



University of  
Zurich <sup>UZH</sup>

S3IT

# Customizing session-based scripts

Riccardo Murri <[riccardo.murri@uzh.ch](mailto:riccardo.murri@uzh.ch)>

*S3IT: Services and Support for Science IT*

University of Zurich

# Outline

1. Customize positional argument processing
2. Customize option processing
3. More complex `new_tasks()` implementations

# What is command-line argument processing?

Problem:

```
$ python ex4a.py --foo=1 bar baz
```

How do we translate the above shell command invocation into a Python procedure call

```
ex4a('bar', 'baz', foo=1)?
```

In other words: how do we access the command-line arguments given to the shell from Python code?

# Command-line argument processing in GC3Pie

GC3Pie scripts use the standard Python library `argparse`.

However, differently from `argparse`, GC3Pie scripts use separate ways to configure processing of options and positional arguments.

# Positional argument processing

Positional arguments are defined in a method called `setup_args()`; override it in derived classes to change what arguments are accepted, their names and number.

This is the default implementation:

```
def setup_args(self):  
    self.add_param(  
        'args', nargs='*', metavar='INPUT',  
        help="Path to input file or directory.")
```

# Positional argument processing

```
def setup_args(self):  
    self.add_param(  
        'args', nargs='*', metavar='INPUT',  
        help="Path to input file or directory.")
```

The value of this command-line argument will be recorded in the variable `self.params.args` in Python code.

# Positional argument processing

```
def setup_args(self):  
    self.add_param(  
        'args', nargs='*', metavar='INPUT',  
        help="Path to input file or directory.")
```

There are 0 or more arguments of this kind.  
Makes `self.params.args` into a Python list.

## Positional argument processing

```
def setup_args(self):  
    self.add_param('args', nargs=..., [...])
```

Other possible values for the `nargs` parameter are:

- `nargs='+'` — there are *1 or more* arguments, i.e., at least one argument is required.
- `nargs=N` — there are exactly  $N > 1$  arguments of this kind.
- `nargs=1` (default) — one single argument of this kind; **in this case `self.params.args` is a *string*, not a list.**



# Positional argument processing

```
def setup_args(self):  
    self.add_param(  
        'args', nargs='*', metavar='INPUT',  
        help="Path to input file or directory.")
```

The name of the command-line argument as displayed to users in usage and help texts:

```
$ python solutions/ex2c.py --help  
usage: ex2c [-h] [-V] [-v] [--config-files CONFIG_FILES] [-c NUM]  
           [-m GIGABYTES] [-r NAME] [-w DURATION] [-s PATH] [-u URL]  
           [-C NUM] [-J NUM] [-o DIRECTORY] [-l [STATES]]  
           [INPUT [INPUT ...]]
```

If not given, defaults to the uppercased version of the “internal” name.

# Positional argument processing

```
def setup_args(self):  
    self.add_param(  
        'args', nargs='*', metavar='INPUT',  
        help="Path to input file or directory.")
```

Description of the command-line argument in `--help` text.

```
$ python solutions/ex2c.py --help  
usage: ex2c [-h] [...]
```

positional arguments:

INPUT

Path to input file or directory.

## Positional argument processing

Calls to `self.add_param()` can be repeated to parse many different command-line arguments in sequence:

```
class AScript(SessionBasedScript):  
    # [...]   
    def setup_args(self):  
        self.add_param('input', help="Input file")  
        self.add_param('radius', help="Convolution radius")  
        self.add_param('sigma', help="Threshold")
```

With the above definition, the following command-line:


```
$ python example.py file.img 10 2.1
```

generates the equivalent of the following Python code:

```
self.params.input = 'example.py'  
self.params.radius = '10' # it's a string!  
self.params.sigma = '2.1' # this one too!
```

## Detour: From grayscale to colors

```
$ convert gray-lena.jpg \  
    ( xc:blue xc:magenta xc:yellow +append ) \  
    -clut color-lena.jpg
```



fig/gray-lena.jpg

fig/arrow.png

fig/color-lena.jpg

**Exercise 4.A:** Write a `colorize.py` script to apply this colorization process to a set of grayscale images.

The `colorize.py` script shall be invoked like this:

```
$ python colorize.py c1 c2 c3 img1 [img2 ...]
```

where *c1*, *c2*, *c3* are color names and *img1*, *img2* are image files.

Each image shall be processed in a separate colorization task.

