Introduction to Biological Computing: MSc/MRes Computational Methods in Ecology & Evolution

(with modules for MSc/Mres EEC, EA, & NHM)

Imperial College London, 2014–15

Samraat Pawar (s.pawar@imperial.ac.uk)
(*with inputs from many others!*)

November 12, 2014

Contents

Chapter 0

Introduction

Donald Knuth, 1995: *Science is what we understand well enough to explain to a computer. Art is everything else we do*

0.1 What is this document all about?

This document contains the content of the modules on Biological Computing in the MSc/MREs Courses on Computational Methods in Ecology and Evolution (CMEE) at Silwood Park, Department of Life Sciences, Imperial College London. These notes are also available to the other Masters courses, including EA, EEC, and NHM, and the chapters on R will be necessary for these courses. Please look at your respective class handbooks to determine when the R weeks are scheduled.

This document is accompanied by data and code on which you can practice your skills in your own time and during the Practical sessions. These materials are available (and will be updated regularly) at: `https://bitbucket.org/mhasoba/cmee2014masterepo/`.

It is important that you work through the exercises and problems in each chapter/section. This document does not tell you every single thing you need to know to perform the exercises in it. In programming, you learn far more from trying to solve problems than from reading about how others have solved them — that is, you have license to google it! You will be provided guidelines for what makes good or efficient solutions. Later, when you have submitted your exercises and practicals (only relevant to the CMEEs, but the rest should have a go!), you will be provided solutions.

0.2 Conventions used in this document

You will find all command line/console arguments, code snippets and output in coloured boxes like this:

```
$ ls
```

The specific prompt ($ in this case, belonging to the UNIX terminal) will vary with the programming language/console (`$` for UNIX, »> for Python, > for R, etc.)

You will type the commands/code that you see in such boxes into the relevant command line (or copy-paste, but not recommended!). I have aimed to make the content of this module computer platform (Mac, Linux or PC) independent. Also note that:

⋆ Lines marked with a star like this will be specific instructions for you to follow

0.3 A note on being organized

In this course, you will write plenty of code, deal with different data files, and produce text and graphic outputs. Please keep all your code, data inputs and results outputs organized in separate directories named `Code`, `Data`, `Results` (or equivalent) respectively.

Note that throughout this document, directory paths will be specified in UNIX (Mac, Linux) style, using / instead of the \ used in Windows. Also, in general, we will be using relative paths throughout the exercises and practicals (more on this later).

0.4 to IDE or not to IDE?

An iteractive Development Environment (IDE) is a text editor with frills that can make life easy by auto-formatting code, Running code, Allowing a graphic view of the workspace (your active functions, variables, etc.), graphic debugging and profiling (you will see these delightful things later), and allowing integrated version control (e.g., using `git`). You will benefit a lot of you use a good code editor that can also offer an IDE (e.g., emacs, vim, geany). At the very least, you IDE should offer:

- Autoindentation
- Automatic code wrapping (e.g., keeping lines <80 characters long)
- Syntax highlighting (e.g., commands vs. variables)
- Code folding
- Keyboard control of commenting/uncommenting, code wrapping, etc.
- Embedded terminal / shell / commandline console
- Sending commands to terminal / shell

And of you end up usilg multiple languages, you will want and IDE that can handle them. For example, RStudio cannot handle more than a fixed set of 2-3 languages. I use `geany`, which has many pluins that make multi-language (multilingual?) code development much easier.

Chapter 1

Biological Computing in Python I

1.1 Outline of the `python` weeks

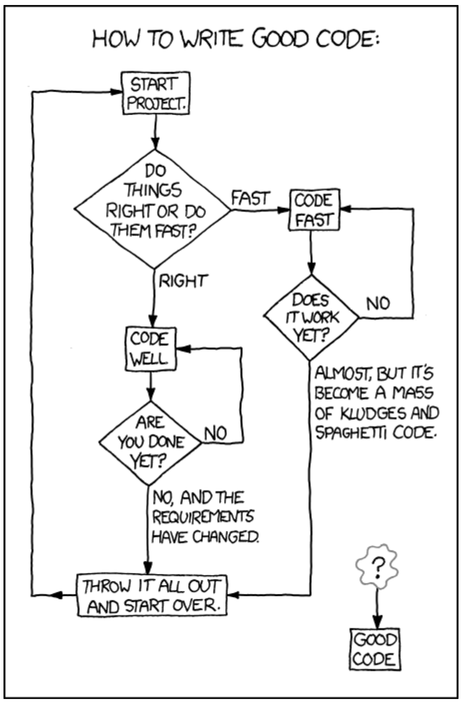
The content and structure of the Python weeks are geared towards teaching you:

- Scientific programming in biology using `python`
- Basics of `python`
- To write and run `python` code, understand and implement control flows
- Learn to use ipython
- Writing, debugging, using, and testing `python` functions
- Efficient numerical programming in `python`
- Regular expressions, certain `python` packages, `python` with databases
- Using `python` to run other “stuff”, and to patch together data analyses or numerical simulation workflows

1.2 Why write computer programs?

Here are my top reasons:

- Programs can do anything (that can be specified)
- As such, no software is typically available to perform exactly the analysis we are planning
- Permits success despite complexity, through modularization and precise specification
- Reproducibility – just re-run the code!
- Modularity – break up your complex analysis in smaller pieces
- Organised thinking – writing code requires this!
- Career prospects – good, scientific coders are in short supply in all fields!


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

`www.xkcd.com`

1.3 Which languages?

There are several hundred programming languages are currently available – which ones should a biologist choose? Ideally, in Scientific computing in biology, one would like to know:

1. A fast, compiled (or semi-compiled) "procedural" language (e.g., `C`) – not so important for most of you

2. A modern, easy-to-write, interpreted (or semi-compiled) language that is still quite fast, like `python`

3. A mathematical/statistical software with programming and graphing capabilities like `R` (also, `mathematica`, `MATLAB`)

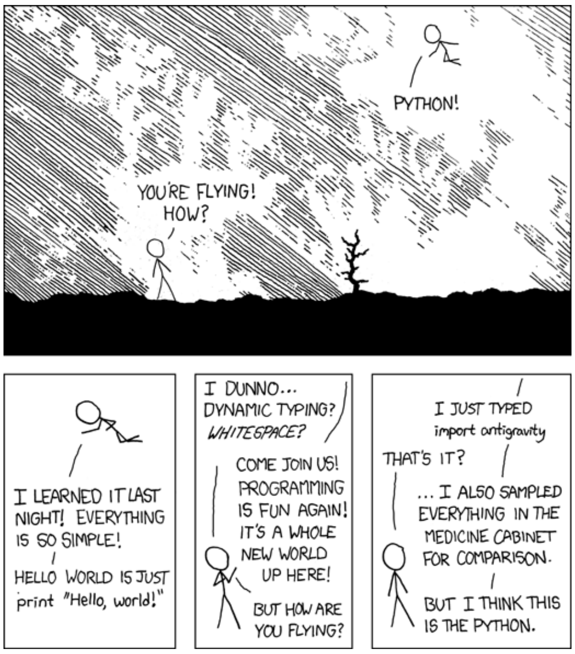
All these because one language doesn't fit all purposes!

1.4 Why `python`?

`python` was designed with readability and re-usability in mind. Time taken by programming + debugging + running is relatively lower in `python` than less intuitive or cluttered languages (e.g., `FORTRAN`, `perl`).

1.4.1 The Zen of python

Open a terminal and type


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

www.xkcd.com

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

1.5 Installing python

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

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

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

Note: You can enable “Send Selection to Terminal” using <Primary>Return in geany

1.6 python warmup

Open a terminal (ctrl+alt+t) and type python. Then:

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

>>> 2 * 2
4

>>> 2 / 2
1

>>> 2 / 2.0
```

```
1.0

>>> 2 / 2.
1.0

>>> 2 > 3
False

>>> 2 >= 2
True
```

Now let's switch to something called `ipython`! We will see later why. Type `ctrl+D`: this will exit you from the python shell and you will see the bash prompt again. Then type `ipython`. You should now see (after some text):

```
In [1]:
```

This is the `interactive python` shell (ergo, `ipython`. If you don't like the blue prompt, type `%colors linux`

1.6.1 Variable types

Now, let's continue our warmup (ignore the prompt numbering)

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

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

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

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

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

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

1.6.2 python operators

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

1.6.3 Assigning and manipulating variables

```
 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
```

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
```

Note that in python, the type of a variable is determined when the program or command is running (dynamic typing) (like R, unlike C or FORTRAN)

1.7 `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!

1.7.1 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']
```

1.7.2 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')
```

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 treat the species associations as sacroscant!

Tuples contain immutable sequences, but their elements can be mutable:

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

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

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

1.7.3 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)

1.7.4 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!

1.7.5 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]]
```

1.7.6 python with strings

```
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'
```

1.8 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`:

```
#############################
# 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
```

The `csv` package makes it easy to manipulate CSV files (get `testcsv.csv` from `CMEEMasteRepo`). Type the followinf 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()
```

1.9 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)`

1.9.1 Writing functions

Let's start with a "boilerplate" 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
import control_flow
# constants can go here

# functions can go here
def main(argv):
    print 'This is a boilerplate' # NOTE: indented using one tab
    # print 'control_flow says: ' + control_flow.even_or_odd(10) + '!'# NOTE:↩
        indented using two tabs
    return 0
```

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

A p̂ython function consists of the following elements:

- First line tells computer where to look for python
- Triple quotes start a "docstring" comment (more on this later)
- "__" signal "internal" variables (never name your variables so!)
- Last few lines tad esoteric, but `main` is important (look up `http://ibiblio.org/g2swap/byteofpython/read/module-name.html`)

Type the above boilerplate and save as `boilerplate.py` in `CMEECourseWork/Week2/Code` and then cd to the directory and run the code:

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

After writing and saving `boilerplate.py`, in terminal, type:

```
$ ipython boilerplate.py
$ ipython -i boilerplate.py
```

The first command launches python and executes the file. The cecond command launches python and "imports" the script (more on this later). You can use `python` instead of `ipython`

1.9.2 Components of the `python` function

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

`sys.argv` – try this in a file called `sysargv.py`:

```
import sys
print "Name of the script is: ", sys.argv[0]
print "Number of arguments are: ", 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
```

OK, what about

```
if __name__ == "__main__"
```

This directs the `python` interpreter to set the special `__name__` variable to have a value "`__main__`"

```
def main(argv):
    print 'This is a boilerplate' # NOTE: indented using two tabs
    print 'control_flow says: ' + control_flow.even_or_odd(10) + '!'
    return 0
```

Run it!

OK, finally, what about this bit:

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

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

1.9.3 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.

1.9.4 Running `python` code from terminal or bash shell

We have been running scripts from our beloved terminal or bash shell. To execute script file from within the python shell, use `execfile("filetoload.py")`. In IPython, use: `%run filetoload.py`

1.10 Control statements

OK, let's get deeper into `python` functions. To begin, first copy and rename boilerplate.py:

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

Then type the following script:

```
#!/usr/bin/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:

```
$ ipython -i control_flow.py
```

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

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

1.10.1 Control flow exercises

```
# 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
    if x > y:
        tmp = y
        y = x
        x = 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)
```

1.11 Loops

```
# 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!
```

1.11.1 List comprehensions

Python offers a way to combine loops, functions and logical tests in a single line of code:

```
## 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 = set()
for taxon in taxa:
    if is_an_oak(taxon):
        oaks.add(taxon)
print oaks
```

```
##Using list comprehensions
oaks = set([t for t in taxa if is_an_oak(t)])
print oaks

##Get names in UPPER CASE using for loops
oaks = set()
for taxon in taxa:
    if is_an_oak(taxon):
        oaks.add(taxon.upper())
print oaks

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

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

1.12 Practical 1.1: Make sure the basics work

1. Review and make sure you can run all the commands, code fragments, and functions we have covered today and get the expected outputs
2. Run `boilerplate.py` and `control_flow.py` from the terminal (try both python and ipython)
3. Run `boilerplate.py` and `control_flow.py` from within the python and ipython shells
4. Open and complete the tasks/exercises in `lc1.py, lc2.py, dictionary.py, tuple.py` (in any order)
5. Keep all code files organized in `CMEECourseWork/Week2/Code`
6. Version control all your work; the updated bitbucket git repository should contain: `boilerplate.py`, `control_flow.py`, `scope.py`, `lc1.py`, `lc2.py`, `dictionary.py`, `tuple.py`

1.13 Readings and Resources

- The Zen of python: open a python shell and type `import this`
- 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` `www.wiki.ipython.org/Cookbook`

Chapter 2

Biological Computing in Python II

2.1 `ipython`

`ipython` stands for `i`nteractive `python` shell. In IPython, you can `TAB` everything — a lot will be revealed!

In particular, `TAB` leads to autocompletion. Also `TAB` after "." reveals object's functions and attributes:

```
In []: alist = ['a', 'b', 'c']

In []: alist. #TAB now!
alist.append   alist.extend   alist.insert   alist.remove
alist.sort     alist.count    alist.index    alist.pop
alist.reverse

In []: adict = {'mickey': 100, 'mouse': 3.14}

In []: adict.
adict.clear       adict.items       adict.pop
adict.viewitems   adict.copy        adict.iteritems
adict.popitem     adict.viewkeys    adict.fromkeys
adict.iterkeys    adict.setdefault  adict.viewvalues
adict.get         adict.itervalues  adict.update
adict.has_key     adict.keys        adict.values
```

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
`%cpaste`	Paste indented code into IPython — very useful when you want to run just part of a function (indentation and all)

Let's try the `%cpaste` function. First, type the following code in a temporary file:

```
def PrintNumbers(x):
    for i in range(x):
        print x
    return 0
```

Now launch ipython and type:

```
In []: x = 11

# Now copy the for loop from the file.
# Note that there is extra indentation!

In []: %cpaste
Pasting code; enter '--' alone on the line to stop
or use Ctrl-D.
:     for i in range(x):
:         print x
:--
11
11
11
11
11
11
11
11
11
11
11
```

Another useful feature is the question mark:

```
In []: ?adict
Type:       dict
Base Class: <type 'dict'>
String Form:{'mickey': 100, 'mouse': 3.14}
Namespace:  Interactive
Length:     2
Docstring:
dict() -> new empty dictionary
dict(mapping) -> new dictionary initialized from a mapping
object's
    (key, value) pairs
dict(iterable) -> new dictionary initialized as if via:
    d = {}
    for k, v in iterable:
    d[k] = v
dict(**kwargs) -> new dictionary initialized with the
name=value pairs
    in the keyword argument list.
For example:  dict(one=1, two=2)
```

2.2 Functions and Modules

Ideally you should aim to compartmentalize your code into a bunch of functions, typically written in a single `.py` file: this is a Python "module". 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 (see next slide)

A Python package is simply a directory of Python modules (quite like an R package)

2.2.1 Importing Modules

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

- `import my_module`, then functions in the module can be called as `my_module.one_of_my_functior`
- `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_`

2.3 Unit testing

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 time is spent debugging (coming up!), not writing code. Unit testing prevents the most common mistakes and yields reliable code. Indeed, there are many reasons for testing:

- Can you prove (to you) that you code do what you think it do?
- 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.

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:

```python
#!/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`

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 (*why?*).

OK, so you unit-tested, let's go look at life through beer-goggles... BUT NO! WHAT IS THIS I SEE?! A BLOODY BUG!

2.4 Debugging

Bugs happen! 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.
    import pdb; pdb.set_trace()
    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:

`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
```

2.4.1 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 an 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...)!

2.5 Practical 2.2

As always, add and commit all your new (functional) code and data to the version control repository: `basic_io.py, basic_csv.py, testout.csv, bodymass.csv, tesp.p, test_control_flow.py, debugme.py, profileme.py`.

Now,

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 write a good doctest, you will note that you found another error that you might not have just by debugging (hint: what happens if you try the doctest with 'Quercuss' instead of 'Quercus'?)

5. How would you fix the new error you found using the doctest?

2.6 The meaning of life (and programming)

...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...

2.7 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". 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`:

```python
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 type `%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...)

