Python 3: Argument parsing

Bob Dowling rjd4@cam.ac.uk

29 October 2012

Prerequisites

This self-paced course assumes that you have a knowledge of Python 3 equivalent to having completed one or other of

- Python 3: Introduction for Absolute Beginners, or
- Python 3: Introduction for Those with Programming Experience

Some experience beyond these courses is always useful but no other course is assumed.

The course also assumes that you know how to use a Unix text editor (`gedit`, `emacs`, `vi`, …).

Facilities for this session

The computers in this room have been prepared for these self-paced courses. They are already logged in with course IDs and have home directories specially prepared. Please do not log in under any other ID.

At the end of this session the home directories will be cleared. Any files you leave in them will be deleted. Please copy any files you want to keep.

The home directories contain a number of subdirectories one for each topic.

For this topic please enter directory `argparse`. All work will be completed there:

```
$ cd argparse
$ pwd
/home/x250/argparse
$
```

These courses are held in a room with two demonstrators. If you get stuck or confused, or if you just have a question raised by something you read, please ask!

These handouts and the prepared folders to go with them can be downloaded from
`www.ucs.cam.ac.uk/docs/course-notes/unix-courses/pythontopics`

The formal Python 3 documentation for the topics covered here can be found online at
`docs.python.org/release/3.2.3/library/argparse.html`

Table of Contents

Prerequisites.....1
Facilities for this session.....1
Notation.....3
Warnings.....3
Exercises.....3
Exercise 0.....3
Input and output.....3
Keys on the keyboard.....3
Content of files.....3
What’s in this course.....4
Quick recap: The primitive command line.....4
Exercise 1.....4
Our target scripts.....4
Exercise 2.....5
A script with no arguments at all.....5
Exercise 3.....7
Adding optional arguments.....7
Exercise 4.....9
Adding a “pure flag”.....10
Exercise 5.....10
Adding a compulsory argument.....10
Exercise 6.....11
Multiple positional arguments.....11
File names as arguments.....12
Exercise 7.....13
And that’s it!.....14
Exercise 8.....14

Notation

Warnings

Warnings are marked like this. These sections are used to highlight common mistakes or misconceptions.

Exercises

Exercise 0

Exercises are marked like this. You are expected to complete all exercises. Some of them do depend on previous exercises being successfully completed.
If you are stuck, please ask a demonstrator.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
--More--(44%)
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: “`Lorem ipsum`”. (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the “A key” will be written “[A]”. Note that the return key (pressed at the end of every line of commands) is written “[↲]”, the shift key as “[⇧]”, and the tab key as “[↹]”. Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as “[⇧]+[A]”. Note that pressing [A] generates the lower case letter “a”. To get the upper case letter “A” you need to press [⇧]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur.                              This is a comment about the line.
```

1 The example text here is the famous “lorem ipsum” dummy text used in the printing and typesetting industry. It dates back to the 1500s. See http://www.lipsum.com/ for more information.

What's in this course

1. A quick recap of the primitive command line
2. Our target scripts
3. A script with no arguments at all
4. Adding optional arguments
5. Adding repeated arguments
6. Adding a compulsory argument
7. Files as arguments

Quick recap: The primitive command line

It never hurts to recap what we know already. The `sys` module contains a member `sys.argv` which is a list containing the command line arguments passed to the Python command.

Exercise 1

Check that you can understand and run the script `args.py` in your directory:

