Chapter 5

Basic Biological Computing in Python

> Science is what we understand well enough to explain to a computer. Art is everything else we do
>
> —*Donald Knuth*

Firstly, chapter 1's UNIX question?

```
find . -type f -exec ls -s {} \; | sort -n | head -10
```

What is the command doing? How has it been built (explain the components)?

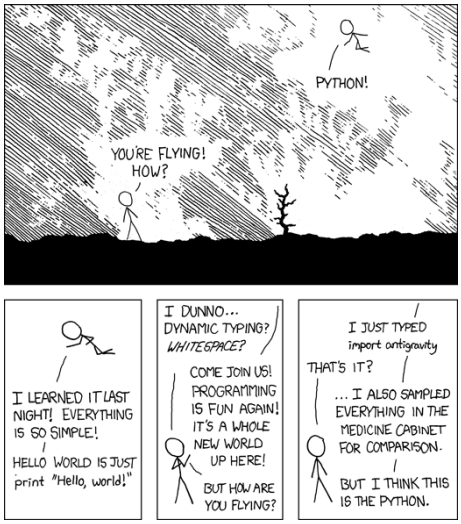
5.1 Outline of the `python` module

The `python` module is geared towards teaching you scientific programming in biology using this modern, and for good reason, immensely popular language. The components of this module across all the chapters (Basic, Advanced, Additional topics) are:

- Basics of `python`
- How to write and run `python` code
- Understand and implement "control flows"
- Learning to use the `ipython` environment
- Writing, debugging, using, and testing `python` functions
- Learning efficient numerical programming in `python`
- Using regular expressions in `python`
- Introduction to certain particularly useful `python` packages
- Using `python` for building and modifying databases
- Using `python` to run other "stuff" and to patch together data analysis and/or numerical simulation work flows

5.2 Why `python`?

`python` was designed with readability and re-usability in mind. Time taken by programming +



www.xkcd.com

Figure 5.1: Is `python` the most common answer to your daily programming needs? Possibly!

	Fortran gcc 5.1.1	Julia 0.4.0	Python 3.4.3	R 3.2.2	Matlab R2015b	Octave 4.0.0	Mathematica 10.2.0	JavaScript V8 3.28.71.19	Go go1.5	LuaJIT gsl-shell 2.3.1	Java 1.8.0_45
fib	0.70	2.11	77.76	533.52	26.89	9324.35	118.53	3.36	1.86	1.71	1.21
parse_int	5.05	1.45	17.02	45.73	802.52	9581.44	15.02	6.06	1.20	5.77	3.35
quicksort	1.31	1.15	32.89	264.54	4.92	1866.01	43.23	2.70	1.29	2.03	2.60
mandel	0.81	0.79	15.32	53.16	7.58	451.81	5.13	0.66	1.11	0.67	1.35
pi_sum	1.00	1.00	21.99	9.56	1.00	299.31	1.69	1.01	1.00	1.00	1.00
rand_mat_stat	1.45	1.66	17.93	14.56	14.52	30.93	5.95	2.30	2.96	3.27	3.92
rand_mat_mul	3.48	1.02	1.14	1.57	1.12	1.12	1.30	15.07	1.42	1.16	2.36

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

http://julialang.org/

Figure 5.2: `python` is pretty fast!

debugging + running is likely to be relatively lower in `python` than less intuitive or cluttered languages (e.g., `FORTRAN`, `perl`). It is a pretty good solution if you want to easily write readable code that is also reasonably efficient (computationally speaking).

5.2.1 The Zen of python

Open a terminal and type

```
$ python -c "import this"
```

5.3 Installing `python`

We will use 2.7.x, not 3.x (you can use 3.x later, if you want)

Your Ubuntu distribution needs `python`, so it will already be installed. However, let's install the interactive python shell `ipython` which we will soon use.

⋆ On Ubuntu/Linux, open a terminal (ctrl+alt+t) and type:

```
$ sudo apt-get install ipython python-scipy python-matplotlib
```

Tip

In Linux, you can easily install python packages that come with the standard python distribution using the usual `sudo apt-get install python-packagename`

5.4 Getting started with `python`

Open a terminal (`ctrl+alt+t`) and type `python` (or use the terminal that you just used to install `ipython`). Then, try the following:

```
>>> 2 + 2 # Summation; note that comments start with #
4

>>> 2 * 2 # Multiplication
4

>>> 2 / 2 # Integer division
1

>>> 2 / 2.0 # "Float" division, note the output is float
1.0

>>> 2 / 2.
1.0

>>> 2 > 3
False

>>> 2 >= 2
True
```

What does “float” mean in the above comment? Why is it necessary to specify this in Python (not necssary in Python 3.x)? You will inevitably run into some such jargon in this chapter. The main ones you need to know are (you will learn more about these along the way):

Workspace	The state of the "environment" of your current python *session*, including all variables, functions, objects, etc.
Variable	A named number, text string, boolean (`True` or `False`), or data structure that can change (more on variable and data types later)
Function	A computer procedure or routine that returns some value, and which can be used again and again
Module	*Variables* and *functions* packaged into a single entity that does something useful
Class	Also, variables and functions packaged into a single entity that does something useful, but unlike modules, you can spawn many copies of a class within a python session or program
Object	A particular instance of a class (every object belongs to a class) that is created in a session and eventually destroyed; pretty much everything that is not a function in your workspace is an object in python!

This Module vs. Class vs. Object business is confusing. These constructs are created to make an (object-oriented) programming language like `python` more flexible and user friendly (though it might not seem so to you currently!). In practice, at least for your current purposes, you will not build python classes yourself much, typically working with modules. More on all this later. Also, have a look at `https://learnpythonthehardway.org/book/ex40.html`

5.4.1 `ipython`

We will now immediately switch to the `i`nteractive `python` shell, `ipython` that you installed above.

OK, now let's continue learning python using `ipython`.

- ⋆ Type `ctrl+D` in the terminal at the python prompt: this will exit you from the python shell and you will see the bash prompt again.
- ⋆ Now type `ipython`

You should now see (after some text):

```
In [ ]:
```

(I have deleted the prompt numbering `[1]`, `[2]`, etc to avoid confusion). This is the `i`nteractive `python` shell (or, "`ipython`"). This shell has many advantages over the bare-bones, non-interactive python shell with the >>> prompt. For example, as in the bash shell, `TAB` leads to auto-completion of a command or file name (try it).

5.4.2 Magic commands

IPython also has "magic commands" (start with %; e.g., `%run`). Some useful magic commands:

`%who`	Shows current namespace (all variables, modules and functions)
`%whos`	Also display the type of each variable; typing `%whos function` only displays functions etc.
`%pwd`	Print working directory
`%history`	Print recent commands

Try any of these now!

5.4.3 Determining an object's type

Another useful IPython feature is the question mark, which can be used to find what a particular Python object is, including variables you created. For example, try:

```
In [1]: a = 1

In [2]: ?a
Type:        int
String form: 1
Docstring:
int(x=0) -> int or long
int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is floating point, the conversion truncates towards zero.
If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or
Unicode object representing an integer literal in the given base.  The
literal can be preceded by '+' or '-' and be surrounded by whitespace.
The base defaults to 10.  Valid bases are 0 and 2-36.  Base 0 means to
interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

5.4.4 Configuring `ipython`

You can configure ipython's environment and behavior by editing the `ipython_config.py` file:

```
$ geany ~/.config/ipython/profile_default/ipython_config.py &
```

This file does not inititally exist, but you can create it by running `ipython profile create` in a bash terminal (try it now).

Now you can configure ipython. For example, If you don't like the blue `ipython` prompt, you can type `%colors linux` (once inside the shell). If you want to make this color the default, then edit `ipython_config.py` — search for "Set the color scheme" in the file.

5.5 Python variables

Now, let's continue our python intro. We will first learn about the `python` variable types that were mentioned above. The types are:

```
In [ ]: a = 2 #integer

In [ ]: ?a
Type:        int
String form: 2
Docstring:
int(x=0) -> int or long
int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is floating point, the conversion truncates towards zero.
If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or
Unicode object representing an integer literal in the given base.  The
literal can be preceded by '+' or '-' and be surrounded by whitespace.
The base defaults to 10.  Valid bases are 0 and 2-36.  Base 0 means to
interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4

In [ ]: a = 2. #Float

In [ ]: ?a
Type:        float
String form: 2.0
Docstring:
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.

In [ ]: a = "Two" #String

In [ ]: ?a
Type:        str
String form: Two
Length:      3
Docstring:
str(object='') -> string

Return a nice string representation of the object.
If the argument is a string, the return value is the same object.

In [10]: a = True #Boolean

In [11]: ?a
Type:        bool
String form: True
Docstring:
bool(x) -> bool

Returns True when the argument x is true, False otherwise.
The builtins True and False are the only two instances of the class bool.
The class bool is a subclass of the class int, and cannot be subclassed.
```

Thus, `python` has integer, float (real numbers, with different precision levels) and string variables.

5.5.1 `python` operators

Here are are the operators in python that you can use on variables:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`**`	Power
`%`	Modulo
`//`	Integer division
`==`	Equals
`!=`	Differs
`>`	Greater
`>=`	Greater or equal
`&, and`	Logical and
`|, or`	Logical or
`!, not`	Logical not

5.5.2 Assigning and manipulating variables

```
 In []: 2 == 2
Out []: True

 In []: 2 != 2
Out []: False

 In []: 3 / 2
Out []: 1

 In []: 3 / 2.
Out []: 1.5

 In []: 'hola, ' + 'mi llamo Samraat' #why not two languages at the same
time?!
Out []: 'hola, mi llamo Samraat'

In []: x = 5

 In []: x + 3
Out []: 8

 In []: y = 8

 In []: x + y
Out []: 13

 In []: x = 'My string'

 In []: x + ' now has more stuff'
Out []: 'My string now has more stuff'

 In []: x + y
Out []: TypeError: cannot concatenate 'str' and 'int' objects
```

OK, so concatenating string and numeric (integer in this case) variables doesn't work. No problem, we can convert from one type to another:

```
 In []: x + str(y)
Out []: 'My string8'

 In []: z = '88'

 In []: x + z
Out []: 'My string88'

 In []: y + int(z)
Out []: 96
```

Tip

In `python`, the type of a variable is determined when the program or command is running (dynamic typing) (like `R`, unlike `C` or `FORTRAN`). This is convenient, but can make programs slow. <More on efficient computing later!

5.6 `python` data types and data structures

`python` number or string variables (or both) can be stored and manipulated in:

- **List**: most versatile, can contain compound data, "mutable", enclosed in brackets, [ ]
- **Tuple**: like a list, but "immutable" — like a read only list, enclosed in parentheses, ( )
- **Dictionary**: a kind of "hash table" of key-value pairs enclosed by curly braces, { } — key can be number or string, values can be any object! (well OK, a python object)
- **numpy arrays**: Fast, compact, convenient for numerical computing — more on this later!

5.6.1 Lists

```
 In []: MyList = [3,2.44,'green',True]

 In []: MyList[1]
Out []: 2.44

 In []: MyList[0] # NOTE: FIRST ELEMENT -> 0
Out []: 3

 In []: MyList[4]
Out []: IndexError: list index out of range

 In []: MyList[2] = 'blue'

 In []: MyList
Out []: [3, 2.44, 'blue', True]

 In []: MyList[0] = 'blue'

 In []: MyList
Out []: ['blue', 2.44, 'blue', True]
```

```
 In []: MyList.append('a new item') # NOTE: ".append"!

 In []: MyList
Out []: ['blue', 2.44, 'blue', True, 'a new item']

 In []: MyList.sort() # NOTE: suffix a ".", hit tab, and wonder!

 In []: MyList
Out []: [True, 2.44, 'a new item', 'blue', 'blue']
```

In the above commands, notice that `python` "indexing" starts at 0, not 1!

5.6.2 Tuples

```
 In []: FoodWeb=[('a','b'),('a','c'),('b','c'),('c','c')]

 In []: FoodWeb[0]
Out []: ('a', 'b')

 In []: FoodWeb[0][0]
Out []: 'a'

 In []: FoodWeb[0][0] = "bbb"  # NOTE: tuples are "immutable"
      TypeError: 'tuple' object does not support item assignment

 In []: FoodWeb[0] = ("bbb","ccc")

 In []: FoodWeb[0]
Out []: ('bbb', 'ccc')
```

Note that tuples are "immutable"; that is, a particular pair or sequence of strings or numbers cannot be modified after it is created.

In the above example, why assign these food web data to a list of tuples and not a list of lists? — because we want to maintain the species associations, no matter what — they are sacrosanct!

Tuples contain immutable sequences, but you can append to them:

```
 In []: a = (1, 2, [])

 In []: a[2].append(1000)

 In []: a
Out []: (1, 2, [1000])
```

5.6.3 Sets

You can convert a list to an immutable "set" — an unordered collection with no duplicate elements. Once you create a set you can perform set operations on it:

```
 In []: a = [5,6,7,7,7,8,9,9]

 In []: b = set(a)

 In []: b
```

```
Out []: set([8, 9, 5, 6, 7])

 In []: c = set([3,4,5,6])

 In []: b & c
Out []: set([5, 6])

 In []: b | c
Out []: set([3, 4, 5, 6, 7, 8, 9])

 In []: list(b | c) # set to list
Out []: [3, 4, 5, 6, 7, 8, 9]
```

The key set operations in `python` are:

`a - b`	a.difference(b)
`a <= b`	a.issubset(b)
`a >= b`	b.issubset(a)
`a & b`	a.intersection(b)
`a | b`	a.union(b)

5.6.4 Dictionaries

A set of values (any `python` object) indexed by keys (string or number), a bit like `R` lists.

