The Informal Python Boot Camp
Lesson 6

1. **Integration into the system environment**

You are not alone

In a Unix-like operating system environment we have:

- command line arguments and options
- standard IO
- environment variables
- the exit value of a process
- subprocesses
- inter-process communication (IPC)
  - signals
  - networking
  - ...

Command line arguments: terminology

Example command line:

```
progi -f --verbose --title=Pifpaf in.txt out.png
```

- *program name:*

  ```
  progi
  ```

- *arguments:*

  ```
  -f --verbose --title=Pifpaf in.txt out.png
  ```

- *options:*

  ```
  -f --verbose --title=Pifpaf
  ```

- *option argument:*

  ```
  Pifpaf
  ```

- *positional arguments:*

  ```
  in.txt out.png
  ```

The Unix option conventions

- short options:

```
-h -v              # may be grouped: -hv
```

- long options:

```
--help --verbose
```

- options may take an argument:

```
--input=bla
--input bla
-i bla
-ibla
```

- sole double hyphen `--` ends option processing:

```
transform -v --in=bla.txt -- -10.0 -5.0
```

- `--help` or/and `-h` give usage message.

The optparse module

Parsing options made simple:

```
from optparse import OptionParser

usage = """usage: %prog [options] in.txt out.png\n
Do something and plot it."""

parser = OptionParser(usage=usage)
parser.add_option( "-t", "--title",
                   dest="title",
                   help="set the TITLE",
                   metavar="TITLE")
parser.add_option( "-v", "--verbose",
                   dest="verbose",
                   action="store_true",
                   help="print more information")

# ... continued on next slide ...
```

The optparse module

```
# ... continued ...

(options, args) = parser.parse_args()
print "Options:", options
print "Positional arguments:", args
```

- Options are returned in a dict.
- Positional arguments are returned in a list.

```
> ./progi --verbose --title=Pifpaf in.txt out.png
Options: {'verbose': True, 'title': 'Pifpaf'}
Positional arguments: ['in.txt', 'out.png']
```

The optparse module

Automatic help:

```
> ./progi --help
Usage: progi [options] in.txt out.png

Do something and plot it.

Options:
  -h, --help            show this help message
  -t TITLE, --title=TITLE
                        set the TITLE
  -v, --verbose         print more information
```

The `optparse` module

Automatic checks:

```
> ./progi --title
Usage: progi [options] in.txt out.png

Do something and plot it.

progi: error: --title option requires an argument
```

Environment variables

Environment variables may be accessed through the dictionary `os.environ`:

```
>>> import os
>>>
>>> for path in os.environ['PATH'].split(':'):
...     print path
...
/usr/local/bin
/usr/bin
/bin
...
```

The exit value

On Unix, a program may return a value to the caller

- 0 indicates success. *(stupid, but that's how it is...)*
- 1 indicates failure.
- Everything else also indicates failure.

```
import sys


...
if failure:
    sys.stderr.write("shit happens...\n")
    sys.exit(1)

sys.exit(0) # <- not needed.
            # Python returns 0 by default.
```

The exit value

$? contains last exit status in shell:

```
> /bin/false
> echo $?
1
> /bin/true
> echo $?
0
```

This allows conditional evaluation in the shell (if the programs follow the convention...)

```
> pdflatex lesson6.tex && gv lesson6.pdf
   ... gv only started, if latex succeeded.
> ./run_simulation || beep -l 10000
   ... alert me if the simulation failed.
```

Subprocesses

We already saw one way to start subprocesses: `os.popen()`

Read from another program:

```
import os
gz = os.popen('gunzip -c compressed_file.gz','r')
uncompressed_data = gz.readlines()
gz.close()
```

Write to another program:

```
import os
gz = os.popen('gzip > compressed_file.gz','w')
gz.write(uncompressed_data)
gz.close()
```

Subprocesses

In some cases `os.popen()` is not the best solution:

- if you need both, `stdout` and `stdin` of the child process.
- if you need `stderr` of the child process.
- if you are concerned about security.
- if you want to examine the exit value of the child process.
- if you need the PID of the child process.

