a single option:

```
from configman import Namespace, ConfigurationManager
...
def define_config():
    definition = Namespace()
    definition.add_option(
      name='devowel',
      default=False
    )
```

Before we can use this option definition, we need to wrap it up in a `ConfigurationManager` instance that is able to cook it up for us correctly:

```
...
config_manager = ConfigurationManager(definition)
config = config_manager.get_config()
```

That's all! That last line returned an instance of what we call a DotDict. It is essentially a standard Python dict that's had its `__getattr__` cross wired with its `__getitem__` method. This means that we can access the values in the dict as if they were attributes. Watch how we access the value of `devowel` in the full example below:

```
from configman import Namespace, ConfigurationManager

def backwards(x, capitalize=False):
    return x[::-1]

import re
vowels_regex = re.compile('[AEIOUY]', re.IGNORECASE)

def devowel(x):
    return vowels_regex.sub('', x)

def define_config():
    definition = Namespace()
    definition.add_option(
      name='devowel',
      default=False
    )
    return definition

if __name__ == '__main__':
    definition = define_config()
    config_manager = ConfigurationManager(definition)
    config = config_manager.get_config()
    output_string = ' '.join(config_manager.args)
    if config.devowel:
        output_string = devowel(output_string)
    print backwards(output_string)
```

When run, you get what you expect:

```
$ ./tutorial02.py Peter was here
ereh saw reteP
$ ./tutorial02.py --devowel Peter was here
rh sw rtP
```

In the `tutorial01.py` example, we fetched the command line arguments using the reference to argv from the sys module. We couldn't do that in the second tutorial because sys.argv included the command line switch `--devowel`. We don't want that as part of the output. `configman` offers a version of the command line arguments with the switches removed. That's the `config_manager.args` reference inside the `join`.

3.2 Intermediate

Now let's expand some of the more powerful features of `configman` to see what it can help us with. Let's start with the help. You invoke the help simply by running it like this:

```
$ ./tutorial02.py --help
```

That's set up automatically for you. As you can see, it mentions, amongst other things, our `--devowel` option there. Let's change the definition of the option slightly to be more helpful:

```
def define_config():
    definition = Namespace()
    definition.add_option(
      name='devowel',
      default=False,
      doc='Removes all vowels (including Y)',
      short_form='d'
    )
```

Now, when running `--help` it will explain our option like this:

```
-d, --devowel    Removes all vowels (including Y)
```

Let's add another option so that we can get our text from a file instead of the command line. The objective is to get a file name from a `--file` or `-f` switch. We'll set the default to be the empty string. If the user doesn't use the switch, the value for this will be the empty string:

```
...
definition.add_option(
  name='file',
  default='',
  doc='Filename that contains our text',
  short_form='f'
)
```

Excellent! The whole thing together looks like this now:

```
from configman import Namespace, ConfigurationManager

def backwards(x, capitalize=False):
    return x[::-1]

import re
vowels_regex = re.compile('[AEIOUY]', re.IGNORECASE)

def devowel(x):
    return vowels_regex.sub('', x)
```

```
def define_config():
    definition = Namespace()
    definition.add_option(
      name='devowel',
      default=False,
      doc='Removes all vowels (including Y)',
      short_form='d'
    )
    definition.add_option(
      name='file',
      default='',
      doc='file name for the input text',
      short_form='f'
    )
    return definition

if __name__ == '__main__':
    definition = define_config()
    config_manager = ConfigurationManager(definition)
    config = config_manager.get_config()
    if config.file:
        with open(config.file) as f:
            output_string = f.read().strip()
    else:
        output_string = ' '.join(config_manager.args)
    if config.devowel:
        output_string = devowel(output_string)
    print backwards(output_string)
```

And it's executed like this:

```
$ cat > foo.txt
Peter works for Mozilla.^d
$ ./tutorial03.py  --file foo.txt
.allizoM rof skrow reteP
$ ./tutorial03.py --file foo.txt -d
.llzM rf skrw rtP
```

3.3 Persistent config files

Our examples so far have been very much about the command line. The whole point of using `configman` is so you can use various config file formats to provide configuration information to your programs. The real power of `configman` isn't to wrap executable command line scripts but its ability to work *ecumenically* with config files.

admin options

controlling configman at run time

`configman` adds some command line parameters to your application that are used to control `configman` itself. To avoid name collisions with command line switches that you define, we've isolated these switches with the namespace, `admin`.

