PWC Python Course: Files, Execution, Objects

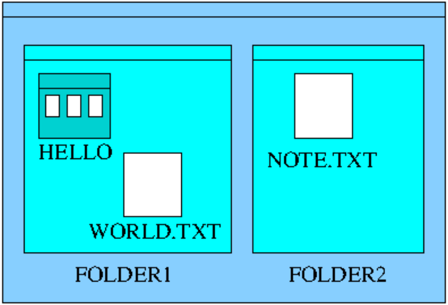
Ramses van Zon

SciNet HPC Consortium

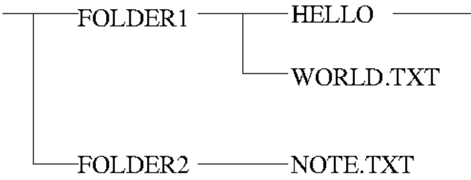
December 1,2, and 11, 2014

File System: Concepts

Files and directories



Tree:



- Files contain your data
- Files organized in directories/folders
- A directory is a file too
- Path: sequence of folders to get to a file

Files:

```
FOLDER1/WORLD.TXT
FOLDER2/NOTE.TXT
FOLDER1/HELLO/...
```

SciNet

Computer Data Storage

Media:

- Memory
- Disks
- Flash (USB)
- DVD
- Tape
- ...

.

All media are essentially linear strings of bits:

1	0	1	0	0	0	1	0	1	1	1	0	1	0	0	1	1	0	1	1	1	0	1	1

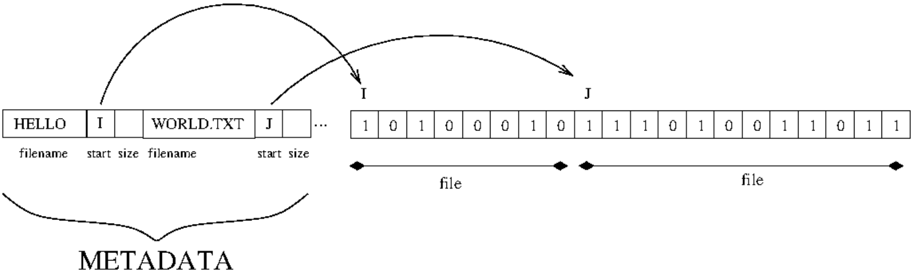
In and of itself, this is useless. What do these bits mean?

File systems

- Many non-volatile media use a file system
- A file system is a way to give meaning to the string of bytes.
- This entails storing data describing the meaning of the data: **metadata**

Files

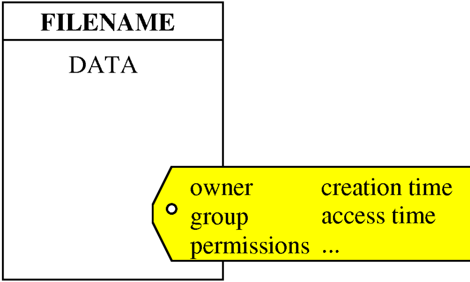
- Storage media is often subdivided into files
- Files have a name, a size and possibly other metadata
- Let's say that the metadata for the files is stored at the beginning of the storage media, e.g.



Metadata

Describes file properties:

- File name
- Within the file system: location on disk, size, etc.
- File type (extensions/magic identifiers)
- Owner, group
- Creation, access and modification times
- Read/write permissions (user, group, world, other access control)



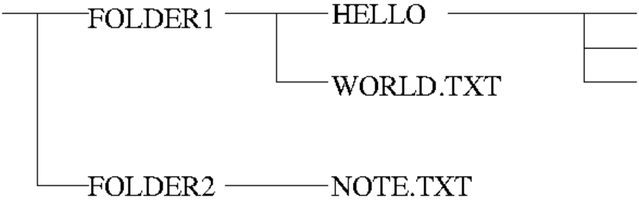
Directories or Folders

So we have files now, but this can get unorganized quickly. Imagine looking for the file ‘NOTE.TXT’ in a list of 10,000,000 files.

.

Directories

- Like special files that contain a list of (metadata for) other files.
- A directory can contain other directories, leading to a tree.



I/O Operations

What really happens if we open a file, write to it, etc.?

Opening a file:

1. Find the file in the directory
   Or create a new entry in the directory
2. Check permissions on the file
3. Find the location of the file on disk
4. Initialize a file 'handle' and file 'pointer'

I/O Operations

What really happens if we open a file, write to it, etc.?

