Introduction to Biological Computing

AKA "The Multilingual Quantitative Biologist"

Department of Life Sciences (Silwood Park Campus)
Imperial College London

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

October 3, 2017

Contents

Chapter 0

Introduction

> It is hard for me to say confidently that, after fifty more years of explosive growth of computer science, there will still be a lot of fascinating unsolved problems at peoples' fingertips, that it won't be pretty much working on refinements of well-explored things. Maybe all of the simple stuff and the really great stuff has been discovered. It may not be true, but I can't predict an unending growth. I can't be as confident about computer science as I can about biology. Biology easily has 500 years of exciting problems to work on, it's at that level.
>
> — *Donald Knuth*

0.1 About this document

These document contains the content of modules on Biological Computing taught in various courses at the Department of Life Sciences, Imperial College London. These courses include Year 1&2 Computational Biostatistics modules at the South Kensington Campus, the MSc/MRes on Computational Methods in Ecology and Evolution (CMEE Masters) at Silwood Park, and the Quantitative Methods in Ecology and Evolution Centre for Doctoral Training (QMEE CDT).

Different subsets of this document will be covered in different courses. Please look up your respective course guidebooks/handbooks to determine when the modules covered in these notes are scheduled in your course. You will be given instructions about which sections are covered in your course.

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/silbiocompmasterepo`. I use git for hosting this course's materials because I want to version-control this course's content, which is constantly evolving to keep up with changing programming/computing technologies. That is, I am treating this course as any computing project that needs to be regularly updated and improved.

Changes to the notes and content will also be made based upon students' feedback. Blackboard is just not set up to handle dynamic updating and version control of this sort! You will see tips like this in the following chapters that you should pay special attention to:

Tip

If you do not use git (or are not required to do so!), you may download the code, data, these notes, and other course materials from the bitbucket repository at one go, by going to `https://bitbucket.org/mhasoba/silbiocompmasterepo/downloads` and then clicking on the "Download repository" link. You can then unzip the downloaded .zip and grab the files you need.

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 and computing, you learn faster by trying to solve problems (including computer crashes!) on your own, often by liberally googling the problem!

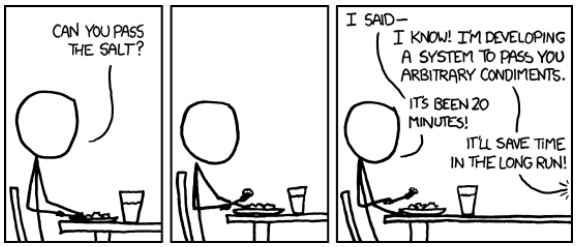


Figure 1: But not every task needs to be converted to a computer program — you will learn to decide when to and when not to write a computer program! `http://xkcd.com/974/`

You will be provided guidelines for what makes good or efficient solutions to the computing exercises. Later, when you have submitted your exercises and practicals, you will be provided solutions.

Also, every time a mysterious, geeky-sounding term like "relative path" or "version control" appears, please google it!

0.2 The goals of this course

The goal of this course is to teach you to become (or at least show you the path towards becoming) a competent quantitative biologist. A large part of this involves learning computer programming. Why do biologists need to write write computer programs? Here are some (hopefully compelling to you!) reasons:

- Short of fieldwork, programs can do anything (that can be specified). In fact, even fieldwork, if you could one day *program* a robot to do it for you [1]!
- As such, no software is typically available to perform exactly the analysis you are planning. You should be unhappy if you are trying to shoehorn your data into methods that don't quite seem right.
- Biological problems and datasets are some of the most complicated imaginable. Programming permits success despite complexity through precise specification and modularization

[1] That way you can potter around the forest catching rare butterflies and frogs while the robot does the boring data collecting for you

of complicated analyses.

- Modularity – programming allows you to break up your complex analysis in smaller pieces, yet keep all the pieces in a single, functional analysis.
- Reproducibility – you (or someone else) can just re-run the code to reproduce your analysis. This is also the key to maintaining scientific accountability, integrity, and accuracy.
- Organised thinking – writing code requires you to do this!
- Career prospects – good, scientific coders are in short supply in all fields, but most definitely in biology!

There are several hundred programming languages currently available – which ones should a biologist choose? Ideally, a quantitative biologist would like to know:

1. A *fast*, compiled (or semi-compiled) 'procedural' language like `C`
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`

And all these because one language doesn't fit all purposes. Therefore you will learn a few different languages in this course — hopefully, just the right number! Among the languages you will learn here — `python`, `R`, and `C` are three of the most popular currently (see `https://www.tiobe.com/tiobe-index/` and `https://goo.gl/vyrqr1`, and for some very good reasons.

Our goal is to teach you not just programming, but also good computing practices. In this course, you will write plenty of code, deal with different data files, and produce text and graphic outputs. You will learn to keep your project and coursework organized in logical, efficient, error-free and reproducible *workflows* (that's a mouthful, but an important mouthful).

0.3 Some guidelines, conventions and rules

0.3.1 Beware the dark forces

You will NOT be using spreadsheet software (e.g., Excel) in these classes. There are times when you will feel the pull of the dark side (ahem!), and imagine a more "comfortable" world where you are mouse-clicking your way happily though Excel-based data manipulations and analyses. NO! You will be doing yourself a disservice. On the long-ish run you will be much better off visualizing and manipulating data on your computer using a programming language like R. This is something you will learn, young *padawan*!

0.3.2 Keep your workflow organized

In the following chapters, you will practice many examples where you are required to write large blocks of code. Please get into the habit of writing code into text files with an appropriate extension (e.g., `*.R` for `R` code, `*.py` for `python` code, etc.). Furthermore, please keep all your code files organized in one or more directories (e.g., named `Code`!). Similarly, some of these scripts will take data files as inputs, and output some results in the form of text or graphics. Please keep these

inputs and outputs organized as well, in separate directories (e.g., named `Data` and `Results`) respectively. Your demonstrators and I will help you get set up and abide by this "workflow".

0.3.3 Conventions used in this document

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, but google it!).

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

0.3.4 To IDE or not to IDE?

As you embark on your journey to becoming a competent prectioner of biological computing, you will be faced with a hamletian question: "To IDE or not to IDE". OK, maybe not that dramatic or hamletian...

An interactive Development Environment (IDE) is a text editor with frills that can make life easy by auto-formatting code, running code though the terminal or command line, 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 code editor that can also offer an IDE (e.g., emacs, vim, geany, atom). At the very least, your IDE should offer:

- Auto-indentation
- Automatic code wrapping (e.g., keeping lines <80 characters long)
- Syntax highlighting (e.g., commands vs. variables)
- Code folding (fold large blocks of code, say an entire function or loop)

Figure 2: Logical workflows are important, but don't get married to yours! `http://xkcd.com/1172/`

- Keyboard control of commenting/uncommenting, code wrapping, etc.
- Embedded terminal / shell / commandline console
- Sending commands to terminal / shell

And if you end up using multiple programming languages, you will want an 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 plugins that make multi-language (multilingual?) code development much easier. I would also recommend `vim` or `emacs`, which have a steeper learning curve, but are very powerful once you have mastered them. There are also some new and (increasingly popular) kids on the block: `atom` and `vstudio code`.