```
$ python3 args.py alpha beta gamma
['args.py', 'alpha', 'beta', 'gamma']
$ python3 args.py 1 2 3
['args.py', '1', '2', '3']
$
```

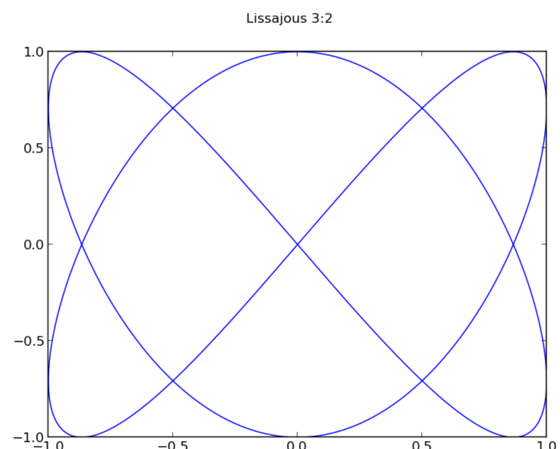
Note that all the arguments are presented as strings.

Our target scripts

We want a script that processes the command line with the same power as a "real" Unix application. The specific features we want from our command line parser are support for:

- short form options such as "`-a`" or "`-b`" which can be combined as "`-ab`".
- long form options such as "`--alpha`" or "`--beta`".
- values set with or without equals signs: "`--alpha foo`" should be the same as "`--alpha=foo`"
- automatic conversion of types
- "`--help`" or "`-h`" giving help on the other options
- command line arguments without any "`-a`" or "`--alpha`" elements

To help introduce (at least some of) these features it will help to have a concrete example in mind. The script `l0issajous.py` creates a graph of a Lissajous figure, based on arguments passed to it on the command line.



The graph shown, for example, was generated by

```
$ python3 lissajous.py --xfreq 2 --yfreq 3 --title='Lissajous 3:2' graph.png
```

We are not going to address the creation of the graph in this topic. See the Python graphics topic for that. We are going to learn how to process that command line. Note that both of the following command lines would have been exactly equivalent:

```
$ python3 lissajous.py -x 2 -y 3 -t 'Lissajous 3:2' graph.png
$ python3 lissajous.py --xfreq=2 --yfreq=3 --title 'Lissajous 3:2' graph.png
```

Also note that the script generates help on request

```
$ python3 lissajous.py --help
usage: lissajous.py [-h] [-n n] [-x a] [-y b] [-p c] [-t title] fname

Plot a graph of x=sin(at), y=sin(bt+c), with graph title provided by the user.

positional arguments:
  fname                 File name for graph (required)

optional arguments:
  -h, --help            show this help message and exit
  -n n, --npts n        Number of points to plot
  -x a, --xfreq a       Frequency of the x-oscillation
  -y b, --yfreq b       Frequency of the y-oscillation
  -p c, --phase c       Phase difference between the x- and y-oscillations
                        measured in radians
  -t title, --title title
                        Title of graph

Author: Bob Dowling <rjd4@cam.ac.uk>
```

and arguably useful error messages if your command line makes no sense to it:

```
$ python3 lissajous.py --xfreq two --yfreq 3 --title='Lissajous 3:2' graph.png
usage: lissajous.py [-h] [-n n] [-x a] [-y b] [-p c] [-t title] fname
lissajous.py: error: argument -x/--xfreq: invalid float value: 'two'
```

All of the help information and the error message shown are produced by the command line parser.

Exercise 2

Try out the `lissajous.py` script with a variety of command line arguments, some correct, some incorrect.

The `lissajous.py` script can act as your guide during this topic. You will be creating a different script that plots Chebyschev polynomials. (Don't worry. We will provide the maths and graphics; you will only have to provide the command line parsing.)

A script with no arguments at all

We will start with the bare minimum: a script with no arguments at all: `example01.py`.

The script starts by importing the `argparse` module:

```
import argparse
```

Then it creates the “argument parser” object which will do the hard work of processing the command line:

```
parser = argparse.ArgumentParser()
```

Then it uses that object to parse the command line. It places the results of that parsing in an object named "`arguments`" which, for the time being, it simply prints:

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

This demonstrates that the arguments returned is a Python "`Namespace`":

```
$ python3 example01.py
Namespace()
```

A namespace is, essentially, a blank object waiting to have some attributes added to it. This one is empty.

Our script, however, is still capable of providing help:

```
$ python3 example01.py --help
usage: example01.py [-h]

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

This output is generated by the `parse_args()` method. If it generates help text it stops execution at that point; the `print()` statement is not reached.