Writing to a file:

1. Convert data to a stream of bytes.
2. Put those bytes in a buffer.
3. Update file pointer.
4. If buffer full: write to file

SciNet

I/O Operations

What really happens if we open a file, write to it, etc.?

Reading from a file:

1. If data not in buffer: read data into a buffer
2. Read bytes from buffer into variable, performing any needed conversion.
3. Update file pointer.

I/O Operations

What really happens if we open a file, write to it, etc.?

Closing a file:

1. Ensure buffers are flushed to disk
2. Update any metadata.
3. Release buffers associated with the file handle.

Minimizing IOPS

- Disk I/O is usually the slowest part of a pipe line.
- If manipulating data from files is most of what you do, try and minimize iops.

Bad

Writing out a string byte-by-byte, reopening the file each time

```
s = 'Hi world\n'
for c in s:
  f = open('hiworld.txt','a')
  f.write(c)
  f.close()
```

Good

Writing out a string in one fell swoop.

```
s = 'Hi world\n'

f = open('hiworld.txt','w')
f.write(s)
f.close()
```

- **Work in memory and reuse data if you can.**

What's in a file?

Text:

- Seems attractive: you can just read it.
- Must assign a bit pattern to each letter or symbol.
- For numerical data, representation in base 10 must be computed.

Binary:

- Usually: use same byte-representation on disk as the computer.
- Can suffer from portability.
- Some binary formats include info on the data, e.g.: hdf5. NetCDF.

Encoded:

- Various non-native, binary looking formats, e.g. pickle.
- Might be used to store non-trivial data structures.
- Example: python's pickle (later).

SciNet

Text format

- ASCII Encoding: 7 bits = character
- 128 possible, but only 95 printable characters
- Uses 8-bit bytes: storage efficiency 82% at best.
- ASCII representation of floating point numbers:
  - Needs about 18 bytes vs 8 bytes in binary: **inefficient**
  - Representation must be computed: **slow**
  - **Non-exact** representation

ASCII

<table>
<tr><th>integers</th><th>characters</th></tr>
<tr><td>32</td><td>(space)</td></tr>
<tr><td>33-47</td><td>!"#$%&'()*+,-./“</td></tr>
<tr><td>48-57</td><td>0-9</td></tr>
<tr><td>58-64</td><td>:;<=>?@</td></tr>
<tr><td>65-90</td><td>A-Z</td></tr>
<tr><td>91-96</td><td>[\]^_</td></tr>
<tr><td>97-122</td><td>a-z</td></tr>
<tr><td>123-126</td><td>{|}~</td></tr>
</table>

Text Encodings

ASCII: 7 bit encoding. For English.

Latin-1: 8 bit encoding. For western European Languages mostly.

UTF-8: *Variable-width* encoding that can represent every character in the Unicode character set.

Unicode: standard containing more than 110,000 characters.

Python can deal with these encodings:

```
# -*- coding: utf-8 -*-
s = u"Comment ça va?"
print s.encode('utf-8')
```

Binary output

- Output the numbers as they are stored in memory
- Why bother: Fast and space-efficient.

Writing 128M doubles:

File system:

ASCII	173 s
binary	6 s

ramdisk

ASCII	174 s
binary	1 s

- Not human readable.
  But is that really so bad? If you have 100 million numbers in a file, are you going to read them all?

Why you should not use raw binary data

Just dumping the memory is fast, but you loose the information on what it meant. E.g.:

- Dump a 2d array of 100x100 floating point numbers
- Gives a file of 800,000 bytes.
- If we give this to someone else, how do they know what it is?
  - 2d array of 100x100 numbers
  - array of 10,000 floating point numbers,
  - string of 800,000 characters,
  - . . . ?

Binary Formats

You could invent your own binary format, but it's better to take an existing standard: Saves you potential bugs, the burden of documentation and/or maintaining an IO library, as one probably already exists.

Pickle: A python specific format. Portable for the same version.

NumPy: Has a binary format called `npy` or `npz`.

NetCDF: A self-describing format: contains not only data but names, descriptions of arrays (`scipy.io.netcdf`).

Hdf5: Another standard, self-describing format (`pytables`) Almost a filesystem in a file.

Some best practices concerning I/O

- If your data is not text, do not save it as text.
- Choose a binary format that is portable.
- Minimize IOPS: write/read big chunks at a time, don't `seek` more than needed, try to reuse data or load more in memory.
- Don't create millions of files: unworkable and slows down directories.
- Stick to letters, numbers, underscores and periods in filenames.

