Python programming — Scripting

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

July 4, 2014
Overview

How to make a command-line script (as oppose to a module)?

Header

Argument parsing

__main__

Command-line input

Standard input/output and piping
Naming

It is not necessary to call a script run from the command-line with the ".py" extension.

Actually it might be better to hide the implementation (that it is written in python) from the user (for some operating systems).
Header in Linux-like environment

The hash-bang at the top

#!/usr/bin/python

enabling you to run the script like (after setting of the execution bit with chmod a+x myscript):

$ myscript

rather than

$ python myscript

or if you are afraid the python program you want is not installed in /usr/bin (think virtualenv):

#!/usr/bin/env python
Header in Windows-like environment

Hashbang does not work in Windows.

If you instead maintain the .py extension then you are able to ASSOC and FTYPE commands to associate a filetype to a specific program (such as the python program. See the suggestion on Stack Overflow.
Command-line argument basics

Command-line arguments are available in the `sys.argv` variable.

With `myscript` consisting of

```python
#!/usr/bin/env python
import sys
print(sys.argv)
```

Called with 3 command-line arguments:

```
$ ./myscript --verbose -a=34 datafile.txt
['myscript', '--verbose', '-a=34', 'datafile.txt']
```

Note there are four items in the list: The first element is the Python program name.
Argument parsing in the old days

For reading/parsing the command-line arguments in `sys.argv` you can write your own code, but there are developers who have written module to ease the handling of the arguments.

In the old days you would have:

- **getopt** — Module in the standard library modeled after the C `getopt` function/library. Not necessarily recommended.

- **optparse** — In the standard library. Not necessarily recommended.

- **argparse** — Added to standard library from 2.7/3.2 see PEP 389. Newest module in the standard library and—argued—better than `getopt` and `optparse`.
argparse example

A lot of code goes here.
Python scripting

But now
docopt
Docopt

Idea: Use the documentation to describe the command-line interface — both for humans and the argument parsing code.

Available for a number of programming languages.

Reference implementation in Python by Vladimir Keleshev.

No longer necessary to write much code only:

```python
import docopt
args = docopt.docopt(__doc__, version=__.version__)
```

The rest is documentation (and the code for actually using the command-line arguments)
Docopt example

#!/usr/bin/env python

mydocopter.

Usage: mydocopter [options] <filename>

Options:
  -v --verbose   Log messages
  -o OUTPUT --output=OUTPUT  Output file
  -a <a>         Initial coefficient for second order term [default: 1.]
  -b <b>         Initial coefficient for first order term [default: 1.]
  -c <c>         Initial coefficient for constant term [default: 1.]

Example:
  $ echo -e "1 4\n2 5\n6 8\n3 3.2" > datafile.txt
  $ ./mydocopter --verbose datafile.txt
  0.315471154631 -1.51271481921 5.64476836068

Description:
  Fit a polynomial to data. The datafile should have x y values in each row
With just the following two lines you get 'usage' and 'help' working:

```python
import docopt
args = docopt.docopt(__doc__, version=1.0)
```

Calling the program with wrong arguments (here `<filename>` is missing):

```
$ python mydocopter
Usage: mydocopter [options] <filename>
```

Calling the program for help (-h or --help) prints the docstring:

```
$ python mydocopter --help
mydocopter.

Usage: mydocopter [options] <filename>
... (and the rest of the docstring)
```
What is in args?

With this program

```python
import docopt
args = docopt.docopt(__doc__, version=1.0)
print(args)
```

Example outputs:

```bash
$ mydocopter datafile.txt
{’--output’: None, ’--verbose’: False, ’-a’: ’1.’, ’-b’: ’1.’, ’-c’: ’1.’, ’<filename>’: ’datafile.txt’}

$ mydocopter --verbose -b 3 datafile.txt
{’--output’: None, ’--verbose’: True, ’-a’: ’1.’, ’-b’: ’3’, ’-c’: ’1.’, ’<filename>’: ’datafile.txt’}
```
the code of a working program
import docopt, logging, scipy.optimize

args = docopt.docopt(__doc__, version=1.0)

if args['--verbose']:
    logging.getLogger().setLevel(logging.INFO)

a, b, c = (float(args['-' + coef]) for coef in ['a', 'b', 'c'])
logging.info("Setting 'a' to \%f" % a)

logging.info('Reading data from ' + args['<filename>'])
data = [ map(float, line.split()) for line in open(args['<filename>']).readlines()]

def cost_function((a, b, c), data):
    return sum(map(lambda (x, y): (a*x**2 + b*x + c - y)**2, data))

parameters = scipy.optimize.fmin(cost_function, [a, b, c],
    args=(data,), disp=False)

if args['--output'] is None:
    print(" ".join(map(str, parameters)))
else:
    with open(args['--output'], 'w') as f:
        f.write(" ".join(map(str, parameters)))
**Docopt details**

