Python:
Operating system access

Bob Dowling

scientific-computing@ucs.cam.ac.uk

www.training.cam.ac.uk/ucs/course/ucs-pythonopsys

Welcome to the “Operating System Access in Python” course.

Our task — 1

Write a script that…

…processes its command line

…navigates the file system

…runs external programs

UCS

What do we mean by “operating system access”? In this session we are going to write a Python script that interacts in the operating system in three particular ways all of which are fairly common.

We are going to interact with our command line, somewhat more powerfully than we do elsewhere.

We are going to navigate around the file system.

We are going to launch external program, for those rare times when we can't find a Python module that does the job!

Our task — 2

…the script must also…

in each directory it visits: `dir`$_1$ `dir`$_2$ `dir`$_3$ `dir`$_4$

read parameters from a file — `params.cfg`

read data from multiple files — `*.dat`

write an output file — `output.dat`

The script is going to visit a number of directories, specified on the command line, and in each is going to run a program using a parameters file as the argument to the program, combining a number of input files to feed to that program and putting its output in a file in that directory. The names of the parameters file and output file will be passed as parameters on the command line as will the wildcard expression for the set of input files to combine.

Our task — detailed spec.

From the command line get a parameters file name

an output file name

an input file name pattern

a list of directories

Visit each directory in the list and find the matching input files

run the commands

```
sort -n input₁ input₂ input₃ … |
  plotter parameter_file > output_file
```

in each directory

This is the sort of detailed specification you might get if you were doing this for real.

The command to run in each directory is essentially made up for the purposes of this course. The `plotter` program is a special just for this course too.

The command line — simple

Recall: `sys.argv`

```
#!/usr/bin/python

import sys
print sys.argv
```

args.py

```
$   ./args.py one two

['./args.py', 'one', 'two']
```


Let's start with the command line.

Other Python courses have met it in its primitive form; the list `sys.argv` from the `sys` module.

The command line — complex

```
$ ./myscript --output=output.dat
--params=params.cfg dir1 dir2 dir3
```

```
$ ./myscript --help
```

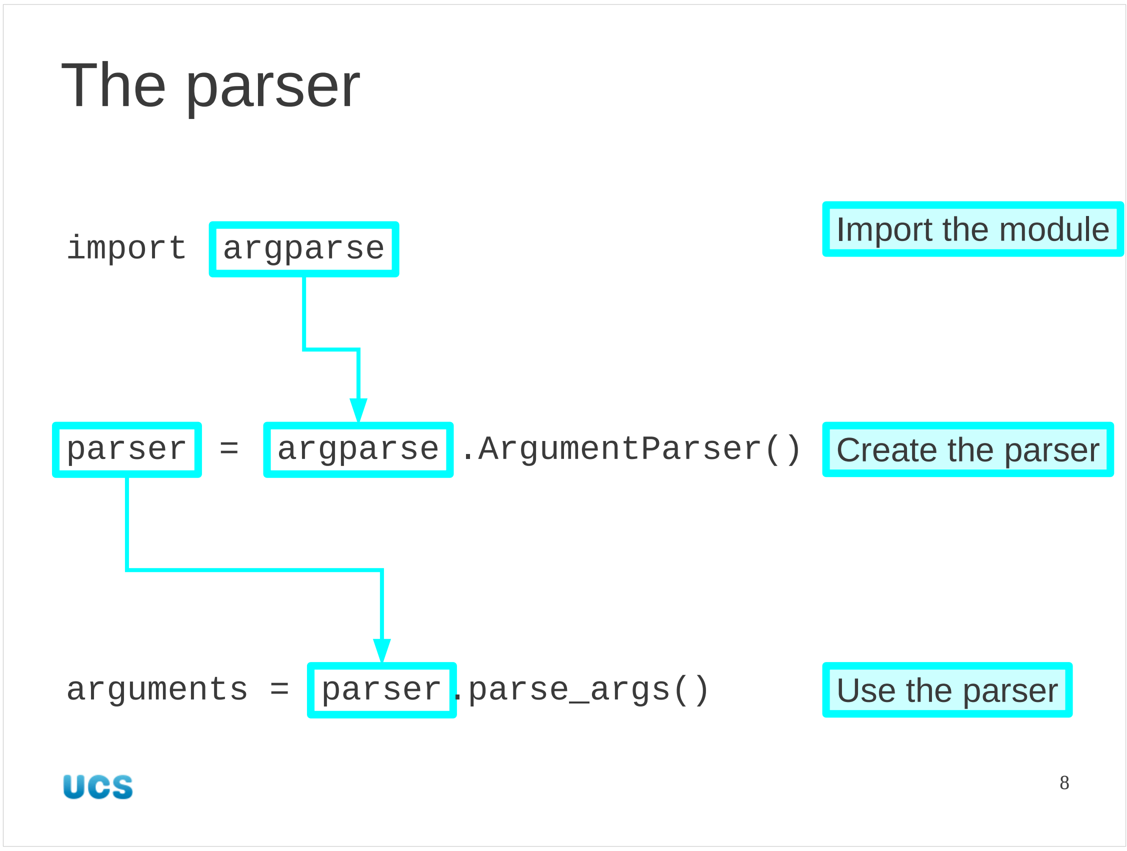

We want to be able to support a script that does something like this: It uses long format options with support for short forms too and it has proper help support too.

The argparse module

You…	It…
Describe the program	Builds help automatically
Define the valid options	Processes options given

There is a Python module specifically for processing the command line. It is called "`argparse`".

There is also an older, less useful module called "`getopt`". We recommend you use `argparse` in preference. Older versions of this course referred to a module called `optparse`. This has now been replaced by `argparse`.



To use it, obviously, we have to import it first. It contains a single function useful to us, called “`ArgumentParser()`”. (Warning: Python is case sensitive so mind the capitals.)

This function hands us a “parser”, a program which interprets the text of the command line.

We haven't told it what to expect on the command line yet, but we can still use it. The parser object has a method “`parse_args()`” which interprets the command line by default and returns a pair of values. The first carries he information gathered about the options and the second the remaining command line arguments. We will come back to it shortly. In the mean time we will consider what this bare bones program actually does.

Functionality!

```
#!/usr/bin/python

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

Only three lines!

```
$ ./parse1.py --help
usage: parse1.py [-h]

optional arguments:
  -h, --help  show this help message and exit
```


You can find this bare bones program in your home directories as “`parse1.py`”. It is executable, but because “`.`” is not on your `PATH` you will have to call it with “`./parse1.py`”:

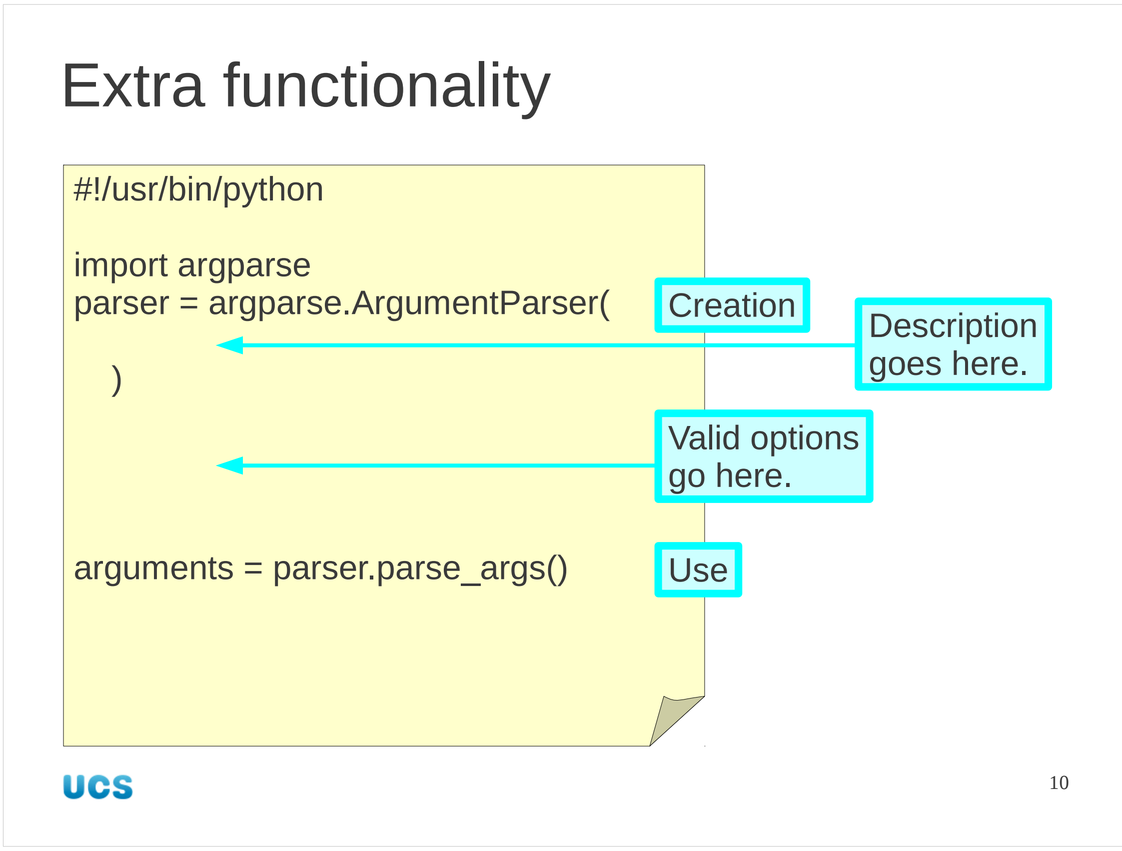
```
$ ./parse1.py --help
```

or by running it under Python explicitly:

```
$ python parse1.py --help
```

Clearly it supports help options. The unconfigured parser object knows how to deal with “`--help`” or “`-h`” on the command line.

p.s.: Never define either of these two options. You do not want to override this built-in behaviour.



Of course, what we have to do is to add some more settings to tell it about the options we expect.

This will come in two phases. The descriptive text that heads the help message gets set when the parser is created. All the various additional options get added to the parser after it is created (but before it is used, obviously).

Setting a description

```
desc_text = """
Visit listed directories and process
the data files in them.
"""

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

A “named argument”

To start with we will expand on the bare bones help text it generates by default. We will define a multi-line string with out text in it. We pass this into the parser as we create it with the named “description” parameter.

The text you give for the description will be automatically line wrapped as it is displayed. Do not attempt clever formatting of your own in this section.

```
import argparse

desc_text = """Visit listed directories and process
the data files found in them.
"""

parser = argparse.ArgumentParser(description=desc_text)

arguments = parser.parse_args()
```

```
$ ./parse2.py --help
usage: parse2.py [-h]

Visit listed directories and process the data
files found in them.
…
```

UCS

The program `parse2.py` has the descriptive text modified. Note how the line wrapping has been done for us.

Setting an optional argument — 1

```
parser.add_argument(
    '-o',
    '--output',
    help = 'Name of output file',
    dest = 'output_file',
    default = 'output.dat',
    type = str
  )
```

Short form option

Long form option

Help text

Variable name

So how do we specify that we want to accept a particular option on the command line? To tell the parser to expect it, again before `parse_args()` needs it, we use the "`add_argument()`" method.

We will focus on three or four of the method's options at this point.

The first two arguments give the long and the short forms. Either or both can be dropped. We will come back to dropping both later.

After that we have named arguments. The "`help`" argument is the single line that will be used to create `--help`'s text for this argument.

The "`dest`" argument (for "**dest**ination") is mostly used for getting at the option later but we will see it in the help output too so we mention it here.

We will return to the other options shortly.

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

parser.add_argument(…)

(options, arguments) = parser.parse_args()
```

```
$ ./parse3.py --help
usage: parse3.py [-h] [-o OUTPUT_FILE]

Visit listed directories and process the
data files found in them.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT_FILE, --output OUTPUT_FILE
                        Name of output file
```


We still don't know what to do with this option once it has been found on the command line by `parse_args()` but we can already see how its existence changes the behaviour of the `--help` option. The `parse3.py` script has just one option added. We can try this with `--help`.

Note that an extra line has been produced for our new option and that it uses our given help text. Also note that the "`OUTPUT_FILE`" comes from the "`dest`" argument we set.

Setting an optional argument — 2

```
parser.add_argument(
    '-o',
    '--output',
    help = 'Name of output file',
    dest = 'output_file',          Variable name
    default = 'output.dat',        Default value
    type = str                     Expected type
   )
```

UCS

Now let's look at actually getting at the option itself.

The `dest` argument specifies where the `--output`'s value is put. It doesn't quite define a simple variable name but it's not far off. We will see a worked example next slide.

The "`default`" parameter specifies the default value to set.

Finally there is a "`type`" parameter. This takes a Python type as its argument and the string read from the command line will be automatically converted into this type on parsing.

```
…
parser.add_argument(
    '-o',
    '--output',
    help    = 'Name of output file',
    dest    = 'output_file',
    default = 'output.dat',
    type    = str,
    )

arguments = parser.parse_args()

print 'Output file:', arguments.output_file
```

parse4.py

So how do we get at the option's value?

You will recall that `parse_args()` returns a value. This is an object that encodes all the arguments. We said we wanted an option parsed with the destination “`output_file`”. As a result the arguments object passed back has a member with that name.

```
$ ./parse4.py --output=foo.out
Output file: foo.out

$ ./parse4.py
Output file: output.dat
```


The `parse4.py` script has had this done to it and prints out the value the parser gets from the `--output` option.

Setting compulsory arguments — 1

```
…
parser.add_argument(
    …something…
    )

arguments = parser.parse_args()

print 'Output file:', arguments.output_file
print 'Directories:', arguments.directories
```

No option

```
$ ./parse4.py --output=foo.out dir1 dir2 dir3
Output file: foo.out
Directories: ['dir1', 'dir2', 'dir3']
```


So what do we do for the arguments that don't have any sort of “double dash” or “single dash” option?

How can we write a version of our parser script that catches a list of the remaining arguments?

Setting compulsory arguments — 2

Try this…

```
parser.add_argument(


    help = 'Directories to be processed',
    dest = 'directories',
    default = [],
    type = str
   )
```

No option strings!

List of strings?

UCS

Given that we have no option string for the compulsory arguments, there is a natural way to specify the compulsory arguments with `add_argument`: simply leave out the option strings!

We set a default empty list and a type of `str`, hoping that we will get back a list of strings.

Let's try this.

Setting compulsory arguments — 3

Almost…

```
$ ./parse5.py --output=foo.dat dir1
Output file: foo.dat
Directories: dir1
```

✓

```
$ ./parse5.py --output=foo.dat dir1 dir2
usage: parse5.py [-h] [-o OUTPUT_FILE] directories
parse5.py: error: unrecognized arguments: dir2
```

✗

Can't cope with more than one positional argument.

UCS

It almost works. If there is a single argument then it works fine. If there is more than one then it fails, only being able to process the first and failing on the second.

Setting compulsory arguments — 4

One last argument…

```
parser.add_argument(
    help = 'Directories to be processed',
    dest = 'directories',
    default = [],
    type = str,
    nargs = '*',
   )
```

UCS

For this to work we need one last parameter in the `add_argument()` method: `nargs` (pronounced “en args”). Setting it to the Python string `'*'` means that `parse_args()` will process as many arguments as there are (zero or more).

If `nargs` is set then the result will always be a list of values (empty if zero items are on the command line, with a single element in the list if there is just one, etc.)

nargs

`nargs = '*'`	0 or more arguments
`'+'`	1 or more arguments
`5`	exactly 5 arguments

If `nargs` is set, the answer is always a list.

```
type = str
```

A list of strings.

`nargs` can also be set to a number, in which case it says that exactly that many arguments are expected, or the string `'+'` to mean “at least one argument”.

The type used in `add_argument()` will now be applied to each element in the list.

Setting compulsory arguments — 5

```
$ ./parse6.py --output foo.out
Output file: foo.out
Directories: []


$ ./parse6.py --output foo.out dir1
Output: foo.out
Directories: ['dir1']