The script now explicitly rejects any command line argument. The parser must know about everything legal and it rejects everything else as illegal:

```
$ python3 example01.py --foo
usage: example01.py [-h]
example01.py: error: unrecognized arguments: --foo

$ python3 example01.py thing
usage: example01.py [-h]
example01.py: error: unrecognized arguments: thing

$
```

All of this is automatically generated. Obviously we want to add to it, both by providing some help text about what the program does and by adding some legal command line arguments.

We start with the additional text, both in front of the automatically generated output and after it.

The text in front is typically a brief description of what the program does and is almost always present. The `argparse` module refers to this text as the "description" in expectation of this usage.

The text afterwards is most commonly an authorship or contact tag and is often omitted. The `argparse` module gives it the more neutral name of the "epilogue" (albeit with an American spelling).

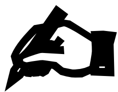
The two pieces of text are specified when the parser is created:

```
help_text = 'Plot a graph of x=sin(at), y=sin(bt+c), with a graph title.'
sign_off = 'Author: Bob Dowling <rjd4@cam.ac.uk>'

parser = argparse.ArgumentParser(description=help_text, epilog=sign_off)
```

The file `example02.py` has this extra functionality. Note how this augments the help output:

```
$ python3 example02.py --help
usage: example02.py [-h]

Plot a graph of x=sin(at), y=sin(bt+c), with a graph title.

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

Author: Bob Dowling <rjd4@cam.ac.uk>
```

And now it is your turn:



Exercise 3

Create a script, `exercise03.py`, with help text as shown. Identify yourself as the author, not Bob Dowling!

```
$ python3 exercise03.py --help
usage: exercise03.py [-h]

Plot graphs of the Chebyshev polynomials of orders M to N
for values between -X and +X.

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

Author: Bob Dowling <rjd4@cam.ac.uk>
```

You can base your `exercise03.py` script on `example02.py`.

Adding optional arguments

Our worked example that generates Lissajous figures takes a number of optional arguments, illustrated by the help output:

```
optional arguments:
  -h, --help            show this help message and exit
  -n n, --npts n        Number of points to plot
  -x a, --xfreq a       Frequency of the x-oscillation
  -y b, --yfreq b       Frequency of the y-oscillation
  -p c, --phase c       Phase difference between the x- and y-oscillations
                        measured in radians
  -t title, --title title
                        Title of graph
```

We will consider one of these to establish our vocabulary:

```
-x a, --xfreq a         Frequency of the x-oscillation
```

`-x`	the short format flag
`--xfreq`	the long format flag
a	the “metavariable” name
`Frequency of…`	the help text

(The “metavariable” is the letter or word used to refer to the value corresponding to the flag in the text generated. It will have no other effect.)

We will now go through the process of telling the parser that it should recognise this optional argument and its corresponding value. To do this we need to specify a little more about the value it will take.

We need to give it a name in the `arguments` object (as opposed to the name in the help text). The argument parser can make a name up based on the long format flag and would call it “`arguments.xfreq`” in our example but it is better practice to name it explicitly. We call this its “destination”.

We need to tell the argument parser what to do with the option. In the case of “`--xfreq`” we expect the flag to be followed by a value which we want stored. This is called the “action” and storing a value is the default action but in this example we will be explicit.

There are alternative actions. For example the flag might not take any values but might be repeated. We would want to count the number of repeats. We will meet these alternative actions later.

We also need to specify what Python type that value ought to be. To specify the type we pass the name of the type itself. What is happening internally is that the parser needs a function that will convert a string (as given to it by the command line) into the type we actually want. Every type in Python has a function named after it that takes a string and creates a value of that type from it (`int()`, `str()`, `float()` etc.)We are passing that function in as an argument to the parser.