File System: Nuts and Bolts

Python modules/packages for files

- built-in python file objects
- `os`, `os.path`
- `shutil`
- `pickle`, `shelve`, `json`
- `zipfile`, `tarfile`, ...
- `csv`, `numpy`, `scipy.io.netcdf`, `pytables`, ...

Directories

Create:

```
>>> import os
>>> os.mkdir('FOLDER')
```

Change current directory:

```
>>> os.chdir('FOLDER')
>>> os.chdir('..')
```

Where am I?

```
>>> os.chdir('FOLDER')
>>> print os.getcwd()
C:\Users\rzon\FOLDER
```

On unix, this would say something like `/home/rzon/FOLDER1`.

Backslash or forward slash?

- Linux and Mac prefer the (forward) slash / to separate directories.
- MS Windows prefers backslash \ for the same purpose. It also separates file trees by file volume (`C:`, `D:`, `...`).

What to do if you want to write cross-platform code?

1. MS Windows will accept the forward slash as well, except on the command-line, so you could use that in python code.
2. You can also use `os.sep` which is set to the operating system's preferred choice.
3. You can assemble and disassemble paths using `os.path.join` and `os.path.split`.

SciNet

Write to a text file

Writing

```
>>> import os
>>> f = open(os.path.join('FOLDER1','WORLD.TXT'),'w')
>>> s = "Hello\n"
>>> f.write(s)
>>> f.close()
```

Appending

```
>>> import os
>>> f = open(os.path.join('FOLDER1','WORLD.TXT'),'a')
>>> s = "World\n"
>>> f.write(s)
>>> f.close()
```

Read a text file

Read-Only

```
>>> import os
>>> f = open(os.path.join('FOLDER1','WORLD.TXT'),'r')
>>> s = f.readline()
>>> print s
Hello
>>> f.close()
```

Read/Write

```
>>> import os
>>> f = open(os.path.join('FOLDER1','WORLD.TXT'),'r+')
>>> f.seek(1)
>>> f.write('i ')
>>> f.seek(0)
>>> s = f.readline()
>>> print s
Hi lo
>>> f.close()
```

Glob

The glob package does only one thing: it finds all files or paths matching a specific Unix-style regular expression pattern, and returns them in a list.

```
>>> import glob
>>> f = glob.glob('*/*.TXT')
>>> print f
['FOLDER1\\NOTE.TXT', 'FOLDER1\\WORLD.TXT']
>>>
```

os.path

There are a number of useful file and directory-testing functions in os.path.

```
>>> print f
['FOLDER1/NOTE.TXT', 'FOLDER1/WORLD.TXT']
>>> import os
>>> print os.path.isfile(f[0])
True
>>> print os.path.isdir(f[1])
False
>>> print os.path.abspath(f[1])
'C:\Users\rzon\FOLDER1\WORLD.TXT'
>>> print os.path.expanduser('~')
'C:\Users\rzon'
```

If you're looking for a directory-testing function, it's likely in os.path.

Example: Copy files

Text file

```
>>> f = open("file1.txt","r")
>>> g = open("file2.txt","w")
>>> for line in f:
>>>   g.write(line)
>>> f.close()
>>> g.close()
```

Binary file

```
>>> f = open("file1.bin","rb")
>>> g = open("file2.bin","wb")
>>> chunk = f.read()
>>> g.write(chunk)
>>> f.close()
>>> g.close()
```

Shutil file/directory management

- Do we really have to open a file read it line by line, write it, and close the file just to copy a file in python?
- In the command shell, you'd do that with a simple cp or copy command.
- In Python, you get shell-like functionality from the `shutil` package.

```
>>> import shutil
>>> shutil.copyfile('file1.txt','file2.txt')
```

Main shutil functions

- `copyfile`
  Copy content of one file to another file.
- `copymode`
  Copy permissions of a file or directory to another.
- `copystat`
  Copy permissions and time-stamps of a file or directory to another.
- `copy`, `copy2`
  Copy content and permissions (and time-stamps, for `copy2`).
- `move`
  Move a file or directory to another place in the file tree.
- `copytree`
  Recursively copy a directory.
- `rmtree`
  Recursively remove a directory.