0.3.5 Course Assessment

Undergraduates

Assessment will be through a computer-based test. You be expected to be able to apply the concepts you have learnt to address questions by using appropriate R input and interpreting R output.

Masters students

If you are a CMEE Masters student, we will assess both your practical computing work itself (including any writeups), and whether you are following good programming and workflow practices, on a weekly basis. This will be done using scripts — yes we will assess your scripts using scripts! A `python` script will check whether your weekly (and version-controlled) directories are neat and organized in a logical workflow, and whether all the scripts run correctly with the expected inputs and outputs, starting from Week 1 (Chapter 1). Specifically, as an example towards learning good workflow practices, you will keep all your coursework code, data inputs and results outputs organized in separate directories named `Code`, `Data`, `Results` (or equivalent) respectively.

The assessment script will then record a log file that summarizes all the issues found in your workflows, which will be emailed to you by the middle of the week (usually on the wednesday) subsequent to the one you submitted your weekly practical work in.

Note that practicals in the weeks/modules not in these notes (e.g., GIS, Genomics, Population Genetics) will also be included in the assessment. The basic rules you must follow, irrespective of a Week's content are:

1. All code/scripts go to a `Code` directory
2. All data go to a `Data` directory
3. All results go to a `Results` directory, but the Results directory should be empty when you submit your week's work, as it will be populated automatically when the assessment script runs.
4. If you have files that don't fit in these categories, put them additional, meaningfully named directories (e.g, "`Writeup`").
5. No single file should be greater than 100 mb, either data or script/code. If a script needs a data file, but the example data file is >100 mb, reduce it to a minimum working dataset and

upload that, keeping the main data file(s) under `.gitignore` (more on this soon!). Keep the main data backed up of course [2].

6. Most importantly, all python, R, bash, and LaTeXscripts should run OK, taking in data and spitting out the results as necessary (these are the scripts the assessment code will try to run)

When necessary, more specific, module-specific details on weekly coursework and assessment will be given when the module starts.

The computing coursework marking criteria are given in the Appendix A.

The final assessment of weekly coursework

A written summary assessment of your overall performance with your marks will be sent after the end of the 9 weeks.

The weekly assessments are to help you spot general, as well as programming language-specific issues with your computing coursework on a regular basis. You may and should fix bugs and other problems that feedback logs bring to your attention. I will have a look at how much you addressed the issues in the final assessment. The final assessment will be necessarily a more subjective than the weekly assessments as we are looking to give you an overall picture of how you did and what you can improve on. To give you an idea about the criteria for the overall assessment, a set of summative marking criteria are also given in Appendix A titled "MARKING CRITERIA for EXAMS and ESSAYS and COURSEWORK".

Alright, full steam ahead then!

[2] You could make a separate directory called `TestData` as the default input and keep the main Data file in the `.gitignore` file — see Chapter 3

Chapter 1

Introduction to UNIX and Linux

1.1 What is UNIX?

UNIX is a machine-independent operating system (OS) developed in the 1970s by AT&T programmers (notably Brian Kernighan and Dennis Ritchie, fathers of `C`) for programmers (you!). It is multi-user and network-oriented by design, uses plain text files for storing data (no proprietary file formats), and has a strictly hierarchical directory structure (more on this below). This makes it an ideal environment for developing your code and storing your data.

Linux and Mac OS are Unix-like (or UN*X or *nix) operating systems that have evolved from UNIX. Ubuntu is a Linux distribution.

1.2 Why UNIX?

Here are some good reasons:

- It was designed for developing code and storing data — an ideal native habitat for programming languages like Python and R!
- Robust, stable, secure (very few UNIX viruses and malware — I have never encountered one!)
- Free and open source!
- Scores of small programs available to perform simple tasks – can be combined easily
- Easy to automate tasks (e.g., using shell scripts)
- Multi-user (multiple users can log in concurrently use computer)
- Multi-tasking (can perform many tasks at the same time)
- Network-ready (easy to communicate between computers)
- UNIX has been around since the early 1970's and will likely be around at the end of your career (the hard work you are putting into learning UNIX will pay off over a lifetime!)
- Amazing support — a large body of tutorials and support web sites are readily available online.
- Basically all resources for High-Performance Computing (computer clusters, large workstations, etc.) run a UNIX or Linux operating system.

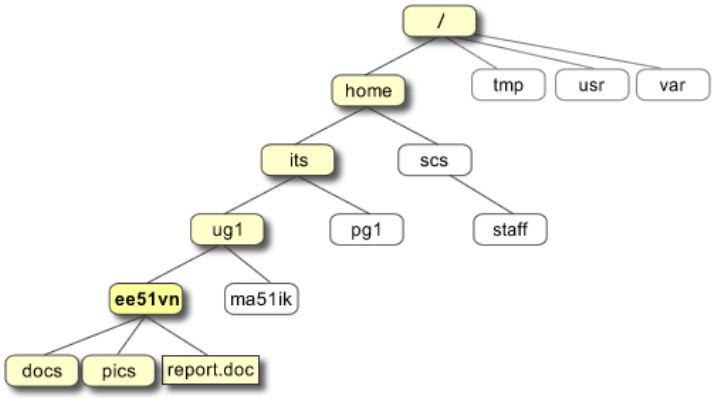
See `http://whylinuxisbetter.net/` (I would have chosen a more subtle domain name though!) to aid your brain-washing (cleaning?).

Also, if you like history: `https://www.howtogeek.com/182649/htg-explains-what-is-unix`

Tip

The terms 32-bit and 64-bit refer to the way a computer's processor (also called a CPU), handles information. A 64-bit OS handles large amounts of random access memory (RAM) more effectively than a 32-bit system. Specifically, while 32 bits of information can only access 4 GB of RAM, a 64-bit machine can access essentially unlimited system memory (though this is not yet physically possible)!

1.3 UNIX directory structure



`https://pathanruet.files.wordpress.com/2012/05/unix-tree.png`

The UNIX directory (same as "Folder") structure is hierarchical, with a single tree starting from the "root" `/`. This is quite unlike Windows or MS-DOS, where there are separate trees for disk partitions, removable media, network, etc. The key UNIX directories are:

`/`	Is the "root" directory
`/bin`	Contains basic programs
`/etc`	Contains configuration files
`/dev`	Contains files connecting to devices (keyboard, mouse, screen, etc.)
`/home`	Your home directory – this is where you will usually work
`/tmp`	Contains Temporary files

This hierarchical directory structure makes navigating your computer from the terminal/shell (coming up next!) or encoding this navigation in your computer programs easier.

Tip

When you install Linux (say, a Ubuntu version), you will have the opportunity to create separate `swap`, `root` and `home` partitions on your hard disk. `swap` necessarily needs to be a separate partition, while `root + home` can be on the same partition. If you are a Linux newbie, I suggest that you create a separate `home` partition, even though it is not necessary — that way, even if you break your linux install, you can easily reinstall it by just wiping the root partition, without losing any of your data (which sits in `home`).

1.4 Meet the UNIX shell

The shell (or terminal) is a text command processor to interface with the Operating System's "kernel". We will use the popular (yes, it's popular!) `bash` shell.

⋆ To launch `bash` shell, do `Ctrl + Alt + t` (or use `Meta` key) — try it now.