```
In []: GenomeSize = {'Homo sapiens': 3200.0, 'Escherichia coli': 4.6,
'Arabidopsis thaliana': 157.0}

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
  'Escherichia coli': 4.6,
  'Homo sapiens': 3200.0}

 In []: GenomeSize['Arabidopsis thaliana']
Out []: 157.0

 In []: GenomeSize['Saccharomyces cerevisiae'] = 12.1

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
'Escherichia coli': 4.6,
'Homo sapiens': 3200.0,
'Saccharomyces cerevisiae': 12.1}

 In []: GenomeSize['Escherichia coli'] = 4.6  # ALREADY IN DICTIONARY!

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
 'Homo sapiens': 3200.0,
 'Saccharomyces cerevisiae': 12.1}

 In []: GenomeSize['Homo sapiens'] = 3201.1

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
```

```
 'Homo sapiens': 3201.1,
 'Saccharomyces cerevisiae': 12.1}
```

So, in summary,

- If your elements/data are unordered and indexed by numbers use **lists**
- If they are ordered sequences use a **tuple**
- If you want to perform set operations on them, use a **set**
- If they are unordered and indexed by keys (e.g., names), use a **dictionary**

But why not use dictionaries for everything? – because it can slow down your code!

5.6.5 Copying mutable objects

Copying mutable objects can be tricky. Try this:

```
# First, try this:
a = [1, 2, 3]
b = a # you are merely creating a new "tag" (b)
a.append(4)
print b
# this will print [1, 2, 3, 4]!!

# Now, try:
a = [1, 2, 3]
b = a[:]  # This is a "shallow" copy
a.append(4)
print b
# this will print [1, 2, 3].

# What about more complex lists?
a = [[1, 2], [3, 4]]
b = a[:]
a[0][1] = 22 # Note how I accessed this 2D list
print b
# this will print [[1, 22], [3, 4]]

# the solution is to do a "deep" copy:
import copy

a = [[1, 2], [3, 4]]
b = copy.deepcopy(a)
a[0][1] = 22
print b
# this will print [[1, 2], [3, 4]]
```

So, you need to employ `deepcopy` to really copy an existing object or variable and assign a new name to the copy.

5.6.6 `python` with strings

One of the things that makes python so useful and versatile, is that it has a powerful set of inbuilt commands to perform string manipulations. For example, try these:

```
s = " this is a string "
len(s)
# length of s -> 18

print s.replace(" ","-")
# Substitute spaces " " with dashes -> -this-is-a-string-

print s.find("s")
# First occurrence of s -> 4 (start at 0)

print s.count("s")
# Count the number of "s" -> 3

t = s.split()
print t
# Split the string using spaces and make
# a list -> ['this', 'is', 'a', 'string']

t = s.split(" is ")
print t
# Split the string using " is " and make
# a list -> [' this', 'a string ']

t = s.strip()
print t
# remove trailing spaces

print s.upper()
# ' THIS IS A STRING '

'WORD'.lower()
# 'word'
```

5.7 Writing `python` code

Now let's learn to write and run python code from a `*.py` file. But first, some some guidelines for good code-writing practices (see `python.org/dev/peps/pep-0008/`):

- Wrap lines to be <80 characters long. You can use parentheses () or signal that the line continues using a "backslash" \
- Use either 4 spaces for indentation or tabs, but not both! (I use tabs!)
- Separate functions using a blank line
- When possible, write comments on separate lines

Make sure you have chosen a particular indent type (space or tab) in `geany` (or whatever IDE you are using) — indentation is all-important in `python`. Furthermore,

- Use "docstrings" to **document how to use the code**, and **comments to explain why and how the code works**
- Naming conventions (bit of a mess, you'll learn as you go!):
  - `_internal_global_variable` (for use inside module only)
  - `a_variable`
  - `SOME_CONSTANT`
  - `a_function`
  - Never call a variable `l` or `O` or `o`
    why not? – you are likely to confuse it with `1` or `0`!

- Use spaces around operators and after commas:
  `a = func(x, y) + other(3, 4)`

5.8 python Input/Output

Let's look at importing and exporting data. Make a textfile called `test.txt` in `Week2/Sandbox/` with the following content (including the empty lines):

```
First Line
Second Line

Third Line

Fourth Line
```

Then, type the following in `Week2/Code/basic_io.py` (note the indentation!):

```
############################
# FILE INPUT
############################
# Open a file for reading
f = open('../Sandbox/test.txt', 'r')
# use "implicit" for loop:
# if the object is a file, python will cycle over lines
for line in f:
    print line, # the "," prevents adding a new line

# close the file
f.close()

# Same example, skip blank lines
f = open('../Sandbox/test.txt', 'r')
for line in f:
    if len(line.strip()) > 0:
        print line,

f.close()

############################
# FILE OUTPUT
############################
# Save the elements of a list to a file
list_to_save = range(100)

f = open('../Sandbox/testout.txt','w')
for i in list_to_save:
    f.write(str(i) + '\n') ## Add a new line at the end

f.close()

############################
# STORING OBJECTS
############################
# To save an object (even complex) for later use
my_dictionary = {"a key": 10, "another key": 11}

import pickle

f = open('../Sandbox/testp.p','wb') ## note the b: accept binary files
pickle.dump(my_dictionary, f)
f.close()
```

```
## Load the data again
f = open('../Sandbox/testp.p','rb')
another_dictionary = pickle.load(f)
f.close()

print another_dictionary
```

Note the following:

- The `for line in f` is an implicit loop — implicit because stating the range of things in `f` to loop over in this way allows python to handle any kind of objects to loop thorugh. For example, if `f` was an array of numbers 1 to 10, it would loop thorugh them; if `f` is a file, as in the case of the script above, it will loop through the lines in the file.
- `is len(line.strip()) > 0` checks if the line is empty. Try `?` to see what `.strip()` does.

The `csv` package makes it easy to manipulate CSV files (get `testcsv.csv` from `CMEEMasteRepo`). Type the following script in `Week2/Code/basic_csv.py`

```
import csv

# Read a file containing:
# 'Species','Infraorder','Family','Distribution','Body mass male (Kg)'
f = open('../Sandbox/testcsv.csv','rb')

csvread = csv.reader(f)
temp = []
for row in csvread:
    temp.append(tuple(row))
    print row
    print "The species is", row[0]

f.close()

# write a file containing only species name and Body mass
f = open('../Sandbox/testcsv.csv','rb')
g = open('../Sandbox/bodymass.csv','wb')

csvread = csv.reader(f)
csvwrite = csv.writer(g)
for row in csvread:
    print row
    csvwrite.writerow([row[0], row[4]])

f.close()
g.close()
```

Tip

Now that you have seen how all-important indentation of python code is, you might find the `ipython %cpaste` function very handy, as it allows you to run fragments of code, indentation and all, directly in the `ipython` commandline. Let's try it. Type the following code in a temporary file:

```
for i in range(x):
        if i > 3: #4 spaces or 2 tabs in this case
                print i
```

Now, assign some integer value to a variable `x`:

```
In [ ]: x = 11
```

Then,

```
In [ ]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:for i in range(x):
:    if i > 3: #4 spaces or 2 tabs in this case
:        print i
:--
4
5
6
7
8
9
10
```

Of course, this code is simple, so directly pasting works as well — `%cpaste` is really useful when you have more complex code fragments you want to try out. Se haow far you have to pus direct pasting till you need `%cpaste`

5.8.1 Writing `python` functions (or modules)

Now let's writing proper `python` functions. We will start with a "boilerplate" code. Type the code below and save as `boilerplate.py` in `CMEECourseWork/Week2/Code`:

```
#!/usr/bin/python

"""Description of this program
    you can use several lines"""

__author__ = 'Samraat Pawar (s.pawar@imperial.ac.uk)'
__version__ = '0.0.1'

# imports
import sys # module to interface our program with the operating system

# constants can go here


# functions can go here
def main(argv):
        print 'This is a boilerplate' # NOTE: indented using two tabs or 4 spaces
        return 0

if (__name__ == "__main__"): #makes sure the "main" function is called from commandline
        status = main(sys.argv)
        sys.exit(status)
```

Running your `python` code

Now `cd` to the directory and run the code:

```
$ cd ~/Documents/../CMEECourseWork/Week2/Code
$ python boilerplate.py
```

You should see "This is a boilerplate" in your terminal window.

Alternatively, you can use ipython:

```
$ ipython boilerplate.py
```

You can also execute a python script file from within the `ipython` shell with `run MyScript.py`. So, enter `ipython` from bash, and do:

```
In [ ]: run boilerplate.py
```

To run the script from the native python shell, you would use `execfile("MyScript.py")`.

5.8.2 Components of the `python` function

Now let's look at the elements of your first, boilerplate code:

The shebang

Just like UNIX shell scripts, the first "shebang" line tells the computer where to look for python. It determines the script's ability to be executed like an standalone executable without typing python beforehand in the terminal or when double clicking it in a file manager (when configured properly to be an executable). It isn't necessary but generally put there so when someone sees the file opened in an editor, they immediately know what they're looking at. However, which shebang line you use is important.

Here by using `#!/usr/bin/python` we are specifying the location to the python executable in your machine that rest of the script needs to be interpreted with. You may want to use `#!/usr/bin/env python` instead, which will prevent failure to run if the Python executable on some other machine or distribution isn't actually located at `#!/usr/bin/python`, but elsewhere.

The Docstring

Triple quotes start a "docstring" comment, which is meant to describe the operation of the script or a function/module within it. docstrings are considered part of the running code, while normal comments are stripped. Hence, you can access your docstrings at run time. It is a good idea to have doctrings at the start of every python script and module as it can provide useful information to the user and you as well, down the line.

You can access the docstring(s) in a script (both for the overall script and the ones in each of its functions), by importing the function (say, `my_func`), and then typing `help(my_func)` in the python or ipython shell. For example, try `import boilerplate.py` and then `help(boilerplate)` (but you have to be in the python or ipython shell).

Internal Variables

"__" signal "internal" variables (never name your variables so!)

Function definitions and "modules"

def indicates the start of a python function; all subsequent lines must be indented.

It's important to know that somewhat confusingly, Pythonistas call a file containing function definitions's) and statements (e.g., assignments of constant variables) a "module". There is a practical reason (there's always one!) for this. You might want to use a particular set of python def's (functions) and statements either as a standalone function, or use it or subsets of it from other scripts. So in theory, every function you define can be a sub-module usable by other scripts.

In other words, definitions from a module can be imported into other modules and scripts, or into the main module itself.

At this juncture, you might also want to know more about a Python "class". Have a look at http://learnpythonthehardway.org/book/ex40.html — a nice, intuitive tutorial that should help you understand functions vs. modules vs. classes in Python.

The last few lines, including the main function/module are somewhat esoteric but important; more on this below.

Why include __name__ == "__main__" and all that jazz

When you run a Python module with or without arguments, the code in the called module will be executed just as if you imported it, but with the __name__ set to "__main__". So adding this code at the end of your module,

```
if (__name__ == "__main__"):
```

directs the python interpreter to set the special __name__ variable to have a value "__main__", so that the file is usable as a script as well as an importable module. How do you import? Simply as (in python or ipython shell):

```
In []: import boilerplate
```

Then type

```
In []: boilerplate
Out[]: <module 'boilerplate' from 'boilerplate.py'>
```

One more script to hopefully clarify this further. Type and save the following in a script file called using_name.py:

```
#!/usr/bin/python
# Filename: using_name.py
```

```
if __name__ == '__main__':
    print 'This program is being run by itself'
else:
    print 'I am being imported from another module'
```

Now run it:

```
In []: run using_name.py
This program is being run by itself
```

Now, try:

```
In []: import using_name
I am being imported from another module
```

The output `I am being imported from another module` will only show up once.

Also please look up `https://docs.python.org/2/tutorial/modules.html`

What on earth is `sys.argv`?

In your boilerplate code, as any other Python code, `argv` is the "argument variable". Such variables are necessarily very common across programming languages, and play an important role — `argv` is a variable that holds the arguments you pass to your Python script when you run it. `sys.argv` is simply an object created by python using the `sys` module (which you imported at the beginning of the script) that contains the names of the argument variables in the current script.

To understand this in a practical way, let's write and save a script called `sysargv.py`:

```
import sys
print "This is the name of the script: ", sys.argv[0]
print "Number of arguments: ", len(sys.argv)
print "The arguments are: " , str(sys.argv)
```

Now run `sysargv.py` with different numbers of arguments:

```
run sysargv.py
run sysargv.py var1 var2
run sysargv.py 1 2 var3
```

As you can see the first variable is always the file name, and is always available as to the Python interpreter.

Then, the command `main(argv=sys.argv)` directs the interpreter to pass the argument variables to the main function. Which brings us to,

```
def main(argv):
    print 'This is a boilerplate' # NOTE: indented using two tabs or four spaces
```

This is the main function. Arguments obtained in the `if (__name__ == "__main__"):` part of the script are "fed" to this main function where the printing of the line "This is a boilerplate" happens.

OK, finally, what about this bit:

```
sys.exit(status)
```

It's just a way to exit in a dignified (or abrupt) manner (from anywhere in the program)! Try putting it elsewhere in a function/module and see what happens.

5.8.3 Variable scope

One important thing to note about functions, in any language, is that variables inside functions are invisible outside of it, nor do they persist once the function has run. These are called "local" variables, and are only accessible inside their function. However, "global" variables are visible inside and outside of functions. In python, you can assign global variables. Type the following script in `scope.py` and try it:

```
## Try this first

_a_global = 10

def a_function():
    _a_global = 5
    _a_local = 4
    print "Inside the function, the value is ", _a_global
    print "Inside the function, the value is ", _a_local
    return None

a_function()
print "Outside the function, the value is ", _a_global



## Now try this

_a_global = 10

def a_function():
    global _a_global
    _a_global = 5
    _a_local = 4
    print "Inside the function, the value is ", _a_global
    print "Inside the function, the value is ", _a_local
    return None

a_function()
print "Outside the function, the value is", _a_global
```

However, in general, avoid assigning globals because you run the risk of "exposing" unwanted variables to all functions within your name work/namespace.

5.9 Control statements

OK, let's get deeper into `python` functions. To begin, first copy and rename `boilerplate.py` (to make use of it's existing structure and save you some typing):

```
$ cp boilerplate.py control_flow.py
```

```
$
```

Then type the following script into `control_flow.py`:

```
#!/usr/bin/env python

"""Some functions exemplifying the use of control statements"""
#docstrings are considered part of the running code (normal comments are
#stripped). Hence, you can access your docstrings at run time.
__author__ = 'Samraat Pawar (s.pawar@imperial.ac.uk)'
__version__ = '0.0.1'

import sys

def even_or_odd(x=0): # if not specified, x should take value 0.