2.8 Regular expressions in `python`

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

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

A regex may consist of a combination of "metacharacters" (modifiers) and "regular" or literal characters. There are 14 metacharacters: [ ] ( ) \ˆ $. | ? * +

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
- `[ˆFf]` means anything except upper or lower case f and `[ˆa-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) 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 excl. 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

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.

Always put `r` in front of your regex — it tells python to read the regex in its "raw" (literal) form. Let's try (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
 #
str = 'an example'
match = re.search(râĂŹ\w*\sâĂŹ, str)
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`:

```

str = 'an example'

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

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

```
# 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()
```

You can group regexes into meaningful blocks using parentheses:

```
str = '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&]*",str)

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&]*)",str)

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'
```

2.8.1 Exercises

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

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

r'^\d{1,2}\/\d{1,2}\/\d{4}$'

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

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

`www.xkcd.com/208/`

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

2.8.2 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!).

2.9 Practical 2.1

Bring under version control: `profileme.py, regexs.py`

2.10 Practical 2.2

Objective: 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` and make sure you understand what each line is doing (hint: you can use `pdb` to do this)
2. Convert `align_seqs.py` to a Python function that takes the DNA sequences as an input from a `.csv` file and saves the best alignment to along with corresponding score in a single text file (hint: remember `pickle`!)
3. Make sure you provide a test `.csv` file that the script can call
4. Extra Credit – align the `.fasta` sequences from Week 1!

2.11 Numerical computing in `python`

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

- Linear algebra
- Numerical integration
- Fourier transforms
- Interpolation
- Special functions (Incomplete Gamma, Bessel, etc.)
- Generation of random numbers
- Statistical functions

Let’s look at `scipy`. In the following, we will use the `array` class in `scipy` for data manipulations and calculations. Scipy arrays are similar in some respects to python lists, but are more naturally multidimensional, homogeneous in type, and allow efficient manipulations. So try this:

```
In [1]: import scipy

In [2]: a = scipy.array(range(5))

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

In [4]: a = scipy.array(range(5), dtype = float)

In [5]: a
Out[5]: array([ 0.,  1.,  2.,  3.,  4.])
```

```
In [6]: a.dtype
Out[6]: dtype('float64')

In [7]: x = scipy.arange(5)

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

In [9]: x = scipy.arange(5.)

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

In [11]: x. # TAB after x. to see methods you can apply to 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 [11]: x.tolist()
Out[11]: [0.0, 1.0, 2.0, 3.0, 4.0]

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

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

In [16]: mat.shape
Out[16]: (2, 2)

In [17]: mat[1]
Out[17]: array([2, 3])

In [18]: mat[0,0]
Out[18]: 0

In [19]: mat.ravel() # flatten!
Out[19]: array([0, 1, 2, 3])

In [20]: scipy.ones((4,2))
Out[20]:
```

```
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])

In [21]: scipy.zeros((4,2))
Out[21]:
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])

In [22]: scipy.identity(4)
Out[22]:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

In [23]: m = scipy.identity(4)

In [24]: m.reshape((8, 2))
Out[24]:
array([[ 1.,  0.],
       [ 0.,  0.],
       [ 0.,  1.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 1.,  0.],
       [ 0.,  0.],
       [ 0.,  1.]])

In [25]: m.fill(16)

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

Let’s perform some common operations on arrays:

```
In [1]: import scipy

In [2]: mm = scipy.arange(16)

In [3]: mm = mm.reshape(4,4)

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

# NOTE: Rows first

In [5]: mm.transpose()
```

```
Out[5]:
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`.

Two other particularly useful `scipy` sub-packages are `scipy.integrate` (what will I need this for?) `scipy.stats` (why not use `R` for this?). 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])
```

2.11.1 Exercise

Let's look at an example using `scipy.integrate`. Create `LV1.py` in your `Week2/Code` directory and run it.

2.12 Practical 2.3

Test and bring under version control: `LV1.py`. Then, convert `LV1.py` into another script called `LV2.py` that does the following:

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

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

2. Runs the Lotka-Volterra model with prey density dependence $rx(1 - \frac{x}{K})$
3. Saves the plot as `.pdf` in an external results directory (`Week2/Results`)

4. The chosen parameter values should show in the plot (e.g., $r = 1, a = .5$, etc)

You can change time length `t` too. Also include a script called `run_LV2.py` in `Code` that will run `LV2.py` with appropriate arguments

Extra credit if you also choose appropriate values for the paramaters such that both predator and prey persist with prey density dependence

Extra-extra credit if you can write a recursion version of the model in discrete time (what's this?) – it should do everything that `run_LV2.py` does

Extra-extra-extra credit if you can write a recursion version of the model in discrete time with a random gaussian fluctuation at each time-step (use `scipy.stats`)

2.13 Practical 2.4

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

2.14 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 best:
  `docs.scipy.org/doc/scipy/reference/`
  Read about the scipy modules you think will be important to you...
- Many illustrative examples at `http://wiki.scipy.org/Cookbook` (including Lotka-Volterra!)
- In general, good module-specific cookbooks are out there (e.g., `biopython`)

Chapter 3

Introduction to R

3.1 Outline of the the R module

The content and structure of the chapters in the R module are geared towards the following objectives:

- Give you a introduction to R syntax and programming conventions, assuming you have never set your eyes on R or any other programming language before — we will breeze through this as you have already been introduced to R in the Stats Week!
- Teach you principles of clean and efficient programming in R, including delightful things like vectorization and debugging.
- Teach you how to generate publication quality graphics in R — publication quality is thesis quality!
- Teach you how to develop reproducible data analyses "work flows" so (or anybody else) run and re-run your analyses, graphics outputs and all, in R.

You will use R a lot during the rest of your courses and probably your thesis and career — the aim is to lay down the foundations for you to become very comfortable with it!

s

3.2 What is `R`?

R is a freely available statistical software with strong programming capabilities. R has become incredibly popular in biology due to several factors: i) many packages are available to perform all sorts of statistical and mathematical analyses; ii) it has been developed and scrutinised by top level academic statisticians; iii) it can produce beautiful, publication-quality graphics; iv) it has a very good support for matrix-algebra.

3.3 Would you ever need anything other than R?

Although we can technically program in R, the programming environment is not the greatest: especially the way types are managed is problematic (e.g., often your matrix will become a vector if it has only one column/row!), and the way errors and warnings are handled and displayed (often unintelligibly!).

Nevertheless, being able to program R means you can develop and automate your statistical analyses and the generation of figures into a reproducible work flow (there's that term again!). However, if your work also includes extensive numerical simulations, manipulation of very large matrices, bioinformatics, or complex workflows including databases, you will be much better off if you *also* know another programming language that is more versatile, computationally efficient (like `python`, `perl` or `C`)).

In particular, `python` is recommended, as it is reasonably efficient and has very nice and clean syntax. With something like `python`, You can embed your R analysis work flow inside a more complex meta-workflow, for example, one that includes interfacing with the internet, manipulating/querying databases, and compiling LaTeXdocuments.

But for many of you, R will do the job!

3.4 Installing R

Linux/Ubuntu: run the following in terminal

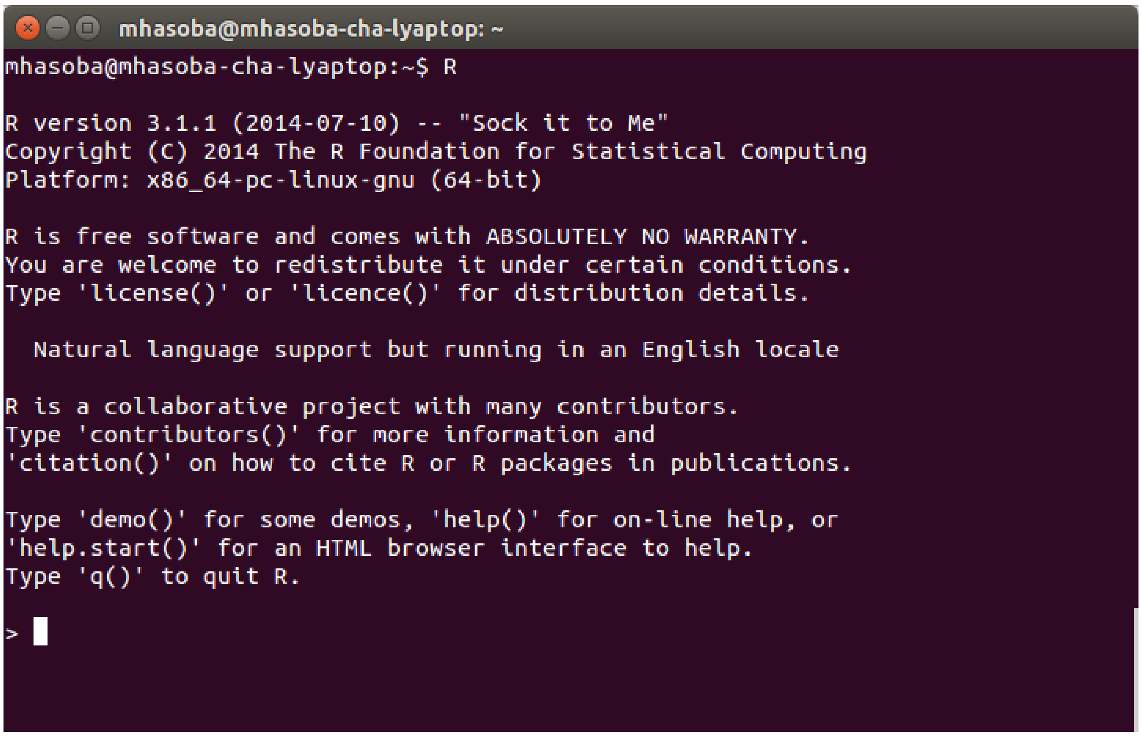
```
sudo apt-get install r-base r-base-dev
```

Mac OS X: download and install from `http://cran.r-project.org/bin/macosx/`

Windows: download and install from `http://cran.r-project.org/bin/windows/base/`

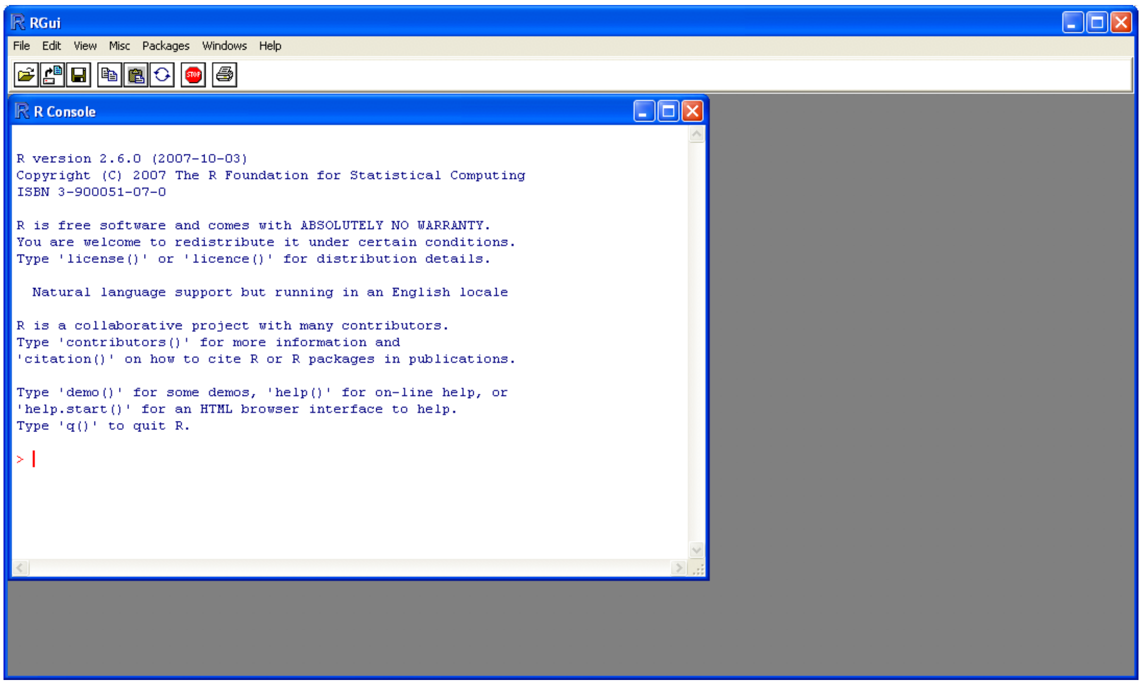
3.5 Getting started

Launch R (From Applications menu on Window or Mac, from terminal in Linux/Ubuntu) — it should look something like this (on Linux/Ubuntu or Mac terminal):



```
mhasoba@mhasoba-cha-lyaptop:~$ R

R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Or like this (Windows “console”, similar in Mac):



```
R version 2.6.0 (2007-10-03)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

You can also use an IDE (Interactive Development Environment) that can offer delights like syntax highlighting (google it!), such as RStudio, `geany`, `vim`, etc.

3.6 Useful R commands

`ls()`	list all the variables in the work space
`rm('a', 'b')`	remove variable(s) `a` and `b`
`rm(list=ls())`	remove all variable(s)
`getwd()`	get current working directory
`setwd('Path')`	set working directory to `Path`
`q()`	quit R
`?Command`	show the documentation of `Command`
`??Keyword`	search the all packages/functions with `Keyword`, "fuzzy search"

3.7 Some Basics

Try out the following in the R console:

```
> a <- 4  # assignment
> a
[1] 4
> a*a  # product
[1] 16
> a_squared <- a*a
> sqrt(a_squared) # square root
[1] 4
> v <- c(0, 1, 2, 3, 4) # c: "concatenate"
```

`c()` (concatenate) is one of the most commonly used functions — DonâĂŹt forget it! (try `?c`)

Note that any text after a "#" is ignored by R — handy for commenting. In general, please comment your code and scripts, for *everybody's* sake!

```
> v # Display the vector variable you created
[1] 0 1 2 3 4
> is.vector(v) # check if it's a vector
[1] TRUE
> mean(v) # mean
[1] 2
> var(v) # variance
[1] 2.5
> median(v) # median
[1] 2
> sum(v) # sum all elements
[1] 10
> prod(v + 1) # multiply
[1] 120
> length(v) # length of vector
[1] 5
```

3.7.1 Variable names and Tabbing

```
> wing.width.cm <- 1.2 #Using dot notation
> wing.length.cm <- c(4.7, 5.2, 4.8)
```

Using tabbing with dot notation can be handy. Type:

```
> wing.
```

And then hit the `tab` key. This is handy, but good style and readability is more important than just convenient variable names. Variable names should be as obvious as possible, not over-long!

3.7.2 R likes `E` Notation

```
> 1E4
[1] 10000
> 1e4
[1] 10000
> 5e-2
[1] 0.05
```

R uses *E* notation to print very large or small numbers:

```
> 1E4 ^ 2
[1] 1e+08
> 1 / 3 / 1e8
[1] 3.333333e-09
```

3.7.3 Operators

The usual operators are available in R (slight differences from `python`):

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

3.7.4 When things go wrong

Syntax errors are those where you've just made a typing mistake. Here are some common problems:

- missing close bracket leads to continuation line.

  ```
  > x <- (1 + (2 * 3)
  +
  ```

  Hit Ctrl C (see below) or keep typing!