OK, so you have met the shell. Note that:

- The shell automatically starts in your home directory `/home/yourname/`, also called ∼ (important to remember!)
- Use the `Tab` key – very handy (try `ls` with `Tab Tab`)
- You can navigate commands you previously typed using the up/down arrows

Other useful keyboard shortcuts are:

`Ctrl + A`	Go to the beginning of the line
`Ctrl + E`	Go to the end of the line
`Ctrl + L`	Clear the screen
`Ctrl + U`	Clear the line before cursor position
`Ctrl + K`	Clear the line after the cursor
`Ctrl + C`	Kill whatever you are running
`Ctrl + D`	Exit the current shell
`Ctrl + right arrow`	Move cursor forward one word
`Ctrl + left arrow`	Move cursor backward one word

1.5 `sudo`, and installing/removing software

You can install software in your `/home` directory. In UNIX you originally had to login as `root` (administrator). But in Ubuntu, it is sufficient to add `sudo` (`s`uper `u`ser `do`) in front of a command:

```
sudo apt-get install geany geany-plugins geany-plugin-latex
geany-plugin-addons
```

You can install anything that is in the Ubuntu "repository". Let's try installing something else — a weather indicator that I think really works very well.

But here you have to first install the repository:

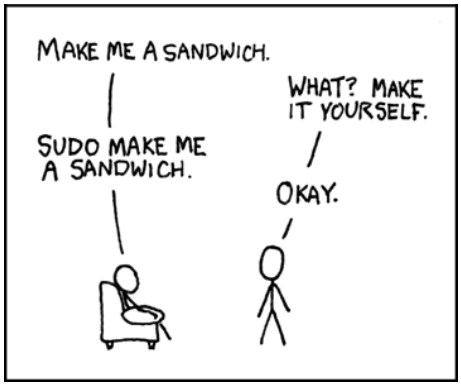
```
$ sudo add-apt-repository ppa:atareao/atareao
```

This will ask you to first approve the additon of this repository. Then update the repository packages,

```
$ sudo apt-get update
```

And then install,

```
$ sudo apt-get install my-weather-indicator
```



http://xkcd.com/149/

You can also easily remove software by, well, using the `remove` command! You will find that commands names are quite intuitive, as they should be. If you think a command with a certain name should exist, it often does!

```
$ sudo apt-get remove indicator-messages
```

This will get rid of the evolution mail indicator — very unlikely that you will use evolution!

1.6 Basic UNIX commands

`man [COMMAND]`	Show help page of a command.
`whoami`	Display your user-name.
`pwd`	Show the current directory.
`ls`	List the files in the directory.
`cd [DIRNAME]`	Change directory.
`cd ..`	Move one directory up.
`cd /`	Go to the root directory.
`cd ~` or just `cd`	Go to the home directory.
`cp [FROM] [TO]`	Copy a file or directory non-recursively (what's this?).
`mv [FROM] [TO]`	Move or rename a file or directory.
`touch [FILENAME]`	Create an empty file.
`echo "My string"`	Print a string (here, "My string").
`rm [TOREMOVE]`	Remove a file or directory non-recursively.
`mkdir [DIRNAME]`	Create a directory.
`rmdir [DIRNAME]`	Remove an empty directory.
`wc [FILENAME]`	Count the number of lines and words in a file.
`sort [FILENAME]`	Sort the lines of a file and print result.
`uniq`	Shows only unique elements of a list.
`cat [FILENAME]`	Print the file on the screen.
`less [FILENAME]`	Progressively print a file on the screen ("q" to exit).
`head [FILENAME]`	Print the first few lines of a file.
`tail [FILENAME]`	Print the last few lines of a file.
`history`	Show the last commands you typed.
`date`	Print current date.
`file [FILENAME]`	Determine the type of a file.
`passwd`	Change user password.
`chmod [FILENAME]`	Change file permissions.

1.7 Building your coursework directory structure

It is time to start building your CMEE coursework directory structure. Please follow these rules:

- Do all your work in `CMEECourseWork`, located in a suitable place in your `/home` (make mama proud, keep your `home` organized!)
- Each week's coursework should be in its respective directory: `CMEECourseWork/Week1`, `CMEECourseWork/Week2`, etc
- Each week's directory will contain directories called `Code`, `Data`, etc (later)
- You will bring `CMEECourseWork` and all it's contents under version control using Git

(later)

Tip

Don't forget to use the `tab` key. It autocompletes directory names for you (same in `R` and `Python`, and/or provides a list of files in the current directory (hit `tab` twice after certain commands. For example, try double tab after typing `ls` at the bash prompt.)

⋆ OK, `mkdir` your `CMEECourseWork` directory now.

Starting by creating `CMEECourseWork`). Then,

```
$ mkdir Week1
$ cd Week1
$ mkdir sandbox
$ cd Sandbox
bash: cd: Sandbox: No such file or directory
$ cd ..
$ rm sandbox
rm: cannot remove `sandbox/`: Is a directory
$ mv sandbox Sandbox # OR, "rm -r sandbox" (careful with the -r option!)
```

Note the hash mark # above — anything after a # is ignored (so you can use it for commenting).

```
$ cd Sandbox
$ pwd
$ ls
$ touch TestFile # OR, "touch TestFile.txt"
$ ls
$ mv TestFile TestFile2
$ rm TestFile2
```

You could have made your project directories and subdirectories in one swoop by using the `-p` option of `mkdir`:

```
$ mkdir -p CMEECourseWork/Week1/{Data,Code,Sandbox}
```

1.8 Command arguments

Most UNIX commands accept arguments that modify their behavior. E.g., `ls -l` (ls "minus"l) lists the files in longer format. Some useful arguments:

`cp -r [DIR1] [DIR2]`	Copy a directory recursively (i.e., including all the subdirectories and files).
`rm -i [FILENAME]`	Remove a file, but asks first (for safety).
`rm -r [DIR]`	Remove a directory recursively (i.e., including all the subdirectories and files).
`ls -a`	List all files, including hidden ones.
`ls -h`	List all files, with human-readable sizes (Mb, Gb).
`ls -l`	List all files, long format.
`ls -S`	List all files, order by size.
`ls -t`	List all files, order by modification time.
`ls -1`	List all files, one file per line.
`mkdir -p Dir1/Dir2/Dir3`	Create the directory Dir3 and Dir1 and Dir2 if they do not already exist.
`sort -n`	Sort all the lines, but use numeric values instead of dictionary (i.e., 11 follows 2).

You can also combine command arguments. Try:

```
$ ls -1t #combines -l and -t
```

1.9 Redirection and pipes

Output of programs can also be "redirected" to a file:

>	Redirect output from a command to a file on disk. If the file already exists, it will be overwritten.
>>	Append the output from a command to a file on disk. If the file does not exist, it will be created.