    """Find whether a number x is even or odd."""
    if x % 2 == 0: #The conditional if
        return "%d is Even!" % x
    return "%d is Odd!" % x

def largest_divisor_five(x=120):
    """Find which is the largest divisor of x among 2,3,4,5."""
    largest = 0
    if x % 5 == 0:
        largest = 5
    elif x % 4 == 0: #means "else, if"
        largest = 4
    elif x % 3 == 0:
        largest = 3
    elif x % 2 == 0:
        largest = 2
    else: # When all other (if, elif) conditions are not met
        return "No divisor found for %d!" % x # Each function can return a value or a ↩
            variable.
    return "The largest divisor of %d is %d" % (x, largest)

def is_prime(x=70):
    """Find whether an integer is prime."""
    for i in range(2, x): #  "range" returns a sequence of integers
        if x % i == 0:
          print "%d is not a prime: %d is a divisor" % (x, i) #Print formatted text "%d↩
               %s %f %e" % (20,"30",0.0003,0.00003)

          return False
    print "%d is a prime!" % x
    return True

def find_all_primes(x=22):
    """Find all the primes up to x"""
    allprimes = []
    for i in range(2, x + 1):
      if is_prime(i):
        allprimes.append(i)
    print "There are %d primes between 2 and %d" % (len(allprimes), x)
    return allprimes

def main(argv):
    # sys.exit("don't want to do this right now!")
    print even_or_odd(22)
    print even_or_odd(33)
    print largest_divisor_five(120)
    print largest_divisor_five(121)
    print is_prime(60)
    print is_prime(59)
    print find_all_primes(100)
    return 0
```

```
if (__name__ == "__main__"):
    status = main(sys.argv)
    sys.exit(status)
```

Now run the code:

```
In []: run control_flow.py
```

You can also call any of the functions within `control_flow.py`:

```
In []: even_or_odd(11)
Out[]: '11 is Odd!'
```

This is possible without explicitly importing the modules because you are only running one script. You would have to do an explicit `import` if you needed a module from another python script file.

5.9.1 Control flow exercises

- ⋆ Write the following, and save them to `cfexercises.py`.
- ⋆ Now try these *function by function*, pasting the block in the ipython command line (hopefully you have set youe code editor to send a selection to the commandline by now)

```
# How many times will 'hello' be printed?
# 1)
for i in range(3, 17):
    print 'hello'

# 2)
for j in range(12):
    if j % 3 == 0:
        print 'hello'

# 3)
for j in range(15):
     if j % 5 == 3:
        print 'hello'
     elif j % 4 == 3:
        print 'hello'

# 4)
z = 0
while z != 15:
    print 'hello'
    z = z + 3

# 5)
z = 12
while z < 100:
    if z == 31:
        for k in range(7):
            print 'hello'
    elif z == 18:
        print 'hello'
    z = z + 1

# What does fooXX do?
def foo1(x):
    return x ** 0.5

def foo2(x, y):
```

```
    if x > y:
        return x
    return y

def foo3(x, y, z):
    if x > y:
        tmp = y
        y = x
        x = tmp
    if y > z:
        tmp = z
        z = y
        y = tmp
    return [x, y, z]

def foo4(x):
    result = 1
    for i in range(1, x + 1):
        result = result * i
    return result

# This is a recursive function, meaning that the function calls itself
# read about it at
# en.wikipedia.org/wiki/Recursion_(computer_science)
def foo5(x):
    if x == 1:
        return 1
    return x * foo5(x - 1)

foo5(10)
```

5.10 Loops

Write the following, and save them to `loops.py`.

```
# for loops in Python
for i in range(5):
    print i

my_list = [0, 2, "geronimo!", 3.0, True, False]
for k in my_list:
    print k

total = 0
summands = [0, 1, 11, 111, 1111]
for s in summands:
    print total + s

# while loops  in Python
z = 0
while z < 100:
    z = z + 1
    print (z)

b = True
while b:
    print "GERONIMO! infinite loop! ctrl+c to stop!"
# ctrl + c to stop!
```

Figure 5.3: In case you were wondering who Geronimo was.

5.10.1 List comprehensions

Python offers a way to combine loops, functions and logical tests in a single line of code. Type the following in a script file called `oaks.py`:

```
## Let's find just those taxa that are oak trees from a list of species

taxa = [ 'Quercus robur',
         'Fraxinus excelsior',
         'Pinus sylvestris',
         'Quercus cerris',
         'Quercus petraea',
       ]

def is_an_oak(name):
    return name.lower().startswith('quercus ')

##Using for loops
oaks_loops = set()
for species in taxa:
    if is_an_oak(species):
        oaks_loops.add(species)
print oaks_loops

##Using list comprehensions
oaks_lc = set([species for species in taxa if is_an_oak(species)])
print oaks_lc

##Get names in UPPER CASE using for loops
oaks_loops = set()
for species in taxa:
    if is_an_oak(species):
        oaks_loops.add(species.upper())
```

```
print oaks_loops

##Get names in UPPER CASE using list comprehensions
oaks_lc = set([species.upper() for species in taxa if is_an_oak(species)])
print oaks_lc
```

Don't go mad with list comprehensions — code readability is more important than squeezing lots into a single line!

5.10.2 Practicals

As always, test, add, commit and push all your new code and data to your git repository.

1. Modify `cfexercises.py` to make it a "module" like `control_flow.py`). That is, all the `fooXX` functions should take arguments from the user (like the functions inside `control_flow.py`. Also, add some test arguments to show that they work (again, like `control_flow.py`) — for example, "foo5(10)". Thus, running `cfexercises.py` should now also output evaluations of all the `fooXX` modules along with a bunch of hellos.
2. Open and complete the tasks in `lc1.py`, `lc2.py`, `dictionary.py`, `tuple.py` (you can tackle them in any order)

5.11 Functions, Modules, and code compartmentalization

Ideally you should aim to compartmentalize your code into a bunch of functions, typically written in a single `.py` file: this are Python "modules", which you were introduced to previously. Why bother with modules? Because:

- Keeping code compartmentalized is good for debugging, unit testing, and profiling (coming up later)
- Makes code more compact by minimizing redundancies (write repeatedly used code segments as a module)
- Allows you to import and use useful functions that you yourself wrote, just like you would from standard python packages (coming up)

5.11.1 Importing Modules

There are different ways to **import** a module:

- `import my_module`, then functions in the module can be called as `my_module.one_of_my_functions()`.
- `from my_module import my_function` imports only the function `my_function` in the module `my_module`. It can then be called as if it were part of the main file: `my_function()`.
- `import my_module as mm` imports the module `my_module` and calls it `mm`. Convenient when the name of the module is very long. The functions in the module can be called as `mm.one_of_my_functions()`.
- `from my_module import *`. Avoid doing this!
  Why? – to avoid name conflicts!

- You can also access variables written into modules: `import my_module`, then `my_module.one_of_my_variables`

5.12 Python packages

A Python package is simply a directory of Python modules (quite like an `R` package). Many packages, such as the following that I find particularly useful, are always available as standard libraries (just require `import` from within python or ipython):

- `io`: file input-output with `*.csv`, `*.txt`, etc.
- `subprocess`: to run other programs, including multiple ones at the same time, including operating system-dependent functionality
- `sqlite3`: for manipulating and querying `sqlite` databases
- `math`: for mathematical functions

Scores of other packages are accessible by explicitly installing them using `sudo apt-get install python-packagename` (as you did previously). Some particularly mentionable ones are:

- `scipy`: for scientific computing
- `matplotlib`: for plotting (very matlab-like, requires `scipy`) (all packaged in `pylab`)
- `scrapy`: for writing web spiders that crawl web sites and extract data from them
- `beautifulsoup`: for parsing HTML and XML (can do what `scrapy` does)
- `biopython`: for bioinformatics

Of course, you have already installed some of these (`scipy`, `matplotlib`).

For those of you interested in bioinformatics, the `biopython` package will be particularly useful. We will not cover bioinformatics in any depth within the python weeks, but you may want to try to use Python for bioinformatics in other weeks, especially the Genomics weeks, and perhaps use it for your Masters project. Therefore, I suggest that if bioinformatics is your thing, check out `biopython` — in particular the worked examples at `http://biopython.org/DIST/docs/tutorial/Tutorial.html`.

5.12.1 Practicals

As always, test, add, commit and push all your new code and data to your git repository.

Align DNA sequences

Align two DNA sequences such that they are as similar as possible.

The idea is to start with the longest string and try to position the shorter string in all possible positions. For each position, count a "score" : number of bases matched perfectly over the number of bases attempted. Your tasks:

1. Open and run `Practicals/Code/align_seqs.py` — make sure you understand what each line is doing to do this)

Now convert `align_seqs.py` to a Python function that takes the DNA sequences as an input from a single external file and saves the best alignment along with its corresponding score in a single text file (your choice of format and file type) to an appropriate location. No external should be needed; that is, you should still only need to use `python align_seq.py` to run it.

For example, the input file can be a single `.csv` file with the two example sequences given at the top of the original script.

Don't forget to add docstrings where necessary/appropriate.

2. Extra Credit – align all the `.fasta` sequences from Week 1; call the new script `align_seqs_fasta.py`. Unlike align_seqs.py, this script should take *any* two fasta sequences (in separate files) to be aligned as input. So this script would typically run by using explicit inputs, by calling something like `python align_seqs_fasta.py seq1.csv seq2.csv`. However, it should still run if no inputs were given, using two fasta sequences from `Data` as defaults.

5.13 Errors in your `python` code

What do you want from your code? Rank the following by importance:

1. it gives me the right answer
2. it is very fast
3. it is possible to test it
4. it is easy to read
5. it uses lots of 'clever' programming techniques
6. it has lots of tests
7. it uses every language feature that you know about

Then, think about this:

- If you are very lucky, your program will crash when you run it
- If you are lucky, you will get an answer that is obviously wrong
- If you are unlucky, you won't notice until after publication
- If you are very unlucky, someone else will notice it after publication

Ultimately, most of your time could well be spent error-checking and fixing them "debugging", not writing code. You can debug when errors appear, but why not just nip as many as you can in the bud? For this, you would use unit testing.

5.13.1 Unit testing

Unit testing prevents the most common mistakes and helps write reliable code. Indeed, there are many reasons for testing:

- Can you prove (to yourself) that your code does what you think it does?
- Did you think about the things that might go wrong?

- Can you prove to other people that your code works?
- Does it still all still work if you fix a bug?
- Does it still all still work if you add a feature?
- Does it work with that new dataset?
- Does it work on the latest version of R?
- Does it work on Mac, Linux, Windows?
- 64 bit and 32 bit?
- Does it work on an old version of a Mac
- Does it work on Harvey, or Imperial's Linux cluster?

The idea is to write *independent* tests for the *smallest units* of code. Why the smallest units? — to be able to retain the tests upon code modification.

Unit testing with `doctest`

Let's try `doctest`, the simplest testing tool in python: simpletests for each function are embedded in the docstring. Copy the file `control_flow.py` into the file `test_control_flow.py` and edit the original function so:

```
#!/usr/bin/python

"""Some functions exemplifying the use of control statements"""

__author__ = 'Samraat Pawar (s.pawar@imperial.ac.uk)'
__version__ = '0.0.1'

import sys
import doctest # Import the doctest module

def even_or_odd(x=0):
    """Find whether a number x is even or odd.

    >>> even_or_odd(10)
    '10 is Even!'

    >>> even_or_odd(5)
    '5 is Odd!'

    whenever a float is provided, then the closest integer is used:
    >>> even_or_odd(3.2)
    '3 is Odd!'

    in case of negative numbers, the positive is taken:
    >>> even_or_odd(-2)
    '-2 is Even!'