Catching errors: exceptions

Exceptions

- Even scripts written by the best programmers will occasionally fail due to unexpected circumstances.
- This is particularly true when dealing with IO, as the files could be in the wrong place, renamed, etc., without the script knowing.
- Python has a mechanism to catch errors and recover from that gracefully if possible.
- This mechanism is called ‘exceptions’
  - exceptions are ‘thrown’ by a function when an error occurs
  - exceptions can be caught by the piece of your python code that called that function
  - there are different kinds of exceptions, and your code could be setup such that it catches only particular types of exceptions

Example

(from https://docs.python.org/2/tutorial/errors.html)

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

A more involved example

(from https://docs.python.org/2/tutorial/errors.html)

```
>>> import sys
>>> try:
>>>     f = open('myfile.txt')
>>>     s = f.readline()
>>>     i = int(s.strip())
>>> except IOError as e:
>>>     print "I/O error({0}): {1}".format(e.errno, e.strerror)
>>> except ValueError:
>>>     print "Could not convert data to an integer."
>>> except:
>>>     print "Unexpected error:", sys.exc_info()[0]
>>>     raise
```

SciNet

Output formats

Pickle

- Base64 encoding using readable ASCII
- Portable for the same version of python.
- In the `pickle` module.
- Flexible, can **serialize** any structure.

```
>>> import pickle,os,numpy
>>> a = numpy.zeros((1000,1000))
>>> f = open('a.pickle','wb')
>>> pickle.dump(a,f)
>>> f.close()
>>> print os.path.getsize('a.pickle')
32000196
>>> g = open('a.pickle','rb')
>>> b = pickle.load(g)
>>> g.close()
```

Shelve

- You can pickle multiple variables in one file, but you must retrieve them sequentially.
- `shelve` allows to store multiple variables in one file, indexed by name, so you can retreive just the variable you want.

```
>>> import shelve,numpy
>>> a = numpy.zeros((1000,1000))
>>> b = {'b':'bb','c':'cc'}
>>> f = shelve.open('b_and_c')
>>> f['a'] = a
>>> f['b'] = b
>>> f.close()
>>> g = shelve.open('b_and_c')
>>> readb = g['b']
>>> g.close()
>>> print readb['b']
'bb'
```

Numpy input/output

`save(FILE,ARR)` save a numpy array to a .npy file

`savez(FILE,NAME1=ARR1,NAME2=ARR2)` save several numpy arrays to an uncompressed zip file with extension .npz

`savez_compressed(FILE,NAME1=ARR1,NAME2=ARR2)` save several numpy arrays to a compressed zip file with extension .npz

`load(FILE)` load numpy array(s) from .npy (.npz) file. If FILE is an .npz, a dictionary with keys equal to the names supplied to savez is returned.

`savetxt(FILE,ARR,delimiter=CH)` save numpy array as text, separated by character CH (thus it can create comma separated files)

`genfromtxt(FILE,ARR,delimiter=CH)` loads numpy array from text file, separated by character CH (thus it can create comma separated files). Options exist to e.g. skip headers.

CSV Format

- Comma Separated Values
- Common format for import/export
- Human readable

Sample csv file (data.csv):

```
3,4,5
4,3,2
5,6,7
```

CSV Format

- Comma Separated Values
- Common format for import/export
- Human readable

Sample csv file (data.csv):

```
3,4,5
4,3,2
5,6,7
```

Reading using the csv module

```
>>> import csv
>>> f = open('data.csv','r')
>>> s = csv.reader(f)
>>> a = [row for row in s]
>>> print a
[['3', '4', '5'], ['4', '3', '2'],
 ['5', '6', '7']]
```

CSV Format

- Comma Separated Values
- Common format for import/export
- Human readable

Sample csv file (data.csv):

```
3,4,5
4,3,2
5,6,7
```

Reading using the csv module

```
>>> import csv
>>> f = open('data.csv','r')
>>> s = csv.reader(f)
>>> a = [row for row in s]
>>> print a
[['3', '4', '5'], ['4', '3', '2'],
 ['5', '6', '7']]
```

... and the numpy module

```
>>> import numpy as np
>>> a = np.genfromtxt('data.csv',
...                   delimiter=',')
>>> print a
[[ 3.   4.   5.]
 [ 4.   3.   2.]
 [ 5.   6.   7.]]
```

Json

- JSON (JavaScript Object Notation) is a lightweight data-interchange format
- Human readable

Reading

```
>>> import json
>>> f = open("data.json","r")
>>> b = json.load(f)
>>> f.close()
>>> print b
[[3, 4, 5], [4, 3, 2]]
```

data.json

```
[[3.4.5],
[4,3,2]]
```

Writing

```
>>> import json
>>> f = open("newdata.json","w")
>>> b = [[3, 4, 5], [4, 3, 2]]
>>> json.dump(b,f)
>>> f.close()
.
```

newdata.json

```
[[3.4.5], [4,3,2]]
```

File Manipulation Exercise

Create a bunch of comma separate value files using the following code:

```
>>> import numpy
>>> for i in xrange(20):
...     a=i*numpy.arange(500)
...     a.shape=(100,5)
...     numpy.savetxt('a%02d.csv'%i, a, delimiter=',', fmt='%.6f')
```

Now create python script that:

1. Finds all .csv files in a directory (pretend not to know the filenames)
2. The script should move the csv files to a new directory 'csv_files'.
3. It should also convert each file to a numpy array and stores these as '.npy' files in a directory 'npy_files'.

