Frequently encountered operating system tasks in Python

Joakim Sundnes[1,2] Hans Petter Langtangen[1,2] Ola Skavhaug[3]

Center for Biomedical Computing, Simula Research Laboratory[1]

Dept. of Informatics, University of Oslo[2]

mCASH (formerly Dept. of Informatics, University of Oslo)[3]

Aug 25, 2014

Program Python instead of Bash?

Essence:

We can do Bash operations in Python too, in a cross-platform fashion (Mac, Windows, Linux) and with much more flexibility for extensions.

Topics to be covered:

- environment variables
- file globbing, testing file types
- copying and renaming files, creating and moving to directories, creating directory paths, removing files and directories
- directory tree traversal
- parsing command-line arguments
- running an application from a script
- Text processing (some repetition)
- use of `split, join`
- searching and substituting

Environment variables

The dictionary-like `os.environ` object holds the environment variables:

```
>>> os.environ['PATH']
'/users/me/bin:/bin:/usr/sbin/:/some/strange/path'
>>> os.environ['HOME']
'/users/me'
>>> os.environ['PYTHONPATH']
'/users/me/mypy/lib:/users/you/yourpy/new/lib'

>>> os.environ['my_self_defined_environment_variable'] = 'ON'
>>> os.environ['my_self_defined_environment_variable']
'ON'
```

Write all the environment variables in alphabethic order:

```
for var in sorted(os.environ):
    print '%s=%s' % (var, os.environ[var])
```

File globbing: list files that match a wildcard expression

List all `.pdf` and `.gif files` in the current directory (Unix):

```
> ls *.pdf *.gif
```

Cross-platform way to do it in Python:

```
import glob
filelist = glob.glob('*.pdf') + glob.glob('*.gif')
```

This is referred to as *file globbing*.

More sophisticated example: find all files not starting with `P` or `p`, followed by any string, then two numbers, and then and ending in `.txt`:

```
filelist = glob.glob('[!Pp]*[0-9][0-9].txt')
```

Finding file type, size, age

```
import os.path
print myfile,

if os.path.isfile(myfile):
     print 'is a plain file'

if os.path.isdir(myfile):
     print 'is a directory'

if os.path.islink(myfile):
     print 'is a link'

# the size and age:
size = os.path.getsize(myfile)
time_of_last_access       = os.path.getatime(myfile)
time_of_last_modification = os.path.getmtime(myfile)

# times are measured in seconds since 1970.01.01
days_since_last_access = \
(time.time() - os.path.getatime(myfile))/(3600*24)
```

More detailed file info via `os.stat`

```
import stat

myfile_stat = os.stat(myfile)

filesize = myfile_stat[stat.ST_SIZE]

mode = myfile_stat[stat.ST_MODE]

if stat.S_ISREG(mode):
    print '%(myfile)s is a regular file '\
          'with %(filesize)d bytes' % vars()
```

Check out the `stat` module in the Python Library Reference

Copy, rename and remove files

Copy a file:

```
import shutil
shutil.copy(myfile, tmpfile)
```

Rename a file:

```
os.rename(myfile, 'tmp.1')
```

Remove a file:

```
os.remove('mydata.dat')
# or os.unlink('mydata.dat')
```

Use `os.path.join` for file path construction

Cross-platform construction of file paths:

```
filename = os.path.join(os.pardir, 'src', 'lib')

# Unix:    ../src/lib
# Windows: ..\src\lib
```

Current directory is `os.curdir`, parent directory is `os.pardir`:

```
# Unix:  cp ../../myfile .
shutil.copy(os.path.join(os.pardir, os.pardir, filename), os.curdir)
```

Directory management

Creating and moving to directories:

```
dirname = 'mynewdir'
if not os.path.isdir(dirname):
    os.mkdir(dirname) # or os.mkdir(dirname, mode='0755')
os.chdir(dirname)
```

Equivalent Bash commands:

```
> dirname=mynewdir
> mkdir $dirname
> cd $dirname
```

Make complete directory path with intermediate directories:

```
path = os.path.join(os.environ['HOME'], 'py', 'src')
os.makedirs(path)

# Unix: mkdirhier HOME/py/src or mkdir  − p HOME/py/src
```

