Slides from INF3331 lectures - Python tasks

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011

Frequently encountered tasks in Python

Overview

- 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
- (more advanced) list operations
- splitting and joining text

Python programming information

Man-page oriented information:

- `pydoc somemodule.somefunc`, `pydoc somemodule`
- `doc.html`! Links to lots of electronic information
- The Python Library Reference (go to the index)
- Python in a Nutshell
- Beazley's Python reference book
- Your favorite Python language book
- Google

These slides (and exercises) are closely linked to the "Python scripting for computational science" book, ch. 3 and 8

File globbing

- List all .ps and .gif files (Unix):

```
ls *.ps *.gif
```

- Cross-platform way to do it in Python:

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

This is referred to as file globbing

Testing file types

```
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

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

Path construction

- Cross-platform construction of file paths:

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

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

shutil.copy(filename, os.curdir)

# Unix:  cp ../src/lib .

# os.pardir : ..
# os.curdir : .
```

Directory management

- Creating and moving to directories:

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

- Remove a non-empty directory tree:

```
shutil.rmtree('myroot')
```

Basename/directory of a path

- Given a path, e.g.,

```
fname = '/home/hpl/scripting/python/intro/hw.py'
```

- Extract directory and basename:

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

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

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

- Extract suffix:

```
root, suffix = os.path.splitext(fname)
# suffix: .py
```

Platform-dependent operations

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

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

# cmd:  string holding command to be run
if os.name == 'posix':                # Unix?
  failure = os.system(cmd + '&')
elif sys.platform[:3] == 'win':       # Windows?
  failure = os.system('start ' + cmd)
else:
  # foreground execution:
  failure, output = commands.getstatusoutput(cmd)
```

Traversing directory trees (1)

- Run through all files in your home directory and list files that are larger than 1 Mb
- A Unix find command solves the problem:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \;
```

- This (and all features of Unix find) can be given a cross-platform implementation in Python

Traversing directory trees (2)

- Similar cross-platform Python tool:

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

walks through a directory tree (`root`) and calls, for each directory `dirname`,

```
myfunc(arg, dirname, files)  # files is list of (local) filenames
```

- `arg` is any user-defined argument, e.g. a nested list of variables

Example on finding large files

```
def checksize1(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:
                print '%.2fMb %s' % (size/1000000.0,filename)

root = os.environ['HOME']
os.path.walk(root, checksize1, None)

# arg is a user-specified (optional) argument,
# here we specify None since arg has no use
# in the present example
```

Make a list of all large files

- Slight extension of the previous example
- Now we use the `arg` variable to build a list during the walk

```
def checksize1(arg, dirname, files):
    for file in files:
        filepath = os.path.join(dirname, file)
        if os.path.isfile(filepath):
            size = os.path.getsize(filepath)
            if size > 1000000:
                size_in_Mb = size/1000000.0
                arg.append((size_in_Mb, filename))

bigfiles = []
root = os.environ['HOME']
os.path.walk(root, checksize1, bigfiles)
for size, name in bigfiles:
    print name, 'is', size, 'Mb'
```

Creating Tar archives

- Tar is a widespread tool for packing file collections efficiently
- Very useful for software distribution or sending (large) collections of files in email
- Demo:

```
>>> import tarfile
>>> files = 'NumPy_basics.py', 'hw.py', 'leastsquares.py'
>>> 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)))
...
NumPy_basics.py: size=11898, mode=33261, mtime=2004.11.23
hw.py: size=206, mode=33261, mtime=2005.08.12
leastsquares.py: size=1560, mode=33261, mtime=2004.09.14
>>> tar.close()
```

- Compressions: uncompressed (`w:`), gzip (`w:gz`), bzip2 (`w:bz2`)

Reading Tar archives

```
>>> tar = tarfile.open('tmp.tar.gz', 'r')
>>>
>>> for file in tar.getmembers():
...     tar.extract(file)       # extract file to current work.dir.
...
>>> # do we have all the files?
>>> allfiles = os.listdir(os.curdir)
>>> for file in files:
...     if not file in allfiles:  print 'missing', file
...
>>> hw = tar.extractfile('hw.py')  # extract as file object
>>> hw.readlines()
```