Finally, we need to specify a value that this argument will take if the user does not specify one. This is called the "default value".

This gives us three more things we will need to specify:

destination	The name of the variable in the `arguments` object.
action	What to do with this flag (e.g. expect a value and to store it).
type	What type that value should be.
default	What value to use if the user doesn't specify one.

The file `example03.py` has the extra stage to add the "`--xfreq`" option to the parser. This is not done at parser creation (`argparse.ArgumentParser()`) but afterwards — typically immediately afterwards — using a method of the parser, `parser.add_argument()`:

```
parser = argparse.ArgumentParser(description=help_text, epilog=sign_off)

parser.add_argument(
    '--xfreq',
    '-x',
    dest='xfreq',
    action='store',
    type=float,
    default=1.0,
    help='Frequency of the x-oscillation',
    metavar='a'
    )

arguments = parser.parse_args()
```

We can now use this attribute and receive help on it.

For example, our help now knows about it:

```
$ python3 example03.py --help
usage: example03.py [-h] [--xfreq a]

Plot a graph of x=sin(at), y=sin(bt+c), with a graph title.

optional arguments:
  -h, --help       show this help message and exit
  --xfreq a, -x a  Frequency of the x-oscillation

Author: Bob Dowling <rjd4@cam.ac.uk>
```

We can also use the option to set a value:

```
$ python3 example03.py --xfreq 2.0
Namespace(xfreq=2.0)

$ python3 example03.py --xfreq=3.0
Namespace(xfreq=3.0)
```

And we can see the default taking effect:

```
$ python3 example03.py
Namespace(xfreq=1.0)
```

We also need to see how to get into this “namespace” object. That’s easy:

```
arguments = parser.parse_args()
print(arguments)
print(arguments.xfreq)
```

You can see this in `example04.py`:

```
$ python3 example04.py --xfreq=4.0
Namespace(xfreq=4.0)
4.0
```

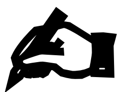
The name that follows the dot is the name specified as the destination by the `dest=`… keyword.

The script `example05.py` has all the options added.

Note that because we told the argument parser what the type of the value should be *it* can produce the error messages for us if the text passed is invalid:

```
$ python3 example05.py --yfreq TWO graph.png
usage: example05.py [-h] [-n n] [-x a] [-y b] [-p c] [-t title] fname
example05.py: error: argument -y/--yfreq: invalid float value: 'TWO'
```

And now it is your turn:



Exercise 4

Copy your script, `exercise03.py`, to `exercise04.py` and update that script to support the optional arguments (all to be stored) shown in this table:

```
-x   --limit   float  1.0  limit X  Range of values of x
-m   --lower   int    1    lower M1 Minimum order of
                                    polynomial
-n   --upper   int    3    upper M2 Maximum order of
                                    polynomial
-k   --npts    int    512  npts  N  Number of points to plot
-t   --title   str    ''   title T  Title of graph
```

The script should run like this:

```
$ python exercise04.py
Namespace(limit=1.0, max=3, min=1, npts=512, title='')
$ python exercise04.py --help
usage: exercise04.py [-h] [-k k] [-x X] [-m M] [-n N]
[-t title]

Plot graphs of the Chebyshev polynomials of orders M to N
for values between -X and +X.

optional arguments:
  -h, --help            show this help message and exit
  -k k, --npts k        Number of points to plot
  -x X, --limit X       Range of values of x.
  -m M1,--min M1        Minimum order of polynomial.
  -n M2,--max M2        Maximum order of polynomial.
  -t title, --title title
                        Title of graph

Author: Bob Dowling <rjd4@cam.ac.uk>
```

Don't try to reset the "-h" or "--help" flags.
You can do it, but it will only end in tears.

Adding a "pure flag"

There is a difference between --xfreq and --help. The former requires an argument after it and the latter doesn't. How do we add flags like --help, --debug or --verbose that take no further arguments?