$ ./parse6.py --output foo.out dir1 dir2
Output: foo.out
Directories: ['dir1', 'dir2']
```

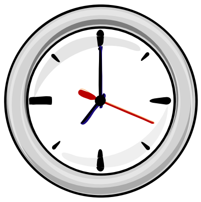

The `parse6.py` script prints out the second argument.

Exercise 1

Complete `exercise1.py`

1. Set help text
2. Set options
3. Print out options variables
4. Print out remaining arguments
   (call the list “directories”)

Ten minutes



UCS

Got that?

Now it's time to put it to the test.

You have a file in your home directory called “exercise1.py”. This has certain critical elements missing and replaced by comments telling you what to do.

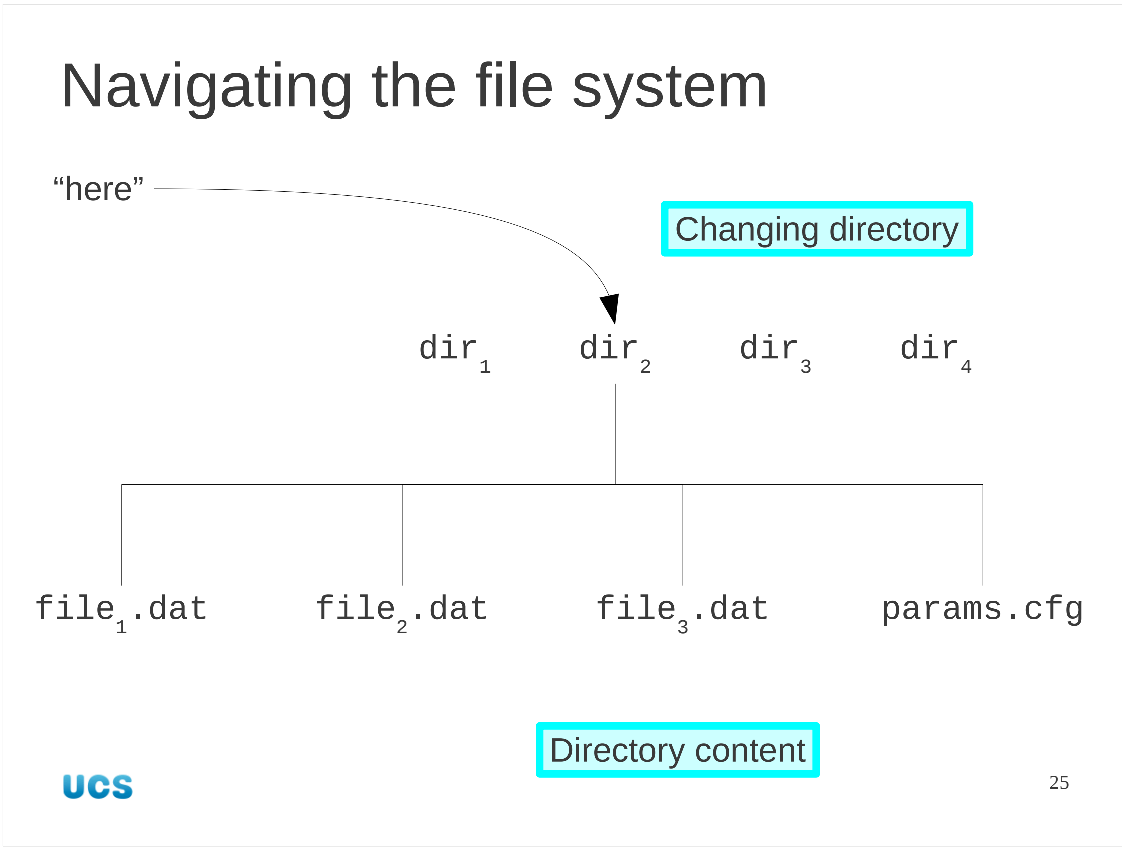
Write some help text of your own.

The options you should set are these:

		dest	default	help
--output	-o	output_file	output.dat	“The output file”
--input	-i	input_pattern	*.dat	“The wildcard for the input files to use”
--params	-p	params_file	params.cfg	“The parameter file in the directory”

Print out all three options and the directories list.

Test the script!



Our next interaction with the operating system is via the file system. Our script is going to be passed a list of directories (our compulsory arguments) where it should go and launch a program. So we need to be able to move around the file system. Our script is also going to combine various input files so we will need to be able to read the content of directories too.

The “os” module

“**o**perating **s**ystem” module

`sys`	Universal systems-y facilities
`os`	Facilities varying between operating systems Some may not be provided by base system This course: Unix & Windows

We will need another module to give us this functionality. This is the “`os`” module and provides simple access to the **o**perating **s**ystem.

We should pause for a moment to contrast it with the `sys` module that gives something very similar. The idea is that `sys` gives those things that are the same on all operating systems and the `os` module gives those that are different (o/s-specific).

The components of the `os` module that you see in this course are common to Unix (Linux & MacOS) and Windows.

Changing directory

```
>>> import os

>>> os.getcwd()
'/home/y550'
>>> os.chdir('/tmp')
>>> os.getcwd()
'/tmp'
```

get current **w**orking **d**irectory

change **dir**ectory

Let's start by changing directory.

This is best illustrated in an interactive Python session. We import the module and can then use the `os.getcwd()` function to get the file path of the current working directory.

We can change directory with the `os.chdir()` function.

Using the current directory

```
>>> os.chdir('/home/y550')          Start here
>>> f = open('first', 'w')          Create a file
>>> os.chdir('/tmp')                Move here
>>> s = open('second', 'w')         Create a file
>>> f.close()
                                    good habit
>>> s.close()
```

```
$ ls -l first /tmp/second
-rw-r--r-- 1 rjd4 rjd4 0 … first
-rw-r--r-- 1 rjd4 rjd4 0 … /tmp/second
```

UCS

Perhaps a quick reminder is called for as to why the current working directory matters. If we create a file with a simple name (so no directories in it) then it appears in our current working directory.

Only in the Python process

```
$ pwd
  /home/y550

$  python
…
…

$ pwd
  /home/y550
```

```
>>>  import os
>>>  os.getcwd()
'/home/y550'
>>>  os.chdir('/tmp')
>>>  os.getcwd()
'/tmp'
>>>  [Ctrl]+[D]
```

No change in the shell

Change directory in the Python process

It's also worth recalling that while the Python process changes directory the shell that launched it continues on with magnificent indifference in what ever directory it was to start with.

Lists of directories

```
…
for directory in directories:
    os.chdir(directory)
    print 'CWD:', os.getcwd()
```

This looks innocent enough…

Now we need to cover a common mistake made by many beginners.

Suppose we have a list of directories (you have `alpha`, `beta` and `gamma` defined in your home directories for this purpose). What could be simpler than to run through a list of these directories one after the other moving into each in turn?

The script shown has a critical bug.

Lists of directories

```
…
for directory in directories:
    os.chdir(directory)
    print 'CWD:', os.getcwd()
```

```
$  ./directory1.py /tmp /home /usr

CWD: /tmp
CWD: /home
CWD: /usr
```


The bug doesn't always trigger. The script directory1.py has the bug but works fine if I give it the list of three directories shown.

Note that these are all given by absolute paths, i.e. their paths start with a leading “/”.

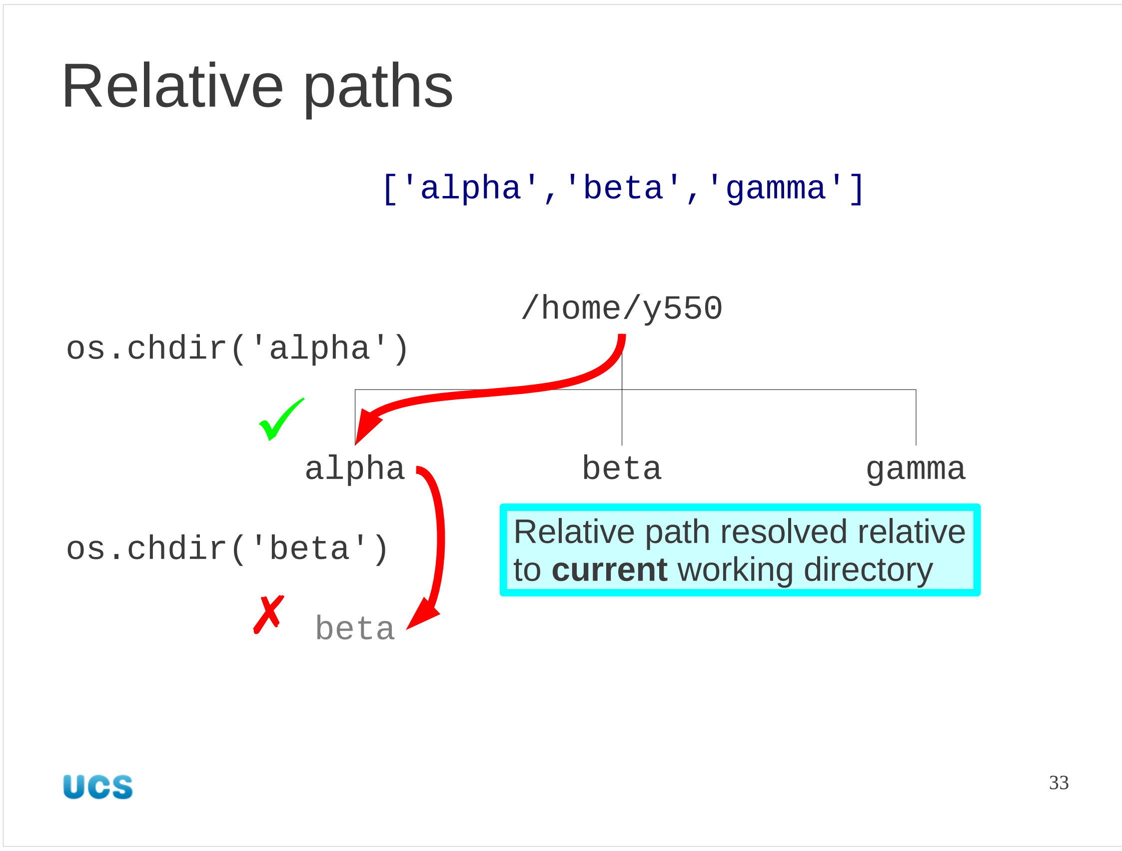
Lists of directories