Parsing command-line arguments

- Running through `sys.argv[1:]` and extracting command-line info 'manually' is easy
- Using standardized modules and interface specifications is better!
- Python's `getopt` and `optparse` modules parse the command line
- `getopt` is the simplest to use
- `optparse` is the most sophisticated

Short and long options

- It is a 'standard' to use either short or long options

```
-d dirname              # short options -d and -h
--directory dirname     # long options --directory and --help
```

- Short options have single hyphen,
  long options have double hyphen
- Options can take a value or not:

```
--directory dirname --help --confirm
-d dirname -h -i
```

- Short options can be combined

```
-iddirname   is the same as  -i -d dirname
```

Using the getopt module (1)

- Specify short options by the option letters, followed by colon if the option requires a value
- Example: `'id:h'`
- Specify long options by a list of option names, where names must end with = if they require a value
- Example: `['help','directory=','confirm']`

Using the getopt module (2)

- `getopt` returns a list of (option,value) pairs and a list of the remaining arguments
- Example:

```
--directory mydir -i file1 file2
```

makes `getopt` return

```
[('--directory','mydir'), ('-i','')]
['file1','file2]'
```

Using the getopt module (3)

- Processing:

```
import getopt
try:
    options, args = getopt.getopt(sys.argv[1:], 'd:hi',
                                  ['directory=', 'help', 'confirm'])
except:
    # wrong syntax on the command line, illegal options,
    # missing values etc.

directory = None; confirm = 0  # default values
for option, value in options:
    if option in ('-h', '--help'):
        # print usage message
    elif option in ('-d', '--directory'):
        directory = value
    elif option in ('-i', '--confirm'):
        confirm = 1
```

Using the interface

- Equivalent command-line arguments:

```
-d mydir --confirm src1.c src2.c
--directory mydir -i src1.c src2.c
--directory=mydir --confirm src1.c src2.c
```

- Abbreviations of long options are possible, e.g.,

```
--d mydir --co
```

- This one also works: `-idmydir`

Writing Python data structures

- Write nested lists:

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

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

- Equivalent statements writing to standard output:

```
print a
sys.stdout.write(str(a) + '\n')

# sys.stdin         standard input as file object
# sys.stdout        standard input as file object
```

Reading Python data structures

- `eval(s)`: treat string `s` as Python code
- `a = eval(str(a))` is a valid 'equation' for basic Python data structures
- Example: read nested lists

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

results in

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

# i.e.
newa = eval(f.readline())
# is the same as
newa = [[1.3, ['text1', 'text2']], 'some text']
```

Remark about str and eval

- `str(a)` is implemented as an object function `__str__`
- `repr(a)` is implemented as an object function `__repr__`
- `str(a)`: pretty print of an object
- `repr(a)`: print of all info for use with `eval`
- `a = eval(repr(a))`
- `str` and `repr` are identical for standard Python objects (lists, dictionaries, numbers)

Persistence

- Many programs need to have persistent data structures, i.e., 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 storing/loading objects

Pickling

- Write *any* set of data structures to file using the cPickle module:

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

- Read data structures in again later:

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

- Run a stand-alone program:

```
cmd = 'myprog -c file.1 -p -f -q > res'
failure = os.system(cmd)
if failure:
  print '%s: running myprog failed' % sys.argv[0]
  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
```

- Better tool: the `commands` module (next slide)

Running applications and grabbing the output

- A nice way to execute another program:

```
import commands
failure, output = commands.getstatusoutput(cmd)

if failure:
    print 'Could not run', cmd; sys.exit(1)

for line in output.splitlines()  # or output.split('\n'):
    # process line
```

(`output` holds the output as a string)

- `output` holds both standard error and standard output (`os.popen` grabs only standard output so you do not see error messages)

Running applications in the background

- `os.system`, pipes, or `commands.getstatusoutput` terminates after the command has terminated
- There are two methods for running the script in parallel with the command:
  - run the command in the background

    ```
    Unix:     add an ampersand (&) at the end of the command
    Windows:  run the command with the 'start' program
    ```

  - run the operating system command in a separate thread
- More info: see “Platform-dependent operations” slide and the `threading` module

The new standard: subprocess

- A module subprocess is the new standard for running stand-alone applications:

```
from subprocess import call
try:
    returncode = call(cmd, shell=True)
    if returncode:
        print 'Failure with returncode', returncode;
        sys.exit(1)
except OSError, message:
    print 'Execution failed!\n', message; sys.exit(1)
```

- More advanced use of subprocess applies its `Popen` object

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

Output pipe

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

```
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
```

Writing to and reading from applications

- In theory, `Popen` allows us to have two-way comminucation with an application (read/write), but this technique is not suitable for reliable two-way dialog (easy to get hang-ups)
- The `pexpect` module is the right tool for a two-way dialog with a stand-alone application

```
# copy files to remote host via scp and password dialog
cmd = 'scp %s %s@%s:%s' % (filename, user, host, directory)
import pexpect
child = pexpect.spawn(cmd)
child.expect('password:')
child.sendline('&%$hQxz?+MbH')
child.expect(pexpect.EOF)  # wait for end of scp session
child.close()
```

File reading

- Load a file into list of lines:

```
infilename = '.myprog.cpp'
infile = open(infilename, 'r')  # open file for reading

# load file into a list of lines:
lines = infile.readlines()

# load file into a string:
filestr = infile.read()
```

- Line-by-line reading (for large files):

```
while 1:
    line = infile.readline()
    if not line: break
    # process line
```

Some notes on lists

- Initializing a list:

```
arglist = [myarg1, 'displacement', "tmp.ps"]
```

- Or with indices (if there are already two list elements):

```
arglist[0] = myarg1
arglist[1] = 'displacement'
```

- Create list of specified length:

```
n = 100
mylist = [0.0]*n
```

- Adding list elements:

```
arglist = []  # start with empty list
arglist.append(myarg1)
arglist.append('displacement')
```

Getting list elements

- Extract elements form a list:

```
 filename, plottitle, psfile  = arglist
(filename, plottitle, psfile) = arglist
[filename, plottitle, psfile] = arglist
```

- Or with indices:

```
filename = arglist[0]
plottitle = arglist[1]
```

Traversing lists

- For each item in a list:

```
for entry in arglist:
    print 'entry is', entry
```

- For-loop-like traversal:

```
start = 0;  stop = len(arglist);  step = 1
for index in range(start, stop, step):
    print 'arglist[%d]=%s' % (index,arglist[index])
```

- Visiting items in reverse order:

```
mylist.reverse()  # reverse order
for item in mylist:
    # do something...
```

List comprehensions

- Compact syntax for manipulating all elements of a list:

```
y = [ float(yi) for yi in line.split() ]  # call function float
x = [ a+i*h for i in range(n+1) ]         # execute expression
```

(called list comprehension)

- Written out:

```
y = []
for yi in line.split():
    y.append(float(yi))
```

etc.

Map function

- `map` is an alternative to list comprehension:

```
y = map(float, line.split())
y = map(lambda i: a+i*h, range(n+1))
```

- `map` is (probably) faster than list comprehension but not as easy to read

Sorting a list

- In-place sort:

  `mylist.sort()`

  modifies `mylist`!

```
>>> print mylist
[1.4, 8.2, 77, 10]
>>> mylist.sort()
>>> print mylist
[1.4, 8.2, 10, 77]
```

- Strings and numbers are sorted as expected

Defining the comparison criterion

```
# ignore case when sorting:

def ignorecase_sort(s1, s2):
    s1 = s1.lower()
    s2 = s2.lower()
    if   s1 <  s2: return -1
    elif s1 == s2: return  0
    else:          return  1

# quicker variant, using Python's built-in
# cmp function:
def ignorecase_sort(s1, s2):
    s1 = s1.lower();  s2 = s2.lower()
    return cmp(s1,s2)