External executable manipulation

Local execution: Subprocess

- You want to call an executable from within your python script
- Often, you'd want to give it some input and capture the output
- The `subprocess` module is designed for this purpose.
- This module defines `Popen`, `call`, `check_output`, ...

Example:

```
>>> from subprocess import Popen, PIPE, STDOUT
>>> c = 'dir'
>>> p = Popen([c],shell=True)
>>> e = p.wait()
 Volume in drive C is Windows
 Volume Serial Number is CE24-9C37

 Directory of C:\Users\rzon

30/11/2014  10:32 PM    <DIR>          .
30/11/2014  10:32 PM    <DIR>          ..
```

Details of module subprocess

- `subprocess.Popen(..)` 'spawns' a *background* process
- The calling script should ensure this process finishes!
- Command is *not* implicitly run through a shell.
- Specifying command as a list reduces ambiguity (spaces).

Example:

```
>>> from subprocess import Popen, PIPE, STDOUT
>>> c = 'dir'
>>> p = Popen([c],shell=True)
>>> e = p.wait()
 Volume in drive C is Windows
 Volume Serial Number is CE24-9C37

 Directory of C:\Users\rzon

30/11/2014  10:32 PM    <DIR>          .
30/11/2014  10:32 PM    <DIR>          ..
```

Interacting through input and output

The process you've spawned may

- receive text input from the command line (standard in),
- produces text output that you want to capture (standard out)

then you can use `stdin`, `stdout` and `stderr` arguments to tell `Popen` how to treat the input and or output streams:

1. taken from a file
2. taken from the same input/output as the parent process
3. should be new, independent input/output streams: `PIPE`s

How to call Popen

Syntax with some of the more common arguments:

```
p = Popen(cmd, stdin=None, stdout=None, stderr=None, shell=False)
```

- `cmd`: The command to execute is specified as a list. The first element is the executable, the rest are the arguments.
- `stdin`: When given a file-like object, redirects input from this object. When set to 'None', shares input from the parent python process. When set to `subprocess.PIPE`, creates a new independent stream.
- `stdout`: Similar as for `stdin`, but for output.
- `stderr`: Similar as for stdout, but for error messages. You can combine stdout and stderr by setting `stderr=subprocess.OUTPUT`.
- `shell`: If True, the command gets executed through a new shell. Do not use unless your command is a shell command.

Pipes

With PIPEs, you can send and receive text from the sub-process.

You get a file-like handle to the input, output and error streams of the sub-process with

```
>>> outputObject = p.stdout
>>> errorOutput  = p.stderr
>>> inputObject  = p.stdin
```

You can also pipe together several subprocesses, such that the output of one becomes the input of the other ("piping"), by setting the `stdin=` argument of one Popen to the `.stdout` property of another.

Warning:

Because of output buffering, simultaneously using stdin to steer the process and stdout to monitor that same process is near impossible.

Example of capturing output

```
>>> from subprocess import Popen, PIPE
>>> cmd = 'dir'
>>> p = Popen([cmd],stdout=PIPE,shell=True)
>>> for line in p.stdout:
...     print line.strip()
>>> e = p.wait()
...
30/11/2014  10:05 PM    <DIR>          FOLDER1
30/11/2014  10:21 PM    <DIR>          FOLDER2
```

There's an easier way, using `communicate`:

```
>>> from subprocess import Popen, PIPE
>>> cmd = 'dir'
>>> p = Popen([cmd],stdout=PIPE,shell=True)
>>> print p.communicate()[0]
...
30/11/2014  10:05 PM    <DIR>          FOLDER1
30/11/2014  10:21 PM    <DIR>          FOLDER2
```

Example of redirecting input

```
>>> from subprocess import Popen, PIPE
>>> open('readandwrite.py','w').write('print raw_input()\n')
>>> cmd = 'python'
>>> arg = 'readandwrite.py'
>>> out = open('output.txt','w')
>>> p = Popen([cmd,arg],stdout=out,stdin=PIPE)
>>> p.stdin.write('hello\n')
>>> e = p.wait()
>>> out.close()
>>> for s in open('output.txt','r'): print s
hello
```