There is no corresponding value to store, so the "action=store" parameter setting is clearly false. Instead we set "action=count". Instead of storing a user-given value in arguments, this action causes argparse to store a count of how often the flag appears. So a command line with "--verbose --verbose" would store a value of 2. The file example06.py has nothing but a verbosity flag:

```
$ python3 example06.py -vvv
Namespace(verbosity=3)

$ python3 example06.py --verbose --verbose
Namespace(verbosity=2)

$ python3 example06.py
Namespace(verbosity=0)
```

Exercise 5

Copy your script, exercise04.py, to exercise05.py and update that script to support a verbosity flag.

Adding a compulsory argument

To date we have been adding *optional* arguments; each of them has had a default value to use if the user didn't specify one. Optional arguments are passed with one of these -x/--xfreq flags. Compulsory arguments are typically passed unadorned. Both the worked example and your exercise take a file name as a compulsory argument. For a single, simple compulsory argument this is just a case of omitting the flags:

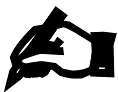
```
parser.add_argument(
    dest='filename',
    action='store',
    type=str,
    help='File name for graph (required)',
    metavar='fname'
    )
```

Because there are no flags specified this is known as a "positional" argument, defined by its position in the command line, rather than by any associated flag:

```
$ python3 example07.py --help
usage: example05.py [-h] [-n n] [-x a] [-y b] [-p c] [-t title] fname

Plot a graph of x=sin(at), y=sin(bt+c), with graph title provided by the user.

positional arguments:
  fname                 File name for graph (required)

optional arguments:
  -h, --help            show this help message and exit
  -n n, --npts n        Number of points to plot
  -x a, --xfreq a       Frequency of the x-oscillation
  -y b, --yfreq b       Frequency of the y-oscillation
  -p c, --phase c       Phase difference between the x- and y-oscillations
                        measured in radians
  -t title, --title title
                        Title of graph

Author: Bob Dowling <rjd4@cam.ac.uk>
```

The `argparse` module makes all positional arguments compulsory unless we override it (which we will do later). There is, therefore, no point in specifying a default value.

The order of the optional arguments in the help is dictated by the order of their `add_argument()` calls. Positional arguments are always placed at the end, regardless.



Exercise 6

Copy your script, `exercise05.py`, to `exercise06.py` and update that script to require a file name as its final argument.

Multiple positional arguments

There is a rather crude file copying program in `example06.py`. It requires an input and output file name simply by adding two positional arguments with `add_argument()`:

```
parser.add_argument(
    dest='source',
    type=str,
    metavar='src',
    help='Source file name'
    )


parser.add_argument(
    dest='target',
    type=str,
    metavar='tgt',
    help='Target file name'
    )
```

This is crude, but effective. The more common case is to have a number of positional arguments but not to know in advance how many and for this circumstance this crude approach doesn’t work.

To take a simple example, suppose we required a list of zero or more file names so that the program could run line/word/character counting statistics on all of them.

To tell `add_arguments()` that it should expect a number of arguments we add a parameter, “`nargs`” (pronounced “en-args”), and tell it how many to expect.

The mere presence of the `nargs` parameter means that the value stored in `arguments` will be a list, even if only one value is returned.

The most common value for the `nargs` parameter is the character "`*`" which means "zero or more":

```
parser.add_argument(
    dest='filenames',
    type=str,
    action='store',
    nargs='*',
    help='Names of files to be analyzed.'
    )
```

Note that the type is the type of the items in the list:

```
$ python3 example08.py *.py
Namespace(filenames=['example01.py', 'example02.py', 'example03.py',
'example04.py', 'example05.py', 'example06.py', 'example07.py',
'example08.py', 'exampleNN.py', 'exercise03.py', 'exercise04.py',
'exercise.py', 'lissajous.py'])
```

(The file `example09.py` actually runs the statistics.)