Remove a non-empty directory tree:

```
shutil.rmtree('somedir')   # dangerous - know what you do!
```

Basename and directory of a path

Given a path, e.g.,

```
path = '/home/me/scripting/python/intro/hw.py'
```

Extract directory and basename:

```
# basename: hw.py
basename = os.path.basename(path)

# dirname: /home/me/scripting/python/intro
dirname  = os.path.dirname(path)

# or
dirname, basename = os.path.split(path)
```

Extract file suffix/extension:

```
stem, suffix = os.path.splitext(path)
# stem: /home/me/scripting/python/intro/hw
# suffix: .py
```

Platform-dependent operations

The operating system interface in Python is the same on Unix, Windows and Mac However, sometimes you need to perform platform-specific operations, but how can you make a portable script?

Key variables:

- `os.name`: operating system name
- `sys.platform`: platform identifier

Example: `cmd` holds a command to be run in the background

```
if os.name == 'posix':            # Unix?
  failure = os.system(cmd + '&')

elif sys.platform[:3] == 'win':   # Windows?
  failure = os.system('start ' + cmd)
```

Traversing directory trees with `find` in Bash

Run through all files in your home directory and list files that are larger than 1 Mb:

```
> find $HOME -type f -size +1000k -exec ls -s -h {} \;
```

This (and all features of Unix find) can be given a cross-platform implementation in Python so it works on Windows too

Traversing directory trees with os.path.walk in Python

Run through all files in your home directory and list files that are larger than 1 Mb:

```
root = os.environ['HOME']            # home directory
os.path.walk(root, myfunc, arg)
```

`os.path.walk` walks through a directory tree (`root`) and calls, for each directory `dirname`,

```
myfunc(arg, dirname, files)
```

where

- `files` is the list of local filenames in directory `dirname`
- `arg` is any user-defined argument, e.g., a nested list of variables

Example on finding large files with os.path.walk

```
import os

def checksize(arg, dirname, files):
    for file in files:
        # construct the file's complete path:
        filename = os.path.join(dirname, file)
        if os.path.isfile(filename):
            size = os.path.getsize(filename)
            if size > 1000000:
                if arg is None:
                    print '%.2fMb %s' % (size/1000000.0,filename)
                elif isinstance(arg, list):
                    arg.append((size/1000000.0,filename))

root = os.environ['HOME']
os.path.walk(root, checksize, None)  # print list of large files

arg = []
os.path.walk(root, checksize, arg)
# arg is now a list of large files
for size, filename in arg:
    print filename, 'has size', size, 'Mb'
```

Time for a bug!

In the previous example, let's use a tuple instead of a list:

```
def checksize2(arg, dirname, files):
    for file in files:
        # construct the file's complete path:
        filename = os.path.join(dirname, file)
        if os.path.isfile(filename):
            size = os.path.getsize(filename)
            if size > 1000000:
                if arg is None:
                    print '%.2fMb %s' % (size/1000000.0,filename)
                elif isinstance(arg, list):
                    arg = arg + ((size/1000000.0,filename))

root = os.environ['HOME']
arg = ()
os.path.walk(root, checksize2, arg)
```

Question.

Why is arg an empty tuple after the call?

Example on finding large files with os.walk

`os.walk(root, followlinks=False)` is an iterator:

```
for dirpath, dirnames, filenames in os.walk(root):
```

Here,

- `dirpath` is the complete path, relative to `root`, to a subdirectory
- `dirnames` are the names of the subdirectories in `dirpath`
- `filenames` are the names of ordinary files in `dirpath`

Previous example programmed with `os.walk`:

```
arg = []
for dirpath, dirnames, filenames in os.walk(root):
    checksize(arg, dirpath, dirnames + filenames)
```

Creating tar archives for file collections

Bash:

```
> tar czf tmp.tar.gz myfile.py yourfile.f mydirectory
```

Cross-platform Python:

```
>>> import tarfile
>>> files = 'myfile.py', 'yourfile.f', 'mydirectory'
>>> tar = tarfile.open('tmp.tar.gz', 'w:gz')  # gzip compression
>>> for file in files:
...     tar.add(file)
...
>>> # check what's in this archive:
>>> members = tar.getmembers()  # list of TarInfo objects
>>> for info in members:
...     print '%s: size=%d, mode=%s, mtime=%s' % \
...           (info.name, info.size, info.mode,
...            time.strftime('%Y.%m.%d', time.gmtime(info.mtime)))
...
myfile.py: size=11898, mode=33261, mtime=2004.11.23
yourfile.f: size=206, mode=33261, mtime=2005.08.12
mydirectory/file1.py: size=1560, mode=33261, mtime=2014.09.14
mydirectory/file2.py: size27560, mode=33261, mtime=2014.08.08
>>> tar.close()
```