- Too many parentheses: `2 + (2*3))`
- wrong/mismatched brackets (see next subsection).
- Do not mix double quotes and single quotes.
- When things seem to take too long, try `Ctrl + C`

3.7.5 Types of parentheses

R has a somewhat confusing array of parentheses that you need to get used to:

- `f(3,4)` – call the function f, with arg1=3, arg2=4.
- `a + (b*c)` – use to enforce order over which statements are executed.
- { expr1; expr2; ...exprn } – group a set of expressions into one compound expression. Value returned is value of last expression; used in looping/conditionals.
- `x[4]` – get the 4th element of the vector x.
- `l[[3]]` – get the 3rd element of some list l, and return it. (compare with l[3] which returns a list with just the 3rd element inside) (more on lists in next section)

3.8 Data types

Like `python` (why `python`? Ask the CMEEs!), R comes with data-types. Mastering these will help you write better, more efficient programs and also handle diverse between datasets. Now get back into R (if you quit R using `q()`), and type:

3.8.1 Vectors

Vectors are a fundamental object for R. Scalars (single numbers) are treated as vector of length 1.

```
> a <- 5
> is.vector(a)
[1] TRUE
> v1 <- c(0.02, 0.5, 1)
> v2 <- c("a", "bc", "def", "ghij")
> v3 <- c(TRUE, TRUE, FALSE)
```

3.8.2 Matrices and arrays

R has many functions to manipulate matrices (two-dimensional vectors) and arrays (multi-dimensional vectors).

```
> m1 <- matrix(1:25, 5, 5)
> m1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
> m1 <- matrix(1:25, 5, 5, byrow=TRUE)
> m1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
> dim(m1)
[1] 5 5
> m1[1,2]
[1] 2
> m1[1,2:4]
[1] 2 3 4
> m1[1:2,2:4]
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    7    8    9
> arr1 <- array(1:50, c(5, 5, 2))
> arr1
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   26   31   36   41   46
[2,]   27   32   37   42   47
[3,]   28   33   38   43   48
[4,]   29   34   39   44   49
[5,]   30   35   40   45   50
```

3.8.3 Data frames

This is a very important data type that is peculiar to R. It is great for storing your data. Basically, it's a two-dimensional table in which each column can contain a different data type (e.g., numbers, strings, boolean). You can think of a dataframe as a spreadsheet. Data frames are great for plotting your data, performing regressions and such. Later (Chapter 2) you will see that fancy plotting using `ggplot` demands the use of dataframes. Now try the following:

```
> Col1 <- 1:10
> Col1
 [1]  1  2  3  4  5  6  7  8  9 10
> Col2 <- LETTERS[1:10]
> Col2
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
> Col3 <- runif(10) # 10 random num. from uniform
> Col3
 [1] 0.29109 0.91495 0.64962 0.95503 0.26589 0.02482 0.59718
 [8] 0.99134 0.98786 0.86168
> MyDF <- data.frame(Col1, Col2, Col3)
> MyDF
     Col1 Col2      Col3
1      1    A 0.2910981
2      2    B 0.9149558
3      3    C 0.6496248
4      4    D 0.9550331
5      5    E 0.2658936
6      6    F 0.0248217
7      7    G 0.5971868
8      8    H 0.9913407
9      9    I 0.9878679
10    10    J 0.8616854
> names(MyDF) <- c("A.name", "another", "another.one")
> MyDF
     A.name another another.one
1         1       A   0.2910981
2         2       B   0.9149558
3         3       C   0.6496248
4         4       D   0.9550331
5         5       E   0.2658936
6         6       F   0.0248217
7         7       G   0.5971868
8         8       H   0.9913407
9         9       I   0.9878679
10       10       J   0.8616854
> MyDF$A.name
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[,1]
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[c("A.name","another")]
     A.name another
1         1       A
2         2       B
3         3       C
4         4       D
5         5       E
6         6       F
7         7       G
8         8       H
9         9       I
10       10       J
> class(MyDF)
[1] "data.frame"
> str(MyDF) # a very useful command!
'data.frame':   10 obs. of  3 variables:
 $ A.name     : int  1 2 3 4 5 6 7 8 9 10
 $ another    : Factor w/ 10 levels "A","B","C","D",..
 $ another.one: num  0.291 0.915 0.65 0.955 0.266 ...
```

3.8.4 Lists

A list is used to collect a group of objects of different sizes and typesA list is simply an ordered collection of objects (that can be other variables) (a bit like `python` lists!).

```
> l1 <- list(names=c("Fred","Bob"), ages=c(42, 77, 13, 91))
> l1
$names
[1] "Fred" "Bob"

$ages
[1] 42 77 13 91

> l1[[1]]
[1] "Fred" "Bob"
> l1[[2]]
[1] 42 77 13 91
> l1[["ages"]]
[1] 42 77 13 91
> l1$ages
[1] 42 77 13 91
```

You can build lists of lists too. Lists are often returned as the result of a complex function (e.g. linear model fitting using `lm()`) to return all relevant information in one object.

3.9 Variable Types, Type Conversion and Special Values

There are different kinds of data variable types such as integer, float (including real numbers), and string (e.g., text words). Beware of the difference between NA (Not Available) and NaN (Not a Number).

```
> as.integer(3.1)
[1] 3
> as.real(4)
[1] 4
> as.roman(155)
[1] CLV
> as.character(155)
[1] "155"
> as.logical(5)
[1] TRUE
> as.logical(0)
[1] FALSE
> b <- NA
> is.na(b)
[1] TRUE
> b <- 0./0.
> b
[1] NaN
```

```
> is.nan(b)
[1] TRUE
> b <- 5/0
> b
[1] Inf
> is.nan(b)
[1] FALSE
> is.infinite(b)
[1] TRUE
> is.finite(b)
[1] FALSE
> is.finite(0/0)
[1] FALSE
```

3.10 Creating and Manipulating Data structures

3.10.1 Sequences

The `:` operator creates vectors of sequential integers:

```
> years <- 1990:2009
> years
[1] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
[11] 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009

> years <- 2009:1990 # or in reverse order
> years
[1] 2009 2008 2007 2006 2005 2004 2003 2002 2001 2000
[11] 1999 1998 1997 1996 1995 1994 1993 1992 1991 1990
```

For sequences of fractional numbers, you have to use `seq()` :

```
> seq(1, 10, 0.5)
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  ↩
     8.0
[16]  8.5  9.0  9.5 10.0
```

You can also `seq(from=1,to=10, by=0.5)` OR `seq(from=1, by=0.5, to=10)` with the same effect (try it) — this explicit, “argument matching” approach is partly why R is so popular.

3.10.2 Strings and Pasting

R’s string handling ain’t elegant or pretty (unlike `python`!), but it works:

```
> species.name <- "Quercus robur" #double quotes
> species.name
[1] "Quercus robur"
> species.name <- 'Fraxinus excelsior' #single quotes
> species.name
```

```
[1] "Fraxinus excelsior"
> paste("Quercus", "robur")
[1] "Quercus robur"
> paste("Quercus", "robur",sep = "") #Get rid of space
"Quercusrobur"
> paste("Quercus", "robur",sep = ", ") #insert comma to separate
```

And as is the case with so many R functions, pasting works on vectors:

```
> paste('Year is:', 1990:2000)
 [1] "Year is: 1990" "Year is: 1991" "Year is: 1992" "Year is: 1993"
 [5] "Year is: 1994" "Year is: 1995" "Year is: 1996" "Year is: 1997"
 [9] "Year is: 1998" "Year is: 1999" "Year is: 2000"
```

Note that this last example creates a vector of 11 strings.

3.10.3 Indices and Indexing

Every element of a vector in R has an order: the first value, second, third, etc. To illustrate this, type:

```
> MyVar <- c( 'a' , 'b' , 'c' , 'd' , 'e' ) # create a simple vector
```

Then, square brackets extract values based on their position in the order:

```
> MyVar[1] # Show element in first position
[1] "a"
> MyVar[4]
[1] "d" # Show element in fourth position
```

The values in square brackets are called “indices” — they give the index (position) of the required value. We can also select sets of values in different orders, or repeat values:

```
> MyVar[c(3,2,1)] # reverse order
[1] "c" "b" "a"
MyVar[c(1,1,5,5)] # repeat indices
[1] "a" "a" "e" "e"
```

So you can manipulate vectors by indexing:

```
> v <- c(0, 1, 2, 3, 4) # Re-create the vector variable v
> v[3] # access one element
[1] 2
> v[1:3] # access sequential elements
[1] 0 1 2
> v[-3] # remove elements
[1] 0 1 3 4
> v[c(1, 4)] # access non-sequential
[1] 0 3
```

3.10.4 Recycling

When vectors are of different lengths, R will recycle the shorter one to make a vector of the same length:

```
a <- c(1,5) + 2
x <- c(1,2); y <- c(5,3,9,2)
x + y
x + c(y,1)  ## somewhat strange!
```

Recycling is convenient, but dangerous!

3.10.5 Basic vector-matrix operations

```
> v2 <- v
> v2 <- v2*2 # whole-vector operation
> v2
[1] 0 2 4 6 8
> v * v2 # product element-wise
[1]  0  2  8 18 32
> t(v) # transpose the vector
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    2    3    4
> v %*% t(v) # matrix/vector product
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    1    2    3    4
[3,]    0    2    4    6    8
[4,]    0    3    6    9   12
[5,]    0    4    8   12   16
> v3 <- 1:7 # assign using sequence
> v3
[1] 1 2 3 4 5 6 7
> v4 <- c(v2, v3) # concatenate vectors
> v4
 [1] 0 2 4 6 8 1 2 3 4 5 6 7
> q() # quit
```

3.11 Your R Analysis Workflow

3.11.1 The Working Directory

Before we go any further, let's get ourselves organized. In R, type:

```
> getwd()
```

This tells you what the current "working directory" is.

In using R for an analysis, you will likely use and create several files. This means that it is sensible to create a folder (directory) to keep all code files together. You can then set R to work from this directory, so that files are easy to find and run — this will be your working directory. So now do the following:

- ⋆ Create a directory called `Week5` in an appropriate location (For CMEEs, in `CMEECourseWork` for others, some place you can remember!
- ⋆ Create subdirectories within `CMEECourseWork/Week5` called `Code`, `Data`, and `Results`

You can create directories using `dir.create()` within R

Use relative paths Using relative paths in in your R scripts and code will make your code computer independent and your life better! For example, in R, `../Data/mydata.txt` specifies a file named `mydata.txt` located in the "parent" of the current directory. Now, set the working directory to be `Week5/Code`:

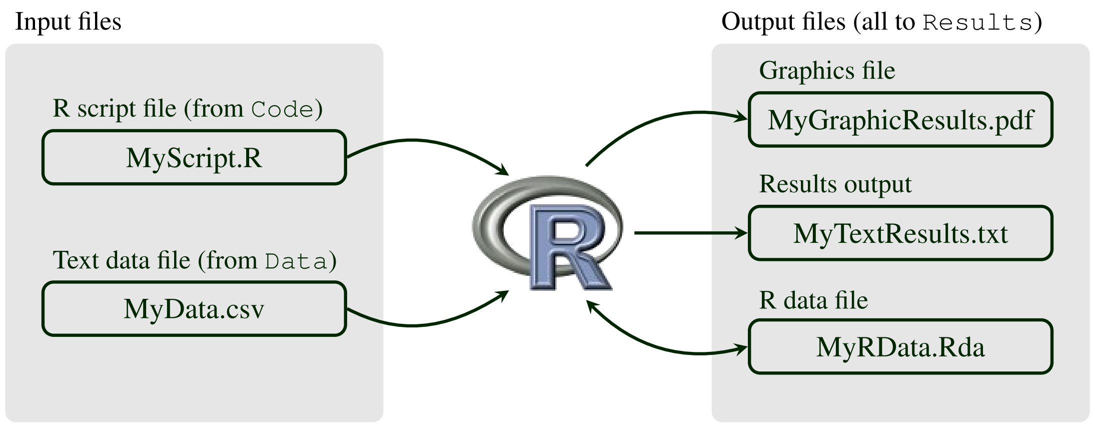
```
> setwd("FullPathUptoHere/Week5/Code")
> dir()
```

Note that `FullPathUptoHere` is not to be taken literally and entered! For example, if you created `Week5` in `H:`, you would use `setwd("H:/MyICStatsModules/Code"`

For CMEE, it would be `setwd("FullPathUptoHere/CMEECourseWork/Week5/Code")`

3.12 The R analysis workflow

Your typical R analysis workflow will be as follows:



Some details on each kind of file:

R script files These are plain text files containing all the R code needed for an analysis. These should always be created with a simple text editor like Notepad (Windows), TextEdit (MacOS) or Geany (Linux) and saved with the extension `*.R`. We will use the built-in editor in R in this class. Alternatively, if RStudio is a good option because it also works across platforms. Try it. A big advantage of something like RStudio is that you will get syntax highlighting, which is very handy and will make R progrmamming far more convenient and error-free. You should *never* use Word to save or edit these files as R can only read code from plain text files.

Text data files These are files of data in plain text format containing one or more columns of data (numbers, strings, or both). Although there are several format options, we will tyoucally be using csv files, where the entries are separated by commas. These are easy to create and export from Excel (if that's what you use...).[1]

Results output files These are a plain text files contain your results, such the the summary of output of a regression or ANOVA analysis. Typically, you will putput your results in a table format where the columns are separated by commas (csv) or tabs (tab-delimited)

Graphics files R can export graphics in a wide range of formats. This can be done automatically from R code and we will look at this later but you can also select a graphics window and click 'File ▷ Save as...'

Rdata files You can save any data loaded or created in R, including model outputs and other things, into a singleRdata file. These are not plain text and can only be read by R, but can hold all the data from an analysis in a single handy location. I never use these, but you can, if you want.

3.13 Importing and Exporting Data

Now we are ready to see how to import and export data in R, typically the first step of your analysis. The best option is to have your data in a comma separated value text file or in a tab separated file. Then, you can use the function `read.csv` (or `read.table`) to import your data (note the relative paths!):

```
> MyData <- read.csv("../Data/trees.csv")
> head(MyData) # Have a quick look at the data frame
> str(MyData) # Have a quick look at the column types
> MyData <- read.csv("../Data/trees.csv", header = TRUE) # with headers
> MyData <- read.table("../Data/trees.csv", sep = ',',
header = TRUE) # A more general way
> head(MyData)
> MyData <- read.csv("../Data/trees.csv", skip = 5) # skip first 5 lines
```

Note that the resulting `MyData` in your workspace is a R dataframe. You can also save your data frames using `write.table` or `write.csv`:

```
> write.csv(MyData, "../Results/MyData.csv")
> dir("../Results/") # Check if it worked
> write.table(MyData[1,], file = "../Results/MyData.csv",append=TRUE) # ↩
   append
> write.csv(MyData, "../Results/MyData.csv", row.names=TRUE) # write row ↩
   names
> write.table(MyData, "../Results/MyData.csv", col.names=FALSE) # ignore col ↩
   names
```

[1]If you are using a computer from elsewhere in the EU, Excel may use a comma ($\pi = 3,1416$) instead of a decimal point ($\pi = 3.1416$). In this case, *csv* files may use a semi-colon to separate columns and you can use the alternative function *read.csv2()* to read them into R.

3.14 Writing Functions

R lets you write your own functions. The syntax is quite simple, with each function accepting arguments and returning a value:

```
MyFunction <- function(Arg1, Arg2){

  ## statements involving Arg1, Arg2

  return (ReturnValue)

}
```

⋆ Type the following in a script file called `TreeHeight.R`, save it in your `Code` directory and run it using `source`:

```
# This function calculates heights of trees
# from the angle of elevation and the distance
# from the base using the trigonometric formula
# height = distance * tan(radians)
#
# Arguments:
# degrees      The angle of elevation
# distance     The distance from base
#
# Output:
# The height of the tree, same units as "distance"

TreeHeight <- function(degrees, distance)
{
  radians <- degrees * pi / 180
  height <- distance * tan(radians)
  print(paste("Tree height is:", height))
  return (height)
}

TreeHeight(37, 40)
```

3.15 Control statements

In R, you can write `if`, `then`, `else` statements, and `for` and `while` loops like any programming language. However, loops are slow in R, so use them sparingly.

⋆ Type the following in a script file called `control.R` (save it in your `Code` directory)

```
## If statement
a <- TRUE
if (a == TRUE){
    print ("a is TRUE")
} else {
    print ("a is FALSE")
}
```

```
## On a single line
z <- runif(1) ##random number
if (z <= 0.5) {
print ("Less than a quarter")}

## For loop using a sequence
for (i in 1:100){
    j <- i * i
    print(paste(i, " squared is", j ))
}

## For loop over vector of strings
for(species in c('Heliodoxa rubinoides',
                 'Boissonneaua jardini',
                 'Sula nebouxii'))
{
  print(paste('The species is', species))
}

## for loop using a vector
v1 <- c("a","bc","def")
for (i in v1){
    print(i)
}

## While loop
i <- 0
while (i<100){
    i <- i+1
    print(i^2)
}
```

3.16 Running R code

You can run the code you wrote in blocks to test and understand it:

- ⋆ Place the cursor on the first line of code and run it by pressing the keyboard shortcut (PC: ctrl+R, Mac: command+enter, Linux: ctrl+enter if you are using `geany`).