# usage:
mywords.sort(ignorecase_sort)

#Best variant:
mywords.sort(key=lambda s: s.lower())
```

Environment variables

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

```
os.environ['PATH']
os.environ['HOME']
os.environ['scripting']
```

- Write all the environment variables in alphabethic order:

```
sorted_env = os.environ.keys()
sorted_env.sort()

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

Find a program

- Check if a given program is on the system:

```
program = 'vtk'
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 '%s found in %s' % (program, program_path)
except:
    print '%s not found' % program
```

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
```

Splitting text

- Split string into words:

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

- Can split wrt other characters:

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

- Very useful when interpreting files

Example on using split (1)

- Suppose you have file containing numbers only
- The file can be formatted 'arbitrarily', e.g,

```
1.432 5E-09
1.0

3.2 5 69 -111
4 7 8
```

- Get a list of all these numbers:

```
f = open(filename, 'r')
numbers = f.read().split()
```

- String objects's `split` function splits wrt sequences of whitespace (whitespace = blank char, tab or newline)

Example on using split (2)

- Convert the list of strings to a list of floating-point numbers, using `map` or list comprehension:

```
numbers = map(float, f.read().split())
numbers = [ float(x) for x in f.read().split() ]
```

- Think about reading this file in Fortran or C!
(quite some low-level code...)
- This is a good example of how scripting languages, like Python, yields flexible and compact code

Joining a list of strings

- Join is the opposite of split:

```
>>> line1 = 'iteration 12:    eps= 1.245E-05'
>>> line1.split()
['iteration', '12:', 'eps=', '1.245E-05']
>>> w = line1.split()
>>> ' '.join(w)  # join w elements with delimiter ' '
'iteration 12: eps= 1.245E-05'
```

- Any delimiter text can be used:

```
>>> '@@@'.join(w)
'iteration@@@12:@@@eps=@@@1.245E-05'
```

Common use of join/split

```
f = open('myfile', 'r')
lines = f.readlines()              # list of lines
filestr = ''.join(lines)           # a single string
# can instead just do
# filestr = file.read()

# do something with filestr, e.g., substitutions...

# convert back to list of lines:
lines = filestr.splitlines()
for line in lines:
    # process line
```

Text processing (1)

- Exact word match:

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

if line.find('double') != -1:
   # line contains 'double'
```

- Matching with Unix shell-style wildcard notation:

```
import fnmatch
if fnmatch.fnmatch(line, 'double'):
   # line contains 'double'
```

Here, double can be any valid wildcard expression, e.g.,

```
double*   [Dd]ouble
```

Text processing (2)

- Matching with full regular expressions:

```
import re
if re.search(r'double', line):
    # line contains 'double'
```

Here, double can be any valid regular expression, e.g.,

```
double[A-Za-z0-9_]*  [Dd]ouble  (DOUBLE|double)
```

Substitution

- Simple substitution:

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

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

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

Various string types

- There are many ways of constructing strings in Python:

```
s1 = 'with forward quotes'
s2 = "with double quotes"
s3 = 'with single quotes and a variable: %(r1)g' \
     % vars()
s4 = """as a triple double (or single) quoted string"""
s5 = """triple double (or single) quoted strings
allow multi-line text (i.e., newline is preserved)
with other quotes like ' and  "
"""
```

- Raw strings are widely used for regular expressions

```
s6 = r'raw strings start with r and \ remains backslash'
s7 = r"""another raw string with a double backslash: \\ """
```

String operations

- String concatenation:

```
myfile = filename + '_tmp' + '.dat'
```

- Substring extraction:

```
>>> teststr = '0123456789'
>>> teststr[0:5]; teststr[:5]
'01234'
'01234'
>>> teststr[3:8]
'34567'
>>> teststr[3:]
'3456789'
```

Brief summary

- Typical Unix shell tasks can also be done using Python:
  - + Combine with more advanced tools and functions
  - + Cross-platform possibility
  - - Usually more code lines
- Python has numerous tools for text processing:
  - Join/split widely used for processing files and simpler operations
  - More advanced tools available, `fnmatch` module, regular expressions