    """
    #Define function to be tested
    if x % 2 == 0:
        return "%d is Even!" % x
    return "%d is Odd!" % x

## I SUPPRESSED THIS BLOCK: WHY?

# def main(argv):
    # print even_or_odd(22)
    # print even_or_odd(33)
    # return 0

# if (__name__ == "__main__"):
    # status = main(sys.argv)
```

```
doctest.testmod()   # To run with embedded tests
```

Now type `run test_control_flow.py -v`:

```
In []: run  test_control_flow.py -v
Trying:
    even_or_odd(10)
Expecting:
    '10 is Even!'
ok
Trying:
    even_or_odd(5)
Expecting:
    '5 is Odd!'
ok
Trying:
    even_or_odd(3.2)
Expecting:
    '3 is Odd!'
ok
Trying:
    even_or_odd(-2)
Expecting:
    '-2 is Even!'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   4 tests in __main__.even_or_odd
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

You can also run doctest "on the fly", without writing `doctest.testmod()` in the code by typing in a terminal: `python -m doctest -v your_function_to_test.py`

Tip

Other unit testing approaches
For more complex testing, see documentation of `doctest` at `www.docs.python.org/2/library/doctest.html`, the package `nose` and the package `unittest` Please start testing as early as possible, but don't try to test everything either! Remember, it is easier to test if code is compartmentalized into functions.

5.13.2 Debugging

OK, so you unit-tested, let's go look at life through beer-goggles... BUT NO! YOU WILL VERY LIKELY RUN INTO BUGS!

Bugs happen, inevitably, in life and programming. You need to find and debug them. Banish all thoughts of littering your code with `print` statements to find bugs.

Enter the debugger. The command `pdb` turns on the python debugger. Type the following in a file and save as `debugme.py` in your `Code` directory:

```
def createabug(x):
    y = x**4
    z = 0.
    y = y/z
    return y

createabug(25)
```

Now run it:

```
In []: %run debugme.py
[lots of text]
createabug(x)
      2     y = x**4
      3     z = 0.
----> 4     y = y/z
      5     return y
      6

ZeroDivisionError: float division by zero
```

OK, so let's `%pdb` it

```
In []: %pdb
Automatic pdb calling has been turned ON

In []: run debugme.py
[lots of text]
ZeroDivisionError: float division by zero
> createabug()
      3     z = 0.
----> 4     y = y/z
      5     return y

ipdb>
```

Now we're in the debugger shell, and can use the following commands to naviagate and test the code line by line or block by block:

Tip

In "normal" python, you would use `pdb` instead of `ipdb`.

`n`	move to the next line
`ENTER`	repeat the previous command
`s`	"step" into function or procedure (i.e., continue the debugging inside the function, as opposed to simply run it)
`p x`	print variable x
`c`	continue until next break-point
`q`	quit
`l`	print the code surrounding the current position (you can specify how many)
`r`	continue until the end of the function

So let's continue our debugging:

```
ipdb> p x
25
ipdb> p y
390625
ipdb> p z
0.0
ipdb> p y/z
*** ZeroDivisionError: ZeroDivisionError
('float division by zero',)
ipdb> l
      1 def createabug(x):
      2     y = x**4
      3     z = 0.
----> 4     y = y/z
      5     return y
      6
      7 createabug(25)

ipdb> q

In []: %pdb
Automatic pdb calling has been turned OFF
```

5.13.3 Paranoid programming: debugging with breakpoints

You may want to pause the program run and inspect a given line or block of code (*why?* — impromptu unit-testing is one reason). To do so, simply put this snippet of code where you want to pause and start a debugging session and then run the program again:

```
import ipdb; ipdb.set_trace()
```

Or, you can use `import pdb; pdb.set_trace()`

Alternatively, running the code with the flag `%run -d` starts a debugging session from the first line of your code (you can also specify the line to stop at). If you are serious about programming, please start using a debugger (R, Python, whatever...)!

5.13.4 Practicals

As always, test, add, commit and push all your new code and data to your git repository.

Missing oaks problem

1. Open and run the code `test_oaks.py` — there's a bug, for no oaks are being found! (where's `TestOaksData.csv`?)

2. Fix the bug (hint: `import ipdb; ipdb.set_trace()`)

3. Now, write doctests to make sure that, bug or no bug, your `is_an_oak` function is working as expected (hint: `>>> is_an_oak('Fagus sylvatica')` should return `False`)

4. If you wrote good doctests, you will note that you found another error that you might not have come across just by debugging (hint: what happens if you try the doctest with 'Quercuss' instead of 'Quercus'?). How would you fix the new error you found using the doctest?

5.14 Practicals wrap-up

1. Review and make sure you can run all the commands, code fragments, and scripts we have till now and get the expected outputs — all scripts should work on any other linux laptop.
2. Run `boilerplate.py` and `control_flow.py` from the bash terminal instead of from within the ipython shell (try both python and ipython from the bash)
3. Include an appropriate docstring (if one is missing) at the beginning of *each* of each of the python script / module files you have written, as well as at the start of every function (or sub-module) in a module.
4. Also annotate your code lines as much and as often as necessary using #.
5. Keep all code files organized in `CMEECourseWork/Week2/Code`

`git add`, `commit` *and* `push` *all your code and data to your git repository by next Wednesday 5 PM.*

5.15 Readings and Resources

- Code like a Pythonista: Idiomatic `python` (Google it)
- Also good: the Google `python` Style Guide
- Browse the python tutorial: `www.docs.python.org/tutorial/`
- For functions and modules:
  `www.learnpythonthehardway.org/book/ex40.html`
- For IPython:
  `www.ipython.org/ipython-doc/stable/interactive/tips.html`

- Cookbooks can be very useful: `www.wiki.ipython.org/Cookbook`
- Look up `docs.python.org/2/library/index.html` – Read about the packages you think will be important to you.
- Some of you might find the python package `biopython` particularly useful — check out www.biopython.org/wiki/Main_Page, and especially, the cookbook
- In general, good module/package-specific cookbooks are out there — google "cookbook" along with the name of the package you are interested in (e.g., "scipy cookbook").

Chapter 6

Advanced Biological Computing in Python

> ...some things in life are bad. They can really make you mad. Other things just make you swear and curse. When you're chewing on life's gristle, don't grumble; give a whistle, and this'll help things turn out for the best. And... always look on the bright side of life...
>
> —*Guess who?*

In this chapter, we will cover a some topics in Python that will round-off your python training:

- "Reading" text data using regular expressions in python
- Numerical computing in python
- Databases, and using python to build and manage them
- Using Python to build workflows

The last topic will be necessary for your Miniproject, which will involve building a reproducible computational workflow.

6.1 Regular expressions in `python`

Let's shift gears now, and look at a very important skill that you should learn, or at least be aware of — *Regular expressions*. Regular expressions (regex) are a tool to find patterns in strings, such as:

- Find DNA motifs in sequence data
- Navigate through files in a directory
- Parse text files
- Extract information from html and xml files

Thus, if you are interested in data mining, need to clean or process data in any other way, or convert a bunch of informatuion into usable data, regex is really handy.

www.xkcd.com/208/

Regex packages are available for most programming languages (`grep` in UNIX / Linux, where regex first became popular).

6.1.1 Metacharacters vs. regular characters

A regex may consist of a combination of "metacharacters" (modifiers) and "regular" or literal characters. There are 14 metacharacters: `[ ] { } ( ) \ ^ $ . | ? * +` These metacharacters do special things, for example:

- `[12]` means match target to `1` and if that does not match then match target to `2`
- `[0-9]` means match to any character in range `0` to `9`
- `[^Ff]` means anything except upper or lower case `f` and `[^a-z]` means everything except lower case `a` to `z`

Everything else is interpreted literally (e.g., `a` is matched by entering `a` in the regex).

`[` and `]`, specify a character "class" — the set of characters that you wish to match. Metacharacters are not active inside classes. For example, `[a-z$]` will match any of the characters `a` to `z`, but also `$`, because inside a character class it loses its special metacharacter status.

6.1.2 regex elements

A useful (not exhaustive) list of regex elements is:

a	match the character a
3	match the number 3
\n	match a newline
\t	match a tab
\s	match a whitespace
.	match any character except line break (newline)
\w	match any alphanumeric character (including underscore)
\W	match any character not covered by \w (i.e., match any non-alphanumeric character excluding underscore)
\d	match a numeric character
\D	match any character not covered by \d (i.e., match a non-digit)
[atgc]	match any character listed: a, t, g, c
at|gc	match at or gc
[^atgc]	any character not listed: any character but a, t, g, c
?	match the preceding pattern element zero or one times
*	match the preceding pattern element zero or more times
+	match the preceding pattern element one or more times
{n}	match the preceding pattern element exactly n times
{n,}	match the preceding pattern element at least n times
{n,m}	match the preceding pattern element at least n but not more than m times
^	match the beginning of a line
$	match the end of a line

6.1.3 regex in python

Regex functions in python are in the module re — so we will import re. The simplest python regex function is re.search, which searches the string for match to a given pattern — returns a *match object* if a match is found and None if not.

Tip

Always put r in front of your regex — it tells python to read the regex in its "raw" (literal) form. Without raw string notation (r"text"), every backslash ('\') in a regular expression would have to be prefixed with another one to escape it.
From https://docs.python.org/2/library/re.html: *If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw*

strings for all but the simplest expressions.

OK, let's try some regexes (type all that follows in `Code/regexs.py`):

```
import re

my_string = "a given string"
# find a space in the string
match = re.search(r'\s', my_string)

print match
# this should print something like
# <_sre.SRE_Match object at 0x93ecdd30>

# now we can see what has matched
match.group()

match = re.search(r's\w*', my_string)

# this should return "string"
match.group()

# NOW AN EXAMPLE OF NO MATCH:
# find a digit in the string
match = re.search(r'\d', my_string)

# this should print "None"
print match

 # Further Example
 #
my_string = 'an example'
match = re.search(r'\w*\s', my_string)

if match:
    print 'found a match:', match.group()
else:
    print 'did not find a match'
```

To know whether a pattern was matched, we can use an `if`:

```
MyStr = 'an example'

match = re.search(r'\w*\s', MyStr)

if match:
    print 'found a match:', match.group()
else:
    print 'did not find a match'
```

Here are some more regexes (add all that follows to the `Code/regexs.py`):

```
# Some Basic Examples
match = re.search(r'\d' , "it takes 2 to tango")
print match.group() # print 2

match = re.search(r'\s\w*\s', 'once upon a time')
match.group() # ' upon '

match = re.search(r'\s\w{1,3}\s', 'once upon a time')
match.group() # ' a '
```

```
match = re.search(r'\s\w*$', 'once upon a time')
match.group() # ' time'

match = re.search(r'\w*\s\d.*\d', 'take 2 grams of H2O')
match.group() # 'take 2 grams of H2'

match = re.search(r'^\w*.*\s', 'once upon a time')
match.group() # 'once upon a '
## NOTE THAT *, +, and { } are all "greedy":
## They repeat the previous regex token as many times as possible
## As a result, they may match more text than you want

## To make it non-greedy, use ?:
match = re.search(r'^\w*.*?\s', 'once upon a time')
match.group() # 'once '

## To further illustrate greediness, let's try matching an HTML tag:
match = re.search(r'<.+>', 'This is a <EM>first</EM> test')
match.group() # '<EM>first</EM>'
## But we didn't want this: we wanted just <EM>
## It's because + is greedy!

## Instead, we can make + "lazy"!
match = re.search(r'<.+?>', 'This is a <EM>first</EM> test')
match.group() # '<EM>'

## OK, moving on from greed and laziness
match = re.search(r'\d*\.?\d*','1432.75+60.22i') #note "\" before "."
match.group() # '1432.75'

match = re.search(r'\d*\.?\d*','1432+60.22i')
match.group() # '1432'

match = re.search(r'[AGTC]+', 'the sequence ATTCGT')
match.group() # 'ATTCGT'

re.search(r'\s+[A-Z]{1}\w+\s\w+', 'The bird-shit frog''s name is Theloderma asper').↩
    group() # ' Theloderma asper'
## NOTE THAT I DIRECTLY RETURNED THE RESULT BY APPENDING .group()
```

Figure 6.1: In case you were wondering what *Theloderma asper*, the "bird-shit frog", looks like. I snapped this one in North-east India ages ago

You can group regexes into meaningful blocks using parentheses. For example, let's try matching a string consisting of an academic's name, email address and research area or interest (no need to type this into any python file):

```
MyStr = 'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and
ecological theory'

# without groups
match = re.search(r"[\w\s]*,\s[\w\.@]*,\s[\w\s&]*",MyStr)

match.group()
'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and ecological theory'

match.group(0)
'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and ecological theory'

# now add groups using ( )
match = re.search(r"([\w\s]*),\s([\w\.@]*),\s([\w\s&]*)",MyStr)

match.group(0)
'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and ecological theory'

match.group(1)
'Samraat Pawar'

match.group(2)
's.pawar@imperial.ac.uk'

match.group(3)
'Systems biology and ecological theory'
```

Have a look at `re4.py` in your code repository for more on parsing email addresses using regexes.

6.1.4 Some RegExercises

These exercises are not for submission as part of your coursework, but we will discuss them in class (in a later week).