To get started, let's have our program itself write a configuration file for us. The easiest way is to use the `--admin.dump_conf` option that is automatically available. It offers different ways to output.

- ini
- conf
- json
- xml (future enhancement, if requested)

Let's, for the sake of this tutorial, decide to use .ini files:

```
$ ./tutorial03.py --admin.dump_conf=./backwards.ini
```

This will write out a default configation file in ini format. configman figured that out by the file extension that you specified. If you had used 'json' instead, configman would have written out a json file:

```
$ ./tutorial03.py --admin.dump_conf=./backwards.ini
$ cat backwards.ini
[top_level]
# name: devowel
# doc: Removes all vowels (including Y)
devowel=False

# name: file
# doc: Filename that contains our text
file=
```

Any of the command line switches that you specify along with the --dump_conf switch will appear as the new defaults in the config file that is written:

```
$ python backwards.py --admin.dump_conf=./backwards.ini --file=/tmp/foo.txt
$ cat backwards.ini
[top_level]
# name: devowel
# doc: Removes all vowels (including Y)
devowel=False

# name: file
# doc: Filename that contains our text
file=/tmp/foo.txt
```

Next, let's make our app always read from this file to get its defaults. To do that, we're going to modify what is known as the hierarchy of value sources. configman, when determining what values to give to your option definitions, uses a list of sources. By default, it first checks the operating system environment. If the names of your options match anything from the environment, configman will pull those values in, overriding any defaults that you specified. Next it looks to the command line. Any values that it fetches will override the defaults as well as the environment variables.

If this default hierarchy of value sources doesn't suit you, you may specify your own hierarchy. In our example, we're going to want our configuration file to be the base value source. Then we want the environment variables and finally the command line. We can specify it like this:

```
value_sources = ('./backwards.ini', os.environ, getopt)
```

configman will walk this list, applying the values that it finds in turn. First it will read your ini file (you may want to use an absolute path to specify your ini file name). Second, we pass in a dict that represents the operating system environment. Interestingly, you can use any dict-like object that you want as a source. Third, we're telling configman to use the getopt module to read the command line. In the future, we'll have the argparse module available here.

To use this value source, we must specify it in the constructor:

```
config_manager = ConfigurationManager(definition,
                                      values_source_list=value_sources)
```

Now, the program will read from the `./backwards.ini` config file whenever the application is run.

Suppose we change the last line of the file `backwards.ini` to instead say:

```
file=/tmp/bar.txt
```

And then create that file like this:

```
$ echo "Socorro" > /tmp/bar.txt
```

Now, our little program is completely self-sufficient:

```
$ ./tutorial04.py
orrocoS
```

Even though we're using a config file, that doesn't mean that we've eliminated the use of the command line. You can override any configuration parameter from command line:

```
$ ./tutorial04.py --devowel
rrcS
$ ./tutorial04.py We both work at Mozilla --file=
allizoM ta krow htob eW
```

3.4 More advanced options

We just covered how to turn a simple application to one where the configuration is done entirely by a `ini` file. Note: we could have chosen `json` or `conf` instead of `ini` and the program would be completely unchanged. Only your taste of config file format changed.

CHAPTER 4

Type conversion

`configman` comes with an advanced set of type conversion utilities. This is necessary since config files don't allow rich python type to be expressed. The way this is done is by turning things into strings and turning strings into rich python objects by labelling what type conversion script to use.

A basic example is that of booleans as seen in the *Tutorial* when it dumps the boolean `devowel` option as into an `ini` file. It looks like this:

```
[top_level]
# name: devowel
# doc: Removes all vowels (including Y)
devowel=False
```

As you can see it automatically figured out that the convertor should be `configman.converters.boolean_converter`. As you can imagine; under the hood `configman` does something like this:

```
# pseudo code
converter = __import__('configman.converters.boolean_converter')
actual_value = converter('False')
```

So, how did it know you wanted a boolean converter? It picked this up from the definition's default value's type itself. Reminder; from the *Tutorial*:

```
definition = Namespace()
definition.add_option(
  'devowel',
  default=False
)
```