Again, communicate can simplify this:

```
>>> cmd = 'python'
>>> arg = 'readandwrite.py'
>>> p = Popen([cmd,arg],stdout=PIPE,stdin=PIPE)
>>> print p.communicate('hello\n')[0]
hello
```

Call: Run command, get result

`subprocess.call(...)` combines `Popen` and `wait`:

```
>>> from subprocess import call
>>> call(['dir'],shell=True)
...
30/11/2014  10:05 PM    <DIR>           FOLDER1
30/11/2014  10:21 PM    <DIR>           FOLDER2
```

Variant that grabs output:

```
>>> from subprocess import check_output
>>> s = check_output(['dir'],shell=True)
>>> print s
...
30/11/2014  10:05 PM    <DIR>           FOLDER1
30/11/2014  10:21 PM    <DIR>           FOLDER2
```

Remote Execution

SSH

- Suppose we want our command to run on another server
- Ssh can run external commands.
- Ssh stands for secure shell. It allows you to login to another server based on passwords or secure keys. Traffic is encrypted.
- We could use the ssh command in combination with subprocess.Popen

```
>>> import subprocess, time
>>> p = subprocess.Popen(['ssh', 'login.scinet.utoronto.ca','ls'])
>>> print "Waiting",
>>> while p.poll() is None:
>>>     print "o",
>>>     time.sleep(0.1)
>>> print "Done"
Waiting o o o o FOLDER1 FOLDER2
 o Done
```

Ssh with Paramiko

- Using Paramiko is often a better approach.
- Paramiko is a python implementation of SSH.
- It even works if ssh is not installed!

```
import paramiko
ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(
  paramiko.AutoAddPolicy())
ssh.connect('142.150.188.52',
        username='rzon',
        password='thisisntit')
c = "ls"
rin,rout,rerr=ssh.exec_command(c)
print rout.readlines()
["FOLDER1\n", "FOLDER2\n"]
```

- Open a paramiko SSH Client
- Set the missing key policy to "auto", so we can connect to new servers
- Connect through an IP address
- execute the 'ls' command
- read the output

Getting the IP address from a hostname

This functionality is not provided by paramiko, but we can use ‘socket’ for this.

```
>>> import socket
>>> print socket.gethostbyname('login.scinet.utoronto.ca')
142.150.188.52
```

FYI: `socket` is a module that allows processes, local or remote, to talk with one another through ports. This can be more convenient than sending data over stdin and stdout. For lack of time, we will not cover this today.

Ssh background process with Paramiko

- As with subprocess, paramiko launches the command asynchronously.
- This allows you to do other stuff while you wait.
- It might seem you would have to parse stdout to see when it is done, but there's the `channel.closed` property to help you with that.

```
>>> import time
>>> rin,rout,rerr=ssh.exec_command("sleep 10; ls")
>>> while not rout.channel.closed:
...     print "o",
...     time.sleep(1)
o o o o o o o o o o
>>> print rout.readlines()
[u"FOLDER1\n", u"FOLDER2\n"]
```

Expecting a lot of output? You will need to `readline` it continuously, to avoid buffer overflow and stallling the remote process.

File transfer

Is also possible with paramiko. Just a small example:

```
>>> import paramiko
>>> ssh = paramiko.SSHClient()
>>> ssh.set_missing_host_key_policy(
>>>   paramiko.AutoAddPolicy())
>>> ssh.connect('142.150.188.52',
>>>           username='rzon',
>>>           password='thisisntit')
>>> ftp=ssh.open_sftp()
>>> ftp.put('localinput.csv','remoteinput.csv')
>>> #some remote command goes here, presumably
>>> ftp.get('remoteoutput.npy','localoutput.npy')
```

Exercise

Write a script that does one of the csv-to-numpy conversions remotely. Use ip address 127.0.0.1, which means it just runs on your local machine.

Debugging and Profiling

Debugging

So you're logging, catching exceptions, doing good resource management. Still the script doesn't work. What to do?

Debugging

- This is the process of systematically finding errors in your code.
- You could add a bunch of print statements, but this tends to be rather unproductive, as it gets you in a cycle of adding more and more print statements, that later have to be removed.
- Within eclipse, there is a 'Debug' mode. It allows you to step through your code line by line, and inspect variable values.