This is when the `subprocess` module is your friend.
`http://docs.python.org/lib/ipc.html`

Subprocesses example: Replacing shell backticks

Shell:

```
> output=‘mycmd myarg‘
```

Python using `subprocess` module:

```
from subprocess import *

p = Popen(['mycmd', 'myarg'], stdout=PIPE)
output = p.communicate()[0]

# the exit value is in p.returncode
returncode = p.returncode
```

Subprocesses example: Replacing shell pipeline

Shell:

```
> output=‘dmesg | grep hda‘
```

Python using `subprocess` module:

```
from subprocess import *

p1 = Popen([’dmesg’], stdout=PIPE)
p2 = Popen([’grep’, ’hda’], stdin=p1.stdout,
                            stdout=PIPE)
output = p2.communicate()[0]
```

Inter-process communication

Some methods allowing programs to ”talk” to each other:

- Files
- Pipes
- Signals
- Named pipes
- Semaphores
- Shared memory
- Message passing
- Network sockets
- RPC, XML-RPC, SOAP

Concurrent file access

Program 1 writes data: `writer1.py`

```
while True:
    f = open('data','w+')
    for i in range(100000):
        f.write( '%i\n' % i )
    f.close()
    print "100000 lines written"
```

Program 2 reads data: `reader1.py`

```
while True:
    f = open('data','r')
    vals = []
    for line in f:
        i = int(line.strip())
        vals.append(i)
    f.close()
    print len(vals), 'lines read'
```

Concurrent file access

Damn...

```
> python writer1.py > writer_output
> python reader1.py
0 lines read
0 lines read
1861 lines read
12775 lines read
54417 lines read
73532 lines read
3499 lines read
5138 lines read
16871 lines read
...
```

File locking

Writer program with file locking:

```
import fcntl

l = open('lock','w')
while True:
    # wait for exclusive lock on file 'lock':
    fcntl.flock( l, fcntl.LOCK_EX )

    f = open('data','w')
    for i in range(100000):
        f.write( '%i\n' % i )
    f.close()
    print "100000 lines written"

    # release the lock:
    fcntl.flock(l, fcntl.LOCK_UN )
```

File locking

Reader program with file locking:

```
import fcntl

l = open('lock','r')
while True:
    # wait for exclusive lock on file 'lock':
    fcntl.flock(l, fcntl.LOCK_EX )

    f = open('data','r')
    vals = []
    for line in f:
        i = int(line.strip())
        vals.append(i)
    f.close()
    print len(vals), 'lines read'

    # release the lock:
    fcntl.flock(l, fcntl.LOCK_UN )
```

File locking

Now they behave...

```
> python writer1.py > writer_output
> python reader1.py
100000 lines read
100000 lines read
100000 lines read
100000 lines read
100000 lines read
100000 lines read
100000 lines read
...
```

Signals

Some useful signals (terminate the process by default):

SIGTERM	default signal sent by `kill`.
SIGINT	sent to process when user hits *Control-C*.
SIGHUP	sent to process when user closes the terminal.
SIGKILL	sent by `kill -9` (cannot be caught).
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2

See also:

- `man kill`
- `man 7 signal`

Using signals

In Python, you may

- install a signal handler to catch a signal with

```
signal.signal( signalnumber, handler )
```

.

- send signals to other processes with

```
os.kill( pid, signalnumber )
```

.

How to allow gracefull program termination

```
import signal, time

def quit_handler(signum, frame):
    print 'caught signal', signum
    quit_handler.goaway = True

signal.signal(signal.SIGINT,  quit_handler)
signal.signal(signal.SIGHUP,  quit_handler)
signal.signal(signal.SIGTERM, quit_handler)

i = 0
quit_handler.goaway = False
while not quit_handler.goaway:
    print i
    i += 1
    sleep(1)

print "landing softly..."
```

Things to remember from today's lesson

- Lock files with `fcntl.flock()`.
- Catch signals with `signal.signal()`.
- Parse command line args with `optparse`.

Keep these pages open, when programming in Python:

- Tutorial:
  `http://docs.python.org/tut/`
- Library Reference:
  `http://docs.python.org/lib/lib.html`