4.1 Built-ins

The list of `configman` built-in converters will get you very far for basic python types. The complete list is this:

- **int**
- **float**
- **str**
- **unicode**
- **bool**

- **datetime.datetime** (`%Y-%m-%dT%H:%M:%S` or `%Y-%m-%dT%H:%M:%S.%f`)
- **datetime.date** (`%Y-%m-%d`)
- **datetime.timedelta** (for example, `1:2:0:3` becomes `days=1, hours=2, minutes=0, seconds=3`)
- **type** (see below)
- **types.FunctionType** (see below)
- **compiled_regexp_type**

The **type** and **types.FunctionType** built-ins are simpler than they might seem. It's basically the same example pseudo code above. This example should demostrate how it might work:

```
import morse
namespace.add_option(
  'morsecode',
  '',
  'Turns morse code into real letters',
  from_string_converter=morse.morse_load
)
```

What this will do is it will import the python module `morse` and expect to find a function in there called `morse_load`. Suppose we have one that looks like this:

```
# This is morse/__init__.py
dictionary = {
  '.-.': 'p',
  '.': 'e',
  '-': 't',
  '.--.': 'r',
}


def morse_load(s):
    o = []
    for e in s.split(','):
        o.append(dictionary.get(e.lower(), '?'))
    return ''.join(o)
```

Another more advanced example is to load a *class* rather than a simple value. To do this you'll need to use one of the pre-defined `configman` converters as the `from_string_converter` value. To our example above we're going to add a configurable class:

```
from configman.converters import class_converter
namespace.add_option(
  'dialect',
  'morse.ScottishDialect',
  'A Scottish dialect class for the morse code converter',
  from_string_converter=class_converter
)
```

That needs to exist as an importable class. So we add it:

```
# This is morse/__init__.py
class ScottishDialect(object):
    def __init__(self, text):
        self.text = text

    def render(self):
        return self.text.replace('e', 'i').replace('E','I')
```

Now, this means that the class is configurable and you can refer to a specific class simply by name and it becomes available in your program. For example, in this trivial example we can use it like this:

```
if __name__ == '__main__':
    config = create_config()
    dialect = config.dialect(config.morsecode)
    print dialect.render()
```

If you run this like this:

```
$ python morse-communicator.py --morsecode=.,-,.--.,-,.
itrti
```

This is just an example to whet your appetite but a more realistic example is that you might have a configurable class for sending emails. In production you might have it wired to be to something like this:

```
namespace.add_option(
  'email_send_class',
  'backends.SMTP',
  'Which backend should send the emails',
  from_string_converter=class_converter
)
namespace.add_option(
  'smtp_hostname',
  default='smtp.mozilla.org',
)
namespace.add_option(
  'smtp_username',
  doc='username for using the SMTP server'
)
namespace.add_option(
  'smtp_password',
  doc='password for using the SMTP server'
)
```

Then, suppose you have different backends for sending SMTP available you might want to run it like this when doing local development:

```
# name: email_send_class
# doc: Which backend should send the emails
dialect=backends.StdoutLogDumper
```

So that instead of sending over the network (which was default) it uses another class which knows to just print the emails being sent on the stdout or some log file or something.

4.2 Not built-ins

Suppose none of the built-ins in `configman` is what you want. There's nothing stopping you from just writing down your own. Consider this tip calculator for example:

```
import getopt
from configman import Namespace, ConfigurationManager


def create_config():
    namespace = Namespace()
    namespace.add_option(
```

```
      'tip',
      default=20
    )
    import decimal
    namespace.add_option(
      'amount',
      from_string_converter=decimal.Decimal
    )
    value_sources = ('tipcalc.ini', getopt, )
    config_manager = ConfigurationManager([namespace], value_sources)
    return config_manager.get_config()


if __name__ == '__main__':
    config = create_config()
    tip_amount = config.amount * config.tip / 100
    print "(exact amount: %r)" % tip_amount
    print '$%.2f' % tip_amount
```

When run it will automatically convert whatever number you give it to a python `Decimal` type. Note how in the example it prints the `repr` of the calculated value:

```
$ python tipcalc.py --amount 100.59 --tip=25
(exact amount: Decimal('25.1475'))
$25.15
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search