Debugging Python in Eclipse

Demonstration

Profiling

- Okay, so our script works but it is very slow, or runs out of memory.
- Profiling is not integrated in eclipse, so we'll need some auxiliary modules.
- Two very common bottlenecks are:
  - Performance issues
  - Memory problems
- These two are separately addresses by the following modules
  - line_profiler
  - memory_profiler
- There are also the standard python profilers `Profile` and `cProfile`, but these consider the cost of whole functions, not lines.

line_profiler

- Use line_profiler to know, line-by-line, where your script spends its time.
- As with debugging, you usually do this on a smaller but representative came.
- First thing to do is to have your code by in a single function (we'll look at functions more tomorrow)
- You also need to include modify your script slightly:
  - *decorate* your function with @profile
  - run your script on the command line with 'kernprof -l -v SCRIPTNAME'

line_profiler script instrumentation

Script before:

```
a=""
a+="lines of\n"
a+="python code\n"
print a
```

Script after:

```
#file: profileme.py
@profile
def profilewrapper():
    a=""
    a+="lines of\n"
    a+="python code\n"
    print a
profilewrapper()
```

Run at the command line:

```
kernprof -l -v profileme.py
```

Output

```
lines of
python code
Wrote profile results to profileme.py.lprof
Timer unit: 1e-06 s

Total time: 0.000193 s
File: profileme.py
Function: profilewrapper at line 1

Line #   Hits     Time  Per Hit   % Time  Line Contents
=======================================================
     2                                     @profile
     3                                     def profilewrapper():
     4      1       13     13.0      5.3       a=""
     5      1        5      5.0      2.0       a+="lines of\n"
     6      1        3      3.0      1.2       a+="python code\n"
     7      1      225    225.0     91.5       print a
```

memory_profiler

- This module/utility monitors the python memory usages and its changes throughout the run.
- Good for catching memory leaks and unexpectedly large memory usage.
- Needs same instrumentation as line_profiler.
- On Windows, requires the `psutil` module.

memory_profiler details

Your decorated script is usable by memory_profiler.

You run your script through the profiler with the command

```
python -m memory_profiler profileme.py
```

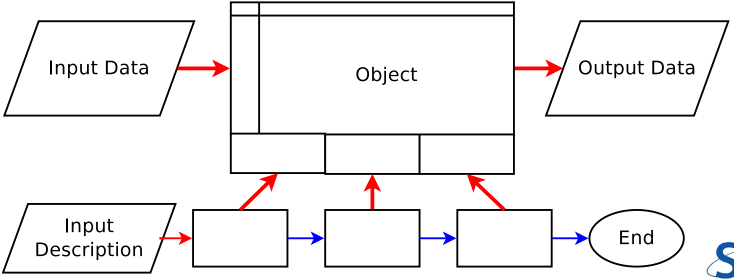
Output

```
lines of
python code

Filename: profileme.py

Line #    Mem usage    Increment   Line Contents
================================================
     2    9.621 MiB    0.000 MiB   @profile
     3                             def profilewrapper():
     4    9.625 MiB    0.004 MiB       a=""
     5    9.625 MiB    0.000 MiB       a+="lines of\n"
     6    9.625 MiB    0.000 MiB       a+="python code\n"
```

Objects

Objects

- Functional programming: data and the functions that can act on that data, are defined separately.
- Object oriented programming, the functions belong to the data structure.
- Better consistency, modularity, and reusability of your code.
- Implementation in python using the `class` construct.



Classes in Python

- Classes are used to group together data and code, accessing them with the `.` operator.
- One could also do this with modules. But there can be only one instance of a module, and many instances of a class.
- Inheritance: multiple base classes, derived class can override any methods of its base class or classes, and method can call a base class method with the same name.
- Objects can contain arbitrary amounts and kinds of data.
- As everything in Python, classes are dynamic: created at runtime, and can be modified further after creation.

Classes as collections of variables

```
class Apple:
    type = "Delicious"
    colour = "Green"
apple1 = Apple()
apple2 = Apple()
Apple.colour = "Golden"
print apple1.colour
```

Outputs: `Golden`

apple1 and apple2 *share* `colour` (class variable): tricky.

```
class Apple: pass
apple1 = Apple()
apple1.type = "Delicious"
apple1.colour = "Green"
apple2 = Apple()
apple2.type = "Delicious"
apple2.colour = "Golden"
print apple1.colour
```

Outputs: `Green`

This works, but now we have to assign each member.
Anything more workable requires writing a constructor.