```
…
for directory in directories:
    os.chdir(directory)
    print 'CWD:', os.getcwd()
```

```
$  ./directory1.py alpha beta gamma

CWD: /home/rjd4/Courses/PythonOS/alpha
Traceback (most recent call last):
  File "./directory1.py", line 7, in <module>
    os.chdir(directory)
OSError: [Errno 2] No such file or directory: 'beta'
```

✓ ✗

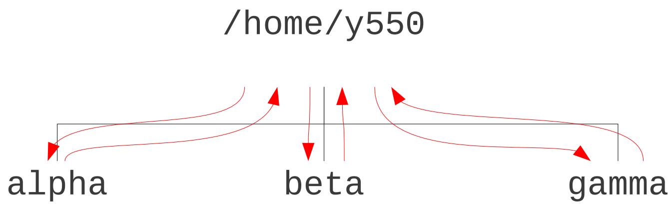
But if we run it with the three directories in your home directory it works for the first and then fails, complaining about the second.



What is going wrong is this: the first change of directory works. We end up in `alpha`. The second change of directory into `beta`, however, is performed *relative to the current working directory*, which is now `alpha`. There is no directory `beta` under `alpha`.

Cleanest solution

`['alpha','beta','gamma']`



Go back after each directory

UCS

The cleanest solution whenever you are working through a list of directories is to always return to where you started after “doing” a directory. Then when you move into the next directory listed you are doing it from where you started, not where you ended up.

Lists of directories

```
…
oldcwd = os.getcwd()
for directory in directories:
    os.chdir(directory)

    print 'CWD:', os.getcwd()

    os.chdir(oldcwd)
```

```
$  ./directory2.py alpha beta gamma

CWD: /home/y550/alpha
CWD: /home/y550/beta
CWD: /home/y550/gamma
```


The script `directory2.py` has this fix and works.

It starts by recording where the script starts before the first `os.chdir()`. It then brackets each set f operations in a different directory by the action of going to that new directory and the action of returning to the old one.

Lists of directories

```
…
oldcwd = os.getcwd()
for directory in directories:
    os.chdir(directory)

    something_useful()

    os.chdir(oldcwd)
```

1. Start here
2. Loop
3. Go
4. Do
5. Return

“All” we have to do now is write `something_useful()` …

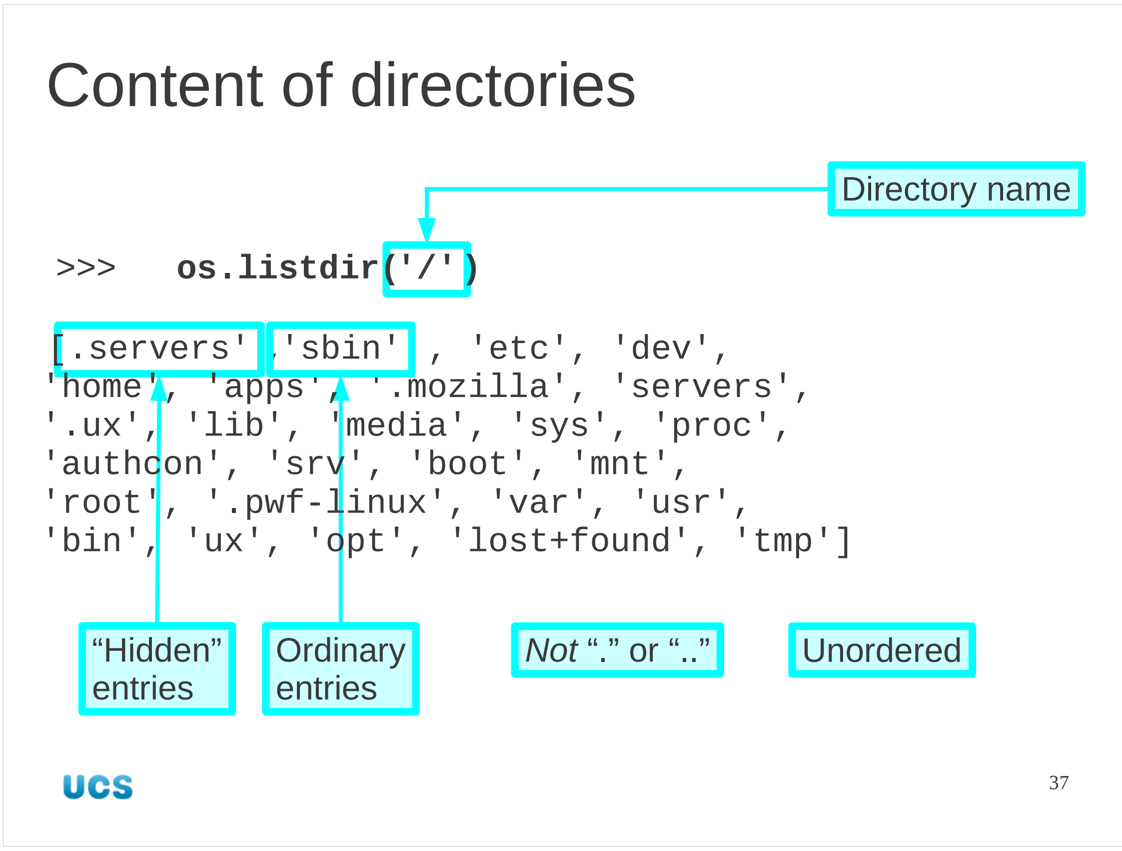
`directory3.py`

The general pattern for working through a list of directories goes like this:

1. We remember where we started.
2. We loop through the set of directories.

   For each directory in the loop…
3. We change to the directory
4. We do whatever it is we came to do. This is most cleanly done in a function so we don't distract from the navigation code.
5. We change directory back to where we started and then move on to the next directory in the list.

The script `directory3.py` is exactly the same as `directory2.py` except that the loop's core activity, printing out the current working directory, has been moved into a new function, called `something_useful()`.



We commented at the start that we needed to be able to see what was in a directory as well as how to move into them. We so this with the `os.listdir()` function.

The output excludes the "`.`" and "`..`" directories that are always present on Unix systems.

The list given is not in any particular order. It's a function of how the directory stores its information. If we want it sorted, we have to sort it ourselves.

Content of directories

```
>>>  os.listdir('/')          Absolute path

>>>  os.listdir('alpha')      Relative path

>>>  os.listdir('../..')      Relative path

>>>  os.listdir('.')          Current working directory
```

UCS

The directory name can be an absolute or relative path. Recall that “`.`” means “the current working directory” and “`..`” means the parent directory.

Content of directories

```
…
def something_useful():
    print 'CWD:', os.getcwd()
    files = os.listdir('.')
    files.sort()
    print files
…
```

1. Where?
2. What?
3. Sorted
4. Output

```
$  ./directory4.py alpha beta gamma
```

UCS

Let's get back to writing a script again.

The script `directory4.py` is the same as `directory3.py` but has an extended function to print a sorted list of the contents of each directory.

We will slowly build up this function. So far it simply builds an ordered list of the files in the directory and prints them out.

Doing something useful

Select the input files from `files`

Output file: `output.dat`

Parameters file: `params.cfg`

Input files: `*.dat`

```
            ✓               ✓               ✓
['input1.dat','input2.dat','input3.dat',
      ✗             ✗              ✗
 'notes','output.dat','params.cfg',
          ✗                ✓
 'readme.txt','variants.dat']
```


Now we must consider our script's purpose again. It is meant to pull out just the input files in the directory that match the pattern given. We must also exclude the output and parameters file in case they are covered by the input pattern too.

So if our input pattern is “`*.dat`”, our parameters file is “`params.cfg`”, and our output file is “`output.dat`” we should take the files shown with ticks as inputs.

Filtering the files — 1

Pass the options into the function.

```
def something_useful(arguments):
    …
```

Definition

```
for directory in directories:
    os.chdir(directory)
    something_useful(arguments)
    …
```

Use

First of all we need to pass the options into the `something_useful()` function. We do this simply by passing the `arguments` object.

Filtering the files — 2

Remove the two named files.

```
def something_useful(arguments):
    print 'CWD:', os.getcwd()
    files = os.listdir('.')
    files.sort()

    files.remove(arguments.output_file)
    files.remove(arguments.params_file)
```

Fails if they are not there

UCS

Next we want to take the files list we already have and strip the output and parameters file from it.

This script will fail, however, if the files are not there to start with!

Filtering the files — 3

Remove the two named files.