## Argument types

You can ask `argparse` and `GC3Pie` to convert a command-line argument to a certain Python type.

For example:

```
class AScript(SessionBasedScript):
    # [...]
    def setup_args(self):

        # this argument is a string (default type)
        self.add_param('input', type=str, help="...")

        # the 'radius' argument is an integer
        self.add_param('radius', type=int, help="...")

        # the 'sigma' argument is a floating-point number
        self.add_param('sigma', type=float, help="...")
```

## Argument types

Declaring argument types makes for better usability: if an argument does not match its type, the script exists immediately and the user is notified with a clear error message.

```
$ python downloads/argp.py foo.txt 123 x
usage: argparse [-h] [...]
               input radius sigma
argp: error: argument sigma: invalid float value: 'x'
```

## Argument types

In addition to the standard Python types, GC3Pie provides other validation functions to ensure arguments meet commonly-found conditions.

```
from gc3libs.cmdline import \
    existing_file, positive_int

class AScript(SessionBasedScript):
    # [...]
    def setup_args(self):
        # reject non-existent input files outright
        self.add_param('input', type=existing_file, ...)
        # force radius to be > 0
        self.add_param('radius', type=positive_int, ...)
        # sigma is a floating-point number
        self.add_param('sigma', type=float, ...)
```

*Reference:* <http://gc3pie.readthedocs.io/en/master/programmers/api/gc3libs/cmdline.html>



# Command-line option processing

Command-line options can be **added** to a session-based script by overriding the `setup_options()` method:

```
class AScript(SessionBasedScript):  
    # [...]   
    def setup_options(self):  
        self.add_param('--e-value', '-e', dest='e_value',  
                        type=float, default=10.0,  
                        help="Expectation value")
```

# Command-line option processing

```
class AScript(SessionBasedScript):  
    # [...]   
    def setup_options(self):  
        self.add_param('--e-value', '-e', dest='e_value',  
                        type=float, default=10.0,  
                        help="Expectation value")
```

Aliases and abbreviations for options can be defined.

# Command-line option processing

```
class AScript(SessionBasedScript):  
    # [...]  
    def setup_options(self):  
        self.add_param('--e-value', '-e', dest='e_value',  
                        type=float, default=10.0,  
                        help="Expectation value")
```

The value of this option will be stored in  
`self.params.e_value`.

# Command-line option processing

```
class AScript(SessionBasedScript):  
    # [...]   
    def setup_options(self):  
        self.add_param('--e-value', '-e', dest='e_value',  
                        type=float, default=10.0,  
                        help="Expectation value")
```

Types work exactly as for positional arguments.

# Command-line option processing

```
class AScript(SessionBasedScript):  
    # [...]   
    def setup_options(self):  
        self.add_param('--e-value', '-e', dest='e_value',  
                        type=float, default=10.0,  
                        help="Expectation value")
```

If the option is not present on the command-line, the associated Python variable (here, `self.params.e_value`) takes this value.

## Detour: BLAST

BLAST is a suite of programs to perform search and alignment of nucleotides and proteins.

One common use of BLAST is the following: given a file describing a new organism, compare it one-to-one to a set of known organisms to find similarities.

The command-line invocation for one such comparison would look like this:

```
$ blastp -query new.faa -subject known.faa \  
-evaluate 1e-6 -outfmt 9
```

**Exercise 4.B:** Write a `topblast.py` script to perform 1-1 BLAST comparisons.

The `topblast.py` script shall be invoked like this:

```
$ python topblast.py [-e T] [-m F] \  
    new.faa k1.faa [k2.faa ...]
```

where:

- Option `-e` (alias: `--e-value`) takes a floating point threshold argument *T*;
- Option `-m` (alias: `--output-format`) takes a single-digit integer argument *F*;
- Arguments `new.faa`, `k1.faa`, etc. are files.

The script should generate and run comparisons between `new.faa` and each of the `kN.faa`. Each 1-1 comparison should run as a separate task. All of them share the same settings for the `-evalue` and `-outfmt` options for `blastp`.

### **Exercise 4.C:** *(Homework)*

Modify the `topblast.py` script that you've written in Exercise 4.B to be invoked like this:

```
$ python topblast.py [-e T] [-m F] new.faa dir
```

Input files describing the “known” subjects should be found by recursively scanning the given directory path.

Bonus points if the modified script exists with a correct error message in case `new.faa` is not an existing file, or `dir` is not a valid directory path.