1. Translate the following regular expressions into regular English (don't type this in `regexs.py`)!

```
r'^abc[ab]+\s\t\d'
% 'abca \t1'

r'^\d{1,2}\/\d{1,2}\/\d{4}$'
% '11/12/2004'

r'\s*[a-zA-Z,\s]+\s*'
% ' aBz  '
```

2. Write a regex to match dates in format YYYYMMDD, making sure that:

   - Only seemingly valid dates match (i.e., year greater than 1900)
   - First digit in month is either 0 or 1
   - First digit in day ≤ 3

6.1.5 Important `re` functions

`re.compile(reg)`	Compile a regular expression. In this way the pattern is stored for repeated use, improving the speed.
`re.search(reg, text)`	Scan the string and find the first match of the pattern in the string. Returns a `match` object if successful and `None` otherwise.
`re.match(reg, text)`	as `re.search`, but only match the beginning of the string.
`re.split(ref, text)`	Split the text by the occurrence of the pattern described by the regular expression.
`re.findall(ref, text)`	As `re.search`, but return a list of all the matches. If groups are present, return a list of groups.
`re.finditer(ref, text)`	As `re.search`, but return an iterator containing the next match.
`re.sub(ref, repl, text)`	Substitute each non-overlapping occurrence of the match with the text in `repl` (or a function!).

6.1.6 Practicals

As always, test, add, commit and push all your new code and data to your git repository.

Blackbirds problem

Complete the code `blackbirds.py` that you find in the `CMEEMasteRepo` (necessary data file is also there).

6.2 Numerical computing in `python`

The python package `scipy` can help you do serious number crunching including,

- Linear algebra (matrix and vector operations)
- Numerical integration (Solving ODEs)
- Fourier transforms
- Interpolation
- calculating special functions (incomplete Gamma, Bessel, etc.)
- Generation of random numbers
- Using statistical functions and transformations

In the following, we will use the `array` data structure in `scipy` for data manipulations and calculations. Scipy arrays are objects, and are similar in some respects to python lists, but are more naturally multidimensional, homogeneous in type (the default is float), and allow efficient (fast) manipulations. Thus scipy arrays are analogous to the R `matrix` data object/structure.

Tip

The same array objects are accessible within the `numpy` package, which is a subset of `scipy`.

So let's try `scipy`:

```
In []: import scipy

In []: a = scipy.array(range(5)) # a one-dimensional array

In []: a
Out[]: array([0, 1, 2, 3, 4])

In []: type(a)
Out[]: numpy.ndarray

In []: type(a[0])
Out []: numpy.int64
```

So all elements in `a` are of type `int` because that is what `range()` returns (try `?range`).

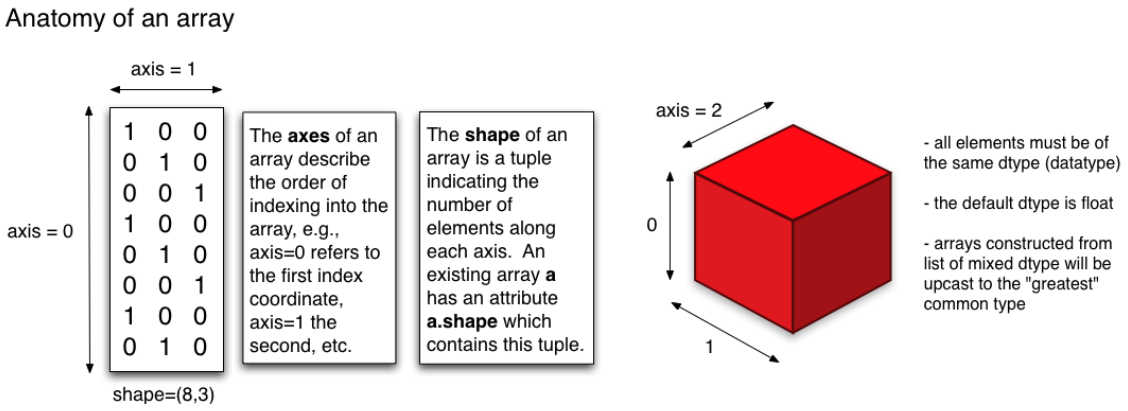


Figure 6.2: A graphical depiction of numpy/scipy arrays, which can have multiple dimensions (even greater than 3). From `http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/python/arrays.html`

You can also specify the data type of the array:

```
In []: a = scipy.array(range(5), float)

In []: a
Out[]: array([ 0.,  1.,  2.,  3.,  4.])

In []: a.dtype # Check type
Out[]: dtype('float64')
```

You can also get a 1-D arrays as follows:

```
In []: x = scipy.arange(5)

In []: x
```

```
Out[8]: array([0, 1, 2, 3, 4])

In [9]: x = scipy.arange(5.) #directly specify float using decimal

In [10]: x
Out[10]: array([ 0.,  1.,  2.,  3.,  4.])
```

As with other Python variables (e.g., created as a list or a dictionary), you can apply methods to variables created as scipy arrays. For example, TAB after `x.` to see methods you can apply to `x`:

```
In [11]: x.
x.T              x.conj           x.fill
x.nbytes         x.round          x.take
x.all            x.conjugate      x.flags
x.ndim           x.searchsorted   x.tofile
x.any            x.copy           x.flat
x.newbyteorder   x.setfield       x.tolist
x.argmax         x.ctypes         x.flatten
x.nonzero        x.setflags       x.tostring
x.argmin         x.cumprod        x.getfield
x.prod           x.shape          x.trace
x.argsort        x.cumsum         x.imag
x.ptp            x.size           x.transpose
x.astype         x.data           x.item
x.put            x.sort           x.var
x.base           x.diagonal       x.itemset
x.ravel          x.squeeze        x.view
x.byteswap       x.dot            x.itemsize
x.real           x.std
x.choose         x.dtype          x.max
x.repeat         x.strides
x.clip           x.dump           x.mean
x.reshape        x.sum
x.compress       x.dumps          x.min
x.resize         x.swapaxes

In [12]: x.shape
Out[12]: (5,)
```

Tip

Remember, you can type `:?x.methodname` to get info on a particular method. For example, try `?x.shape`.

You can also convert to and from Python lists:

```
In []: b = scipy.array([i for i in range(100) if i%2==1]) #odd numbers
between 1 and 100

In []: c = b.tolist() #convert back to list
```

To make a matrix, you need a 2-D scipy array:

```
In [14]: mat = scipy.array([[0, 1], [2, 3]])

In []: mat.shape
Out[]: (2, 2)
```

6.2.1 Indexing and accessing arrays

As with other Python data objects such as lists, scipy array elements can be accessed using square brackets ([]) with the [row,column] reference. Indexing of scipy arrays works like that for other data strauctures, with index values starting at 0. So, you can obtain all the elements of a particular row as:

```
In []: mat[1] # accessing whole 2nd row, remember indexing starts at
0
Out[]: array([2, 3])

In [57]: mat[:,1] #accessing whole second column
Out[57]: array([1, 3])
```

And accessing particular elements:

```
In []: mat[0,0] # 1st row, 1st column element
Out[]: 0
In []: mat[1,0] # 2nd row, 1st column element
Out[]: 2
```

Note that (like all other programming languages) row index always comes before column index. That is, `mat[1]` is always going to mean "whole second row", and `mat[1,1]` means 1st row and 1st column element. Therefore, to access the whole second column, you need:

```
In []: mat[:,1] #accessing whole second column
Out[]: array([0, 2])
```

Tip

Python indexing also accepts negative values for going back to the start from the end of an array:

```
In []: mat[0,1]
Out[]: 1

In []: mat[0,-1] #interesting!
Out[]: 1

In []: mat[0,-2] #very interesting, perhaps useless!
Out[]: 0
```

6.2.2 Manipulating arrays

Manipulating `scipy` arrays is pretty straightforward.

Replacing, adding or deleting elements

Let's look at how you can replace, add, or delete an array element (a single entry, or whole row(s) or whole column(s)):

```
In []: mat[0,0] = -1 #replace a single element

In []: mat
Out[]:
array([[-1,  1],
       [ 2,  3]])

In []: mat[:,0] = [12,12] #replace whole column

In []: mat
Out[]:
array([[12,  1],
       [12,  3]])

In []: scipy.append(mat, [[12,12]], axis = 0) #append row, note axis
specification
Out[]:
array([[12,  1],
       [12,  3],
       [12, 12]])

In []: scipy.append(mat, [[12],[12]], axis = 1) #append column
Out[]:
array([[12,  1, 12],
       [12,  3, 12]])

In []: newRow = [[12,12]] #create existing row

In []: mat = scipy.append(mat, newRow, axis = 0) #append that existing row
Out[]:
array([[12,  1],
       [12,  3],
       [12, 12]])

In []: scipy.delete(mat, 2, 0) #Delete 3rd row
Out[]:
array([[12,  1],
       [12,  3]])
```

And concatenation:

```
In []: mat = scipy.array([[0, 1], [2, 3]])

In []: mat0 = scipy.array([[0, 10], [-1, 3]])

In []: scipy.concatenate((mat, mat0), axis = 0)
Out[]:
array([[ 0,  1],
       [ 2,  3],
       [ 0, 10],
       [-1,  3]])
```

Flattening or reshaping arrays

You can also "flatten" or "melt" arrays, that is, change array dimensions (e.g., from a matrix to a vector):

```
In []: mat.ravel()
Out[]: array([0, 1, 2, 3]) # NOTE: ravel is row-priority

In []: mat.reshape((4,1)) # this is different from ravel - check ?scipy.reshape
Out[66]:
array([[0],
       [1],
       [2],
       [3]])

In []: mat.reshape((1,4)) # NOTE: reshaping is also row-priority
Out[]: array([[0, 1, 2, 3]])

In []: mat.reshape((3, 1)) # But total elements must remain the same!
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-81-ba16cb0744eb> in <module>()
----> 1 mat.reshape((3, 1))

ValueError: total size of new array must be unchanged
```

This is a bit different than how R behaves (coming up in Chapters (7–9)), where you won't get an error (R "recycles" data), which is more dangerous!

6.2.3 Pre-allocating arrays

As in other computer languages, it is usually more efficient to preallocate a array rather than append / insert / concatenate addtional elelents, rows, or columns. For example, if you know the size of your matrix or array, you can inititalize it with ones or zeros:

```
In []: scipy.ones((4,2)) #(4,2) are the (row,col) array dimensions
Out[]:
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])

In []: scipy.zeros((4,2)) # or zeros
Out[]:
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])

In []: m = scipy.identity(4) #create an identity matrix

In []: m
Out[]:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

In []: m.fill(16) #fill the matrix with 16

In []: m
Out[26]:
array([[ 16.,  16.,  16.,  16.],
       [ 16.,  16.,  16.,  16.],
       [ 16.,  16.,  16.,  16.],
       [ 16.,  16.,  16.,  16.]])
```

6.2.4 scipy matrices

Scipy also has a matrix data structure class. Scipy matrices are strictly 2-Dimensional, while scipy arrays are N-Dimensional. Matrix objects are a subclass of scipy arrays, so they inherit all the attributes and methods of scipy arrays (also called "ndarrays").

Tip

The main advantage of scipy matrices is that they provide a convenient notation for matrix multiplication: if a and b are matrices, then a*b is their matrix product.

Matrix-vector operations

Now let's perform some common matrix-vector operations on arrays (you also try the same using matrices instead of arrays):

```
In []: mm = scipy.arange(16)

In []: mm = mm.reshape(4,4) #Convert to matrix

In []: mm.transpose()
Out[]:
array([[ 0,  4,  8, 12],
       [ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15]])

In [6]: mm + mm.transpose()
Out[6]:
array([[ 0,  5, 10, 15],
       [ 5, 10, 15, 20],
       [10, 15, 20, 25],
       [15, 20, 25, 30]])

In [7]: mm - mm.transpose()
Out[7]:
array([[ 0, -3, -6, -9],
       [ 3,  0, -3, -6],
       [ 6,  3,  0, -3],
       [ 9,  6,  3,  0]])

In [8]: mm * mm.transpose()
## Elementwise!

Out[8]:
array([[  0,   4,  16,  36],
       [  4,  25,  54,  91],
       [ 16,  54, 100, 154],
       [ 36,  91, 154, 225]])

In [9]: mm / mm.transpose()
Warning: divide by zero encountered in divide

# Note the integer division
Out[9]:
array([[0, 0, 0, 0],
       [4, 1, 0, 0],
       [4, 1, 1, 0],
       [4, 1, 1, 1]])
```

```
In [10]: mm * scipy.pi
Out[10]:
array([[  0.      ,   3.14159,   6.28318531,   9.42477796],
       [ 12.566370,  15.70796,  18.84955592,  21.99114858],
       [ 25.132741,  28.27433,  31.41592654,  34.55751919],
       [ 37.699111,  40.84070,  43.98229715,  47.1238898 ]])

In [11]: mm.dot(mm) # MATRIX MULTIPLICATION
Out[11]:
array([[ 56,  62,  68,  74],
       [152, 174, 196, 218],
       [248, 286, 324, 362],
       [344, 398, 452, 506]])
```

We can do a lot more (but won't!) by importing the `linalg` sub-package: `scipy.linalg`.

6.2.5 Two useful `scipy` sub-packages

Two particularly useful `scipy` sub-packages are `scipy.integrate` (what will I need this for?) and `scipy.stats`. Why not use `R` for this? — because often you might just want to calculate some summary stats of your simulation results within Python.