```
def something_useful(arguments):
    print 'CWD:', os.getcwd()
    files = os.listdir('.')
    files.sort()

    if arguments.output_file in files:
        files.remove(arguments.output_file)
    if arguments.params_file in files:
        files.remove(arguments.params_file)
```

Test to see if they exist

UCS

We need to do a simple test. If the two files are in the list, remove them. If they are not, do nothing.

So now our script prints a list of every file in the directory except for those two if they exist already.

Matching against a pattern

fnmatch module — **file**name **match**ing module

`fnmatch.filter()` — The useful function

```
files = fnmatch.filter(
    files,
    arguments.input_pattern
    )
…
```

Filtered list out

Unfiltered list in

The filter

Now our files list has to have its final pruning. We want to consider only those files that match the input pattern. To do this we will call upon a further module, `fnmatch` (“**f**ile**n**ame **match**ing”).

Note that we can import that module purely within the `something_useful()` function. It could be done globally as well; it makes no difference.

The `fnmatch` module provides us with one function useful for our purposes: `fnmatch.filter()`.

This function takes a list of file names and a wildcard pattern. It returns the list of names that match the pattern. This is what we will use.

Example

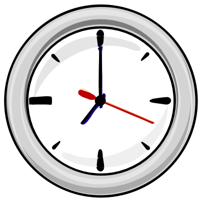
```
>>> import fnmatch

>>> files = ['input1.dat', 'input2.dat', 'notes']

>>> files = fnmatch.filter(files, '*.dat')

>>> files
['input1.dat', 'input2.dat']

>>>
```

Exercise 2

Complete `exercise2.py`
All edits should be in `something_useful()`

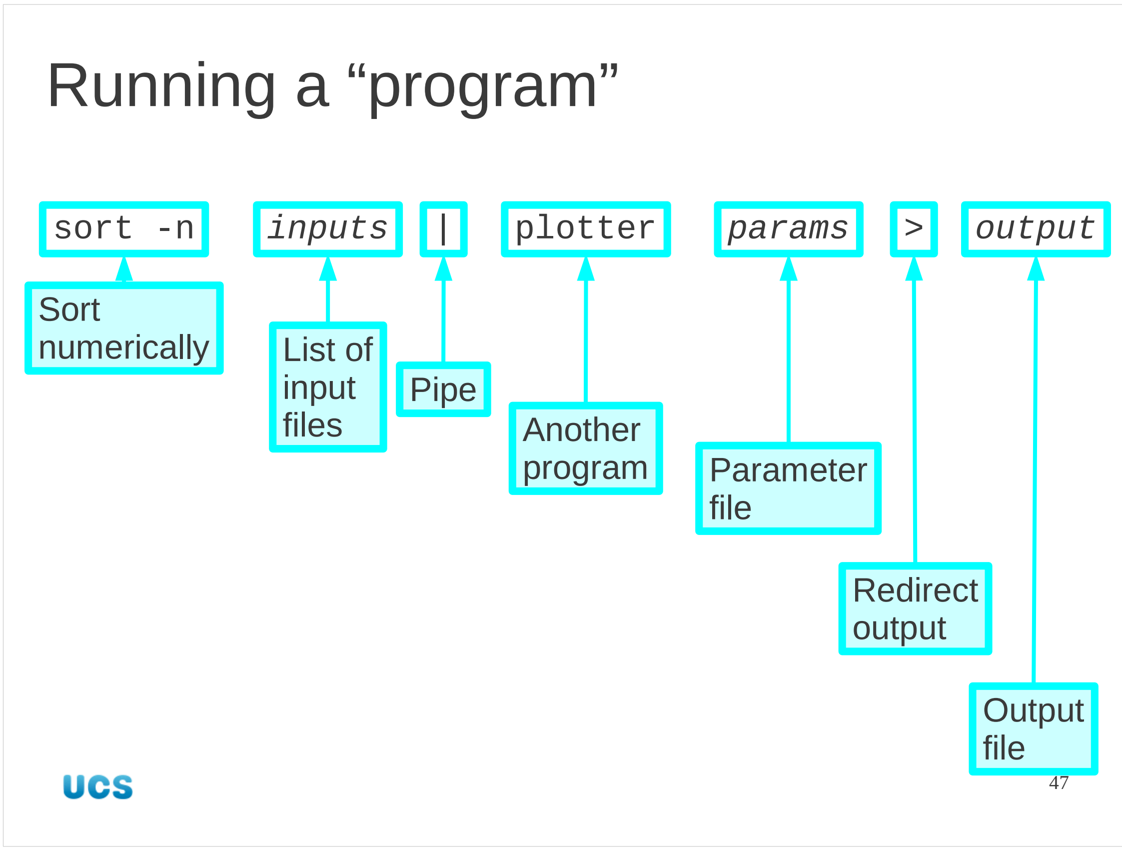
1. List the directory contents.
2. Remove the output and parameter files.
3. Filter the files against the input pattern.
4. Sort the resulting list.
5. Print the sorted list.

Ten minutes



So, it's time to put all that to use. The script `exercise2.py` has within it all the option parsing and the directory traversal code. Its `something_useful()` function is incomplete, though, and you have to get it working.

Your function should get the list of directory contents, remove the output and parameter files if they are present and then filter it against the input pattern. The final list (which is all or input files) should be sorted.



Now let's look at the last bit of our work: we have to get our script to call another program.

The slide shows the shell version of what we need to achieve. We will use the `sort` program to combine our input files and pipe its output into another program which plots the data to the output file.

If we didn't have the piping this would be a lot simpler as we will soon see, but this lets us see the full power of Python's facilities.

The `plotter` program is not a standard application but a made up one for this course. It can be found on the PWF in `/ux/Lessons/PythonOS/plotter`.

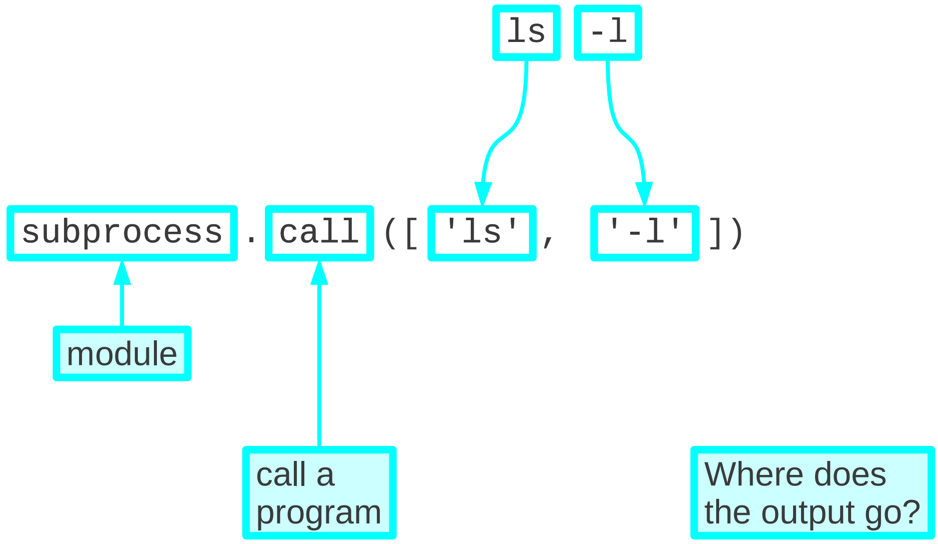
Running programs

“subprocess” module