Typing in commands interactively or running in blocks is good for starters, but you will want to switch to putting your sequence of commands into a script file, and then ask R to run those commands. This is necessary also because you will want to just run your full analysis and outputs all the results. The way to run `*.R` script/code from the command line is to `source` it. This causes R to accept code input from a named file and run it:

```
> source("control.R") # Assuming you are in Code directory!
```

Note that you will need to add the directory path to the file name (`control.R` in the above example), if the file is not in your working directory. For example, `../Code/control.R` if you are in, say, `Data`.

3.17 Useful R Functions

There are a number of very useful functions available by default (in the “base packages”).

3.17.1 Mathematical

`log(x)`	Natural logarithm
`log10(x)`	Logarithm in base 10
`exp(x)`	e^x
`abs(x)`	Absolute value
`floor(x)`	Largest integer $< x$
`ceiling(x)`	Smallest integer $> x$
`pi`	π
`sqrt(x)`	$\sqrt{x}$
`sin(x)`	Sinus function

3.17.2 Strings

`strsplit(x,';')`	Split the string according to ';'
`nchar(x)`	Number of characters
`toupper(x)`	Set to upper case
`tolower(x)`	Set to lower case
`paste(x1,x2,sep=';')`	Join the strings inserting ';'

3.17.3 Statistical

`mean(x)`	Compute mean (of a vector or matrix)
`sd(x)`	Standard deviation
`var(x)`	Variance
`median(x)`	Median
`quantile(x,0.05)`	Compute the 0.05 quantile
`range(x)`	Range of the data
`min(x)`	Minimum
`max(x)`	Maximum
`sum(x)`	Sum all elements

3.17.4 Random number distributions

`rnorm(10, m=0, sd=1)`	Draw 10 normal random numbers with mean 0 and s.d. 1
`dnorm(x, m=0, sd=1)`	Density function
`qnorm(x, m=0, sd=1)`	Cumulative density function
`runif(20, min=0, max=2)`	Twenty random numbers from uniform [0,2]
`rpois(20, lambda=10)`	Twenty random numbers from Poisson(λ)

3.18 Packages

The main strength of R is that users can easily build packages and share them through `cran.r-project.org`. There are packages to do most statistical and mathematical analysis you might conceive, so check them out before reinventing the wheel! Visit `cran.r-project.org` and go to Packages to see a list and a brief description. To install a package, within R type `install.packages()` and choose the package to install.

3.19 Practical 1.1

Modify the script `TreeHeight.R` so that it does the following:

⋆ Loads `trees.csv` and calculates tree heights for all trees in the data. Note that the distances have been measured in meters. (Hint: use relative paths))

⋆ Creates a csv output file called `TreeHts.csv` in `Results` that contains the calculated tree heights along with the original data in the following format (only first two rows and headers shown):

```
"Species","Distance.m","Angle.degrees","Tree.Height.m"
"Populus tremula",31.6658337740228,41.2826361937914,25.462680727681
"Quercus robur",45.984992608428,44.5359166583512,46.094124200205
```

3.20 Readings

3.21 Readings

- Use the internet! Google “R tutorial”, and plenty will pop up. Choose one that seems the most intuitive to you.

- The Use R! series (the yellow books) by Springer are really good. In particular, consider: “A Beginner’s Guide to R”, “R by Example”, “Numerical Ecology With R”, “ggplot2” (we’ll see this in Chapter 2), “A Primer of Ecology with R”, “Nonlinear Regression with R”, “Analysis of Phylogenetics and Evolution with R”.

- For more focus on dynamical models: Soetaert & Herman. 2009 “A practical guide to ecological modelling: using R as a simulation platform”.

- There are excellent websites besides cran. In particular, check out `www.statmethods.net` and `http://en.wikibooks.org/wiki/R_Programming`.

- For those who are coming with Matlab experience: `http://www.math.umaine.edu/~hiebeler/comp/matlabR.html`

- Bolker, B. M.: Ecological Models and Data in R (eBook and Hardcover available).

- Beckerman, A. P. & Petchey, O. L. (2012) Getting started with R: an introduction for biologists. Oxford, Oxford University Press.
  Very basic, good if you are really stuck at the outset.

- Crawley, R. (2013) The R book. 2nd edition. Chichester, Wiley.
  Excellent but enormous reference book, code and data available from `www.bio.ic.ac.uk/research/mjcraw/therbook/index.htm`

Chapter 4

Plotting and graphics in R

4.1 Basic plotting

R can produce beautiful graphics, without the time consuming and fiddly methods that you might have used in Excel or equivalent (it's not you, it's Excel!). Later, you can explore how the package `ggplot2`, can allow the rapid creation of truly elegant publication-grade graphics. However, in many cases you just want to quickly plot the data for exploratory analysis of your data. Here are the basic plotting commands:

`plot(x,y)`	Scatterplot
`plot(y~x)`	Scatterplot with `y` as a response variable
`hist(mydata)`	Histogram
`barplot(mydata)`	Bar plot
`points(y1~x1)`	Add another series of points
`boxplot(y~x)`	Boxplot

Let's try some basic plotting. As a case study, we will use a simplified version of a dataset on predator-prey body mass ratios taken from the Ecological Archives of the ESA (Barnes *et al.* 2008, Ecology 89:881).

Foraging in 3*D*: a school of salema (*Xenocys jessiae*) keep safe distance from a hungry sea lion (*Zalophus wollebaeki*) off the Galápagos Islands, Ecuador
Photo courtesy: David Doubilet

These data should be in your `Data` directory. Let's import the data:

```
> MyDF <- read.csv("../Data/EcolArchives-E089-51-D1.csv")
> dim(MyDF)
[1] 34931    15
> MyDF$
MyDF$Record.number                  MyDF$Predator.mass
MyDF$In.refID                       MyDF$Prey
MyDF$IndividualID                   MyDF$Prey.common.name
MyDF$Predator                       MyDF$Prey.taxon
MyDF$Predator.common.name           MyDF$Prey.mass
MyDF$Predator.taxon                 MyDF$Prey.mass.unit
MyDF$Predator.lifestage             MyDF$Location
MyDF$Type.of.feeding.interaction
```

4.1.1 Scatter Plot

Let's start by plotting Predator mass vs. Prey mass:

```
> plot(MyDF$Predator.mass,MyDF$Prey.mass)
```

That doesn't look very nice! Let's try taking logarithms (why?).

```
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass))
```

We can change almost any aspect of the resulting graph; letâĂŹs change the symbols by specifying the plot characters using pch:

```
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass),pch=20) # Change marker
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass),pch=20,
    xlab = "Predator Mass (kg)", ylab = "Prey Mass (kg)") # Add labels
```

4.1.2 Histograms

Now plot a histogram of Predator body masses:

```
> hist(MyDF$Predator.mass)
> hist(log(MyDF$Predator.mass),
    xlab = "Predator Mass (kg)", ylab = "Count") # labels
> hist(log(MyDF$Predator.mass),xlab="Predator Mass (kg)",ylab="Count",
    col = "lightblue", border = "pink") # Change bar and borders colors
```

4.1.3 Subplots

We can also plot both predator and prey body masses in different sub-plots using par

```
> par(mfcol=c(2,1), lwd = 1.5) #initialize multi-paneled plot
> par(mfg = c(1,1))
> hist(log(MyDF$Predator.mass),
    xlab = "Predator Mass (kg)", ylab = "Count",
    col = "lightblue", border = "pink",
    main = 'Predator') # Add title
> par(mfg = c(2,1));
> hist(log(MyDF$Prey.mass),
    xlab="Prey Mass (kg)",ylab="Count",
    col = "lightgreen", border = "pink",
    main = 'prey')
```

4.1.4 Overlaying plots

Better still, we would like to see if the predator mass and prey mass distributions are similar by overlaying them.

```
> hist(log(MyDF$Predator.mass), # Predator histogram
    xlab="Body Mass (kg)", ylab="Count",
    col = rgb(1, 0, 0, 0.5), # Note 'rgb'
    main = "Overlay")
> hist(log(MyDF$Prey.mass), col = rgb(0, 0, 1, 0.5), add = T) # Plot prey
> legend('topleft',c('Predators','Prey'),   # Add legend
    fill=c(rgb(1, 0, 0, 0.5), rgb(0, 0, 1, 0.5))) # Define legend colors
```

4.1.5 Exercise

We can do a lot more beautification! As an exercise, try adjusting the bin widths to make them same for the predator and prey, and making the x and y labels larger and in boldface.

4.1.6 Saving your graphics

And you can also save the figure in a vector graphics format (important to learn to do this!). PDF is a good option:

```
> pdf("../Results/Pred_Prey_Overlay.pdf", # Open blank pdf page
    11.7, 8.3) # These numbers are page dimensions in inches
> hist(log(MyDF$Predator.mass), # Plot predator histogram (note 'rgb')
    xlab="Body Mass (kg)", ylab="Count",
    col = rgb(1, 0, 0, 0.5),
    main = "Overlay")
> hist(log(MyDF$Prey.mass), # Plot prey weights
    col = rgb(0, 0, 1, 0.5),
    add = T)  # Add to same plot = TRUE
> legend('topleft',c('Predators','Prey'), # Add legend
    fill=c(rgb(1, 0, 0, 0.5), rgb(0, 0, 1, 0.5)))
> dev.off()
```

You can also try other graphic output formats. For example, `png()` (a raster format) instead of `pdf()`.

4.1.7 Boxplots

Now, let's try plotting boxplots instead of histograms. These are useful for getting a visual summary of your data :

```
> boxplot(log(MyDF$Predator.mass) ~ MyDF$Location, # Why the tilde?
    xlab = "Location", ylab = "Predator Mass",
    main = "Predator mass by location")
```

That's a lot of locations! You will need an appropriately wide plot to see all the boxplots adequately. Let's try boxplots by feeding interaction type:

```
> boxplot(log(MyDF$Predator.mass) ~ MyDF$Type.of.feeding.interaction,
    xlab = "Location", ylab = "Predator Mass",
    main = "Predator mass by feeding interaction type")
```

4.1.8 Practical 2.1

You can also make lattice graphs to avoid the somewhat laborious `par()` approach above. For this, you will need:

```
> library(lattice)
```

A lattice plot of the above data for predator mass could look like Fig. 4.1.8 (as a density plot). This was generated using (and printing to a pdf with particular dimensions):

```
> densityplot(~log(Predator.mass) | Type.of.feeding.interaction, data=MyDF)
```

Look up `http://www.statmethods.net/advgraphs/trellis.html` and the `lattice` package help.

In this practical, you will write script that draws and saves three lattice graphs by feeding interaction type: one of predator mass , one of prey mass and one of the size ratio of prey mass over predator mass. Note that you would want to use logarithms of masses (or mass-ratios) for all three plots. In addition, the script will calculate the mean and median predator mass, prey mass and predator-prey size-ratios to a csv file. The workflow would be:

- ⋆ Write a script file called `PP_Lattice.R` and save it in the `Code` directory — sourcing or running this script should result in three files called `Pred_Lattice.pdf`, `Prey_Lattice.pdf`, and `SizeRatio_Lattice.pdf` being saved in the `Results` directory (the names are self-explanatory, I hope).
- ⋆ In addition, the script should calculate the mean log predator mass, prey mass, and predator-prey size ratio, and save it as a single csv output table called `PP_Results.csv` to the `Results` directory. The table should have appropriate headers (e.g., Feeding type, mean, median). (Hint: you will have to initialize a new dataframe in the script to first store the calculations)
- ⋆ The script should be self-sufficient and not need any external inputs — it should import the above predator-prey dataset from the appropriate directory, and save the graphic plots to the appropriate directory (Hint: use relative paths).

4.2 Publication-quality figures in R

`R` can produce beautiful graphics, but it takes a lot of work to obtain the desired result. This is because the starting point is pretty much a "bare" plot, and adding features commonly required for publication-grade figures (legends, statistics, regressions, etc.) can be quite involved.

Moreover, it is very difficult to switch from one representation of the data to another (i.e., from boxplots to scatterplots), or to plot several dataset together. To overcome these issues, the `R` package `ggplot2` is very powerful. It can can be used to produce truly high-quality graphics for papers, theses and reports. *One thing to note though is that at present, ggplot2 cannot be used to create 3D graphs or mosaic plots.*

`ggplot2` differs from other approaches as it attempts to provide a "grammar" for graphics in which each layer is the equivalent of a verb, subject etc. and a plot is the equivalent of a sentence. All graphs start with a layer showing the data, other layers and commands are added to modify the plot.

For a complete reference, please see the book "ggplot2: Elegant Graphics for Data Analysis", by H. Wickham. Also, the website `ggplot2.org` a great resource. To install `ggplot2`, open a session of `R` and type (launch R using `sudo R`) in Linux first):

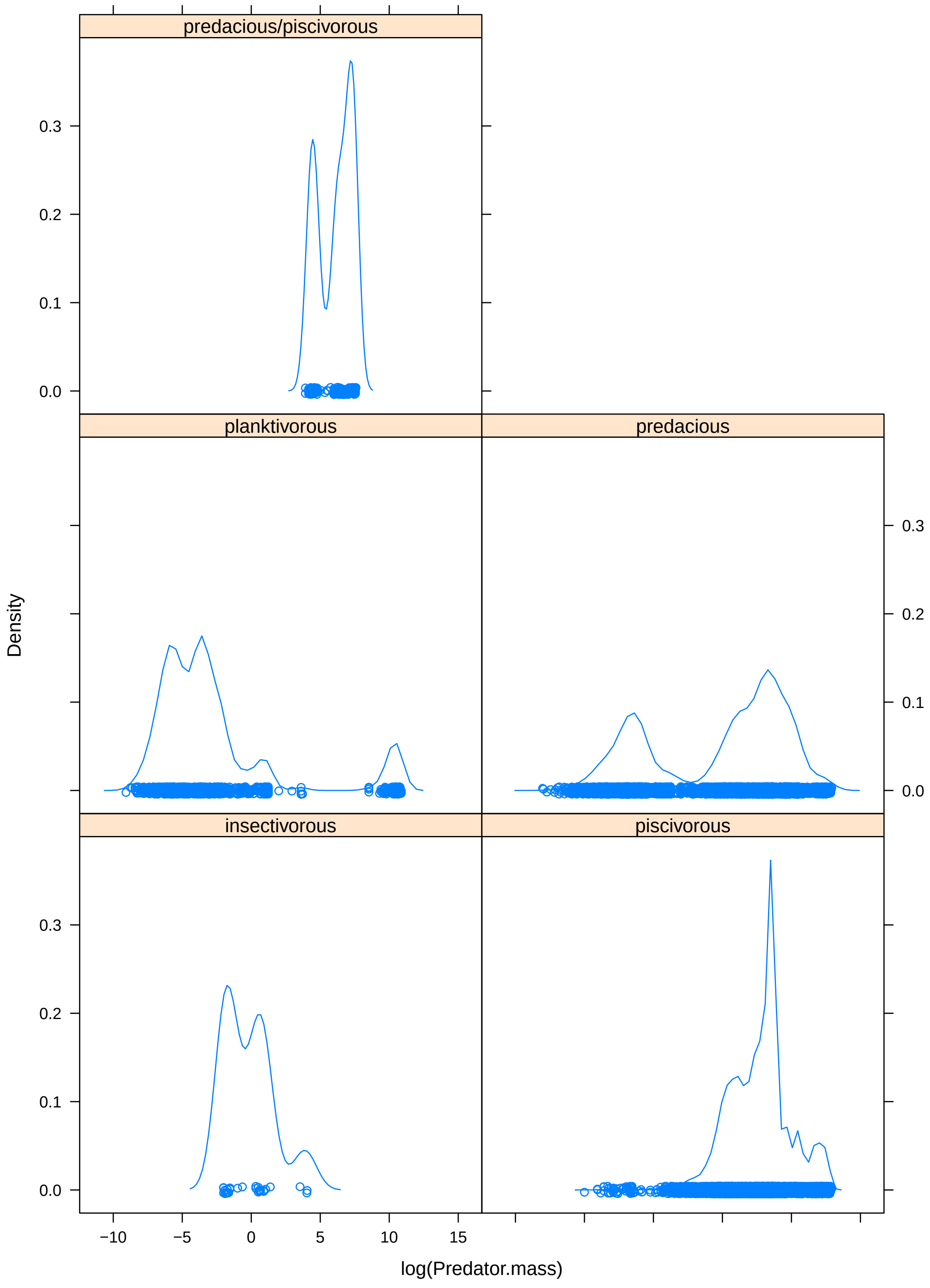


Figure 4.1: A `lattice` representation of the predator size data

```
> install.packages("ggplot2")
> install.packages("reshape") #A handy additional package
```

4.2.1 Basic graphs with `qplot`

`qplot` stands for quick plot, and is the basic plotting function provided by `ggplot2`. It can be used to quickly produce graphics for exploratory data analysis, and as a base for more complex graphics.

