

10 – Command line arguments, packages

Bálint Aradi

Course: Scientific Programming / Wissenschaftliches Programmieren (Python)



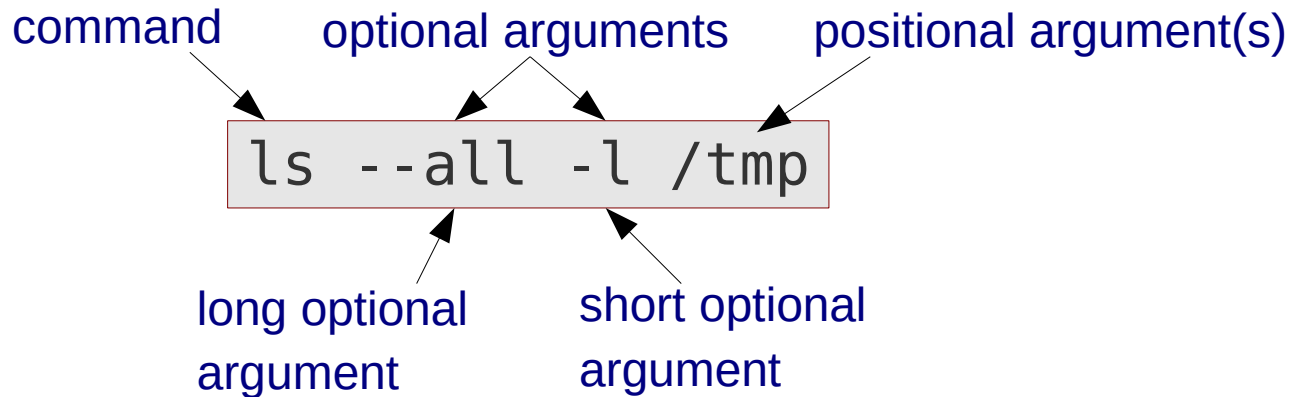
Outline

- Command line argument parsing
- Modularization via packages
- Privat / public variables

Command line argument parsing

Command line arguments

- Unix programs usually accept various arguments which control their behaviour



Positional arguments (“what should the program act on”)

- Last arguments on the command line
- Order may matter
- Number of required positional arguments may be fixed (0, 1, 2) or arbitrary

Optional arguments (“how should the program behave”)

- Start with single dash (short form) or double dash (long form)
- Always optional (program must work without any optional arguments)

Simple argparse example

- The command line arguments can be parsed in Python with the **ArgumentParser** (or in older scripts with **OptionParser**).

```
#!/usr/bin/env python3
```

test.py

```
import argparse
```

```
_DESCRIPTION = 'Test script demonstrating argparse'
```

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

```
msg = 'Directory (default: .)'
```

```
parser.add_argument('-d', '--directory', default='.',  
                    help=msg)
```

```
msg = 'Arbitrary integer number'
```

```
parser.add_argument('number', type=int, metavar='NUM',  
                    help=msg)
```

```
args = parser.parse_args()
```

```
print("Directory: {}".format(args.directory))
```

```
print("Number: {:d}".format(args.number))
```

Testing the example

- Make the Python-script executable from your shell:

```
chmod +x test.py
```

- Passing the **-h** option shows a help page with the description of the script and its arguments and options:

```
./test.py -h
```

```
usage: test.py [-h] [-d DIRECTORY] NUM
```

```
Test script demonstrating argparse
```

```
positional arguments:
```

```
  NUM                Arbitrary integer number
```

```
optional arguments:
```

```
  -h, --help          show this help message and exit
```

```
  -d DIRECTORY, --directory DIRECTORY
```

```
                        Directory (default: .)
```

Testing the example

- Passing one positional parameter only

```
./test.py 2  
Directory: .  
Number: 2
```

- Passing one (short) optional parameter and one positional one

```
./test.py -d /tmp 2  
Directory: /tmp  
Number: 2
```

- Passing one (long) optional parameter and one positional one

```
./test.py --directory /tmp 2  
Directory: /tmp  
Number: 2
```

Testing the example

- Invoking without the required positional argument

```
./test.py
```

```
usage: test.py [-h] [-d DIRECTORY] NUM
```

```
test.py: error: the following arguments are required: NUM
```

- Invoking without a required positional argument of the wrong type

```
./test.py 3.2
```

```
usage: test.py [-h] [-d DIRECTORY] NUM
```

```
test.py: error: argument NUM: invalid int value: '3.2'
```

- Invoking with an invalid optional argument

```
./test.py -a 2
```

```
usage: test.py [-h] [-d DIRECTORY] NUM
```

```
test.py: error: unrecognized arguments: -a
```