```
import subprocess
```


The module needed to run external programs is called “`subprocess`”.

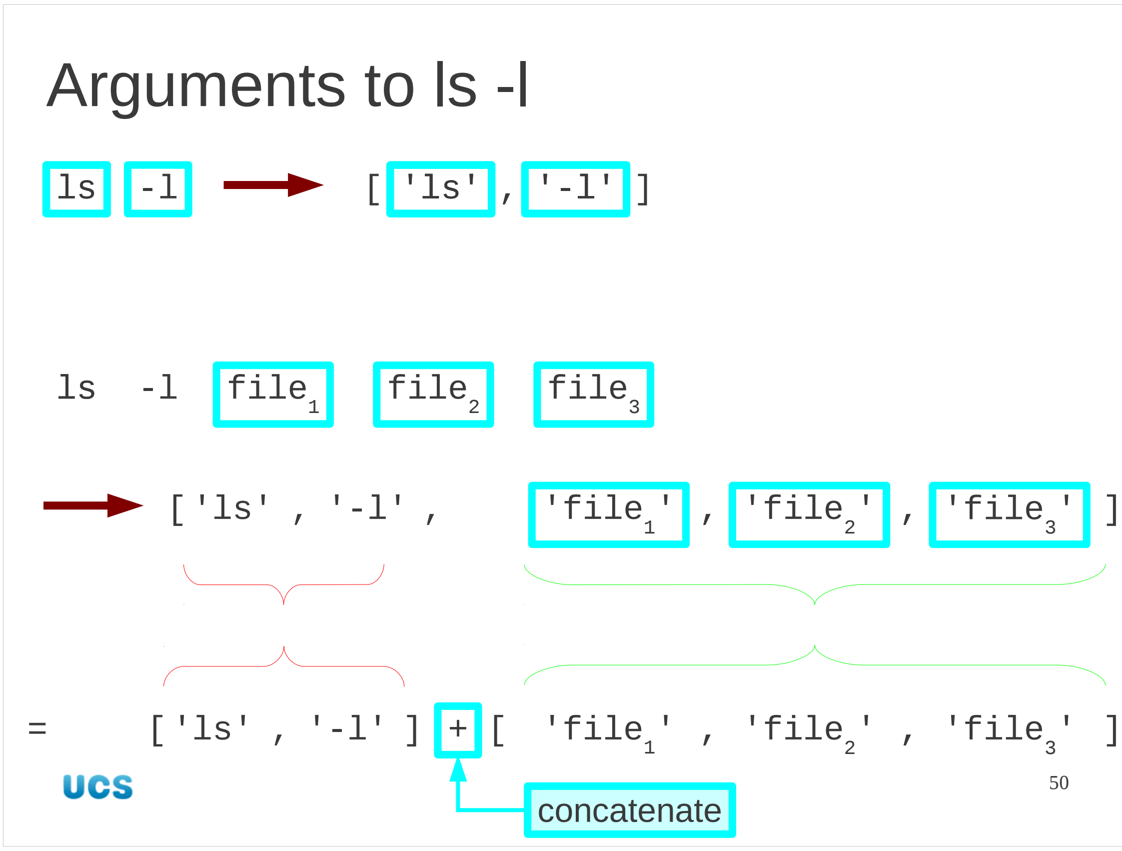
Simplest case



Suppose we had no piping. Suppose all we wanted to do was to call a single program.

In this case the module has a function called `subprocess.call()` which does exactly that.

This function takes, in its simplest form, a list of strings. This list is the argument string of a command with the lead item (number zero) being the command to run. So if we want to run “`ls -l`” in each directory that this is the Python to run.



Of course, “`ls -l`” can take a list of specific files to look at as additional arguments. The use of lists for the command line passed to `subprocess.call()` leads to a very useful way to treat the command (“`ls`”) with its options (“`-l`”) and the file name argument (“`file`$_1$ `file`$_2$ `file`$_3$”).

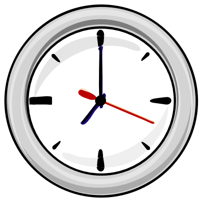
The list passed to `subprocess.call()` has all of these items but we can build it by concatenating two separate lists, one for the command and its options (“`ls -l`”) and the other for the arguments (“`file`$_1$ `file`$_2$ `file`$_3$”).

Exercise 3

1. Copy `exercise2.py` → `exercise3.py`
2. `import subprocess`
3. `print files` →

   `subprocess.call(['ls','-l'] + files)`

Five minutes



UCS

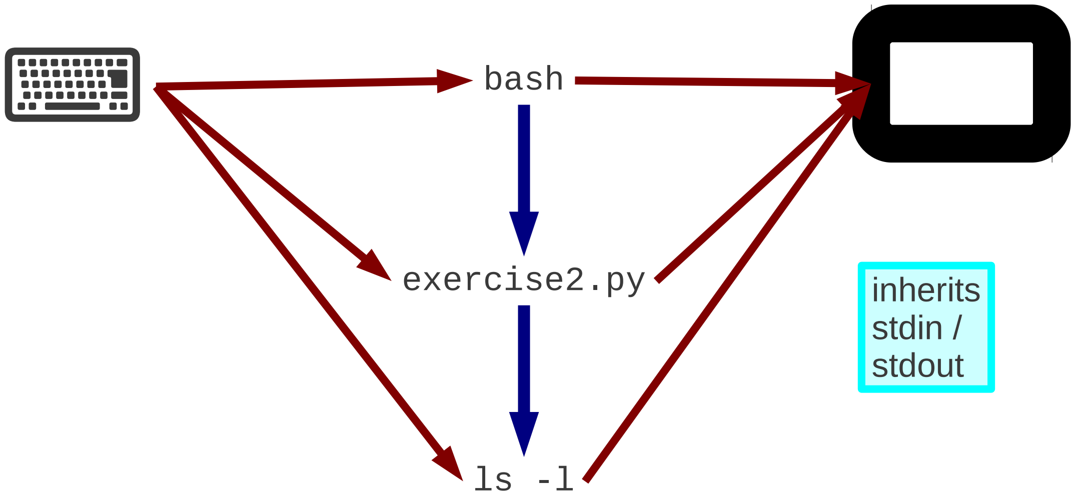
For the next exercise, do just that, but instead of running “`ls -l`”, run “`ls -l` *`files`*” where the list of files is the filtered set of input files we will be processing later. To do this, simply add the list of file names to the “`ls -l`” list.

So `['ls', '-l'] + ['input1.dat', 'input2.dat']` becomes
`['ls', '-l', 'input1.dat', 'input2.dat']`
which corresponds to the command

```
$ ls -l input1.dat input2.dat
```

in the shell.

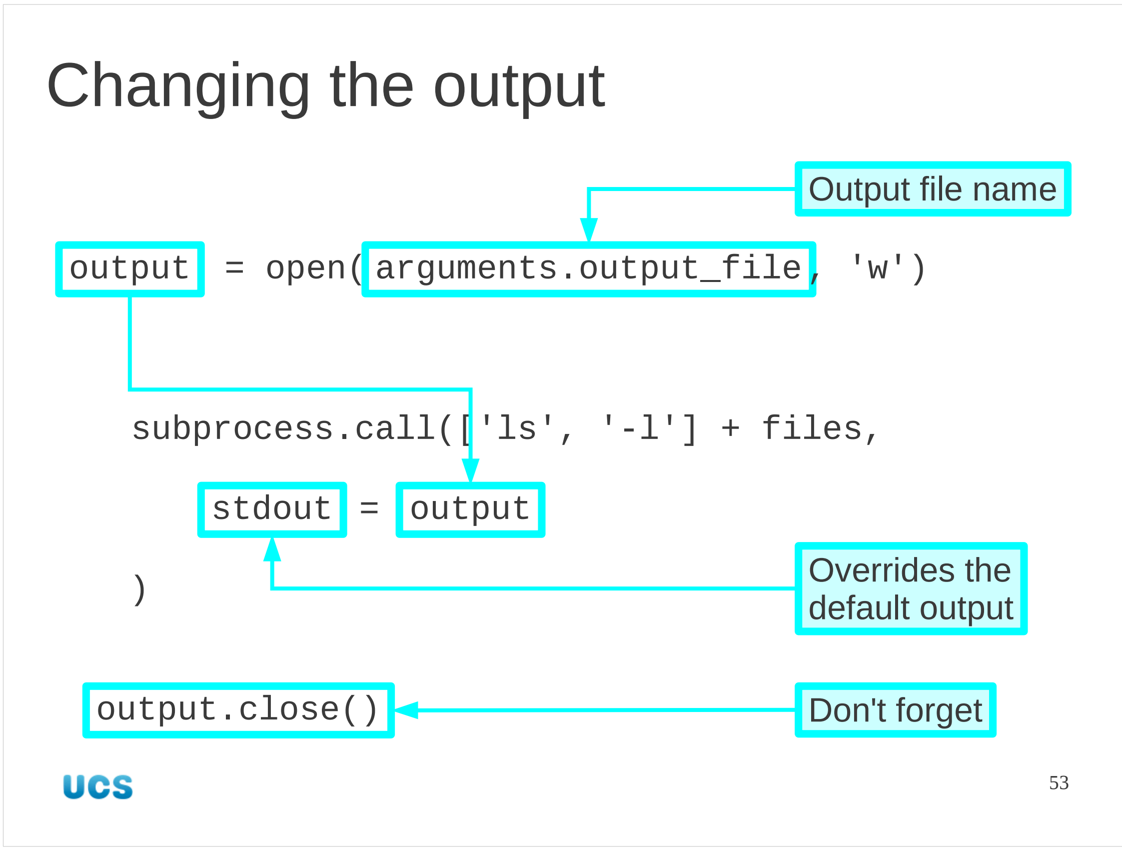
Input & output?



Ultimately we want to put our output in a file in each directory called `output.dat` or some other name passed on the command line. Our script, `exercise3.py`, outputs to the screen.

It does this because the shell starts with getting input from the keyboard and sending output to the terminal window. It launched the `exercise3.py` script and that script's process inherits exactly the same input and output. The script launched “`ls -l`” and it inherits the same again.

Standard input and output are inherited.



We can change the output (and input) of a program launched by `subprocess.call()`.

First we have to open the output file name. The procedure works with real file objects, not just file names.

Then we set the optional parameter “stdout” to be this file object.

The program runs and sends its output to that file rather than the screen.

When we are finished we should close our file. Some programs close their own output and others don't. There is no fault in calling the `close()` method on a file object more than once and it pays to be careful.

Exercise 4

1. Copy `exercise3.py` → `exercise4.py`

2. Open the output file for writing

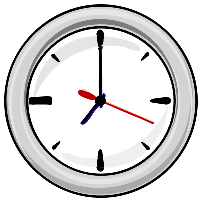
   `options.output_file`

3. Set the standard output for

   `subprocess.call()`

4. Check the output files.

Five minutes



In this exercise you need to set the output to be a file rather than the screen. This is really just putting the previous slide's content to use.

`subprocess.call()` may be enough for you