Also note that it is perfectly happy with no arguments at all:

```
$ python3 example08.py
Namespace(filenames=[])
```

List arguments like this automatically default to the empty list so no default parameter is needed in `add_arguments()`.

The `nargs` parameter can take other values. If it is given an integer value then it requires precisely that many arguments. There are also other single character values it can take:

`'*'`	Zero or more values. (The usual.)
`'+'`	One or more values. (Not uncommon.)
`'?'`	Zero or one value. (Rare.)
`5`	Exactly five values. (For any value of five.)
`1`	Exactly one value. (NB: Because `nargs` is set it will be a *list* of one value, not just the value.)
`argparse.REMAINDER`	All the remaining values, if any.

File names as arguments

Earlier we noted in passing that because we told the argument parser the expected types of values it could do the sanity checking of the command line arguments for us. We can take this a step further with file names. We can get the argument parser to check that a file exists and can be read if we want to read it or even open it for us and pass the Python file object as the value. Similarly if we want a file name to open for writing we can get the argument parser to open it for us and pass us back the file object. In both cases we can leave the error reporting to the parser.

What would this look like in practice?

We would change our compulsory argument definition to look like the following:

```
parser.add_argument(
    dest='file',
    action='store',
    type=argparse.FileType('wb'),
```

```
    help='File name for graph (required)',
    metavar='fname'
    )
```

Note the exotic-looking `type=argparse.FileType('wb')`. That is the only change we have made and is the difference between `example05.py` and `example10.py`. This type means that the argument should be opened as a **b**inary file for **w**riting. The argument to `argparse.FileType()` is the mode string that gets passed to the `open()` function.

The object stored in `arguments` is the open file object:

```
$ python3 example10.py graph.png
Namespace(file=<_io.BufferedWriter name='graph.png'>, npts=512, phase=0.0,
title='', xfreq=1.0, yfreq=1.0)
```

It also means that if the file can't be opened for writing for whatever reason the parser will handle the error for you:

```
$ python3 example10.py /etc/motd
usage: example10.py [-h] [-n n] [-x a] [-y b] [-p c] [-t title] fname
example10.py: error: argument fname: can't open '/etc/motd':
[Errno 13] Permission denied: '/etc/motd'
```

Note that the file will be created (or overwritten and truncated) just by parsing the command line:

```
$ ls -l graph.png
ls: cannot access graph.png: No such file or directory
$ python3 example10.py graph.png
Namespace(file=<_io.BufferedWriter name='graph.png'>, npts=512, phase=0.0,
title='', xfreq=1.0, yfreq=1.0)
$ ls -l graph.png
-rwx------ 1 rjd4 rjd4 0 Oct  4 17:29 graph.png
$
```

The argument parser will *open* the file for you. The onus is still on you to close it again when you are done with it:

```
arguments.file.close()
```

Is it a good idea to get the argument parser to do this for you? Should you do it yourself?

If you are doing something “clever” that might mean that you wouldn't open the files under certain circumstances then you must do it yourself. If you just want them opened and errors handled consistently and cleanly then leave it to the argument parser. *Don't reinvent the wheel.*

Exercise 7

Copy your script, `exercise06.py`, to `exercise07.py` and update that script to open the file for writing as part of the parsing.

And that's it!

You now know enough to write a very powerful parsing of the command line.

The file `example11.py` has exactly the same parsing as `example10.py` (though written a little more compactly) and is a fully functional script. Try it out!

Exercise 8

The script, `exercise08.py`, contains all the Python to create graphs of the Chebyshev polynomials. Copying the parsing lines from your `exercise07.py` to the top of the script should be all it takes to make it a fully functional script.

Is it more trouble than it's worth to write a fully-fledged command line parser?

Once you get used to the module it does not take too much time to write the lines of Python. It also makes your script vastly easier for other people to use. No academics work in isolation today. If you write a script that even you use more than once or twice the chances are that you will have to share it with a colleague. They will be grateful if you make it easy to use, and may even return the favour when they lend you one of theirs.

Good luck!