Notice short and long forms (`-v` and `--verbose`, `-h` and `--help`)

Required fixed arguments, required variable argument (`<filename>`) and optional (e.g., `-a 3`)

Options with (e.g., `-a 3`) and without values (e.g., `--verbose`)

Options with default values `[default: 1.]`

Furthermore:

You can have “or” arguments, e.g., `(set|remove)`

You can have multiple input arguments to a single name with “…”, e.g., program `<filename>...` with parsed command-line element available in a list `<filename>`: `[‘a.txt’, ‘b.txt’, ‘c.txt’]`
'Variable constants' and input arguments

Do not usually use 'constants that varies'(!?) in programs. Put them as input arguments. It might be filename for output and input:

""
Usage:
    myprogram [--output=<filename>] <input>
""
# Here goes program
...

Rather than hardcoded 'constants':

# Here goes program
INPUT_FILENAME = 'data_2014_first_recording.csv'
OUTPUT_FILENAME = 'data_2014_first_recording_analysis_results.txt'
...
The “if __name__ == '__main__':” thing

To distinguish a script from a module.

```python
print('This is executed when the file is executed or imported')
if __name__ == '__main__':
    print('This is executed when the file is executed, '
    'not when imported')
```

It allows a script to be used both as a script as well as a module (if it has the .py extension.

Documentation tools that require import (but does not execute the code) will benefit from this trick.
The def main() thing

Instead of putting code in the __name__ == '__main__' block add a function (usual name: main), e.g., here with a module named onemodule.py:

```python
def main():
    print("This is the main function")

if __name__ == '__main__':
    main()
```

This construct allows you to call the “script” (i.e., onemodule.py) from another module, e.g., like:

```python
import onemodule
onemodule.main()
```

This would not have been possible if you put the line with print in the block with __name__ == '__main__'. See also python - why use def main() and the Google Python Style Guide.
Python scripting

Command-line input

Python interactive command-line interface to Python with coloring of '5'

```python
import blessings, re, readline, rlcompleter

readline.parse_and_bind("tab: complete")  # For tab completion
_term = blessings.Terminal()  # For coloring text output

while True:
    expr = raw_input(">>> ")
    try:
        _ = eval(expr)
        print(re.sub('5', _term.bold_red_on_green('5'), str(_),
                      flags=re.UNICODE))
    except:
        exec(expr)
```

Note the behavior and existence of `raw_input()` and `input()` is different between Python 2 and Python 3.
Input/output streams

raw_input (Python 2) in Python 3 called input

input (Python 2), the same as eval(input())

gtapass.getpass Input with hidden output

sys.stdin Standard input stream for interpreter input

sys.stdout Standard output stream

sys.stderr Standard error stream

The original objects of the three latter are in sys.__stdin__ etc.
Unix pipe example

Example with a Unix pipe:

$ echo "Hallo" | python -c "import sys; sys.stdout.write('<' + 
    sys.stdin.read().strip() + '>
"

<Hallo>
try:
    from msvcrt import getch
except ImportError:
    def getch():
        import sys, tty, termios
        fd = sys.stdin.fileno()
        old_settings = termios.tcgetattr(fd)
        try:
            tty.setraw(fd)
            ch = sys.stdin.read(1)
        finally:
            termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
        return ch
...Reading unbuffered

def getchs():
    while True:
        yield getch()

import sys

# see also getpass module.
for ch in getchs():
    sys.stdout.write('*')
    if ch == 'c':
        break
    elif ch == '':
        sys.stdout.write('
-------------
')
More information

Vladimir Keleshev’s YouTube video about docopt: PyCon UK 2012: Create *beautiful* command-line interfaces with Python
Summary

Use `docopt`: Easiest handling of command-line arguments, forces you to document your script, ensures that your documentation and implementation do not get out of sync.

Use `__name__ == '__main__' + main()` blocks rather than placing code in the global namespace of the module.

Consider different ways of getting input: command-line arguments, interactive input, standard out and in, files.

Put ‘variable constants’ as input arguments.