Examples (make sure you are in `Week1/Sandbox`):

```
$ echo "My first line." > test.txt
$ cat test.txt
$ echo "My second line" >> test.txt
$ ls / >> ListRootDir.txt
$ cat ListRootDir.txt #Isn't that cool?!
```

We can also concatenate commands using "pipes" with "|" e.g., to count how many files are in root (*/*) directory:

```
$ ls / | wc -l # what does this do? Look up "man wc"
```

Or try

```
$ ls -lt | head -5 #what does this do?
```

1.10 Wildcards

We can use wildcards to find files based on their names (again, in `Week1/Sandbox` !):

```
$ mkdir TestWild
$ cd TestWild
$ touch File1txt
$ touch File2.txt
$ touch File3.txt
$ touch File4.txt
$ touch File1.csv
$ touch File2.csv
$ touch File3.csv
$ touch File4.csv
$ touch Anotherfile.csv
$ touch Anotherfile.txt
$ ls
$ ls | wc -l
```

We will use the following wildcards:

`?`	Any single character, except a leading dot (hidden files).
`*`	Zero or more characters, except a leading dot (hidden files).
`[A-Z]`	Define a class of characters (e.g., upper-case letters).

Now let's try to find the files using wildcards:

```
$ ls *
$ ls File*
$ ls *.txt
$ ls File?.txt
$ ls File[1-2].txt
$ ls File[!3].*
```

1.11 Using `grep`

`grep` is a command that matches strings in a file (why is this useful?). It is based on regular expressions (more on this later). Let's explore some basic usage of `grep`. For a test file let's download a list of protected species from the UN website (to `Sandbox`):

```
$ wget http://www.cep.unep.org/pubs/legislation/spawannxs.txt #Cool!
$ head -n 50 spawannxs.txt #You will see "head" in R as well
```

Now,

```
$ mkdir ../Data #Note the relative path "../"
$ mv spawannxs.txt ../Data/
$ cd ../Data
$ head -n 50 spawannxs.txt
```

Note that now you have a `Data` directory.

OK, what about falcons?

```
$ grep Falco spawannxs.txt
Falconidae  Falco       femoralis septentrionalis
Falconidae  Falco       peregrinus
Falconidae  Polyborus   plancus
Falconidae  Falco       columbarius
```

Using `-i` make the matching case-insensitive:

```
$ grep -i Falco spawannxs.txt
Order:  FALCONIFORMES
Falconidae  Falco       femoralis septentrionalis
Falconidae  Falco       peregrinus
Falconidae  Polyborus   plancus
Order:  FALCONIFORMES
Order:  FALCONIFORMES
Order:  FALCONIFORMES
Falconidae  Falco       columbarius
```

Now let's find the beautiful "Ara" macaws:

But this poses a problem (what is the problem?):

```
$ grep -i ara spawannxs.txt
Flacourtiaceae  Banaras     vanderbiltii
Order:  CHARADRIIFORMES
Charadriidae    Charadrius  melodus
Psittacidae Amazona     arausica
Psittacidae Ara     macao
Dasyproctidae   Dasyprocta  guamara
Palmae      Syagrus (= Rhyticocos)  amara
Psittacidae Ara     ararauna
Psittacidae Ara     chloroptera
Psittacidae Arao        manilata
Mustelidae  Eira        barbara
Order:  CHARADRIIFORMES
```

We can solve this by specifying `-w` to match only full words:

```
$ grep -i -w ara spawannxs.txt
Psittacidae Ara     macao
Psittacidae Ara     ararauna
Psittacidae Ara     chloroptera
```

And also show line(s) after the one that was matched, we can use `-A x`, where `x` is number of lines to use:

```
$ grep -i -w -A 1 ara spawannxs.txt
Psittacidae Ara     macao

--
Psittacidae Ara     ararauna
Psittacidae Ara     chloroptera
Psittacidae Arao        manilata
```

Similarly, `-B` shows the lines before:

```
$ grep -i -w -B 1 ara spawannxs.txt
Psittacidae Amazona     vittata
Psittacidae Ara     macao
--
Psittacidae Amazona     ochrocephala
Psittacidae Ara     ararauna
Psittacidae Ara     chloroptera
```

Use `-n` to show the line number of the match:

```
$ grep -i -w -n ara spawannxs.txt
216:Psittacidae Ara     macao
461:Psittacidae Ara     ararauna
462:Psittacidae Ara     chloroptera
```

To print all the lines that do not match a pattern, use `-v`:

```
$ grep -i -w -v ara spawannxs.txt
```

To match one of several strings, use `grep "string1\|string2" file`. `grep` can be used on multiple files, all files, using wildcards for filenames, etc – explore as and when you need.

1.12 Finding files with `find`

Its's easy to find files in UNIX using `find`! Let's test it (make sure you are in `Sandbox`, not `Data`!)

```
$ mkdir TestFind
$ cd TestFind
$ mkdir -p Dir1/Dir11/Dir111 #what does -p do?
$ mkdir Dir2
$ mkdir Dir3
$ touch Dir1/File1.txt
$ touch Dir1/File1.csv
$ touch Dir1/File1.tex
$ touch Dir2/File2.txt
```

```
$ touch Dir2/file2.csv
$ touch Dir2/File2.tex
$ touch Dir1/Dir11/Dir111/File111.txt
$ touch Dir3/File3.txt
```

Now find particular files:

```
$ find . -name "File1.txt"
./Dir1/File1.txt
```

Using `-iname` ignores case, and you can use wildcards:

```
$ find . -iname "fi*.txt"
./Dir1/File1.txt
./Dir1/Dir11/Dir111/File111.txt
./Dir3/File3.txt
./Dir2/File2.txt
```

You can limit the search to exclude sub-directories:

```
$ find . -maxdepth 2 -name "*.txt"
./Dir1/File1.txt
./Dir3/File3.txt
./Dir2/File2.txt
```

You can exclude certain files:

```
$ find . -maxdepth 2 -not -name "*.txt"
.
./Dir1
./Dir1/File1.tex
./Dir1/File1.csv
./Dir1/Dir11
./Dir3
./Dir2
./Dir2/File2.tex
./Dir2/File2.csv
```

To find only directories:

```
$ find . -type d
.
./Dir1
./Dir1/Dir11
./Dir1/Dir11/Dir111
./Dir3
./Dir2
```

Tip

There are are many ways in which you can tweak your Linux/UNIX environment and bash/terminal to your likes. For example, see `http://www.howtogeek.com/tag/ubuntu/ubuntu-tips/`.

But be careful, it can be addictive, and sometimes dangerous to your system's stability!
Here are a couple of tweaks that I really find useful:

Opening nautilus from terminal
In terminal you can enter "f" to open nautilus in current directory by doing the following. Open your `.bashrc` **for editing:**

```
$ sudo gedit ~/.bashrc
```

Then add to the last line (type it, don't copy and paste!):
`alias f='nautilus .'`
Then restart terminal or in current terminal:

```
$ source ~/.bashrc
```

Enabling autocomplete in terminal
What happens when you use up and down keys in terminal? If nothing, then you need to enable reverse searching history. To do so, open `/etc/inputrc`

```
$ sudo geany /etc/inputrc
```

Then, add the following to it:

```
## arrow up
"\e[A":history-search-backward
## arrow down
"\e[B":history-search-forward
```

Then close current terminal, open new one, and try up and down keys again.

1.13 Practical: Make sure the basics work

1. **Some instructions**:

   Review (especially if you got lost along the way) and make sure you can run and understand all the commands and get the expected outputs we have covered today.

   Make sure you have your directory organized with `Data` and `Sandbox` with the necessary files, under `CMEECourseWork/Week1`.