`scipy.stats`

Let's take a quick spin in `scipy.stats`.

```
In [18]: import scipy.stats

In [19]: scipy.stats.
scipy.stats.arcsine               scipy.stats.lognorm
scipy.stats.bernoulli             scipy.stats.mannwhitneyu
scipy.stats.beta                  scipy.stats.maxwell
scipy.stats.binom                 scipy.stats.moment
scipy.stats.chi2                  scipy.stats.nanstd
scipy.stats.chisqprob             scipy.stats.nbinom
scipy.stats.circvar               scipy.stats.norm
scipy.stats.expon                 scipy.stats.powerlaw
scipy.stats.gompertz              scipy.stats.t
scipy.stats.kruskal               scipy.stats.uniform

In [19]: scipy.stats.norm.rvs(size = 10) # 10 samples from
N(0,1)
Out[19]:
array([-0.951319, -1.997693,  1.518519, -0.975607,  0.8903,
       -0.171347, -0.964987, -0.192849,  1.303369,  0.6728])

In [20]: scipy.stats.norm.rvs(5, size = 10)
# change mean to 5
Out[20]:
array([ 6.079362,  4.736106,  3.127175,  5.620740,  5.98831,
        6.657388,  5.899766,  5.754475,  5.353463,  3.24320])

In [21]: scipy.stats.norm.rvs(5, 100, size = 10)
# change sd to 100
Out[21]:
array([ -57.886247,   12.620516,  104.654729,  -30.979751,
         41.775710,  -31.423377,  -31.003134,   80.537090,
          3.835486,  103.462095])

# Random integers between 0 and 10
In [23]: scipy.stats.randint.rvs(0, 10, size =7)
Out[23]: array([6, 6, 2, 0, 9, 8, 5])
```

`scipy.integrate` – the Lotka-Volterra model

Numerical integration is the approximate computation of an integral using numerical techniques. You need numerical integration whenever you have a complicated function that cannot be integrated analytically using anti-derivatives. A common application is solving ordinary differential equations (ODEs), commonly used for modelling biological systems.

Let's try `scipy.integrate` for solving a classical model in biology — the Lotka-Volterra model for a predator-prey system.

⋆ Create `LV1.py` in your weekly directory and run it.

The Lotka-Volterra model is:

$$\begin{aligned} \frac{dR}{dt} &= rR - aCR \\ \frac{dC}{dt} &= -zC + eaCR \end{aligned} \tag{6.1}$$

where C and R are consumer (e.g., predator) and resource (e.g., prey) population sizes (either biomass or numbers), r is the intrinsic growth rate of the resource population, a is a "search rate" that determines the encounter rate between consumer and resource, z is mortality rate and e is the consumer's efficiency in converting resource to consumer biomass.

`LV1.py` runs (numerically solves the ODE) this model and plots the equilbrium. Have good look at the code, line by line, and make sure that you understand what's going on. A subsequent practical will require you to use this code to simulate modified version od the LV model.

6.3 The need for speed: Profiling in Python

Donald Knuth says: *Premature optimization is the root of all evil*. Indeed, computational speed may not be your initial concern. Also, you should focus on developing clean, reliable, reusable code rather than worrying first about how fast your code runs. However, speed will become an issue when and if your analysis or modeling becomes complex enough (e.g., food web or large network simulations). In that case, knowing which parts of your code take the most time is useful – optimizing those parts may save you lots of time. To find out what is slowing down your code you need to use "profiling".

6.3.1 Profiling

Profiling is easy in `ipython` – simply type the magic command `%run -p your_function_name`.

Let's write a simple illustrative program and name it `profileme.py`:

```
def a_useless_function(x):
    y = 0
    # eight zeros!
    for i in range(100000000):
        y = y + i
    return 0

def a_less_useless_function(x):
    y = 0
    # five zeros!
```

```
    for i in range(100000):
        y = y + i
    return 0

def some_function(x):
    print x
    a_useless_function(x)
    a_less_useless_function(x)
    return 0

some_function(1000)
```

Now `%run -p profileme.py`, and you should see something like:

```
         54 function calls in 3.652 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.744    2.744    3.648    3.648 profileme.py:1(a_useless_function)
        2    0.905    0.452    0.905    0.452 {range}
        1    0.002    0.002    0.003    0.003 profileme.py:8(a_less_useless_function)
[more output]
```

The function `range` is taking long – we should use `xrange` instead. When iterating over a large number of values, `xrange`, unlike `range`, does not create all the values before iteration, but creates them "on demand". E.g., `range(1000000)` yields a 4Mb+ list. So let's modify the script:

```
def a_useless_function(x):
    y = 0
    # eight zeros!
    for i in xrange(100000000):
        y = y + i
    return 0

def a_less_useless_function(x):
    y = 0
    # five zeros!
    for i in xrange(100000):
        y = y + i
    return 0

def some_function(x):
    print x
    a_useless_function(x)
    a_less_useless_function(x)
    return 0

some_function(1000)
```

Again running the magic command `%run -p` yields:

```
         52 function calls in 2.153 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.150    2.150    2.150    2.150 profileme2.py:1(a_useless_function)
        1    0.002    0.002    0.002    0.002 profileme2.py:8(a_less_useless_function)
        1    0.001    0.001    2.153    2.153 {execfile}
 [more output]
```

So we saved 1.499 s! (not enough to grab a pint, but ah well...).

6.3.2 Quick profiling with `timeit`

Alternatively, if you are writing your script and want to figure out what the best way to do something (say a particular command or a loop) might be, then you can use `timeit`.

Type an run the following code in a python script called `timeitme.py`:

```
################################################################################
# range vs. xrange.
################################################################################
import time
import timeit

def a_not_useful_function():
    y = 0
    for i in range(100000):
        y = y + i
    return 0

def a_less_useless_function():
    y = 0
    for i in xrange(100000):
        y = y + i
    return 0

# One approach is to time it like this:
start = time.time()
a_not_useful_function()
print "a_not_useful_function takes %f s to run." % (time.time() - start)

start = time.time()
a_less_useless_function()
print "a_less_useless_function takes %f s to run." % (time.time() - start)

# But you'll notice that if you run it multiple times, the time taken changes a
# bit. So instead, you can also run:
#   %timeit a_not_useful_function()
#   %timeit a_less_useless_function()
# in iPython.

################################################################################
# for loops vs. list comprehensions.
################################################################################

my_list = range(1000)

def my_squares_loop(x):
    out = []
    for i in x:
        out.append(i ** 2)
    return out


def my_squares_lc(x):
    out = [i ** 2 for i in x]
    return out

# %timeit my_squares_loop(my_list)
# %timeit my_squares_lc(my_list)


################################################################################
# for loops vs. join method.
################################################################################
```

```
import string
my_letters = list(string.ascii_lowercase)

def my_join_loop(l):
    out = ''
    for letter in l:
        out += letter
    return out


def my_join_method(l):
    out = ''.join(l)
    return out

# %timeit(my_join_loop(my_letters))
# %timeit(my_join_method(my_letters))


################################################################################
# Oh dear.
################################################################################

def getting_silly_pi():
    y = 0
    for i in xrange(100000):
        y = y + i
    return 0


def getting_silly_pii():
    y = 0
    for i in xrange(100000):
        y += i
    return 0

# %timeit(getting_silly_pi())
# %timeit(getting_silly_pii())
```

Now run it these different instances of timeit-ing by tying the `% timeit` command followed by the function call. Note that You can import all the functions using `from timeitme import *`

But remember, don't go crazy with profiling for the sake of shaving a couple of milliseconds, tempting as that may be!

6.3.3 Practicals

As always, test, add, commit and push all your new code and data to your git repository.

Lotka-Volterra model problem

Copy and modify `LV1.py` into another script called `LV2.py` that does the following:

1. Take arguments for the four LV model parameters r, a, z, e from the commandline

```
LV2.py arg1 arg2 ... etc
```

2. Runs the Lotka-Volterra model with prey density dependence $rR(1-\frac{R}{K})$, which changes the coupled ODEs to,

$$\begin{aligned}\frac{dR}{dt} &= rR(1-\frac{R}{K}) - aCR \\ \frac{dC}{dt} &= -zC + eaCR\end{aligned} \tag{6.2}$$

3. Saves the plot as `.pdf` in an external results directory in kyour weekly directory
4. The chosen parameter values should show in the plot (e.g., $r = 1, a = .5$, etc) You can change time length t too.
5. Include a script called `run_LV.py` in `Code` that will run both `LV1.py` and `LV2.py` with appropriate arguments. This script should also profile the two scripts and print the results to screen for each of the scripts using the `%run -p` approach. Look at and compare the speed bottlenecks in `LV1.py` and `LV2.py`. Think about how you could further speed up the scripts.

Extra credit if you also choose appropriate values for the parameters such that both predator and prey persist with prey density dependence — the final (non-zero) population values should be printed to screen.

Write every extra credit script file with a new name such as `LV3.py`,`LV4.py`*, etc.*

Extra-extra credit: Write a discrete-time version of the LV model called `LV3.py`. The discrete-time model is:

$$\begin{aligned}R_{t+1} &= R_t(1 + r(1-\frac{R_t}{K}) - aC_t) \\ C_{t+1} &= C_t(1 - z + eaR_t)\end{aligned} \tag{6.3}$$

Include this script in `run_LV.py`, and profile it as well.

Extra-extra-extra credit: Write a version of the discrete-time model (eqn 6.3) simulation with a random gaussian fluctuation in resource's growth rate at each time-step:

$$\begin{aligned}R_{t+1} &= R_t(1 + (r+\varepsilon)(1-\frac{R_t}{K}) - aC_t) \\ C_{t+1} &= C_t(1 - z + eaR_t)\end{aligned} \tag{6.4}$$

where ε is a random fluctuation drawn from a gaussian distribution (use `scipy.stats`). Include this script in `run_LV.py`, and profile it as well.

You can also add fluctuations to both populations simultaneously this way:

$$\begin{aligned}R_{t+1} &= R_t(1 + \varepsilon + r + (1-\frac{R_t}{K}) - aC_t) \\ C_{t+1} &= C_t(1 - z + \varepsilon + eaR_t)\end{aligned} \tag{6.5}$$

6.4 Networks in `python` (and R!)

ALL biological systems have a network representation, consisting of nodes for the biological entities of interest, and edges or links for the relatioinships between them. Here are some examples:

- Metabolic networks
- Gene regulatory networks

- Individual-Individual (e.g., social networks)
- Food webs
- Pollination networks

Can you think of a few more examples from biology?

You can easily simulate, analyze, and visualize biological networks in both `python` and `R` using some nifty packages. A full network anaalysis tutorial is out of the scope of our Python module's objectives, but let's try a simple visualization using the `networkx` python package.

For this you need to first install the package:

```
$ sudo apt-get install python-networkx
```

Now type the code file `DrawFW.py` and run it:

```
"""
    Plot a snapshot of a food web graph/network.

    Needs: Adjacency list of who eats whom (consumer name/id in 1st
    column, resource name/id in 2nd column), and list of species
    names/ids and properties such as biomass (node abundance), or average
    body mass.

"""
import networkx as nx
import scipy as sc
import matplotlib.pyplot as plt
# import matplotlib.animation as ani #for animation

def GenRdmAdjList(N = 2, C = 0.5):
    """
    Generate random adjacency list given N nodes with connectance
    probability C
    """
    Ids = range(N)
    ALst = []
    for i in Ids:
        if sc.random.uniform(0,1,1) < C:
            Lnk = sc.random.choice(Ids,2).tolist()
            if Lnk[0] != Lnk[1]: #avoid self loops
                ALst.append(Lnk)
    return ALst

## Assign body mass range
SizRan = ([-10,10]) #use log scale

## Assign number of species (MaxN) and connectance (C)
MaxN = 30
C = 0.75

## Generate adjacency list:
AdjL = sc.array(GenRdmAdjList(MaxN, C))

## Generate species (node) data:
Sps = sc.unique(AdjL) # get species ids
Sizs = sc.random.uniform(SizRan[0],SizRan[1],MaxN)# Generate body sizes (log10 scale)

###### The Plotting #####
plt.close('all')

##Plot using networkx:

## Calculate coordinates for circular configuration:
## (See networkx.layout for inbuilt functions to compute other types of node
```

```
# coords)
pos = nx.circular_layout(Sps)

G = nx.Graph()
G.add_nodes_from(Sps)
G.add_edges_from(tuple(AdjL))
NodSizs= 10**-32 + (Sizs-min(Sizs))/(max(Sizs)-min(Sizs)) #node sizes in proportion to ↩
    body sizes
nx.draw(G, pos, node_size = NodSizs*1000)
plt.show()
```

Look thorugh the code carefully (line-by-line) as there are some new python objects introduced by `networkx` for storing node and edge data.

6.4.1 Practical (Extra Credit)

You can also do nice network visualizations in R. Here you will convert a network visualization script written in `R` using the `igraph` package to a python script that does the same thing.

First copy the script file called `Nets.R` and the data files it calls and run it. This script visualizes the QMEE CDT collaboration network (see `http://www.imperial.ac.uk/qmee-cdt/`).

1. Now, modify the nodes in the R script so that the ndoes are colored by the type of node (organization type: "University","Hosting Partner", "Non-hosting Partner"). The script already has hints for how you can do this.
2. Now, convert this script to a `python` script that does the same thing, including writing to an `*.svg` file using the same QMEE CDT link and node data. You can use `networkx` or any other python network visualization package!

6.5 Databases and `python`

Many of you will deal with complex data — and often, lots of it. Ecological and Evolutionary data are particularly complex because they contain large numbers of attributes, often measured in very different scales and units for individual taxa, populations, etc. In this scenario, storing the data in a database makes a lot of sense! You can easily include the database in your analysis workflow — indeed, that's why people use databases. And you can use python (and R) to build, manipulate and use your database.