Initializing objects with constructors

- Collection of variables
- Same `def` keyword to define methods.
- Constructor name is `__init__`

```
class Apple:
   def __init__(self):
      self.type="Delicious"
      self.colour="Green"
apple1 = Apple()
apple2 = Apple()
print apple1.colour
```

Outputs `Green`

Class syntax in Python

- Methods take a first argument that is an instance of the class
- This argument is explicit `self` in definition but implicit in calls.
- In methods, refer to member fields as `self.field`.
- No separation interface/implementation

```
class Apple:
   def __init__(self):
      self.type="Delicious"
      self.colour="Green"
   def describe(self):
      print self.type,
            self.colour

apple1 = Apple()
apple2 = Apple()
print apple1.colour
[Green]

apple1.describe()
[Delicious Green]
```

More special methods

- `__del__`
  A kind of destructor.
- `__str__`
  Converts object to a string for output. Used by `print`. Intended to be readable by users.
- `__repr__`
  Returns a string representation for the object. Used by python (e.g., if you just type the name of an object). Intended to be understandable by developers.

Example: Particle

```
class Particle(object):
   def __init__(self,m,x0,v0):
      self.t = 0.0
      self.m = m
      self.x = x0
      self.v = v0
   def timeStep(self,dt):
      self.t += dt
      self.x += dt*self.v
   def __str__(self):
      return str(self.t)+" "+str(self.x)+" "+str(self.v)
p = Particle(2.0,0.0,-1.0)
while p.t <= 10.0:
   p.timeStep(0.1)
   print p
```

Inheritance in Python

- A class can de derived from another class
- This means that class variables and methods are carried over to the new class.
- Put classes to derive from between parenthesis in the definition.

```
class NamedParticle(Particle):
   def __init__(self,m,x,v,name):
     Particle.__init__(self,m,x,v)
     self.name = name
   def __str__(self):
     return self.name+": "+Particle.__str__(self)
t = NamedParticle(1.0,2.0,-1.0,"Al")
print t
```

SciNet

New-style classes

- Two types of classes in Python:
  - Old style
  - New style: must derive (ultimately) from ‘object’ class
- New style allows for operator overloading, properties, and better multiple inheritance.

```
class Particle(object):
  #...
```

Accessing derived data with properties

Suppose we have a function that computes the kinetic enery of a particle:

```
def kineticEnergy(particle):
   return 0.5*particle.p**2/particle.m
```

This definition assumes that `particle` stores the momentum of the particle. This is not the case for object of the Particle class, which stores the velocity. So it would appear that we'll need to rewrite this function, using that momentum is mass times velocity.

However, using **properties**, one has a syntax to access the momentum as if it were a member variable, but which really calls a getter or setter function.

Derived property example

The momentum property is derived from the velocity variable:

```
class PParticle(Particle):
   #...
   def pget(self):
      return self.m*self.v
   def pset(self,p):
      self.v = p/self.m
   p = property(pget,pset)
```

We can then use p as if it were an object variable.

Slightly better example

Often one uses properties to enforce a validation on allowed values.

For instance:

```
class Particle(object):
   c = 3.0e8
   def vget(self):
      return self._v
   def vset(self,v):
      if (v<=self.c):
         self._v=v
      else:
         raise ValueError("Can't go faster than light!")
   v=property(vget,vset)
```

Overloading operators

- If you define you own object, you may want to define whatit means to e.g. add or multiply these objects.
- In Python you can overload an operator by defining a member function that is equivalent to the operator.
- For instance, the member function that is equivalent to addition is __add__

```
>>> class pricedItem(object):
...  def __init__(self,item,price)
...   self.item = item
...   self.price = price
...  def __add__(self,b):
...   item2=self.item+"+"+b.item
...   price2=self.price+b.price
...   a=pricedItem(item2,price2)
...   return a
...
>>> a = pricedItem("Apple", 1.0)
>>> b = pricedItem("Pear", 0.5)
>>> c = a+b
>>> print c.item, c.price
Apple+Pear 1.5
```

SciNet

Operators

Operation	Notation	Functional equivalent
Addition	a + b	a.__add__(b)
Subtraction	a - b	a.__sub__(b)
Multiplication	a * b	a.__mul__(b)
Power	a ** b	a.__pow__(b)
Division	a / b	a.__truediv__(b)
Floor Division	a // b	a.__floordiv__(b)
Remainder	a % b	a.__mod__(b)
Left Shift	a << b	a.__lshift__(b)
Right Shift	a >> b	a.__rshift__(b)
AND	a & b	a.__and__(b)
OR	a | b	a.__or__(b)
XOR	a ^ b	a.__xor__(b)
NOT	~a	a.__invert__()