In `ggplot2`, it is necessary to use data frames to store the data. Again we can start plotting the `Predator.mass` vs `Prey.mass`.

```
> require(ggplot2)  ## Load the package
Loading required package: ggplot2
> qplot(Prey.mass, Predator.mass, data = MyDF)
```

Again, let's take logarithms and plot:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF)
```

Now, color the points according to the type of feeding interaction:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    colour = Type.of.feeding.interaction)
```

The same as above, but changing the shape:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    shape = Type.of.feeding.interaction)
```

To manually set a color or a shape, you have to use `I()` (meaning "Identity"):

```
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, colour = I("red"))
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, shape= I(3))
```

Because there are so many points, we can make them semi-transparent using `alpha` so that the overlaps can be seen:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    colour = Type.of.feeding.interaction, alpha = I(.5))
```

Now add a smoother to the points.

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"))
```

If we want to have a linear regression, we can specify the method `lm`:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"), method = "lm")
```

We can add a smoother for each type of interaction:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"), method = "lm",
    colour = Type.of.feeding.interaction)
```

To extend the lines to the full range, use `fullrange = TRUE`:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"), method = "lm",
    colour = Type.of.feeding.interaction,
    fullrange = TRUE)
```

Now we want to see how the ratio between prey and predator mass changes according to the type of interaction:

```
> qplot(Type.of.feeding.interaction,
        log(Prey.mass/Predator.mass), data = MyDF)
```

Because there are so many points, we can “jitter” them to get a better idea of the spread:

```
> qplot(Type.of.feeding.interaction,
        log(Prey.mass/Predator.mass), data = MyDF,
        geom = "jitter")
```

Or we can draw a boxplot of the data (note the `geom` argument):

```
> qplot(Type.of.feeding.interaction,
        log(Prey.mass/Predator.mass), data = MyDF,
        geom = "boxplot")
```

Now let’s draw an histogram of predator-prey mass ratios:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "histogram")
```

Color the histogram according to the interaction type:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "histogram",
    fill = Type.of.feeding.interaction)
```

You may want to define binwidth (in units of x axis):

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "histogram",
    fill = Type.of.feeding.interaction,
    binwidth = 1)
```

To make it easier to read, we can plot the smoothed density of the data:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "density", fill = Type.of.feeding.interaction)
```

And you can make the densities transparent so that the overlaps are visible:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "density", fill = Type.of.feeding.interaction, alpha =
    I(0.5))
```

Or using `colour` instead of `fill` draws only the edge of the curve:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "density", colour = Type.of.feeding.interaction)
```

Similarly, `geom = “bar”` produces a barplot, `geom = “line”` a series of points joined by a line, etc.

An alternative way of displaying data belonging to different classes is using “faceting”. A simple example:

```
> qplot(log(Prey.mass/Predator.mass),
    facets = Type.of.feeding.interaction ~ .,
    data = MyDF, geom =  "density")
```

A more elegant way of drawing logarithmic quantities is to set the axes to be logarithmic:

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy")
```

Let’s add a title and labels:

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy",
    main = "Relation between predator and prey mass",
    xlab = "log(Prey mass) (g)",
    ylab = "log(Predator mass) (g)")
```

Adding `+ theme_bw()` makes it suitable for black and white printing.

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy",
    main = "Relation between predator and prey mass",
    xlab = "Prey mass (g)",
    ylab = "Predator mass (g)") + theme_bw()
```

Finally, let's save a pdf file of the figure (same approach as we used before):

```
> pdf("../Results/MyFirst-ggplot2-Figure.pdf")
> print(qplot(Prey.mass, Predator.mass, data = MyDF,log="xy",
    main = "Relation between predator and prey mass",
    xlab = "log(Prey mass) (g)",
    ylab = "log(Predator mass) (g)") + theme_bw())
> dev.off()
```

Uing `print` ensures that the whole command is kept together and that you can use the command in a script.

Other important options to keep in mind:

`xlim`	limits for x axis: `xlim = c(0,12)`
`ylim`	limits for y axis
`log`	log transform variable `log = "x"`, `log = "y"`, `log = "xy"`
`main`	title of the plot `main = "My Graph"`
`xlab`	x-axis label
`ylab`	y-axis label
`asp`	aspect ratio `asp = 2`, `asp = 0.5`
`margins`	whether or not margins will be displayed

4.2.2 Various `geom`

`geom` Specifies the geometric objects that define the graph type. The geom option is expressed as a character vector with one or more entries. geom values include "point", "smooth", "boxplot", "line", "histogram", "density", "bar", and "jitter". Try the following:

```
# load the package
require(ggplot2)

# load the data
MyDF <- as.data.frame(
  read.csv("../Data/EcolArchives-E089-51-D1.csv"))

# barplot
qplot(Predator.lifestage,
      data = MyDF, geom = "bar")

# boxplot
qplot(Predator.lifestage, log(Prey.mass),
      data = MyDF, geom = "boxplot")

# density
qplot(log(Predator.mass),
      data = MyDF, geom = "density")

# histogram
qplot(log(Predator.mass),
      data = MyDF, geom = "histogram")

# scatterplot
qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "point")

# smooth
qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "smooth")

qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "smooth", method = "lm")
```

4.2.3 Practical 2.2

In this practical, you will write script that draws and saves a pdf file of Fig. 4.2.3, and writes the accompanying regression results to a formatted table in csv. Note that the plots show that the analysis must be subsetted by the `Predator.lifestage` field of the dataset. The guidelines are:

⋆ Write a script file called `PP_Regress.R` and save it in the `Code` directory — sourcing or running this script should result in one pdf file containing (Fig. 4.2.3) being saved in the `Results` directory (HInt: Use the `print()` command to write to the pdf).

⋆ In addition, the script should calculate the regression results corresponding to the lines fitted in Fig. 4.2.3 and save it to a csv delimited table called (`PP_Regress_Results.csv`), in the `Results` directory. (Hint: you will have to initialize a new dataframe in the script to first store the calculations and then `write.csv()` or `write.table()` it.)
All that you are being asked for here is results of an analysis of Linear regression on subsets of the data corresponding to available Feeding Type × Predator life Stage combination — not a multivariate linear model with these two as separate covariates!

⋆ The regression results should include the following with appropriate headers (e.g., slope, intercept, etc, in each Feeding type × life stage category): regression slope, regression intercept, R^2, F-statistic value, and p-value of the overall regression (Hint: Review Practical 8 of the Stats week).

⋆ The script should be self-sufficient and not need any external inputs — it should import the above predator-prey dataset from the appropriate directory, and save the graphic plots to the appropriate directory (Hint: use relative paths). I should be able to `source` it without errors.

Extra Credit: Do the same as above, but the analysis this time should be separate by the dataset's `Location` field. This is a substantial extra effort, so I do think it deserves substantial extra credit!

4.2.4 Advanced plotting: `ggplot`

The command `qplot` allows you to use only a single dataset and a single set of "aesthetics" (x, y, etc.). To make full use of `ggplot2`, we need to use the command `ggplot`. We need:

- The data to be plotted, in a data frame;
- Aesthetics mappings, specifying which variables we want to plot, and how;
- The `geom`, defining how to draw the data;

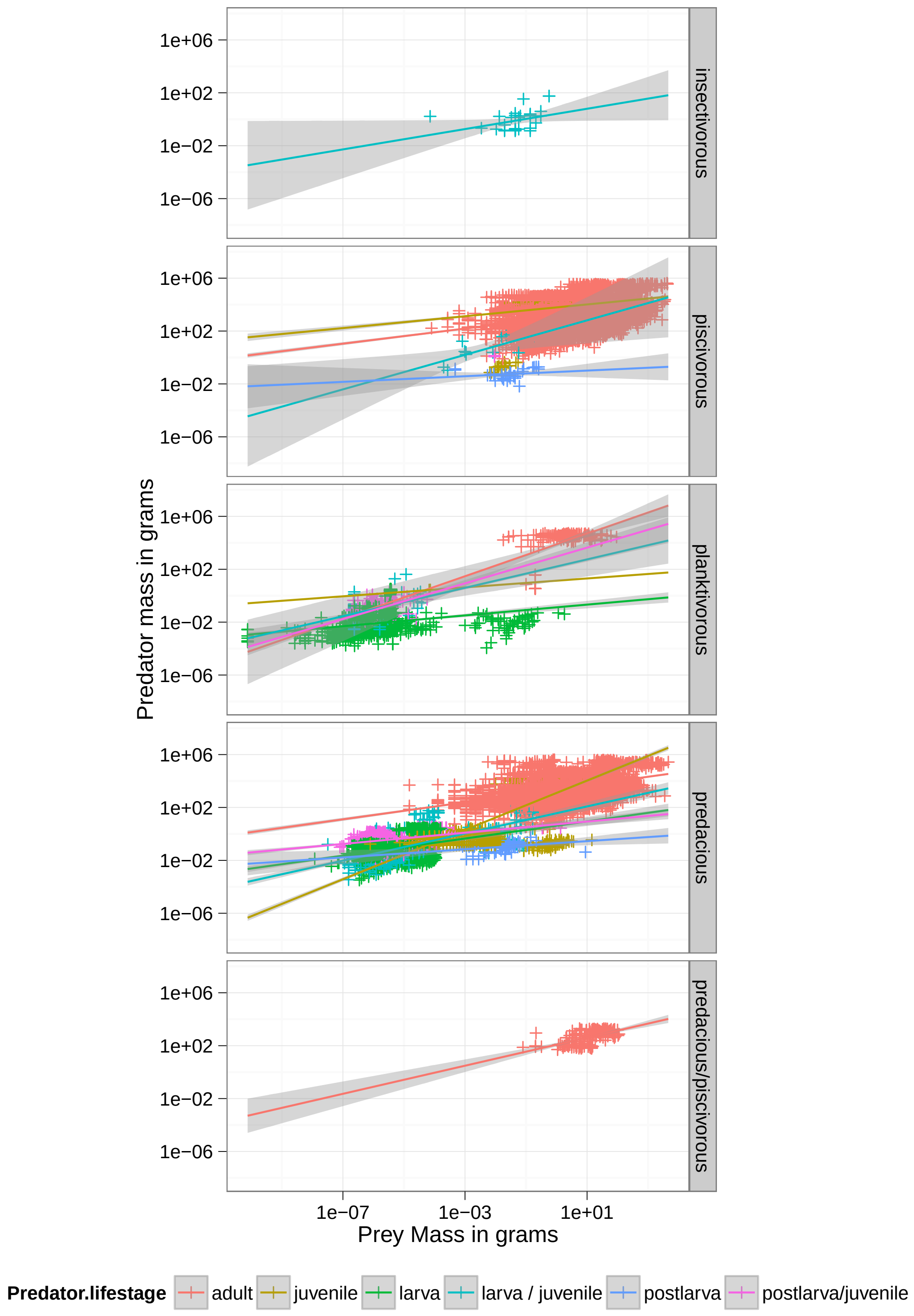


Figure 4.2: Write a script that generates this figure.

- (Optionally) some `stat` that transform the data or perform statistics using the data.

To start a graph, we can specify the data and the aesthetics:

```
> p <- ggplot(MyDF, aes(x = log(Predator.mass),
                y = log(Prey.mass),
                colour = Type.of.feeding.interaction ))
```

Now try to plot the graph:

```
> p
Error: No layers in plot
```

In fact, we have to specify a geometry in order to see the graph:

```
> p + geom_point()
```

We can use the "plus" sign to concatenate different commands:

```
> p <- ggplot(MyDF, aes(x = log(Predator.mass),
                y = log(Prey.mass),
                colour = Type.of.feeding.interaction ))
> q <- p + geom_point(size=I(2), shape=I(10)) + theme_bw()
> q
```

Let's remove the legend:

```
> q + opts(legend.position = "none")
```

4.2.5 Case study 1: plotting a matrix

In this section we will plot a matrix of random values taken from a normal distribution $\mathscr{U}[0,1]$. Our goal is to produce the plot in Figure 4.3. Because we want to plot a matrix, and `ggplot2` accepts only dataframes, we use the package `reshape` that can "melt" a matrix into a dataframe:

```
require(ggplot2)
require(reshape)

GenerateMatrix <- function(N){
    M <- matrix(runif(N * N), N, N)
    return(M)
}

> M <- GenerateMatrix(10)

> M[1:3, 1:3]
            [,1]       [,2]       [,3]
[1,] 0.2700254 0.8686728 0.7365857
[2,] 0.1744879 0.8488169 0.4165879
[3,] 0.3980783 0.7727821 0.4271121
```

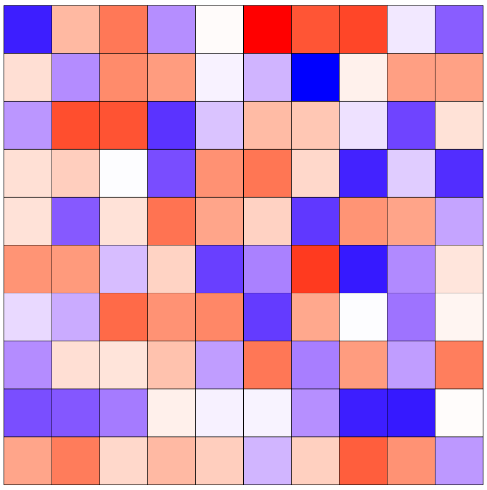


Figure 4.3: Random matrix with values sampled from uniform distribution.

```
> Melt <- melt(M)

> Melt[1:4,]
    X1 X2     value
1  1  1 0.2700254
2  2  1 0.1744879
3  3  1 0.3980783
4  4  1 0.3196671

> ggplot(Melt, aes(X1, X2, fill = value)) + geom_tile()

# adding a black line dividing cells
> p <- ggplot(Melt, aes(X1, X2, fill = value))
> p <- p + geom_tile(colour = "black")

# removing the legend
> q <- p + opts(legend.position = "none")

# removing all the rest
> q <- p + opts(legend.position = "none",
     panel.background = theme_blank(),
     axis.ticks = theme_blank(),
     panel.grid.major=theme_blank(),
     panel.grid.minor=theme_blank(),
     axis.text.x = theme_blank(),
     axis.title.x=theme_blank(),
     axis.text.y = theme_blank(),
     axis.title.y=theme_blank())

# exploring the colors
> q + scale_fill_continuous(low = "yellow",
                        high = "darkgreen")
> q + scale_fill_gradient2()
> q + scale_fill_gradientn(colours = grey.colors(10))
> q + scale_fill_gradientn(colours = rainbow(10))
> q + scale_fill_gradientn(colours =
                c("red", "white", "blue"))
```

4.2.6 Case study 2: plotting two dataframes

According to Girko's circular law, the eigenvalues of a matrix M of size $N \times N$ are approximately contained in a circle in the complex plane with radius $\sqrt{N}$. We are going to draw a simulation displaying this result (Figure 4.4).