6.5.1 Relational databases

A *relational* database is a collection of interlinked (*related*) tables that altogether store a complex dataset in a logical, computer-readable format. Dividing a dataset into multiple tables minimizes redundancies. For example, if your data were sampled from three sites — then, rather than repeating the site name and description in each row in a text file, you could just specify a numerical "key" that directs to another table containing the sampling site name and description.

Finally, if you have many rows in your data file, the type of sequential access we have been using in our `python` and `R` scripts is inefficient — you should be able to instantly access any row regardless of its position

Data columns in a database are usually called *fields*, while the rows are the *records*. Here are a few things to keep in mind about databases:

- Each field typically contains only one data type (e.g., integers, floats, strings)
- Each record is a "data point", composed of different values, one for each field — somewhat like a python tuple
- Some fields are special, and are called *keys*:
  - The *primary key* uniquely defines a record in a table (e.g., each row is identified by a unique number)
  - To allow fast retrieval, some fields (and typically all the keys) are indexed — a copy of certain columns that can be searched very efficiently
  - *Foreign keys* are keys in a table that are primary keys in another table and define relationships between the tables
- The key to designing a database is to minimize redundancy and dependency without losing the logical consistency of tables — this is called *normalization* (arguably more of an art than a science!)

Let's look at a simple example.

Imagine you recorded body sizes of species from different field sites in a single text file (e.g., a `.csv` file) with the following fields:

`ID`	Unique ID for the record
`SiteName`	Name of the site
`SiteLong`	Longitude of the site
`SiteLat`	Latitude of the site
`SamplingDate`	Date of the sample
`SamplingHour`	Hour of the sampling
`SamplingAvgTemp`	Average air temperature on the sampling day
`SamplingWaterTemp`	Temperature of the water
`SamplingPH`	PH of the water
`SpeciesCommonName`	Species of the sampled individual
`SpeciesLatinBinom`	Latin binomial of the species
`BodySize`	Width of the individual
`BodyWeight`	Weight of the individual

It would be logical to divide the data into four tables:

Site table:

`SiteID`	ID for the site
`SiteName`	Name of the site
`SiteLong`	Longitude of the site
`SiteLat`	Latitude of the site

Sample table:

SamplingID	ID for the sampling date
SamplingDate	Date of the sample
SamplingHour	Hour of the sample
SamplingAvgTemp	Average air temperature
SamplingWaterTemp	Temperature of the water
SamplingPH	PH of the water

Species table:

SpeciesID	ID for the species
SpeciesCommonName	Species name
SpeciesLatinBinom	Latin binomial of the species

Individual table:

IndividualID	ID for the individual sampled
SpeciesID	ID for the species
SamplingID	ID for the sampling day
SiteID	ID for the site
BodySize	Width of the individual
BodyWeight	Weight of the individual

In each table, the first ID field is the primary key. The last table contains three foreign keys because each individual is associated with one species, one sampling day and one sampling site.

These structural features of a database are called its *schema*.

6.5.2 SQLite

SQLite is a simple (and very popular) SQL (Structured Query Language)-based solution for managing localized, personal databases. I can safely bet that most, if not all of you unknowingly (or knowingly!) use SQLite — it is used by MacOSX, Firefox, Acrobat Reader, iTunes, Skype, iPhone, etc. SQLite is also the database "engine" underlying your Siwlood Masters Web App: http://silwoodmasters.co.uk

We can easily use SQLite through Python scripts. First, install SQLite by typing in the Ubuntu terminal:

```
$ sudo apt-get install sqlite3 libsqlite3-dev
```

Also, make sure that you have the necessary package for python by typing `import sqlite3` in the python or ipython shell. Finally, you may install a GUI for SQLite3 :

```
$ sudo apt-get install sqliteman
```

Now type `sqlite3` in the Ubuntu terminal to check if SQLite successfully launches.

SQLite has very few data types (and lacks a boolean and a date type):

`NULL`	The value is a NULL value
`INTEGER`	The value is a signed integer, stored in up to or 8 bytes
`REAL`	The value is a floating point value, stored as in 8 bytes
`TEXT`	The value is a text string
`BLOB`	The value is a blob of data, stored exactly as it was input (useful for binary types, such as bitmap images or pdfs)

Typically, you will build a database by importing csv data — be aware that:

- Headers: the csv should have no headers
- Separators: if the comma is the separator, each record should not contain any other commas
- Quotes: there should be no quotes in the data
- Newlines: there should be no newlines

Now build your first database in SQLite! We will use as example a global dataset on metabolic traits called *Biotraits* that we are currently developing in our lab (should be in your `Data` directory). This dataset contains 164 columns (fields). Thermal response curves for different traits and species are stored in rows. This means that site description or taxonomy are repeated as many times as temperatures are measured in the curve. You can imagine how much redundancy can be here!!!

For this reason, it is easier to migrate the dataset to SQL and split it into several tables:

- TCP: Includes the thermal curve performance for each species and trait (as many rows per trait and species as temperatures have been measured within the TCP)
- TraitInfo: Contains site description and conditions under the traits were measured (one row per thermal curve)
- Consumer: Consumer description including taxonomy (one row per thermal curve).
- Resource: Resource description including taxonomy (one row per thermal curve).
- Size: Size data for each species (one row per thermal curve)
- DataSource: Contains information about the data source (citation, contributors) (one row per thermal curve).

So all these tables compose the *Biotraits* `schema`.

Navigate to your `Data` directory and in an Ubuntu terminal type:

```
$ sqlite3 Biotraits.db
SQLite version 3.7.9
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

This creates an empty database in your `Data` directory. Now, you need to create a table with some fields. Let's start with the *TraitInfo* table:

```
sqlite> CREATE TABLE TraitInfo (Numbers integer primary key,
   ...>                                 OriginalID text,
   ...>                                 FinalID text,
   ...>                                 OriginalTraitName text,
   ...>                                 OriginalTraitDef text,
   ...>                                 Replicates integer,
   ...>                                 Habitat  integer,
   ...>                                 Climate text,
```

```
   ...>                                 Location text,
   ...>                                 LocationType text,
   ...>                                 LocationDate text,
   ...>                                 CoordinateType text,
   ...>                                 Latitude integer,
   ...>                                 Longitude integer);
```

Note that I am writing all SQL commands in upper case, but it is not necessary. I am using upper case here because SQL syntax is long and clunky, and it quickly becomes hard to spot (and edit) commands in long strings of complex queries.

Now let's import the dataset:

```
sqlite> .mode csv

sqlite> .import TraitInfo.csv TraitInfo
```

So we built a table and imported a csv file into it. Now we can ask SQLite to show all the tables we currently have:

```
sqlite> .tables

TraitInfo
```

Let's run our first *Query* (note that you need a semicolon to end a command):

```
sqlite> SELECT * FROM TraitInfo LIMIT 5;

1,1,MTD1,"Resource Consumption Rate","The number of resource consumed per number of ↩
    consumers per time",6,freshwater,temperate,"Eunice Lake; Ontario; Canada",NA,NA,NA↩
    ,51.254,-85.323
2,1,MTD1,"Resource Consumption Rate","The number of resource consumed per number of ↩
    consumers per time",6,freshwater,temperate,"Eunice Lake; Ontario; Canada",NA,NA,NA↩
    ,51.254,-85.323
3,1,MTD1,"Resource Consumption Rate","The number of resource consumed per number of ↩
    consumers per time",6,freshwater,temperate,"Eunice Lake; Ontario; Canada",NA,NA,NA↩
    ,51.254,-85.323
4,2,MTD2,"Resource Consumption Rate","The number of resource consumed per number of ↩
    consumers per time",6,freshwater,temperate,"Eunice Lake; Ontario; Canada",NA,NA,NA↩
    ,51.254,-85.323
5,2,MTD2,"Resource Consumption Rate","The number of resource consumed per number of ↩
    consumers per time",6,freshwater,temperate,"Eunice Lake; Ontario; Canada",NA,NA,NA↩
    ,51.254,-85.323
```

Let's turn on some nicer formatting:

```
sqlite> .mode column

sqlite> .header ON

sqlite> SELECT * FROM TraitInfo LIMIT 5;

Numbers  OriginalID  FinalID     OriginalTraitName           ...
-------  ----------  ----------  ------------------------    ...
1        1           MTD1        Resource Consumption Rate   ...
4        2           MTD2        Resource Consumption Rate   ...
6        3           MTD3        Resource Consumption Rate   ...
9        4           MTD4        Resource Mass Consumption   ...
12       5           MTD5        Resource Mass Consumption   ...
```

The main statement to select records from a table is `SELECT`:

```
sqlite> .width 40  ## NOTE: Control the width

sqlite> SELECT DISTINCT OriginalTraitName FROM TraitInfo; # Returns unique values

OriginalTraitName
----------------------------------------
Resource Consumption Rate
Resource Mass Consumption Rate
Mass-Specific Mass Consumption Rate
Voluntary Body Velocity
Forward Attack Distance
Foraging Velocity
Resource Reaction Distance
....

sqlite> SELECT DISTINCT Habitat FROM TraitInfo
   ...> WHERE OriginalTraitName = "Resource Consumption Rate"; # Sets a condition

Habitat
----------------------------------------
freshwater
marine
terrestrial

sqlite> SELECT COUNT (*) FROM TraitInfo;  # Returns number of rows

Count (*)
--------------------
2336

sqlite> SELECT Habitat, COUNT(OriginalTraitName) # Returns number of rows for each ↩
    group
   ...> FROM TraitInfo GROUP BY Habitat;

Habitat     COUNT(OriginalTraitName)
----------  ------------------------
NA          16
freshwater  609
marine      909
terrestria  802

sqlite> SELECT COUNT(DISTINCT OriginalTraitName) # Returns number of unique values
   ...> FROM TraitInfo;

COUNT(DISTINCT OriginalTraitName)
---------------------------------
220

sqlite> SELECT COUNT(DISTINCT OriginalTraitName) TraitCount # Assigns alias to the ↩
    variable
   ...> FROM TraitInfo;

TraitCount
----------
220

sqlite> SELECT Habitat,
   ...> COUNT(DISTINCT OriginalTraitName) AS TN
   ...> FROM TraitInfo GROUP BY Habitat;

Habitat     TN
----------  ----------
NA          7
freshwater  82
marine      95
terrestria  96
```

```
sqlite> SELECT * # WHAT TO SELECT
   ...> FROM TraitInfo # FROM WHERE
   ...> WHERE Habitat = "marine" # CONDITIONS
   ...> AND OriginalTraitName = "Resource Consumption Rate";

Numbers     OriginalID  FinalID     OriginalTraitName          ...
----------  ----------  ----------  -------------------------  ...
778         308         MTD99       Resource Consumption Rate  ...
798         310         MTD101      Resource Consumption Rate  ...
806         311         MTD102      Resource Consumption Rate  ...
993         351         MTD113      Resource Consumption Rate  ...
```

The structure of the `SELECT` commend is as follows (*Note:* **all** *characters are case* **in***sensitive*):

```
SELECT [DISTINCT] field
FROM table
WHERE predicate
GROUP BY field
HAVING predicate
ORDER BY field
LIMIT number
;
```

Let's try some more elaborate queries:

```
sqlite> SELECT Numbers FROM TraitInfo LIMIT 5;

Numbers
----------
1
4
6
9
12

sqlite> SELECT Numbers
   ...> FROM TraitInfo
   ...> WHERE Numbers > 100
   ...> AND Numbers < 200;

Numbers
----------
107
110
112
115

sqlite> SELECT Numbers
   ...> FROM TraitInfo
   ...> WHERE Habitat = "freshwater"
   ...> AND Number > 700
   ...> AND Number < 800;

Numbers
----------
704
708
712
716
720
725
730
735
740
```

```
744
748
```

You can also match records using something like regular expressions. In SQL, when we use the command `LIKE`, the percent % symbol matches any sequence of zero or more characters and the underscore matches any single character. Similarly, `GLOB` uses the asterisk and the underscore.