Reading tar archives

```
>>> tar = tarfile.open('tmp.tar.gz', 'r')
>>>
>>> for file in tar.getmembers():
...     tar.extract(file)   # extract file to current working dir.
...
>>> f = tar.extractfile('myfile.py')  # extract as file object
>>> f.readlines()
```

Parsing command-line arguments

- Three types of command-line options:
  `myprog.py arg1 arg2 -m 1.5 --mass 2.5`
  1. positional arguments: `arg1`, `arg2`
  2. option-value pair with short option: `-m`
  3. option-value pair with long option: `--mass`
- Running through `sys.argv[1:]` and extracting command-line info "manually" is easy
- Using standardized modules and interface specifications is better!
- The standard module is `argparse`

A case involving option-value pairs and the `argparse` module

We have some program with four parameters v_0, s_0, a, and t, corresponding to logical names *initial velocity*, *initial position*, *acceleration*, and *time*.

All these have sensible default values. We use command-line option-value pairs to override default values (cf. keyword arguments).

The user can choose between long and short options, but we let short options also have double hyphen:

```
> python position.py --v0 1.0 --acceleration 0.9
> python position.py --initial_velocity 1.0 --a 0.9 --t 3
```

How to use `argparse`

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

# Define command-line arguments
parser.add_argument('--v0', '--initial_velocity', type=float,
                    default=0.0, help='initial velocity')

parser.add_argument('--s0', '--initial_position', type=float,
                    default=0.0, help='initial position')

parser.add_argument('--a', '--acceleration', type=float,
                    default=1.0, help='acceleration')

parser.add_argument('--t', '--time', type=float,
                    default=1.0, help='time')

# Read the command line and interpret the arguments
args = parser.parse_args()

# Extract values
s = args.s0 + args.v0*t + 0.5*args.a*args.t**2
# or
s0 = args.s0; v0 = args.v0; a = args.a; t = args.t
s = s0 + v0*t + 0.5*a*t**2
```

See the documentation of argparse for more examples

Writing Python data structures to file

Write nested lists to file using `str` for converting an object to a string:

```
somelist = ['text1', 'text2']
a = [[1.3,somelist], 'some text']
f = open('tmp.dat', 'w')

# convert data structure to its string representation:
f.write(str(a))
f.close()
```

`tmp.dat`:

```
[[1.3, ['text1', 'text2']], 'some text']
```

Writing to standard output: `print a` implies `print str(a)`

```
print a      # v2.x
print(a)     # v3.x
sys.stdout.write(str(a) + '\n')   # standard output file object
```

Reading Python data structures from file

- `eval(s)`: turn string `s` into live Python code
- `a = eval(str(a))` is a valid "equation" for basic Python data structures (since `str(a)` turns `a` into a string with valid Python syntax for defining `a`, and `eval` turns the string back into a living Python object)

Last slide created `tmp.dat`:

```
[[1.3, ['text1', 'text2']], 'some text']
```

Read this nested list back into a Python object `a`:

```
f = open('tmp.dat', 'r')
# evaluate first line in file as Python code:
a = eval(f.readline())
```

Now, `a` is

```
[[1.3, ['text1', 'text2']], 'some text']

# i.e.
```

Remark about `str`, `repr`, and `eval`

- `str(a)` is used for "pretty print" and defined by the object's special method `__str__`
- `repr(a)` is used for "string representation" of the object and defined by the object's special method `__repr__`
- Strictly speaking, `eval(repr(a))` recreates `a`, while `eval(str(a))` is not always meaningful if `str(a)` is a pretty print of the object and not exactly the syntax needed to create the object
- For the Python standard objects (numbers, lists, tuples, dicts), `str(a)` and `repr(a)` are the same

A class with tailored `str` and `repr`

```
class LinearFunction:
    """Represent a linear function a*x + b."""