   Along with the completeness of the practicals/exercises themselves, you will be marked on the basis of how complete and well-organized your directory structure and content is – in all coming weeks as well.

2. Here is a more complicated bash command using two pipes `you are not expected to include the answer to this one as part of your weekly submission`:

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

What does this command do (Hint: try it on the test directories and files we created in `Sandbox`)?

Note that along with the `man` command, you can use the internet to get help on practically everything about UNIX!

3. In the directory `UNIX/Data/fasta` you find some FASTA files. These files have an header starting with > followed by the name of the sequence and other metadata. Starting from the second line, we have the sequence data. Write a file called `UnixPrac1.txt` with UNIX shell commands that do the following (number each command with a hashed comment like so – # 1, # 2, etc):

   (a) Count how many lines are in each file

   (b) Print everything starting from the second line for the `E. coli` genome

   (c) Count the sequence length of this genome

   (d) Count the matches of a particular sequence, "ATGC" in the genome of *E. coli* (hint: Start by removing the first line and removing newline characters)

   (e) Compute the AT/GC ratio

   Save `UnixPrac1.txt` in the `Code` directory. Please make sure that each command calls the data from the `Data` directory! Do not write any of the above as shell scripts (that's not been covered yet; see Chapter 2) — each one should be a single line solution made of (potentially piped together) UNIX commands.

   Please put (judicious) comments in any of your script files. But you won't be penalized if you haven't put in comments in the first week in practicals. From the first Python week (Chapter 5) onwards, you will be penalizeed if you don't properly document and comment code (more on this next week), even if you weren't explicitly asked to.

1.14 Readings & Resources

IC library gives you with access to several e- and paper books on UNIX, some specific to Ubuntu. Browse or search and find a good intro book.

- Lots of UNIX tutorials out there. Try `http://software-carpentry.org/lessons.html` (Chapter "shell").

- Some good UNIX usage habits: `http://www.ibm.com/developerworks/aix/library/au-badunixhabits.html`

- List of UNIX commands along with man page: `http://archive.oreilly.com/linux/cmd/`

Chapter 2

Advanced UNIX: Shell scripting

2.1 Shell scripting: What and Why

Instead of typing all the UNIX commands we need to perform one after the other, we can save them all in a file (a "script") and execute them all at once.

The `bash` shell we are using provides a proper syntax that can be used to build complex command sequences and scripts.

Scripts can be used to automate repetitive tasks, to do simple data manipulation or to perform maintenance of your computer (e.g., backup). Indeed, most data manipulation can be handled by scripts without the need of writing a proper program.

2.2 Scripting: How

There are two ways of running a script, say `myscript.sh`:

1. The first is to call the interpreter bash to run the file (try this, but won't work as you don't have a `myscript.sh` script !)

   ```
   $ bash myscript.sh # OR sh myscript.sh
   ```

   (A script that does something specific in a given project)

2. OR, make the script executable and execute it:

   ```
   $ chmod +x myscript.sh
   $ myscript.sh
   ```

   (A script that does something generic, and is likely to be reused again and again – can you think of examples?)

The generic scripts of type (2) can be saved in `username/bin/` and made executable (the `.sh` extension not needed)

```
$ mkdir ~/bin
$ PATH=$PATH:$HOME/bin #Tell UNIX to look in /home/bin for commands
```

2.3 Your first shell script

Let's write our first shell script! For starters,

- ⋆ Write and save `boilerplate.sh` in `CMEECourseWork/Week1/Code`, and add the following script to it (type it in a code editor like geany):

```
#!/bin/bash
# Author: Your Name your.login@imperial.ac.uk
# Script: boilerplate.sh
# Desc: simple boilerplate for shell scripts
# Arguments: none
# Date: Oct 2015

echo -e "\nThis is a shell script! \n" #what does -e do?

#exit
```

The first line is a "shebang" (or sha-bang or hashbang or pound-bang or hash-exclam or hash-pling! – Wikipedia). It can also can be written as `#!/bin/sh`. It tells the bash iterpreter that this is a bash script and that it should be interpreted and run as such. The hash marks in the following lines tell the interpreter that it should ignore the lines following them (that's how you put in script documentation (who wrote the script and when, what the script does, etc.) and comments on particular line of script.

Tip

Geany users can enable send lines of code directly to terminal using a keyboard key combination through two configuration steps:

1. **Enable "Send Selection to Terminal" with the `<Primary>Return` keys by going to geany's `Edit > Preferences > Keybindings` menu item.**
2. **Now edit (e.g., using geany) the file `geany.conf`. You can use geany itself to this:**

```
$ geany ~/.config/geany/geany.conf
```

This will open `geany.conf` in geany. In this file, set `send_selection_unsafe=true`, then close the file, and restart geany.

Now run your boilerplate shell script by typing in the terminal:

```
$ bash boilerplate.sh
```

2.4 A useful shell-scripting example

Let's write a shell script to transform comma-separated files (csv) to tab-separated files and vice-versa. This can be handy — for example, in certain computer languages, it is much easier to read tab or space separated files than csv (e.g., `C`)

To do this, in the bash we can use `tr`, which deletes or substitute characters. Here are some examples.

```
$ echo "Remove    excess      spaces." | tr -s "\b" " "
Remove excess spaces.
$ echo "remove all the as" | tr -d "a"
remove ll the s
$ echo "set to uppercase" | tr [:lower:] [:upper:]
SET TO UPPERCASE
$ echo "10.00 only numbers 1.33" | tr -d [:alpha:] | tr -s "\b" ","
10.00,1.33
```

Now write a shell script to substitute all tabs with commas called `tabtocsv.sh` in `Week1/Code`:

```
#!/bin/bash
# Author: Your name you.login@imperial.ac.uk
# Script: tabtocsv.sh
# Desc: substitute the tabs in the files with commas
#       saves the output into a .csv file
# Arguments: 1-> tab delimited file
# Date: Oct 2015

echo "Creating a comma delimited version of $1 ..."

cat $1 | tr -s "\t" "," >> $1.csv

echo "Done!"

exit
```

Now test it (note where the output file gets saved)

```
echo -e "test \t\t test" >> ../SandBox/test.txt
bash tabtocsv.sh ../SandBox/test.txt
```

2.5 Variables in shell scripting

There are three ways to assign values to variables (note lack of spaces!):

1. Explicit declaration: `MYVAR=myvalue`
2. Reading from the user: `read MYVAR`
3. Command substitution: `MYVAR=$( (ls | wc -l) )`

Here are some examples of assignments (try it out save as `Week1/Code/variables.sh`):

```
#!/bin/bash
# Shows the use of variables
MyVar='some string'
```

```
echo 'the current value of the variable is' $MyVar
echo 'Please enter a new string'
read MyVar
echo 'the current value of the variable is' $MyVar
## Reading multiple values
echo 'Enter two numbers separated by space(s)'
read a b
echo 'you entered' $a 'and' $b '. Their sum is:'
mysum=`expr $a + $b`
echo $mysum
```

And also (save as `Week1/Code/MyExampleScript.sh`):

```
#!/bin/bash

msg1="Hello"
msg2=$USER
echo "$msg1 $msg2"

echo "Hello $USER"
echo
```

2.6 Some more Examples

Here are a few more illustrative examples (test each one out, save in `Week1/Code/` with the given name):

`CountLines.sh`:

```
#!/bin/bash
NumLines=`wc -l < $1`
echo "The file $1 has $NumLines lines"
echo
```

`ConcatenateTwoFiles.sh`:

```
#!/bin/bash
cat $1 > $3
cat $2 >> $3
echo "Merged File is"
cat $3
```

2.7 Practical

1. **Some instructions**:

   Along with the completeness of the practicals/exercises themselves, you will be marked on the basis of how complete and well-organized your directory structure and content is.