```
sqlite> SELECT DISTINCT OriginalTraitName
   ...> FROM TraitInfo
   ...> WHERE OriginalTraitName LIKE "_esource Consumption Rate";

OriginalTraitName
------------------------
Resource Consumption Rate

sqlite> SELECT DISTINCT OriginalTraitName
   ...> FROM TraitInfo
   ...> WHERE OriginalTraitName LIKE "Resource%";

OriginalTraitName
--------------------------------------
Resource Consumption Rate
Resource Mass Consumption Rate
Resource Reaction Distance
Resource Habitat Encounter Rate
Resource Consumption Probability
Resource Mobility Selection
Resource Size Selection
Resource Size Capture Intent Acceptance
Resource Encounter Rate
Resource Escape Response Probability

sqlite> SELECT DISTINCT OriginalTraitName
   ...> FROM TraitInfo
   ...> WHERE OriginalTraitName GLOB "Resource*";


OriginalTraitName
--------------------------------------
Resource Consumption Rate
Resource Mass Consumption Rate
Resource Reaction Distance
Resource Habitat Encounter Rate
Resource Consumption Probability
Resource Mobility Selection
Resource Size Selection
Resource Size Capture Intent Acceptance
Resource Encounter Rate
Resource Escape Response Probability

# NOTE THAT GLOB IS CASE SENSITIVE, WHILE LIKE IS NOT

sqlite> SELECT DISTINCT OriginalTraitName
   ...> FROM TraitInfo
   ...> WHERE OriginalTraitName LIKE "resource%";

OriginalTraitName
--------------------------------------
Resource Consumption Rate
Resource Mass Consumption Rate
Resource Reaction Distance
Resource Habitat Encounter Rate
Resource Consumption Probability
Resource Mobility Selection
Resource Size Selection
Resource Size Capture Intent Acceptance
Resource Encounter Rate
```

```
Resource Escape Response Probability
```

We can also order by any column:

```
sqlite> SELECT OriginalTraitName, Habitat FROM
   ...>  TraitInfo LIMIT 5;

OriginalTraitName          Habitat
-------------------------  ----------
Resource Consumption Rate  freshwater
Resource Consumption Rate  freshwater
Resource Consumption Rate  freshwater
Resource Mass Consumption  freshwater
Resource Mass Consumption  freshwater

sqlite> SELECT OriginalTraitName, Habitat FROM
   ...> TraitInfo ORDER BY OriginalTraitName LIMIT 5;

OriginalTraitName           Habitat
--------------------------  ----------
48-hr Hatching Probability  marine
Asexual Reproduction Rate   marine
Attack Body Acceleration    marine
Attack Body Velocity        marine
Attack Body Velocity        marine
```

Until now we have just queried data from one single table, but as we have seen, the point of storing a database in SQL is that we can use multiple tables minimizing redundancies within them. And of course, querying data from those different tables at the same time will be necessary at some point.

Let's import then one more table to our database:

```
 sqlite> CREATE TABLE Consumer (Numbers integer primary key,
   ...>                                 OriginalID text,
   ...>                                 FinalID text,
   ...>                                 Consumer text,
   ...>                                 ConCommon text,
   ...>                                 ConKingdom text,
   ...>                                 ConPhylum text,
   ...>                                 ConClass text,
   ...>                                 ConOrder text,
   ...>                                 ConFamily text,
   ...>                                 ConGenus text,
   ...>                                 ConSpecies text);


sqlite> .import Consumer.csv Consumer

# Now we have two tables in our database:

sqlite> .tables
Consumer   TraitInfo


# These tables are connected by two differents keys: OriginalID
# and FinalID. These are unique IDs for each thermal curve. For each
# FinalID we can get the trait name (OriginalTraitName) from the TraitInfo
# table and the corresponding species name (ConSpecies) from the Consumer table.

sqlite> SELECT A1.FinalID, A1.Consumer, A2.FinalID,  A2.OriginalTraitName
   ...> FROM Consumer A1, TraitInfo A2
   ...> WHERE A1.FinalID=A2.FinalID LIMIT 8;
```

```
FinalID     Consumer                FinalID     OriginalTraitName
----------  ----------------------  ----------  ------------------------
MTD1        Chaoborus trivittatus   MTD1        Resource Consumption Rate
MTD2        Chaoborus trivittatus   MTD2        Resource Consumption Rate
MTD3        Chaoborus americanus    MTD3        Resource Consumption Rate
MTD4        Stizostedion vitreum    MTD4        Resource Mass Consumption
MTD5        Macrobrachium rosenbe   MTD5        Resource Mass Consumption
MTD6        Ranatra dispar          MTD6        Resource Consumption Rate
MTD7        Ceriodaphnia reticula   MTD7        Mass-Specific Mass Consum
MTD8        Polyphemus pediculus    MTD8        Voluntary Body Velocity

# In the same way we assign alias to variables, we can use them for tables.
```

This example seems easy because both tables have the same number of rows. But the query is still as simple when we have tables with different rows.

```
# Let's import the TCP table:

sqlite> CREATE TABLE TCP (Numbers integer primary key,
   ...>                   OriginalID text,
   ...>                   FinalID text,
   ...>                   OriginalTraitValue integer,
   ...>                   OriginalTraitUnit text,
   ...>                   LabGrowthTemp integer,
   ...>                   LabGrowthTempUnit text,
   ...>                   ConTemp integer,
   ...>                   ConTempUnit text,
   ...>                   ConTempMethod text,
   ...>                   ConAcc text,
   ...>                   ConAccTemp integer);


sqlite> .import TCP.csv TCP
sqlite> .tables
Consumer   TCP        TraitInfo

# Now imagine we want to query the thermal performance curves that we have
# stored for the species Mytilus edulis. Using the FinalID to match the tables,
# the query can be as simple as:

sqlite> SELECT A1.ConTemp, A1.OriginalTraitValue, A2.OriginalTraitName, A3.Consumer
   ...> FROM TCP A1, TraitInfo A2, Consumer A3
   ...> WHERE A1.FinalID=A2.FinalID AND A3.ConSpecies="Mytilus edulis" AND A3.FinalID=↩
       A2.FinalID LIMIT 8

ConTemp     OriginalTraitValue    OriginalTraitName              Consumer
----------  --------------------  -----------------------------  --------------------
25          2.707075              Filtration Rate                Mytilus edulis
20          3.40721               Filtration Rate                Mytilus edulis
5           3.419455              Filtration Rate                Mytilus edulis
15          3.711165              Filtration Rate                Mytilus edulis
10          3.875465              Filtration Rate                Mytilus edulis
5           0.34                  In Vitro Gill Particle Transpo  Mytilus edulis
10          0.46                  In Vitro Gill Particle Transpo  Mytilus edulis
15          0.595                 In Vitro Gill Particle Transpo  Mytilus edulis
```

So on and so forth (joining tables etc. would come next...). But if you want to keep practicing and learn more about sqlite commands, this is a very useful site: `http://www.sqlite.org/sessions/sqlite.html`. You can store your queries and database management commands in an `.sql` file (`geany` will take care of syntax highlighting etc.)

6.5.3 SQLite with python

It is easy to access, update and manage SQLite databases with `python` (you should have this script file in your `Code` directory):

```
# import the sqlite3 library
import sqlite3

# create a connection to the database
conn = sqlite3.connect('../Data/test.db')

# to execute commands, create a "cursor"
c = conn.cursor()

# use the cursor to execute the queries
# use the triple single quote to write
# queries on several lines
c.execute('''CREATE TABLE Test
           (ID INTEGER PRIMARY KEY,
           MyVal1 INTEGER,
           MyVal2 TEXT)''')

#~c.execute('''DROP TABLE test''')

# insert the records. note that because
# we set the primary key, it will auto-increment
# therefore, set it to NULL
c.execute('''INSERT INTO Test VALUES
           (NULL, 3, 'mickey')''')

c.execute('''INSERT INTO Test VALUES
           (NULL, 4, 'mouse')''')

# when you "commit", all the commands will
# be executed
conn.commit()

# now we select the records
c.execute("SELECT * FROM TEST")

# access the next record:
print c.fetchone()
print c.fetchone()

# let's get all the records at once
c.execute("SELECT * FROM TEST")
print c.fetchall()

# insert many records at once:
# create a list of tuples
manyrecs = [(5, 'goofy'),
            (6, 'donald'),
            (7, 'duck')]

# now call executemany
c.executemany('''INSERT INTO test
                 VALUES(NULL, ?, ?)''', manyrecs)

# and commit
conn.commit()

# now let's fetch the records
# we can use the query as an iterator!
for row in c.execute('SELECT * FROM test'):
    print 'Val', row[1], 'Name', row[2]

# close the connection before exiting
conn.close()
```

You can create a database in memory, without using the disk — thus you can create and discard an SQLite database within your workflow!:

```
import sqlite3

conn = sqlite3.connect(":memory:")

c = conn.cursor()

c.execute("CREATE TABLE tt (Val TEXT)")

conn.commit()

z = [('a',), ('ab',), ('abc',), ('b',), ('c',)]

c.executemany("INSERT INTO tt VALUES (?)", z)

conn.commit()

c.execute("SELECT * FROM tt WHERE Val LIKE 'a%'").fetchall()

conn.close()
```

6.6 Using `python` to build workflows

You can use python to build an automated data analysis or simulation workflow that involves multiple applications, especially the ones you have already learnt: `R`, LaTeX, & UNIX `bash`. For example, you could, in theory, write a single Python script to generate and update your masters dissertation, tables, plots, and all. Python is ideal for building such workflows because it has packages for practically every purpose (see Section on Packages above).

6.6.1 Using `subprocess`

The `subprocess` module is particularly important as it can run other applications, including R. Let's try – first launch `ipython`, then `cd` to your python code directory, and type:

```
import subprocess
subprocess.os.system("geany boilerplate.py")
subprocess.os.system("gedit ../Data/TestOaksData.csv")
subprocess.os.system("python boilerplate.py") # A bit silly!
```

Easy as pie! You will notice that the terminal remains "connected" to geany after you run the first of the three lines above, and you have to quit geany to go on to launcing gedit. To avoid this, you can do:

```
subprocess.os.system("geany boilerplate.py &")
subprocess.os.system("gedit ../Data/TestOaksData.csv &")
subprocess.os.system("python boilerplate.py &") # A bit silly!
```

Adding a `&` after a program call, i.e., `geany boilerplate.py &` instead of `geany boilerplate.py` disconnects the terminal and allows you to run sequential commands in the terminal/bash.

Similarly, to compile your LaTeXdocument (using `pdflatex` in this case):

```
subprocess.os.system("pdflatex yourlatexdoc.tex")
```

You can also do this (instead of using `subprocess.os`):

```
subprocess.Popen("geany boilerplate.py", shell=True).wait()
```

You can also use `subprocess.os` to make your code OS (Linux, Windows, Mac) independent. For example to assign paths:

```
subprocess.os.path.join('directory', 'subdirectory', 'file')
```

The result would be appropriately different on Windows (with backslashes instead of forward slashes).

Note that in all cases you can "catch" the output of `subprocess` so that you can then use the output within your python script. A simple example, where the output is a platform-dependent directory path, is:

```
MyPath = subprocess.os.path.join('directory', 'subdirectory', 'file')
```

Explore what `subprocess` can do by tabbing `subprocess.`, and also for submodules, e.g., type `subprocess.os.` and then tab.

6.6.2 Running `R`

R is likely an important part of your workflow, for example for statistical analyses and pretty plotting (hmmm... `ggplot2`!). Try the following.

Create an R script file called `TestR.R` in your `CMEECourseWork/Week6/Code` with the following content:

```
print("Hello, this is R!")
```

Then, create `TestR.py` in `CMEECourseWork/Week6/Code` with the following content :

```
import subprocess
subprocess.Popen("/usr/lib/R/bin/Rscript --verbose TestR.R > \
../Results/TestR.Rout 2> ../Results/TestR_errFile.Rout",\
 shell=True).wait()
```

Note the backslashes — this is so that `python` can read the mutiline script as a single line.

It is possible that the location of `RScript` is different in your Ubuntu install. To locate it, try `find /usr -name 'Rscript'` in the linux terminal (not in `python`!).

Now run `TestR.py` (or `%cpaste`) and check `TestR.Rout` and `TestR_errorFile.Rout`.

Also check what happens if you run (type directly in `ipython` or `python` console):

```
subprocess.Popen("/usr/lib/R/bin/Rscript --verbose NonExistScript.R > \
../Results/outputFile.Rout 2> ../Results/errorFile.Rout", \
shell=True).wait()
```

What do you see on the screen? Now check `outputFile.Rout` and `errorFile.Rout`.

6.6.3 Practicals

As always, test, add, commit and push all your new code and data to your git repository.

Using `os` problem 1

Open `using_os.py` and complete the tasks assigned
(hint: you might want to look at `subprocess.os.walk()`)

Using `os` problem 2

Open `fmr.R` and work out what it does; check that you have `NagyEtAl1999.csv`. Now write python code called `run_fmr_R.py` that:

- Runs `fmr.R` to generate the desired result
- `run_fmr_R.py` should also print to the python screen whether the run was successful, and the contents of the R console output

6.7 Practicals wrap-up

1. Review and make sure you can run all the commands, code fragments, and scripts we have till now and get the expected outputs — all scripts should work on any other linux laptop.
2. Include an appropriate docstring (if one is missing) at the beginning of *each* of each of the python script / module files you have written, as well as at the start of every function (or sub-module) in a module.
3. Also annotate your code lines as much and as often as necessary using #.
4. Keep all files organized in `CMEECourseWork`.
5. `git add`, `commit` and `push` all your week's code and data to your git repository by next Wednesday.

6.8 Readings and Resources

- `docs.python.org/2/howto/regex.html`
- Google's short class on regex in python:
  `https://developers.google.com/edu/regular-expressions`

- www.regular-expressions.info has a good intro, tips and a great array of canned solutions
- Use and abuse of regex:
  www.codinghorror.com/blog/2005/02/regex-use-vs-regex-abuse.html
- www.matplotlib.org/
- For SciPy, the official documentation is great:
  docs.scipy.org/doc/scipy/reference/
  Read about the scipy modules you think will be important to you.
- The "ecosystem" for Scientific computing in python: http://www.scipy-lectures.org/
- A Primer on Scientific Programming with Python http://link.springer.com/book/10.1007%2F978-3-642-54959-5; Multiple copies of this book are available from the central library and can be requested to Silwood from the IC library website.
- Many illustrative examples at http://wiki.scipy.org/Cookbook (including Lotka-Volterra!)
- "The Definitive Guide to SQLite" is a pretty complete guide to SQLite and freely available from http://evalenzu.mat.utfsm.cl/Docencia/2012/ SQLite.pdf
- For databses in general, try the Stanford Introduction to Databases course: https://www.coursera.org/course/db