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a*x + self.b

    def __str__(self):
        """Return pretty print of function: -1.3*x - 4."""
        return '%s*x %s %s' % (self.a,
                               '+ if self.b > 0 else '-',  # right sig
                               abs(self.b))

    def __repr__(self):
        return 'LinearFunction(%s, %s)' % (self.a, self.b)
```

Illustration of `str`, `repr`, and `eval`

```
>>> f = LinearFunction(-1, 2)
>>> str(a)   # calls f.__str__()
'-1*x + 2'
>>> eval(str(a))
Traceback (most recent call last):
...
NameError: name 'x' is not defined
>>> repr(a)
'LinearFunction(-1, 2)'
>>> new_a = eval(repr(a))
>>> str(new_a)
'-1*x + 2'
```

Persistence: permanent storage of data structures

- Many programs need to have persistent data structures, i.e., the data live after the program is terminated and can be retrieved the next time the program is executed
- `str`, `repr` and `eval` are convenient for making data structures persistent
- `pickle`, `cPickle` and `shelve` are other (more sophisticated) Python modules for implementing persistence

Pickling is used for storing a sequence of objects

Write *any* set of data structures to file using the `pickle` module:

```
f = open(filename, 'w')
import pickle
pickle.dump(a1, f)
pickle.dump(a2, f)
pickle.dump(a3, f)
f.close()
```

Remark:

In v2.x, cPickle is a faster version than pickle, but works in the same way. In 3.x, `pickle` is `cPickle`.

Read data structures in again later (the sequence is important):

```
f = open(filename, 'r')
a1 = pickle.load(f)
a2 = pickle.load(f)
a3 = pickle.load(f)
```

Shelving

Think of shelves as dictionaries with file storage:

```
import shelve
database = shelve.open(filename)
database['a1'] = a1  # store a1 under the key 'a1'
database['a2'] = a2
database['a3'] = a3
# or
database['a123'] = (a1, a2, a3)

# retrieve data:
if 'a1' in database:
    a1 = database['a1']
# and so on

# delete an entry:
del database['a2']

database.close()
```

Running an application (old-style)

Run a stand-alone program:

```
cmd = 'myprog -c file.1 -p -f -q > res'
failure = os.system(cmd)
if failure:
    print cmd, 'failed'
    sys.exit(1)
```

Redirect output from the application to a list of lines:

```
pipe = os.popen(cmd)
output = pipe.readlines()
pipe.close()

for line in output:
  # process line
```

The new standard: subprocess

```
from subprocess import call
cmd = 'myprog -c file.1 -p -f -q > res'
try:
    returncode = call(cmd, shell=True)
    if returncode:
        print 'failure'  # returncode has no meaning when != 0
        sys.exit(1)
except OSError, message:
    print 'execution failed!\n', message
    sys.exit(1)
```

Grab the output using `subprocess.Popen`:

```
from subprocess import Popen, PIPE
p = Popen(cmd, shell=True, stdout=PIPE)
output, errors = p.communicate()

# output: text written to standard output
# errors: text written to standard error

for line in output.splitlines():
    print line
```

Output pipe

Open (in a script) a dialog with an interactive program:

```
from subprocess import Popen, PIPE
pipe = Popen('gnuplot -persist', shell=True, stdin=PIPE).stdin
pipe.write('set xrange [0:10]; set yrange [-2:2]\n')
pipe.write('plot sin(x)\n')
pipe.write('quit')  # quit Gnuplot
```

Same as “here documents” in Unix shells:

```
gnuplot <<EOF
set xrange [0:10]; set yrange [-2:2]
plot sin(x)
quit
EOF
```

Find a program

Check if a given program is on the system:

```
program = 'someprog'
path = os.environ['PATH']
# PATH can be /usr/bin:/usr/local/bin:/usr/X11/bin
# os.pathsep is the separator in PATH
# (: on Unix, ; on Windows)
paths = path.split(os.pathsep)
for d in paths:
    if os.path.isdir(d):
        if os.path.isfile(os.path.join(d, program)):
            program_path = d; break

try:  # program was found if program_path is defined
    print program, 'resides in', program_path
except:
    print program, 'was not found'
```

Corresponds to `which someprog` in Unix

Cross-platform fix of previous script

On Windows, programs usually end with `.exe` (binaries) or `.bat` (DOS scripts), while on Unix most programs have no extension

We test if we are on Windows:

```
if sys.platform[:3] == 'win':
    # Windows-specific actions
```

Cross-platform snippet for finding a program:

```
for d in paths:
    if os.path.isdir(d):
        fullpath = os.path.join(dir, program)
        if sys.platform[:3] == 'win':   # windows machine?
            for ext in '.exe', '.bat':  # add extensions
                if os.path.isfile(fullpath + ext):
                    program_path = d; break
        else:
            if os.path.isfile(fullpath):
                program_path = d; break
```

Simple text processing

When doing operating system tasks in Python, one often needs to manipulate text and commands. The next slides list the basic text processing tools in Python. More powerful tools are accessible via regular expressions.

Splitting text

Split string into words:

```
>>> files = 'case1.png case2.png case3.png'
>>> files.split()
['case1.png', 'case2.png', 'case3.png']
```

Can split wrt other characters:

```
>>> files = 'case1.png, case2.png, case3.png'
>>> files.split(', ')
['case1.png', 'case2.png', 'case3.png']
>>> files.split(',  ')  # extra erroneous space after comma...
['case1.png, case2.png, case3.png']  # unintended split??
```

Very useful when interpreting text in files!

Example on using split to read data from file

Suppose you have file containing numbers only. The numbers appear without any structure:

```
1.432 5E-09
1.0

3.2 5 69 -111
4 7 8
```

It is easy to get a list of all these numbers:

```
f = open(filename, 'r')
numbers = [float(n) for n in f.read().split()]

# or equivalently with map
numbers = map(float, f.read().split())
```

Joining a list of strings

Join is the opposite of split:

```
>>> filelist = ['case1.png', 'case2.png', 'case3.png']
>>> text = ', '.join(filelist)
>>> text
'case1.png, case2.png, case3.png'
```

Any delimiter text can be used:

```
>>> '@@@'.join(filelist)
'case1.png@@@case2.png@@@case3.png'
```

Common use of join/split

```
# Read file into a string
f = open('myfile', 'r')
filestr = file.read()
f.close()

filestr = filestr.replace('directory', 'folder')

# convert back to list of lines and process lines
lines = filestr.splitlines()
for i, line in enumerate(lines):
    # if line starts with >>> and previous line is not empty,
    # remove the leading >>>
    if line.startswith('>>>') and i>0 \
       and not lines[i-1].strip() == '':
        lines[i] = lines[i][3:]  # strip off >>>

filestr = '\n'.join(lines)  # make a string again

f = open('myfile.new', 'w')
f.write(filestr)
f.close()
```

Searching in strings

Exact word match:

```
if 'somestring' in line

if line == 'somestring':
   # line equals 'somestring'
```

Matching with Unix shell-style wildcard notation:

```
>>> import fnmatch
>>> filename = 'package-python-sympy.txt'
>>> wildcard_pattern = '*-python-*.t*xt'
>>> fnmatch.fnmatch(filename, wildcard_pattern)
True
>>> filename = 'package-python-sympy.text'
>>> fnmatch.fnmatch(filename, wildcard_pattern)
True
>>> filename = 'package-python-sympy'
>>> fnmatch.fnmatch(filename, wildcard_pattern)
False
```

Searching in strings with regular expressions

Matching with full regular expressions:

```
import re

if re.search(r'somestring', line):
    # line contains 'somestring'

if re.search(r'^somestring$', line):
    # line equals 'somestring'

pattern = '.*-python-.*\.t.*xt'
if re.search(pattern, line):
    # line contains some text of the form specified by pattern
```

Substitution

Simple substitution:

```
newstring = oldstring.replace(substring, newsubstring)
```

Substitute regular expression `pattern` by `replacement` in `str`:

```
import re
str = re.sub(pattern, replacement, str)
```

Brief summary

- Typical Unix shell tasks can also be done using Python in a cross-platform fashion so it works on Mac, Windows, and Linux
- Key Python modules for shell operations; `os`, `shutil`, `glob`, `tarfile`, `subprocess`
- Python has numerous tools for text processing:
  - `join` and `split`
  - `fnmatch` for Unix wildcard matching
  - regular expressions