```
require(ggplot2)

# function that returns an ellipse
build_ellipse <- function(hradius, vradius){
  npoints = 250
  a <- seq(0, 2 * pi, length = npoints + 1)
  x <- hradius * cos(a)
  y <- vradius * sin(a)
  return(data.frame(x = x, y = y))
}

# Size of the matrix
N <- 250
# Build the matrix
M <- matrix(rnorm(N * N), N, N)
# Find the eigenvalues
eigvals <- eigen(M)$values
# Build a dataframe
eigDF <- data.frame("Real" = Re(eigvals),
                    "Imaginary" = Im(eigvals))

# The radius of the circle is sqrt(N)
my_radius <- sqrt(N)
# Ellipse dataframe
ellDF <- build_ellipse(my_radius, my_radius)
# rename the columns
names(ellDF) <- c("Real", "Imaginary")

# Now the plotting:
# plot the eigenvalues
p <- ggplot(eigDF, aes(x = Real, y = Imaginary))
p <- p +
  geom_point(shape = I(3)) +
  opts(legend.position = "none")


# now add the vertical and horizontal line
p <- p + geom_hline(aes(intercept = 0))
p <- p + geom_vline(aes(intercept = 0))

# finally, add the ellipse
p <- p + geom_polygon(data = ellDF,
                      aes(x = Real,
                          y = Imaginary,
                          alpha = 1/20,
                          fill = "red"))
pdf("Girko.pdf")
print(p)
dev.off()
```

4.2.7 Case study 3: annotating the plot

In the plot in Figure 4.5, we use the geometry “text” to annotate the plot.

```
require(ggplot2)

filename <- "Results.txt"
a <- read.table(filename, header = TRUE)
# here's how the data looks like
print(a[1:3,])
print(a[90:95,])

# append a col of zeros
a$ymin <- rep(0, dim(a)[1])

# print the first linerange
p <- ggplot(a)
p <- p + geom_linerange(data = a, aes(
                          x = x,
                          ymin = ymin,
                          ymax = y1,
                          size = (0.5)
                          ),
                        colour = "#E69F00",
                        alpha = 1/2, show_guide = FALSE)

# print the second linerange
p <- p + geom_linerange(data = a, aes(
                          x = x,
                          ymin = ymin,
                          ymax = y2,
                          size = (0.5)
                          ),
                        colour = "#56B4E9",
                        alpha = 1/2, show_guide = FALSE)

# print the third linerange
p <- p + geom_linerange(data = a, aes(
                          x = x,
                          ymin = ymin,
                          ymax = y3,
                          size = (0.5)
                          ),
                        colour = "#D55E00",
                        alpha = 1/2, show_guide = FALSE)

# annotate the plot with labels
p <- p + geom_text(data = a,
                   aes(x = x, y = -500, label = Label))

# now set the axis labels,
# remove the legend, prepare for bw printing
p <- p + scale_x_continuous("My x axis",
                            breaks = seq(3, 5, by = 0.05)
                            ) +
  scale_y_continuous("My y axis") + theme_bw() +
  opts(legend.position = "none")
```

```
# Finally, print in a pdf
pdf("MyBars.pdf", width = 12, height = 6)
print(p)
dev.off()
```

4.2.8 Case study 4: mathematical display

In Figure 4.6, you can see the mathematical annotation of the axis and on the plot.

```
require(ggplot2)

# create an "ideal" linear regression data!
x <- seq(0, 100, by = 0.1)
y <- -4. + 0.25 * x +
  rnorm(length(x), mean = 0., sd = 2.5)

# now a dataframe
my_data <- data.frame(x = x, y = y)

# perform a linear regression
my_lm <- summary(lm(y ~ x, data = my_data))

# plot the data
p <-  ggplot(my_data, aes(x = x, y = y,
                          colour = abs(my_lm$residual))
             ) +
  geom_point() +
  scale_colour_gradient(low = "black", high = "red") +
  opts(legend.position = "none") +
  scale_x_continuous(
    expression(alpha^2 * pi / beta * sqrt(Theta)))

# add the regression line
p <- p + geom_abline(
  intercept = my_lm$coefficients[1][1],
  slope = my_lm$coefficients[2][1],
  colour = "red")
# throw some math on the plot
p <- p + geom_text(aes(x = 60, y = 0,
                       label = "sqrt(alpha) * 2* pi"),
                       parse = TRUE, size = 6,
                       colour = "blue")

# print in a pdf
pdf("MyLinReg.pdf")
print(p)
dev.off()
```

4.3 Readings

- The classic Tufte `www.edwardtufte.com/tufte/books_vdqi` (btw, check out what Tufte thinks of PowerPoint!)

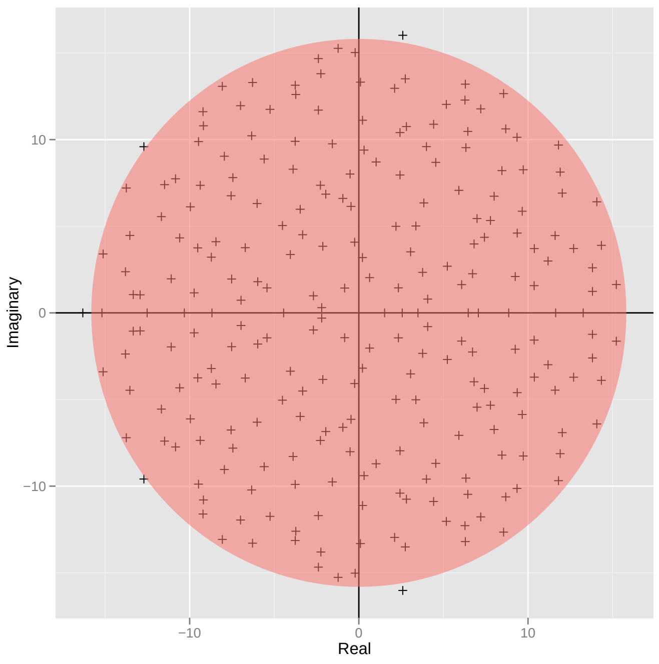


Figure 4.4: Girko's circular law.

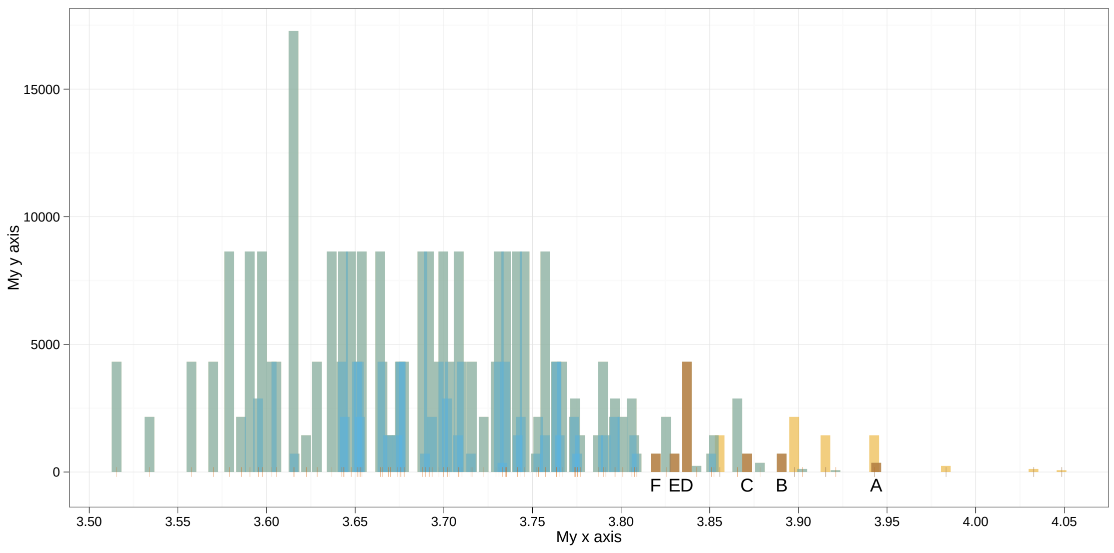


Figure 4.5: Overlay of three lineranges and a text geometry.

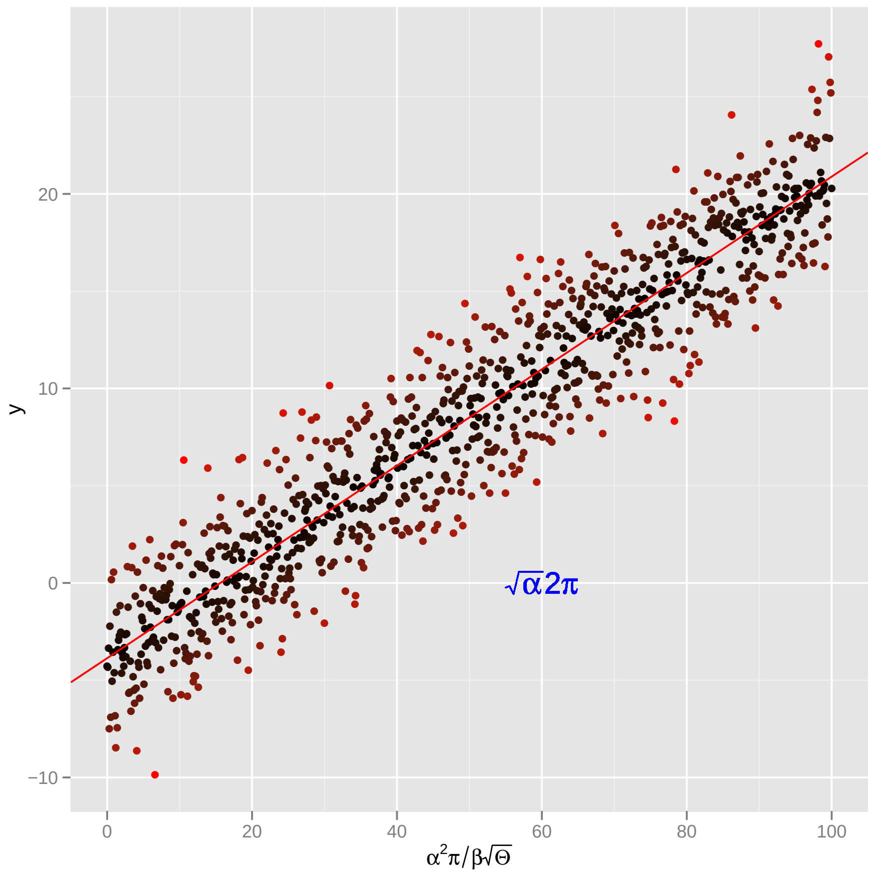


Figure 4.6: Linear regression with colors expressing residuals and mathematical annotations.

Available in the Central Library, I have also added extracts and a related book in pdf (on Blackboard)

- Rolandi et al. “A Brief Guide to Designing Effective Figures for the Scientific Paper”, doi:10.1002/adma.201102518 (on Blackboard)
- Lauren `et al.` “Graphs, Tables, and Figures in Scientific Publications: The Good, the Bad, and How Not to Be the Latter”, doi:10.1016/j.jhsa.2011.12.041 (on Blackboard)
- Effective scientific illustrations: `www.labtimes.org/labtimes/issues/lt2008/lt05/lt_2008_05_52_53.pdf` (on Blackboard)
- `https://web.archive.org/web/20120310121708/http://addictedtor.free.fr/graphiques/thumbs.php`
- Make `xkcd` style graphs in R: `http://xkcd.r-forge.r-project.org/`

Chapter 5

Advanced topics in R

5.1 Vectorization

R is very slow at running cycles (`for` and `while` loops). This is because R is a “nimble” language: at execution time R does not know what you’are going to perform until it “reads” the code to perform. Compiled languages such as `C`, know exactly what the flow of the program is, as the code is compiled before execution. As a metaphor, `C` is a musician playing a score she has seen before – optimizing each passage, while R is playing it “a prima vista” (i.e., at first sight).

Hence, in R you should try to avoid loops like the plague. In practical terms, sometimes it is much easier to throw in a for loop, and then optimize the code to avoid the loop if the running time is not satisfactory. R has several functions that can operate on entire vectors and matrices.

⋆ For example, type (save in `Code`) in `Vectorize1.R` and run it (it sums all elements of a matrix):

```r
M <- matrix(runif(1000000),1000,1000)

SumAllElements <- function(M){
  Dimensions <- dim(M)
  Tot <- 0
  for (i in 1:Dimensions[1]){
    for (j in 1:Dimensions[2]){
      Tot <- Tot + M[i,j]
    }
  }
  return (Tot)
}

## This on my computer takes about 1 sec
print(system.time(SumAllElements(M)))
## While this takes about 0.01 sec
print(system.time(sum(M)))
```

Both approaches are correct, and will give you the right answer. However, one is 100 times faster than the other!

Fortunately, R offers several ways of avoiding loops. Here are the main ones:

```
## apply:
# applying the same function to rows/colums of a matrix

## Build a random matrix
M <- matrix(rnorm(100), 10, 10)
## Take the mean of each row
RowMeans <- apply(M, 1, mean)
print (RowMeans)
## Now the variance
RowVars <- apply(M, 1, var)
print (RowVars)
## By column
ColMeans <- apply(M, 2, mean)
print (ColMeans)

## You can use it to define your own functions
## (What does this function do?)
SomeOperation <- function(v){
  if (sum(v) > 0){
    return (v * 100)
  }
  return (v)
}
print (apply(M, 1, SomeOperation))

## by:
## apply the function to a dataframe, using some factor
## to define the subsets

## import some data
attach(iris)
print (iris)

## use colMeans (as it is better for dataframes)
by(iris[,1:2], iris$Species, colMeans)
by(iris[,1:2], iris$Petal.Width, colMeans)

## There are many other methods: lapply, sapply, eapply, etc.
## Each is best for a given data type (lapply -> lists)

## replicate:
## this is quite useful to avoid a loop for function that typically
## involve random number generation
print(replicate(10, runif(5)))
```

5.2 Breaking out of loops

Often it is useful (or necessary) to `break` out of a loop when some condition is met. Use `break` (like in preyy much any other programming language, like `python`) in situations when you cannot set a target number of iterations, as you would with a `while` loop (Chapter 1). Try this (type into `break.R` and save in `Code`):

```
i <- 0 #Initialize i
    while(i < Inf) {
        if (i == 20) {
            break } # Break out of the while loop!
        else {
            cat("i equals " , i , " \n")
            i <- i + 1 # Update i
    }
}
```

5.3 Using next

You can also skip to next iteration of a loop. Both `next` and `break` can be used within other loops (`while`, `for`). Try this (type into `next.R` and save in `Code` (what does this script do?)):

```
for (i in 1:10) {
  if ((i %% 2) == 0)
    next # pass to next iteration of loop
  print(i)
}
```

Reminder: Indent your code! Indentation helps you see the flow of the logic, rather than flattened version, which is hard for you and everybody else to read. I recommend using the *tab* key to indent.

5.4 "Catching" errors

Often, you don't know if a simulation or a R function will work on a particular data or variable, or a value of a variable (can happen in many stats functions). Rather than having R throw you out of the code, you would rather catch the error and keep going. This can be done using `try`. Type the following into `try.R` and save in `Code` (what does this script do?):

```
## run a simulation that involves sampling from a population

x <- stats::rnorm(50)
doit <- function(x){
    x <- sample(x, replace = TRUE)
    if(length(unique(x)) > 30) {
         mean(x)}
    else {
        stop("too few unique points")
        }
}

## Try using "try" with vectorization:
result <- lapply(1:100, function(i) try(doit(x), FALSE))

## Or using a for loop:
result <- vector("list", 100) #Preallocate/Initialize
for(i in 1:100) {
    result[[i]] <- try(doit(x), FALSE)
}
```

Note the functions `sample` and `stop` in the above script. Also check out `tryCatch`.

5.5 Generating Random Numbers

Computers don't really generate mathematically random numbers, but instead a sequence of numbers that are close to random: "pseudo-random numbers". They are generated based on some iterative formula:

$$x_{new} = f(x_{old}) \mod N$$

where modulo operation provides the "remainder" division.

To generate the first random number, you need a **seed**. Setting the seed allows you to reliably generate the same sequence of numbers, which can be useful when debugging programs (next section).

R has many routines for generating random samples from various probability distributions — we have already used `runif()`, `rnorm()`. Try this:

```
> set.seed(1234567)
> rnorm(1)
 0.1567038
```

What happened?! If this were truly a random number, how would everybody get the same answer? Now try `rnorm(10)` and compare the results with your neighbour. Thus "random" numbers generated in R and in any other software are in fact "deterministic", but from a very complex formula that yields numbers with properties like random numbers.

Effectively, `rnorm` has an enormous list that it cycles through. The random seed starts the process, i.e., indicates where in the list to start. This is usually taken from the clock when you start R.

But why bother with this? Well, for debugging (next section). Bugs in code can be hard to find — harder still if you are generating random numbers, so repeat runs of your code may or may not all trigger the same behaviour. You can set the seed once at the beginning of the code — ensuring repeatability, retaining (pseudo) randomness. Once debugged, if you want, you can remove the set seed line.

5.6 Debugging

Indeed, as most of you must have already experienced by now, there can be frustrating, puzzling bugs in programs that lead to mysterious errors. Often, the error and warning messages you get are un-understandable, especially in R! Some useful debugging functions in R:

- Warnings vs Errors; converting warnings to errors: `stopifnot()` — a bit like `try`
- What to do when you get an error: `traceback()`
- Simple `print` commands in the right places can be useful for testing (but not strongly recommended)
- Use of `browser()` at key points in code — my favourite option (also look up `recover()`)
- `debug(fn)`, `undebug(fn)` : More technical approach to debugging — explore them

Let's look at an example using `browser()`. `browser()` is handy because it will allow you to "single-step" through your code. Place it within your function at the point you want to examine (e.g.) local variables.

Here's an example usage of `browser()` (type in `browse.R` and save in `Code`):

```
Exponential <- function(N0 = 1, r = 1, generations = 10){
  # Runs a simulation of exponential growth
  # Returns a vector of length generations

  N <- rep(NA, generations)     # Creates a vector of NA

  N[1] <- N0
  for (t in 2:generations){
    N[t] <- N[t-1] * exp(r)
#      browser()
  }
  return (N)
}
plot(Exponential(), type="l", main="Exponential growth")
```

Now, within the browser, you can enter expressions as normal, or you can use a few particularly useful debug commands:

- `n`: single-step
- `c`: exit browser and continue
- `Q`: exit browser and abort, return to top-level.

5.7 Practical 3.1

Non-CMEE students can ignore this one!

The Ricker model is a classic discrete population model which was introduced in 1954 by Ricker to model recruitment of stock in fisheries. It gives the expected number (or density) N_{t+1} of individuals in generation $t+1$ as a function of the number of individuals in the previous generation t:

$$N_{t+1} = N_t e^{r\left(1-\frac{N_t}{k}\right)} \tag{5.1}$$

Here r is intrinsic growth rate and k as the carrying capacity of the environment. Try this script that runs it:

```
Ricker <- function(N0=1, r=1, K=10, generations=50)
{
  # Runs a simulation of the ricker model
  # Returns a vector of length generations

  N <- rep(NA, generations)     # Creates a vector of NA

  N[1] <- N0
  for (t in 2:generations)
  {
    N[t] <- N[t-1] * exp(r*(1.0-(N[t-1]/K)))
  }
  return (N)
}
plot(Ricker(generations=10), type="l")
```

Now open and run the script `Vectorize2.R` (available on the bitbucket Git repository). This is the stochastic Ricker model (compare with the above script to see where the stochasticity (random error) enters. Now modify the script to complete the exercise given. CMEEs, As always, bring your functional code and data under version control!

5.8 Launching/Running R in batch mode

Often, you may want to run the final analysis without opening R in interactive mode. In in Mac or linux, you can do so by typing:

```
R CMD BATCH MyCode.R MyResults.Rout
```

This will create an `MyResults.Rout` file containing all the output. On Microsoft Windows, its more complicated — (change the path to `R.exe` and output file as needed:

```
"C:\Program Files\R\R-3.1.1\bin\R.exe" CMD BATCH -vanilla -slave
"C:\PathToMyResults\Results\MyCode.R"
```

5.9 Accessing databases using R

R can be used to link seamlessly to on-line databases such as PubMed and GenBank. Such computational Biology datasets are often quite large, and it makes sense to access their data by querying the databases instead of manually downloading. There are many R packages that provide an interface to databases (SQLite, MySQL, Oracle, etc). Check out R packages `DBI` (`http://cran.r-project.org/web/packages/DBI/index.html`) and `RMySQL` (`cran.r-project.org/web/packages/RMySQL/index.html`.

5.10 Building your own R packages

You can packaging up your code, data sets and documentation to make a *bona fide* R package. You may wish to do this for particularly large projects that you think will be useful for others. Read *Writing R Extensions* (`cran.r-project.org/doc/manuals/r-release/R-exts.html` manual and see *package.skeleton* to get started. The R tool set EcoDataTools (`https://github.com/DomBennett/EcoDataTools`) and the package `cheddar` were written by Silwood Grad Sudents!

5.11 Sweave

Sweave is a tool that allows you to write your Dissertation or Report such that it can be updated automatically if data or R analysis change. Instead of inserting a prefabricated graph or table into the report, the master document contains the R code necessary to obtain it. When run through R, all data analysis output (tables, graphs, etc.) is created on the fly and inserted into a final latex document. The report can be automatically updated if data or analysis change, which allows for truly reproducible research. Check out the webpage: `http://www.statistik.lmu.de/~leisch/Sweave`

5.12 Practical 3.2 (CMEE Extra Credit)

Autocorrelation in weather

⋆ Make a new script named `TAutoCorr.R`, and save in `Code` directory
⋆ At the start of the script, load and examine and plot `KeyWestAnnualMeanTemperature.Rdata`, using `load()` — This is the temperature in Key West, Florida for the 20th century
⋆ The question this script will help answer is: Is the temperature one year significantly correlated with the next year?
⋆ You will need the `sample` function that we used today — read the help file for `sample` and experiment with it
⋆ Now,
   1. Compute the appropriate correlation coefficient and store it (look at the help file for `cor()`
   2. Then repeat this 10000 times
      – Randomly permute the time series
      – Compute the same correlation coefficient
      – Store it

   3. Then see what fraction of the coefficients from step 2 were greater than that from step 1
⋆ How do you interpret these results? Why? Present your interpretation in a pdf document.

5.13 Practical 3.3 (No *tangible* Credit!)

Your project may not really need GIS, but you would still like to do some mapping.Yo you can do it in R using the `maps` package. In this practical, you will map the Global Population Dynamics Database (`https://www.imperial.ac.uk/cpb/gpdd/gpdd.aspx`) (GPDD). This is a freely available database that was developed at Silwood).
If any of you are interested in doing a project around this database, please contact David Orme or Samraat Pawar! It is a goldmine of as yet under-utilized information. Note that the Living Planet Index (`http://livingplanetindex.org/home/index`) is based upon these data.

⋆ Use `load()` from `GPDDFiltered.RData` that is available on the Git repository — have a look at the database field headers and contents.
⋆ What you need is latitude and logitude information for a bunch of species for which population time series are available in the GPDD
⋆ Now use `install.packages()` to install the package `maps`, as you did with `ggplot2` — hopefully without any problems!
⋆ Now create a script (saved under a sensible name in a sensible location — hint hint!) that:
   1. Loads the maps package
   2. Loads the GPDD data

3. Creates a world map (use the map function, read its help file, also google examples using
4. maps)
5. Superimposes on the map all the locations from which we have data in the GPDD dataframe
6. Compare your map with a fellow student to check

Based on this map, what biases might you expect in any analysis based on the data represented?

5.14 R Module Wrap up

5.14.1 Some comments and suggestions

Thanks for enduring through the week! Learning to program in R or any other language, especially if it's you first-ever effort to learn programming, demands perseverance. Most of you have shown an admirable quantities of this necessary quality, Keep going! I believe most of you have climbed a significant part of a steep learning curve. Here are some things to keep in mind:

- There are many R nerds at Silwood that you can talk to — `They walk among us!`
- There is a Silwood R list that you can subscribe to: `https://mailman.ic.ac.uk/mailman/listinfo/silwood-r`
- However, post questions only as a last resort! Google it first, and even before that, make sure you revise this week's (and stats week's) work.
- Solutions to this weeks Pracs/Exercises will become available by Thursday (13th Nov) next week.

5.14.2 CMEE weekly coursework submission

Test and bring under version control: `TreeHeight.R`, `control.R`, `PP_Lattice.R`, `PP_Regress.R` (where's the extra credit?), `TAutoCorr.R`, `Vectorize1.R`, `Ricker.R`, `Vectorize2.R`, `break.R`, `next.R`, `try.R`, `browse.R`, `TAutoCorr.R` (Extra Credit).

Git commit and push by: **Wednesday, 12 November 2014**

5.15 Readings

- See **An introduction to the Interactive Debugging Tools in R, Roger D Peng** for detailed usage. `http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf`
- Friedrich Leisch. (2002) Sweave: Dynamic generation of statistical reports using literate data analysis. Proceedings in Computational Statistics, pages 575-580. Physica Verlag, Heidelberg, 2002. `http://www.statistik.lmu.de/~leisch/Sweave`
- Remember, R packages come with pdf guides/documentation!

Chapter 6

Back to Python: building workflows

6.1 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 MSc/MRes dissertation, tables, plots, and all. Python is ideal for building such workflows because it has packages for practically every purpose.

The following useful packages 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

The following packages you have to install in terminal using
`sudo apt-get install python-packagename` (as you did for `ipdb` and `scipy` previously):

- `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 biological computation, including bioinformatics

6.2 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:

```python
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! Similarly, to compile your Latex document (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. As imple 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.`. So also for submodules, e.g., type `subprocess.os.` and then tab.

6.2.1 Running `R`

R is likely to an important part of your workflow, for example for statistica 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, you can use in the linux terminal (not in `python`!): `find /usr -name 'Rscript'`.

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

Also check what happens if you run instead (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`.

6.3 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.

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 as 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 text file with 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*

6.3.1 `python` with 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.

We can easily use SQLite through Python scripting. 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 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 build it from a global dataset on wood density from Chave *et al.* (2009), Ecology Letters (should be in your `Data` directory). Navigate to your `Code` directory and in a ubuntu terminal type:

```
$ sqlite3 ../Data/wood.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:

```

sqlite> CREATE TABLE woodb (Number integer primary key,
   ...> Family text,
   ...> Binomial text,
   ...> WoodDensity real,
   ...> Region text,
   ...> ReferenceNumber integer);
```

Now import the dataset:

```
sqlite> .mode csv

sqlite> .import wood.csv woodb
```

So we built a table and imported a csv file into it.

We can ask SQLite to show all the tables we currently have:

```
sqlite> .tables

woodb
```

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

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

1,Fabaceae,"Abarema jupunba",0.78,"South America...
2,Fabaceae,"Abarema jupunba",0.66,"South America...
3,Fabaceae,"Abarema jupunba",0.551,"South America...
4,Fabaceae,"Abarema jupunba",0.534,"South America...
5,Fabaceae,"Abarema jupunba",0.551,"South America...
```

Let's turn on some nicer formatting:

```
sqlite> .mode column

sqlite> .header ON

sqlite> SELECT * FROM woodb LIMIT 5;

Number      Family      Binomial         WoodDensity ...
----------  ----------  ---------------  ----------- ...
1           Fabaceae    Abarema jupunba  0.78        ...
2           Fabaceae    Abarema jupunba  0.66        ...
3           Fabaceae    Abarema jupunba  0.551       ...
4           Fabaceae    Abarema jupunba  0.534       ...
5           Fabaceae    Abarema jupunba  0.551       ...
```

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

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

sqlite> SELECT DISTINCT Region FROM woodb;

Region
----------------------------------------
Africa (extratropical)
Africa (tropical)
Australia
Australia/PNG (tropical)
....
```

```
sqlite> SELECT Binomial FROM woodb
   ...> WHERE Region = "Africa (tropical)";

Binomial
--------------------
Acacia holosericea
Adansonia digitata
Afzelia africana
Afzelia africana
Afzelia africana
....

sqlite> SELECT COUNT (*) FROM woodb;

COUNT (*)
--------------------
16468

sqlite> SELECT Region, COUNT(Binomial)
   ...> FROM woodb GROUP BY Region;

Region                COUNT(Binomial)
--------------------  ---------------
Africa (extratropica  351
Africa (tropical)     2482
Australia             678
Australia/PNG (tropi  1560
Central America (tro  420
....

sqlite> SELECT COUNT(DISTINCT Family)
   ...> FROM woodb;

COUNT(DISTINCT Family
--------------------
191

sqlite> SELECT COUNT(DISTINCT Family)
   ...> AS FamCount
   ...> FROM woodb;

FamCount
--------------------
191

sqlite> SELECT Region,
   ...> COUNT(DISTINCT Family) AS FC
   ...> FROM woodb GROUP BY Region;

Region                FC
--------------------  ----------
Africa (extratropica  74
Africa (tropical)     66
Australia             49
Australia/PNG (tropi  91
Central America (tro  68
....

sqlite> SELECT * # WHAT TO SELECT
   ...> FROM woodb # FROM WHERE
   ...> WHERE Region = "India" # CONDITIONS
   ...> AND Family = "Pinaceae";

Number                Family      Binomial       WoodDensity
--------------------  ----------  -------------  -----------
29                    Pinaceae    Abies pindrow  0.38
3287                  Pinaceae    Cedrus deodar  0.47
12057                 Pinaceae    Picea morinda  0.4
12126                 Pinaceae    Pinus longifo  0.48
12167                 Pinaceae    Pinus wallich  0.43
15858                 Pinaceae    Tsuga brunoni  0.38
```

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 Number FROM woodb LIMIT 5;

Number
--------------------
1
2
3
4
5

sqlite> SELECT Number
   ...> FROM woodb
   ...> WHERE Number > 100
   ...> AND Number < 105;

Number
--------------------
101
102
103
104

sqlite> SELECT Number
   ...> FROM woodb
   ...> WHERE Region = "India"
   ...> AND Number > 700
   ...> AND Number < 800;

Number
--------------------
716
```

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 Region
   ...> FROM woodb
   ...> WHERE Region LIKE "_ndia";

Region
--------------------
India

sqlite> SELECT DISTINCT Region
   ...> FROM woodb
   ...> WHERE Region LIKE "Africa%";

Region
--------------------
Africa (extratropical)
Africa (tropical)

sqlite> SELECT DISTINCT Region
   ...> FROM woodb
   ...> WHERE Region GLOB "Africa*";

Region
--------------------
Africa (extratropica
Africa (tropical)

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

sqlite> SELECT DISTINCT Region
   ...> FROM woodb
   ...> WHERE Region LIKE "africa%";

Region
--------------------
Africa (extratropical)
Africa (tropical)
```

We can also order by any column:

```
sqlite> SELECT Binomial, Family FROM
   ...> woodb LIMIT 5;

Binomial              Family
--------------------  ----------
Abarema jupunba       Fabaceae
Abarema jupunba       Fabaceae
Abarema jupunba       Fabaceae
Abarema jupunba       Fabaceae
Abarema jupunba       Fabaceae

sqlite> SELECT Binomial, Family FROM
   ...> woodb ORDER BY Family LIMIT 5;

Binomial              Family
--------------------  -----------
```

```
Avicennia alba          Acanthaceae
Avicennia alba          Acanthaceae
Avicennia alba          Acanthaceae
Avicennia germinans     Acanthaceae
Avicennia germinans     Acanthaceae
```

So on and so forth (joining tables etc would come next...). You can store your queries and database management commands in an `.sql` file (`geany` will take care of syntax highlighting etc.)

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()
```

6.4 Practical 6.1

1. Bring today's work under version control: `TestR.py`,`TestR.R`, woodb, `pythonsql.py`, `sqlitemem.py`
2. Open `using_os.py` and complete the tasks assigned (hint: you might want to look at `subprocess.os.walk()`)
3. 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

6.5 Readings and Resources

- look up `docs.python.org/2/library/index.html` – Read about the packages you think will be important to you...
- "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`
- Some of you might find the python package `biopython` particularly useful — check out www.biopython.org/wiki/Main_Page, and especially, the cookbook

6.6 Long Practical

We have talked a lot about workflows and confronting models with data. It's time to do something concrete with all the stuff you have learnt!