   Review (especially if you got lost along the way) and make sure all your shell scripts are functional: `boilerplate.sh`, `ConcatenateTwoFiles.sh`, `CountLines.sh`, `MyExampleScript.sh`, `tabtocsv.sh`, `variables.sh`

Don't worry about how some of these scripts will run on my computer without explicit inputs (e.g., `ConcatenateTwoFiles.sh` needs two input files) — I will run them with my own test files.

Make sure you have your weekly directory organized with `Data`, `Sandbox`, `Code` with the necessary files, under `CMEECourseWork/Week1`. *All scripts should run on any other Unix/Linux machine* — for example, always call data from the `Data` directory using relative paths.

Make sure there is a `readme` file in every week's directory. This file should give an overview of the weekly directory contents, listing all the scripts and what they do. This is different from the `readme` for your overall git repository, of which `Week 1` is a part. You will write a similar `readme` for each subsequent weekly submission.

Don't put any scripts that are part of the submission in your `home/bin` directory! You can put a copy there, but a working version should be in your repository.

2. Finally, a small exercise: write a `csvtospace.sh` shell script that takes a comma separated values and converts it to a space separated values file. However, it must not change the input file — it should save it as a differently named file.

   Save the script in `CMEECourseWork/Week1/Code`, and run it on the `csv` data files that are in `Temperatures` in the master repository's `Data` directory.

   Don't modify anything (or refer to anything) in your local copy of the master repository. All changes you make in the master repository will be lost. Copy whatever you need from the master repository to your own repository.

Commit and push everything by next Wednesday 5 PM.

This includes `UnixPrac1.txt`! Check the updated instructions from Chapter 1 on this practical.

2.8 Readings & Resources

- Plenty of shell scripting resources and tutorials out there; in particular, look up `http://www.tutorialspoint.com/unix/unix-using-variables.htm`

Chapter 3

Version control with Git

3.1 What is Version Control?

Version control, also known as revision control or source control, is the management and tracking of changes to documents, computer programs, large web sites, and other collections of information in an automated way.

Any project (collections of files in directories) under version control has changes and additions/deletions to its files and directories recorded and archived over time so that you can recall specific versions later. With version control of biological computing projects, you can:

- record of all changes made to a set of files and directories, including text (usually ASCII) data files, so that you can access any previous version of the files
- branch (and merge) new projects
- "roll back" data, code, documents (but less efficient with formats like *.docx though!)

Version control is embedded in various word processors and spreadsheets, e.g., Google Docs and Sheets

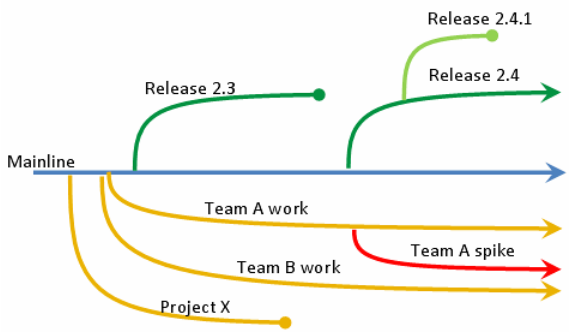


Figure 3.1: A general idea of how version control works.

3.2 Why Version Control?

maktoons.blogspot.com/2009/06/if-dont-use-version-control-system.html

3.3 git

We will use git, developed by Linus Torvalds, the "Linu" in Linux. In git each user stores a complete local copy of the project, including the history and all versions. So you do not rely as much on a centralized (remote) server. We will use bitbucket.org – it gives you unlimited free private repositories if you register with an academic email! First, install and configure `git`:

```
$ sudo apt-get install git
$ git config --global user.name "Your Name"
$ git config --global user.email "your.login@imperial.ac.uk"
$ git config --list
```

3.4 Your first repository

Time to bring your `CMEECourseWork` under version control:

```
$ cd CMEECourseWork
$ git init
$ echo "My CMEE 2016-17 Coursework Repository" > README.txt
$ git config --list
$ ls -all
$ git add README.txt #Staging
$ git status
$ git commit -m "Added README file." #you can use -am too
$ git status #what does it say now?
$ git add -A
$ git status
```

Nothing has been sent to a remote server yet (see section 3.5.1)! So let's go to bitbucket.org and setup:

- Login to your account
- Set up your `ssh` based access to bitbucket – `https://confluence.atlassian.com/bitbucket/set-up-ssh-for-git-728138079.html`
- Then create repository there with name `CMEECourseWork`
- Then grab the repository url and use `git remote add origin https...` — check bitbucket help online.

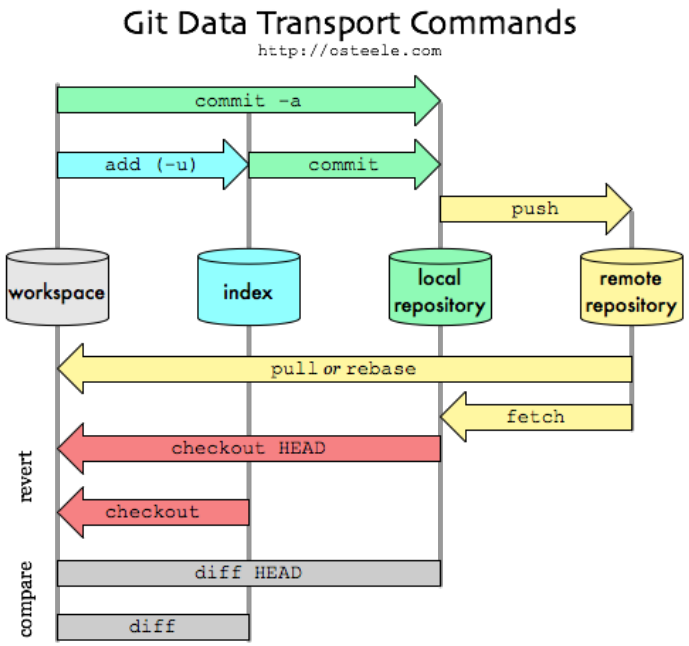
3.5 `git` commands

Here are some basic git commands:

`git init`	Initialize a new repository
`git clone`	Download a repository from a remote server
`git status`	Show the current status
`git diff`	Show differences between commits
`git blame`	Blame somebody for the changes!
`git log`	Show commit history
`git commit`	Commit changes to current branch
`git branch`	Show branches
`git branch name`	Create new branch
`git checkout name`	Switch to a different commit/branch
`git pull`	Upload from remote repository
`git push`	Send changes to remote repository

3.5.1 `git` command structure

Here is a graphical outline of the git command structure. Note that only when you `push` or `fetch` do you need an internet connection, as before that you are only archiving in a local (hidden) repository.