Building an argument parser step by step

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

- Creates an argument parser
- An optional description has been provided, which will be used as the general description in the help.

```
msg = 'Directory (default: .)'  
parser.add_argument('-d', '--directory', default='.',  
                    help=msg)
```

- Adds an optional argument with short form (**-d**) and long form (**--directory**) to the parser.
- As argument type is not explicitly defined, it assumes a string argument.
- If the optional argument has not been specified at the command line, the given default value (.) will be used
- The option description has been provided with the **help=** option

Building an argument parser step by step

```
msg = 'Arbitrary integer number'  
parser.add_argument('number', type=int, metavar='NUM',  
                    help=msg)
```

- Adds a mandatory positional argument, named **number**
- Argument type should be an integer
- In the help NUM should be used as argument variable
- Help message for the argument is provided

Parsing the command line

```
args = parser.parse_args()
```

- Using the arguments added so far, parse the command line arguments
- Returns the result as a Namespace object, which can be queried

```
print("Directory: {}".format(args.directory))  
print("Number: {:d}".format(args.number))
```

- The parsed values can be accessed in the returned Namespace object using the dot notation.



Packages

Packages

- Module names of different projects can easily collide (e.g. solver, io)
- Packages enable a fine grained structuring of modules via name-spacing
- Modules within a package can be accessed with the dot-notation:

```
import PACKAGENAME.MODULENAME
```

- Many built-in modules and popular 3rd party modules use packages

```
import os.path  
...  
fname = os.path.join(directory, filename)
```

```
import numpy.linalg as linalg  
  
xx = linalg.solve(aa, bb)
```

Custom packages

- Packages are directories in the file system
- In order to indicate that a directory is a package, it must contain an (empty) file `__init__.py`

`linsolve`

`linsolver/`

`__init__.py`

`io.py`

`solver.py`

Empty file

Modules in
the package

```
#!/usr/bin/env python3 linsolve
'''Solves a linear ...'''
...
import linsolver.solvers
import linsolver.io
...
aa, bb = linsolver.io.get_input()
xx = linsolver.solvers.gauss_elim(aa, bb))
```

Custom package initialization

- The `__init__.py` file may itself contain Python code (package initialization)
- The content of `__init__.py` is imported if the package is imported as a whole
- It is often used to import the modules of the package with one statement

`linsolve`

`linsolver/`

`__init__.py`

`io.py`

`solver.py`

```
import linsolvers.solvers as solvers
import linsolvers.io as io
```

```
#!/usr/bin/env python3
```

`linsolve`

```
'''Solves a linear ...'''
```

```
...
```

```
import linsolver
```

```
...
```

```
aa, bb = linsolver.io.get_input()
```

```
xx = linsolver.solvers.gauss_elim(aa, bb)
```

Finding modules / packages

- When Python encounters a module / package name, it looks for it at various places:
 - Current working directory
 - Directories specified in the **PYTHONPATH** environment variable
 - Directories of the Python standard installation
- The PYTHONPATH environment variable can be set up / adjusted in the shell by the user to let Python find packages at non-standard locations:

```
export PYTHONPATH=/home/aradi/.local/lib/python3.6/site-packages:/home/aradi/some_other_directory
```

- The directories, where Python should look up modules/packages should be separated by colon (:)
- The order of the directories corresponds to the search order
- The current value of the environment variable can be printed in the shell:

```
echo $PYTHONPATH
```

```
/home/aradi/.local/lib/python3.6/site-packages:/home/...
```




Private / public entitites

Hiding module internals

- A module should only export functions / objects which should be invoked from outside
- Implementational details (e.g. helper routines, internal constants, etc.) should remain hidden.

```
_TOLERANCE = 1e-12                Internally used constant

def gaussian_eliminate(aa, bb):      Public routine
    ...
    _eliminate(aa, pp)
    _make_back_substitute(pp, bb)

def _eliminate(aa, pp):             Internal helper routine
    ...

def _substitute_back(pp, bb):      Internal helper routine
    ...
```

Public / private entities

- Whether an entity should be accessed from outside is indicated by its name.
- If its name starts with a letter [a-zA-Z], the entity is considered to be a public entity
- If its name starts with a **single underscore** (`_`), the entity is considered to be a **private** entity.
- Private entities should only be accessed from within the scope (module, object) where they are defined.

Note

- The public / private rule above is only a convention and is not enforced by the language.
- Theoretically, one can also access “private” entities also from outside.

Always stick to the public / private naming convention

Make sure, you mark only the necessary entities in your module public and mark everything else as private (hiding implementation details)

Public entities should be always documented via their doc-strings