This practical gives you an opportunity to try the "whole nine yards" of developing and implementing a workflow and delivering a "finished product" to answer a scientific question in biology.

This practical is optional for the MRes', but can provide extra credit if you choose to participate!

huh? how will this work? — One MSc and *upto two* Mres' can collaborate and co-author the report. The MSc is expected to do the bulk of the work, of course! The contributions will be stated in the report (see report subsection below). Every participating MRes, concommitant with her/his contribution, gets some extra credit that can go towards bolstering weaknesses in the weekly practical assessments. I cannot give any more detauls, because I have bever tried this before!

6.6.1 Background and Objectives

The main question you will address is: *How well does a mechanistic model based upon biochemical principles fit a dataset of thermal responses of individual fitness in phytoplankton?*

This is currently a "hot" (no pun intended) topic in biology, with both ecological and evolutionary consequences, as we discussed in the earlier lecture. On the *ecological side*, because the temperature-dependence of metabolic rate sets the rate of intrinsic r_{max} (papers by Savage et al, Brown et al) as well as interactions between species, it has a strong effect on population dynamics. In this context, note that 99.9% of life on earth is ectothermic! On the *evolutionary side*, the temperature-dependence of fitness and species interactions also means that warmer environments may have stronger rates of evolution. This may be compounded by the fact that mutation rates may also increase with temperature (papers by Gillooly et al)!

6.6.2 The Data

These data are for subset of growth rates (time^{-1}) of phtoplankton, collected by experimental studies from across the world, and compiled by Thomas et al and Chen et al. The figure in your `Data` folder shows an example curve along with a fitted curve of the Schoolfield model.

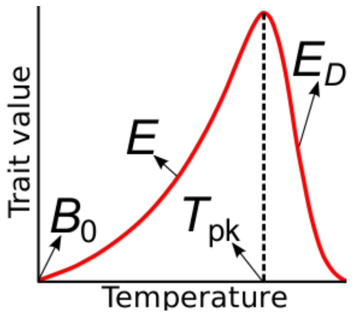


Figure 6.1: Illustration of a unimodal thermal response curve for a biological trait according to the Schoolfiled model (Equation 6.1). B_0 is the trait value at low temperature and controls the offset of the curve. E is the activation energy (eV) which controls the rise of the curve up to the peak, whereas E_D is the de-activation energy (eV) which controls the fall of the curve after the peak. T_{pk} (K) is the temperature where trait performance peaks (B_{pk}).

6.6.3 The Models

There are at least two alternatve models.

The Schoolfield model (paper is in `Readings` directory) is a mechanistic alternative:

$$B(T) = B_0 \cdot \frac{e^{\left[\frac{-E}{k}\left(\frac{1}{T}-\frac{1}{283.15}\right)\right]}}{1+\frac{E}{E_D-E}\cdot e^{\left[\frac{E_D}{k}\left(\frac{1}{T_{pk}}-\frac{1}{T}\right)\right]}} \tag{6.1}$$

Here, k is the Boltzmann constant (8.617×10^{-5} eV $\cdot$ K^{-1}), B the value of the trait at a given temperature T (K) (K = °C + 273.15), while B_0 is the trait value at 283.15 K (10°C) which stands for the value of the growth rate at low temperature and controls the vertical offset of the curve. E is the activation energy (eV) which controls the rise of the curve up to the peak, E_D is the de-activation energy (eV) which controls the fall of the curve after the peak and T_{pk} (K) is the temperature where trait performance is maximal.

The Gaussian-Gompertz model (Martin et al) (paper is in `Readings` directory) is a phenomenological alternative:

$$B(T) = B_{max} \cdot e^{\left[-E\cdot\left(T-T_{pk}\right)^2 - e^{\left[E_D\cdot\left(T-T_{pk}\right)-\theta\right]}\right]} \tag{6.2}$$

where B_{max} is the maximum value of the available measurements and θ a quantity that improves the fit, without any biological interpretation. The temperature parameters (T, T_{pk}) of the Gaussian - Gompertz model (Equation 6.2) are in °C.

6.6.4 Fitting data to the models

You will use Nonlinear Least Squares (NLLS) to fit the two alternative models to data, followed by model selection with AIC and BIC (also known as the Schwartz Criterion — recall the Johnson and Omland paper).

6.6.5 The Workflow

You will build a workflow that starts with the data and ends with a report written in LaTeX. I suggest the following components and sequence in your workflow (you can choose to do it differently!):

1. A `R` script that imports the data and prepares it for NLLS fitting, with the following features:
   - It should create unique ids so that you can identify unique thermal responses (what does this mean?)
   - It should filter out datasets with less than 5 data points (why?)
   - It should deal with negative and zero trait values (why?)
   - The script should add columns containing starting values of the model parameters for the NLLS fitting
   - Save the modified data to a new csv file
2. A python script that opens the new modified dataset (from step 1) and does the NLLS fitting, with the following features:
   - Uses `lmfit` — look up submodules `minimize`, `Parameters`, `Parameter`, and `report_fit`.
     Have a look through `http://lmfit.github.io/lmfit-py`, especially `http://lmfit.github.io/lmfit-py/fitting.html#minimize`

     You will have to install lmfit using `pip` or `easy_install` (you may get some errors while installing but nothing un-solvable!) – One of us can help you out if needed.
     Also have a look (and try out) the examples at: `http://lmfit.github.io/lmfit-py/intro.html` & `http://lmfit.github.io/lmfit-py/parameters.html`, and whatever other examples you can find.
   - Will use the `try` construct because not all runs will converge. Recall the `try` example from R
   - Will calculate AIC, BIC, R^2, and other statistical measures of fit (you decide what you want to include)
   - Will export the results to a csv that the plotting R script (next item) can read.
3. A `R` script that imports the results from the previous step and plots every thermal response with both models (or none, if nothing converges) overlaid — all plots should be saved in a single separate sub-directory. Use `ggplot` for pretty results!
4. A LaTeXsource code that generates your report.
5. A python script (saved in `Code`) called `run_LongPrac.py` that should run the whole project, right down to compilation of the LaTeX document.

Doing all this may seem a bit scary at the start. However, you need to approach the problem systematically and methodically, and you will be OK. I suggest the following to get you started:

- Explore the data in R and get a preliminary version of the plotting script without the fitted models overlaid worked out. That will also give you a feel for the data.
- Explore the two models – be able to plot them. Write them as functions in your `python` script, because that's where you will use them (step 2 above) (you can use `matplotlib` for quick and dirty plotting and then suppress those code lines later).
- Figure out, using a minimal example (say, with one, "nice-looking" thermal response dataset) to see the `python lmfit` module works. Dimitris can help you work out the minimal example, including the usage of `try` to catch errors in case the fitting doesn't converge.

- One thing to note is that you will need to do the NLLS fitting on the logarithm of the the function to facilitate convergence — please ask me or Dimitris if you need help on this.

The Report

The report should,

- be written in LaTeX using the `article` document class
- be double-spaced, with *continuous* line numbers
- have a title, author(s) name(s) with affiliation, brief introduction with objectives of the study, and appropriate additional sections such as methods, data, results, discussion, etc — multiple authors if it is a MSc–Mres collaboration.
- must contain $\leq$4000 words, including references! — there should be a wordcount at the beginning of the document
- all references should be properly cited using bibtex
- if there are multiple authors, the contributions should be stated in detail (who did what part of the coding, etc.) . I am assuming the MSc lead author will have done most of the work!

6.6.6 Patching together your workflow components

Use `python` for this. You can call an run all the components of the above suggested workflow, including compilation of the LaTeXdocument. Look back at the notes to see you would run these different components — we have already covered how to run `R` and compile LaTeXusing `subprocess`.

6.6.7 Submission

Commit and push all your work to your bitbucket repository using to a directory called `CMEELongPrac` at the same level as the `Week1`, `Week2` etc directories.

At this stage, I am not going to tell you how to organize your project — that's one of my marking criteria (see next section).

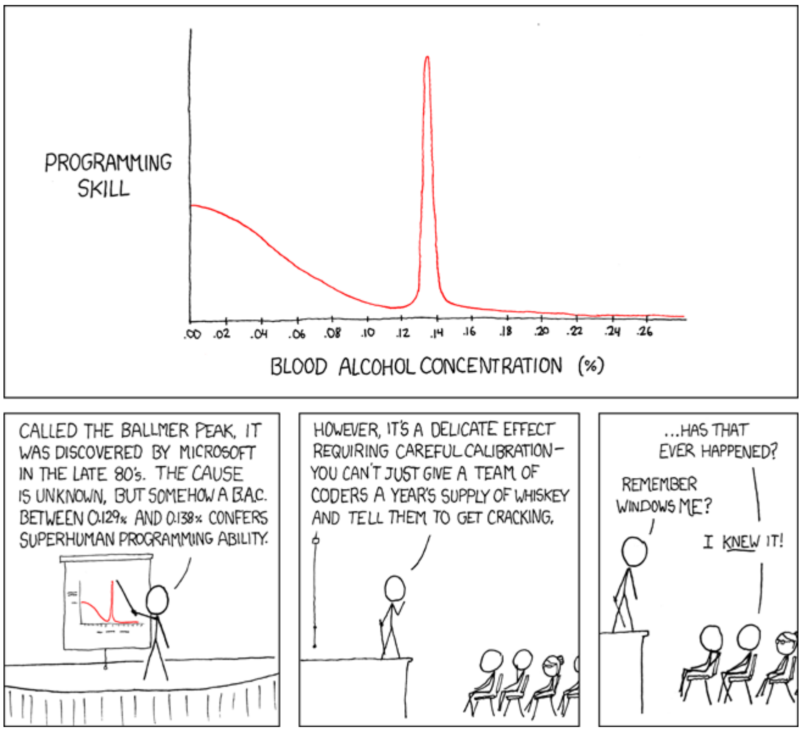
Marking criteria

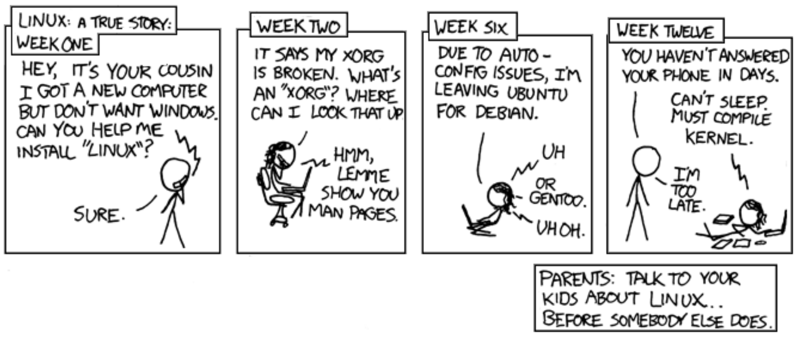
I will be looking for the following in your submission:

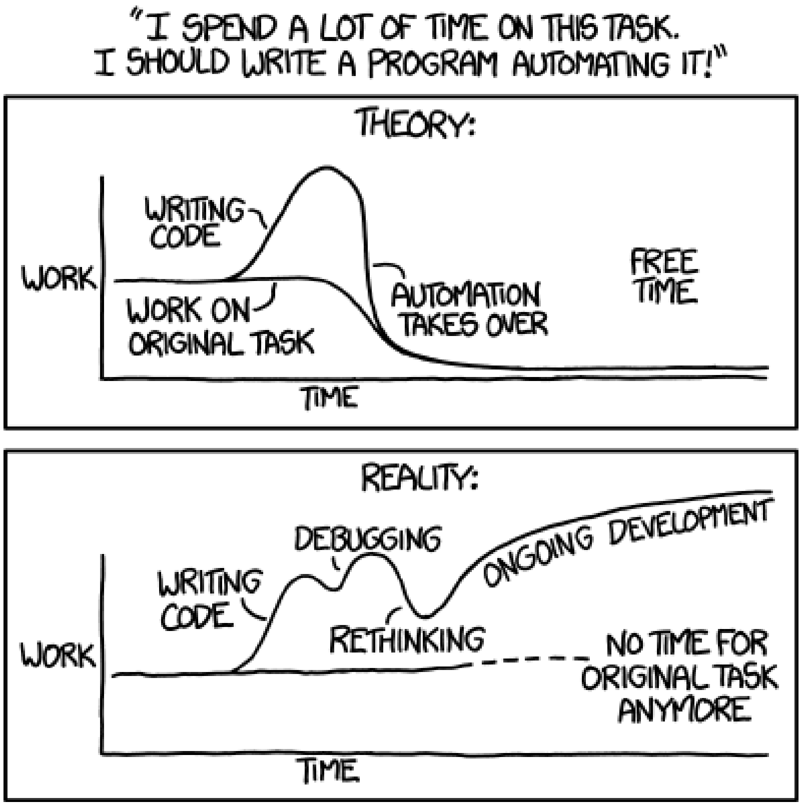
- A well-organized project where code, results, data, etc., are easy to locate and inspect
- A script called `run_LongPrac.py` that runs the project successfully
- A report that contains all the components indicated above in the subsection called "The Report" — I will be looking for some original thought and synthesis in the intro and disussion
- The more thermal response curves you are able to fit, the better — That is part of the challenge!
- Quality of the presentation of the graphics and tables in your report, as well as the plots showing the fits.

Deadline

Commit and push your work to your git repository by December 18 2014, 5PM


PROGRAMMING SKILL
.00 .02 .04 .06 .08 .10 .12 .14 .16 .18 .20 .22 .24 .26
BLOOD ALCOHOL CONCENTRATION (%)
CALLED THE BALLMER PEAK, IT WAS DISCOVERED BY MICROSOFT IN THE LATE 80's. THE CAUSE IS UNKNOWN, BUT SOMEHOW A B.A.C. BETWEEN 0.129% AND 0.138% CONFERS SUPERHUMAN PROGRAMMING ABILITY.
HOWEVER, IT'S A DELICATE EFFECT REQUIRING CAREFUL CALIBRATION – YOU CAN'T JUST GIVE A TEAM OF CODERS A YEAR'S SUPPLY OF WHISKEY AND TELL THEM TO GET CRACKING.
...HAS THAT EVER HAPPENED?
REMEMBER WINDOWS ME?
I KNEW IT!


LINUX: A TRUE STORY:
WEEK ONE
HEY, IT'S YOUR COUSIN I GOT A NEW COMPUTER BUT DON'T WANT WINDOWS. CAN YOU HELP ME INSTALL "LINUX"?
SURE.
WEEK TWO
IT SAYS MY XORG IS BROKEN. WHAT'S AN "XORG"? WHERE CAN I LOOK THAT UP
HMM, LEMME SHOW YOU MAN PAGES.
WEEK SIX
DUE TO AUTO-CONFIG ISSUES, I'M LEAVING UBUNTU FOR DEBIAN.
UH
OR GENTOO.
UH OH.
WEEK TWELVE
YOU HAVEN'T ANSWERED YOUR PHONE IN DAYS.
CAN'T SLEEP. MUST COMPILE KERNEL.
I'M TOO LATE.
PARENTS: TALK TO YOUR KIDS ABOUT LINUX... BEFORE SOMEBODY ELSE DOES.


"I SPEND A LOT OF TIME ON THIS TASK. I SHOULD WRITE A PROGRAM AUTOMATING IT!"
THEORY:
WORK
WRITING CODE
WORK ON ORIGINAL TASK
AUTOMATION TAKES OVER
FREE TIME
TIME
REALITY:
WORK
WRITING CODE
DEBUGGING
RETHINKING
ONGOING DEVELOPMENT
NO TIME FOR ORIGINAL TASK ANYMORE
TIME

www.xkcd.com