Keep in mind, the main mantra is, "commit often, comment always"!

3.6 Ignoring Files

You will have some files you don't want to track (log files, temporary files, executables, etc). You can ignore entire classes of files with `.gitignore` (be in your `CMEECourseWork`!):

```
$ echo -e "*~ \n*.tmp" > .gitignore

$ cat .gitignore
*~
*.tmp

$ git add .gitignore

$ touch temporary.tmp

$ git add *
The following paths are ignored by one of your .gitignore
files:
temporary.tmp
Use -f if you really want to add them.
```

```
fatal: no files added
```

3.6.1 Dealing with large files

As such, git was designed for version control of workflows and software projects, /textitnot large files (say, >100mb). This includes binary files. A binary file is computer-readable but not human-readable, that is, it cannot be read by opening them in a text viewer. Examples of binary files include compiled executables, zip files, images, and videos. In contrast, text files are stored in a form (usually ASCII) that is human-readable by opening in a suitable text reader, gedit or geany on Ubuntu.

So the fact is that that binary files cannot be version-controlled — this is something you have to live with. This is because each version of the file is saved in a hidden directory in the repository (`.git`). When you're having a large number of version (say 100) it means that there are the same number of binary files in the hidden directory (for example 100 $\times$ >100mb files!). But there are some reasonable workarounds; see `http://blogs.atlassian.com/2014/05/handle-big-repositories-git/`.

In this course at least, you should not try to keep large files including binary files under version control. You will run into this problem in the GIS week in particular. None of the computing weeks assessments will require you to use such large files.

My suggestion is to include files larger than some size in your `.gitignore`. For example, you can use the following bash command:

```
find . -size +100M | cat >> .gitignore
```

The 100M means 100 mb – you can reset it to whatever you want.

3.7 Removing files

To remove a file (i.e. stop version controlling it) use `git rm`:

```
$ echo "Text in a file to remove" > FileToRem.txt

$ git add FileToRem.txt

$ git commit -am "added a new file that we'll remove later"
master 5df9e96 added a new file that we'll remove later
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 FileToRem.txt

$ git rm FileToRem.txt
rm 'FileToRem.txt'

$ git commit -am "removed the file"
master b9f0b1a removed the file
 1 files changed, 0 insertions(+), 1 deletions(-)
 delete mode 100644 FileToRem.txt
```

I typically just do all my stuff and then just use `git add -A`

3.8 Accessing history of the repository

To see particular changes introduced, read the repo's log :

```
$ git log
commit 08b5c1c78c8181d4606d37594681fdcfca3149ec
Author: Your Name <your.login@imperial.ac.uk>
Date:   Wed Oct 8 16:41:51 2014 -0500

    removed the file

commit 13f701775bce71998abe4dd1c48a4df8ed76c08b
Author: Your Name <your.login@imperial.ac.uk>
Date:   Wed Oct 5 16:41:16 2015 -0500

    added a new file that we'll remove later

commit a228dd3d5b1921ef18c5efd926ef11ca47306ed5
Author: Your Name <your.login@imperial.ac.uk>
Date:   Wed Oct 5 10:03:40 2015 -0500

    Added README file
```

For a more detailed version, add `-p` at the end.

3.9 Reverting to a previous version

If things go horribly wrong with new changes, you can revert to the previous, "pristine" state:

```
$ git reset --hard
$ git commit -am "returned to previous state" #Note I used -am here
```

If instead you want to move back in time (temporarily), first find the "hash" for the commit you want to revert to, and then check-out:

```
$ git status
# On branch master
nothing to commit (working directory clean)

$ git log
commit c797824c9acbc59767a3931473aa3c53b6834aae
Author: Your Name <your.login@imperial.ac.uk>
Date:   Wed Aug 22 16:59:02 2014 -0500
.
.
.

$ git checkout c79782
```

Now you can play around. However, if you commit changes, you create a "branch" (git plays safe!). To go back to the future, type `git checkout master`

3.10 Branching

Imagine you want to try something out, but you're not sure it will work well. For example, say you want to rewrite the Introduction of your paper, using a different angle, or you want to see whether switching to a library for a piece of code improves speed. What you then need is branching, which creates a project copy in which you can experiment:

```
$ git branch anexperiment

$ git branch
  anexperiment
* master

$ git checkout anexperiment
Switched to branch 'anexperiment'

$ git branch
* anexperiment
  master

$ echo "Do I like this better?" >> README.txt

$ git commit -am "Testing experimental branch"
[anexperiment 9f17dc1] Testing experimental branch
 1 files changed, 2 insertions(+), 0 deletions(-)
```

If you decide to merge the new branch after modifying it:

```
$ git checkout master

$ git merge anexperiment
Updating 08b5c1c..9f17dc1
Fast-forward
 README.txt |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

$ cat README.txt
My CMEE 2015-16 Coursework Repository
Do I like this better?
```

If there are no conflicts (i.e., some files that you changed also changed in the master in the meantime), you are done, and you can delete the branch:

```
$ git branch -d anexperiment
Deleted branch anexperiment (was 9f17dc1).
```

If instead you are not satisfied with the result, and you want to abandon the branch:

```
$ git branch -D anexperiment
```

When you want to test something out, always branch! Reverting changes, especially in code, is typically painful. Merging can be tricky, especially if multiple people have simultaneously worked on a particular document. In the worst-case scenario, you may want to delete the local copy and re-clone the remote repository.

3.11 Running git commands on a different directory

Since `git` version 1.8.5, you can run git directly on a different directory than the current one using absolute or relative paths. For example, using a relative path, you can do:

```
git -C ../SomeDir/ status
```

3.12 Running git commands on multiple repositories at once

For git pulling in multiple subdirectories (each a separate repository):

```
    $ find . -mindepth 1 -maxdepth 1 -type d -print -exec git -C {} pull \;
```

Breaking down these commands one by one,

`find .` searches the current directory
`-type d` to find directories, not files
`-mindepth 1` for setting min search depth to one sub-directory
`-maxdepth 1` for setting max search depth to one sub-directory
`-exec git -C {} pull \` runs a custom git command for every git repo found

3.12.1 Practicals

1. The only practical submission for `git` is the `.gitgnore` and overall git repository `readme` file — make sure these in your coursework repository.

And of course, if you haven't gotten git with bitbucket going, you won't be able to submit any of you practicals anyway!

3.13 Practical wrap-up

- Invite me (s.pawar@imperial.ac.uk) to your `CMEECourseWork` repository
- `CMEE2015MasteRepo` will contain data and code files for upcoming practicals
- You will clone `CMEE2015MasteRepo` using git clone git@bitbucket.org:mhasoba/cmee2015masterepo.git
- You will thereafter `git pull CMEE2015MasteRepo`
- You will `git pull` inside `CMEE2015MasteRepo` thereafter (always use `git status` first)
- `cp` files from `CMEE2015MasteRepo` to your `CMEECourseWork` as and when needed — don't work in the amster repo, as you will lose your work when I next update it!