```
subprocess.call(
    [program, options, arguments],
    stdin  = input,
    stdout = output
    )
```

Single program

Input from file

Output to file

Now, this may be all you need. While we are going to move on from `subprocess.call()` we are going to extend our use in ways that you don't need.

If you want to launch a single program and the input and output for that program are files, then you have everything you need.

Our “program”

```
sort -n inputs    |    plotter params > output
```

```
call(['sort','-n'] + files)
```

```
call(['plotter',arguments.params_file],
    stdout = output)
```


But our task is more complex. We want to run two programs, hooked together by a pipe. There are two problems we need to overcome and we will address them one after the other.

First problem: timing

Pipe: programs run at the same time

`sort -n` *`inputs`* `|` `plotter` *`params`* `>` *`output`*

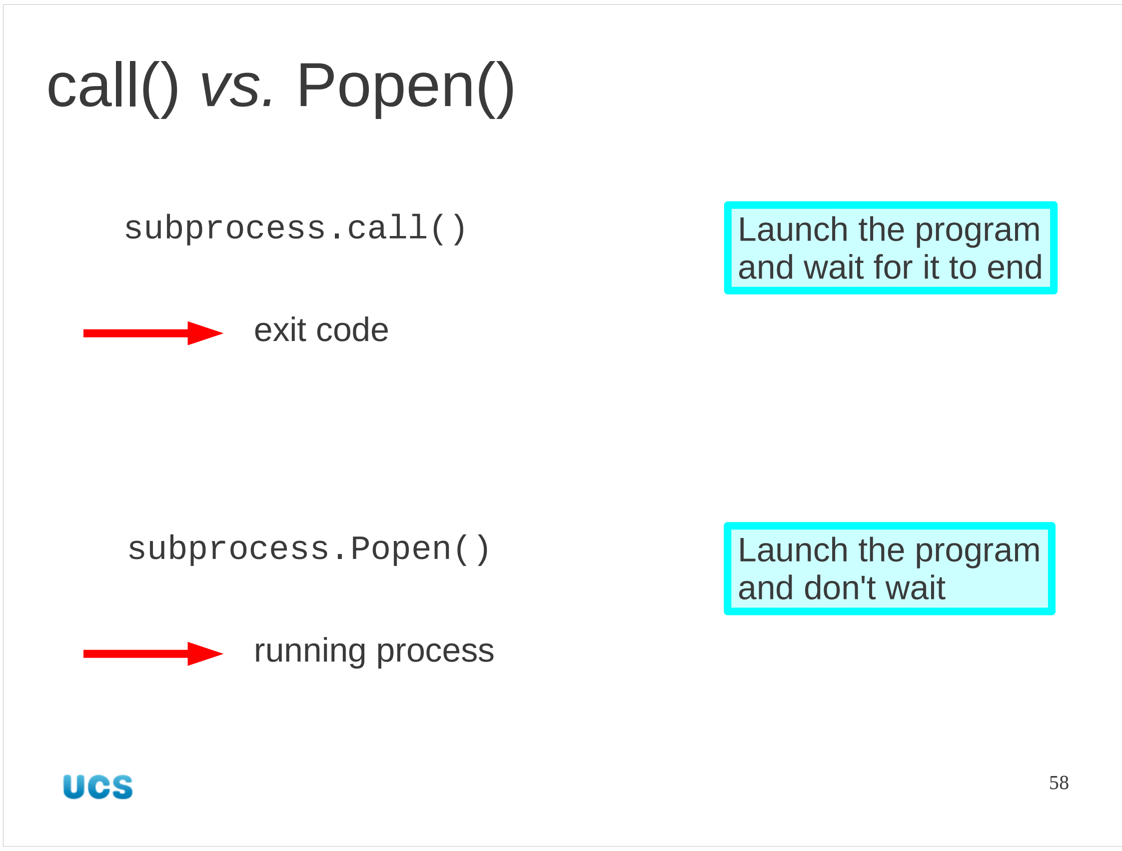
```
call(['sort','-n'] + files)
```

Call: one after the other

```
call(['plotter',arguments.params_file],
    stdout = output)
```

UCS

When two commands are piped together they both run simultaneously. The `call()` function we are using runs its program and only completes when the program is run. We can't run two `call()`s side-by-side.



We need to move away from the simplicity of `call()` to something rather more complicated.

The `Popen()` (n.b.: capital "P") function launches a command but doesn't wait for it to finish. It is the Python equivalent of running a command in the background in the shell.

Its arguments are exactly the same as for `call()` but instead of returning the command's return code, which is what `call()` does, it returns an object corresponding to the running process.

No waiting!

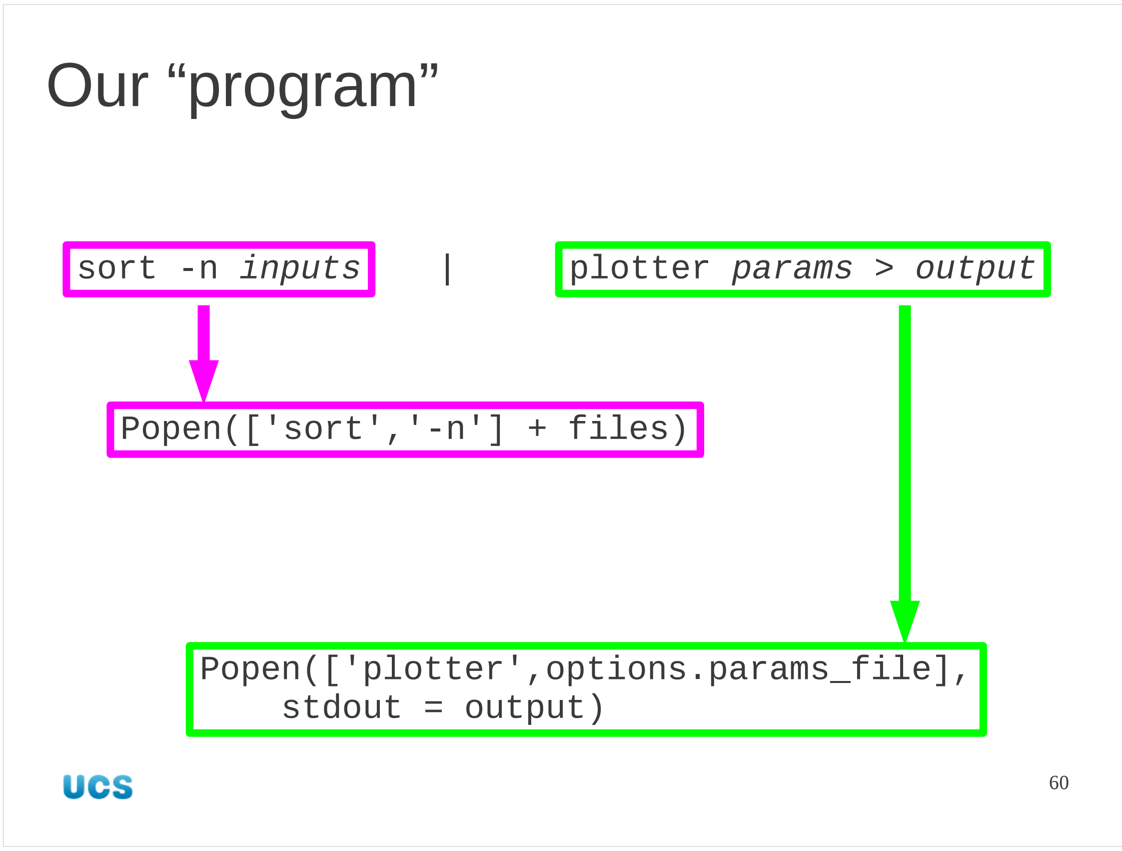
```
$ ./process2.py alpha/ beta/ gamma/
CWD: /home/rjd4/Courses/PythonOS/alpha
CWD: /home/rjd4/Courses/PythonOS/beta
-rw-r--r-- 1 rjd4 … input1.dat
-rw-r--r-- 1 rjd4 … input2.dat
-rw-r--r-- 1 rjd4 … input3.dat
-rw-r--r-- 1 rjd4 … input4.dat
CWD: /home/rjd4/Courses/PythonOS/gamma
-rw-r--r-- 1 rjd4 … input1.dat
-rw-r--r-- 1 rjd4 … input2.dat
-rw-r--r-- 1 rjd4 … input3.dat
-rw-r--r-- 1 rjd4 … variants.dat
-rw-r--r-- 1 rjd4 … input1.dat
-rw-r--r-- 1 rjd4 … input2.dat
-rw-r--r-- 1 rjd4 … input3.dat
-rw-r--r-- 1 rjd4 … input4.dat
```

All three runs simultaneous!

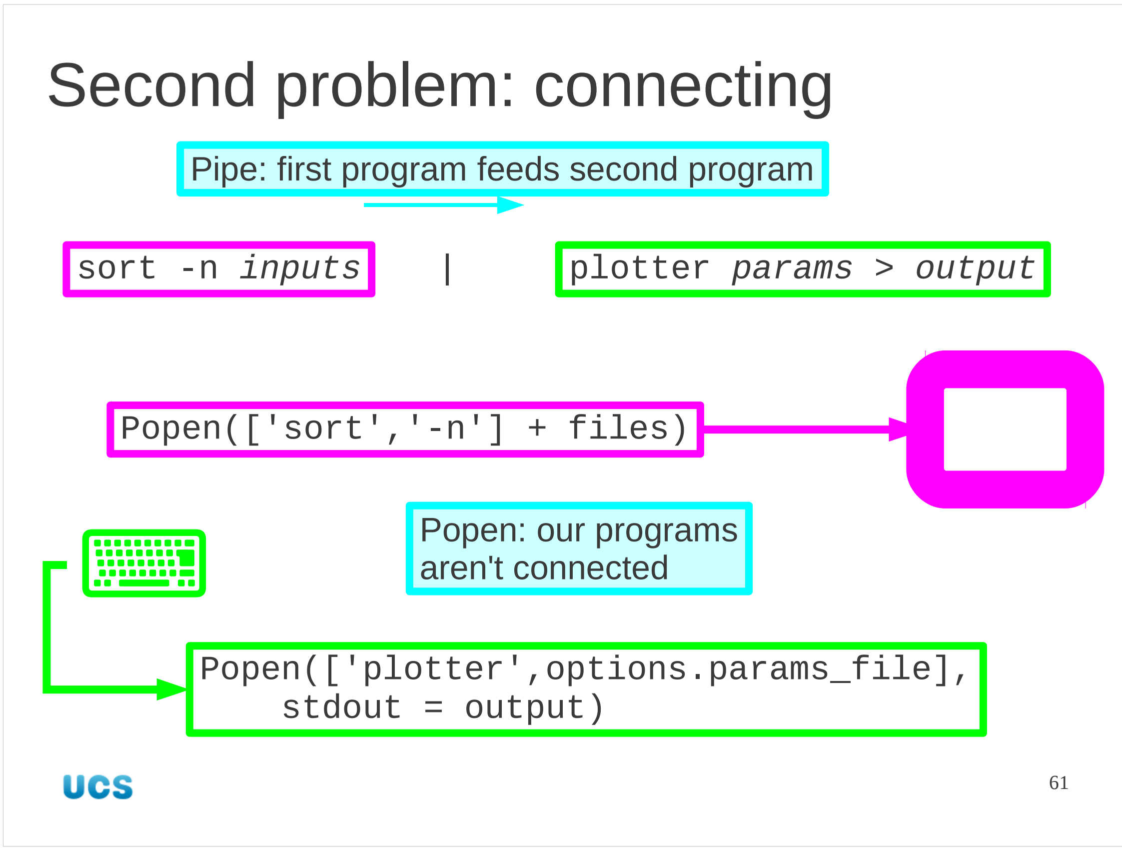
UCS

The script `process2.py` has this simple change made for the "`ls -l`" example. We notice immediately that we would get confusion if the various commands running simultaneously all have the same output. They get mixed up.

We must specify distinct `stdout` parameters if we are going to use `Popen()`. But that's all right; we are.



So we will tackle the timing issue by using `Popen()`. There is still one other problem with reproducing the functionality of a pipe.



We know how to divert standard output to a file but we don't know how to send it to another `Popen()`ed process. We can't hook up the commands in our pipeline.

Connectivity

Need to refer to the running process

```
p1  =   subprocess.Popen(
            ['sort','-n'] + files,
            stdout = subprocess.PIPE,
            )
```

Prepare stdout to be passed on

```
p2  =    subprocess.Popen(
             ['plotter',options.params_file],
             stdin = p1.stdout,
             stdout = output,
             )
```

p2's stdin is p1's stdout

Connecting two running processes requires three things.

First, we need to be able to refer to the process of the command whose output we want to pass on. So we don't just call `Popen()`, we assign its output to a variable, `p1` say.

Next we tell it that its output is going to be passed on. We do this by assigning a special value to its `stdout` parameter: `subprocess.PIPE`.

Finally, in the subprocess to which we are going to pass the data we set its standard input parameter, “`stdin`” the `stdout` member of the earlier process object, `p1.stdout`. This is the “make its output my input” step.

Putting it all together

```
def something_useful(options):
    …

    sort_proc = subprocess.Popen(
        ['sort', '-n'] + files,
        stdout=subprocess.PIPE
        )

    plot_proc = subprocess.Popen(
        ['plotter', options.params_file],
        stdin = sort_proc.stdout,
        stdout = output
        )
```

Closing output?

So now we put it all together. We remove the trivial “`ls -l`” instruction in `something_useful()` and put in two `subprocess.Popen()` instructions.

Note that we can't close the output until we know the processes are finished with it!

One last nicety

```
def something_useful(options):
    …
    sort_proc.wait()
    plot_proc.wait()
```

Don't move on until both processes have finished

```
    output.close()
```

Then close output!

There's one last tweak we need to make right at the end of the function. We have launched the processes in the background, running in parallel. We ought to wait for them to finish before we move on.

Strictly we only need to wait for them before quitting the script altogether. It's easier for us to wait for them in pairs. So we add these two lines at the very end of the function.

If we wanted to check the return codes from these commands (and we ought to) they would be the returned values from these two `wait()` methods:

```
sort_rc = sort_proc.wait()
plot_rc = plot_proc.wait()
```

Once we have both processes finished we can safely close the output file. Note that the closing must not happen before the waiting. If we close the output file prematurely the plotting process may get cut off before it has finished writing its output to the file.

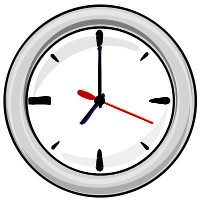
Exercise 5

1. Copy `exercise4.py` → `exercise5.py`

1. Launch sort process
2. Launch plotter process

   `/ux/Lessons/PythonOS/plotter`

3. Wait for processes to finish



Fifteen minutes

So let's do that.

The `exercise5.py` script is another script with a few critical lines missing and replaced by comments.

Please note that the “`plotter`” command I am using is not a standard command . You will need to use its full path name in the first item in the list:

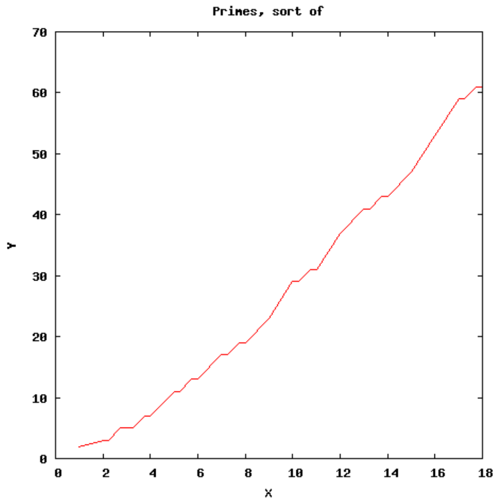
`/ux/Lessons/PythonOS/PythonOS/plotter`

Viewing the output

$ **./exercise5.py alpha beta gamma**

Output file is a graph

$ **eog alpha/output.dat**



The output file is a graph, as might have been guessed from the name of the “plotter” program. If you would like to see your handicraft use the eog program (“**e**ye **o**f **g**nome; don't ask).

And that's it!

`argparse`	Command line argument parsing
`os`	File system navigation
`fnmatch`	File name wild cards
`subprocess`	Launching external programs

`scientific-computing@ucs.cam.ac.uk`

`www.training.cam.ac.uk/ucs/course/ucs-pythonopsys`