3.14 Readings & Resources

- Excellent book on Git: `http://git-scm.com/book`
- Also, `https://www.atlassian.com/git/`, including Bitbucket 101

Chapter 4

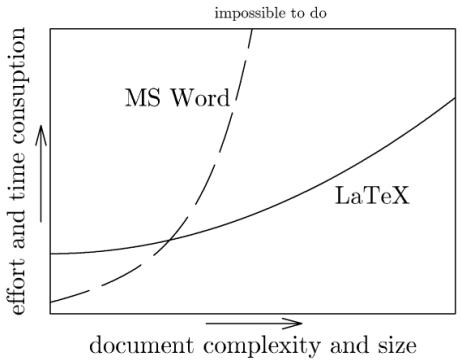
Using LaTeX for scientific documents

4.1 What's LaTeX?

In your research, you will produce papers, reports and – very importantly – your thesis. These documents can be written using a WYSIWYG (What You See Is What You Get) editor (e.g., Word). However, an alternative especially suited for scientific publications is LaTeX. In LaTeX, the document is simply a text file (`.tex`). Text formatting is using markups (like HTML). The file is then "compiled" (like source code of a programming language) into a file – typically `.pdf`.

4.2 Why LaTeX?

- The input is a small, portable text file
- LaTeX compilers are freely available for all OS'
- Exactly the same result on any computer (not true for Word)
- LaTeX produces beautiful, professional looking docs (e.g., like this one!)
- Mathematical formulas (esp complex ones) are easy to write
- LaTeX is very stable – current version basically same since 1994! (9 major versions of MS Word since 1994 – with compatibility issues)
- LaTeX is free!
- Many journals provide LaTeX templates, making formatting quicker
- Bibliographies are a breeze and work with Mendeley and Zotero
- Plenty of online support available – your question has probably already been answered
- You can integrate LaTeX into a workflow to auto-generate lengthy and complex documents (like your thesis).



4.2.1 Limitations of LaTeX

- It has a steeper learning curve.
- Can be difficult to manage revisions with multiple authors – especially if they don't use LaTeX! (I have a dark secret)
- Typesetting tables can be a bit complex.
- Images and floats don't jump like Word, but if you don't use the right package, they can be difficult to place where you want!

4.3 Installing LaTeX

```
sudo apt-get install texlive-full texlive-fonts-recommended
latex-beamer texpower texlive-pictures texlive-latex-extra
texpower-examples imagemagick
```

We will use a text editor in this lecture, but you can use one of a number of WYSIWYG frontends (e.g., `Lyx`, `TeXmacs`), as well as GUI's (`texmaker`, `Gummi`, `TeXShop`, etc)

4.4 A first LaTeX example

⋆ Open geany and type the following in a file `Week1/Code/FirstExample.tex`:

```
\documentclass[12pt]{article}
\title{A Simple Document}
\author{Your Name}
\date{}
\begin{document}
  \maketitle

  \begin{abstract}
    This paper must be cool!
  \end{abstract}

  \section{Introduction}
    Blah Blah!

  \section{Materials \& Methods}
  One of the most famous equations is:
  \begin{equation}
    E = mc^2
  \end{equation}
  This equation was first proposed by Einstein in 1905
  \cite{einstein1905does}.

  \bibliographystyle{plain}
  \bibliography{FirstBiblio}
\end{document}
```

Now, let's get a citation for Einstein's paper:

⋆ In Google Scholar, go to "settings" (upper right corner) and choose BibTeX as bibliography manager.

- ⋆ Now type “does the energy of a body einstein 1905”
- ⋆ The paper should be the one on the top.
- ⋆ Click “ Import into BibTeX” should show the following text, that you will save in the file `FirstBiblio.bib` (in the same directory as `FirstExample.tex`)

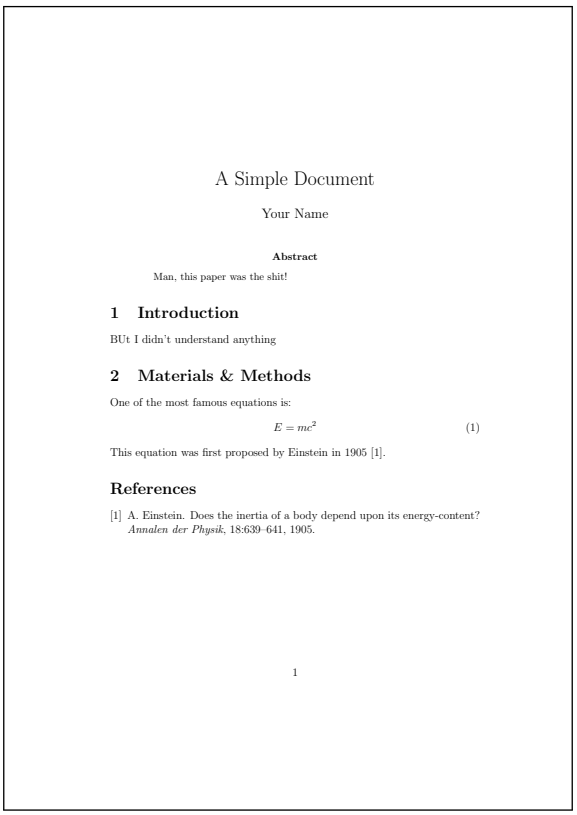
```
@article{einstein1905does,
  title={Does the inertia of a body depend upon its energy-content?},
  author={Einstein, A.},
  journal={Annalen der Physik},
  volume={18},
  pages={639--641},
  year={1905}
}
```

Now we can create a `.pdf` of the article.

- ⋆ In the terminal type (are you the right directory?!):

```
$ pdflatex FirstExample.tex
$ pdflatex FirstExample.tex
$ bibtex FirstExample
$ pdflatex FirstExample.tex
$ pdflatex FirstExample.tex
```

This should produce the file `FirstExample.pdf`:



A Simple Document

Your Name

Abstract

Man, this paper was the shit!

1 Introduction

BUt I didn't understand anything

2 Materials & Methods

One of the most famous equations is:

$$E = mc^2 \quad (1)$$

This equation was first proposed by Einstein in 1905 [1].

References

[1] A. Einstein. Does the inertia of a body depend upon its energy-content? *Annalen der Physik*, 18:639–641, 1905.

1

4.4.1 A bash script to compile LATEX

You can of course write a useful little bash script to compile latex with bibtex!

Type the following script and call it `CompileLaTeX.sh` (you know where to put it!):

```
#!/bin/bash
pdflatex $1.tex
pdflatex $1.tex
bibtex $1
pdflatex $1.tex
pdflatex $1.tex
evince $1.pdf &

## Cleanup
rm *~
rm *.aux
rm *.dvi
rm *.log
rm *.nav
rm *.out
rm *.snm
rm *.toc
```

How do you run this script? The same as your previous bash scripts, so

```
$ bash CompileLaTeX.sh FirstExample
```

Why have I not written the `*.tex` extension of `FirstExample` in the command above?

4.5 A brief LaTeX tour

- Spaces, new lines and special characters:
  - Several spaces in your text editor are treated as one space in the typeset document
  - Several empty lines are treated as one empty line
  - One empty line defines a new paragraph
  - Some characters are "special":# $ % ^ & _ { } ~ \.
  - To type these special characters, you have to add a "backslash" in front, e.g., \$ produces $.

- Document structure:
  - Each LaTeX command starts with \(e.g., to get LaTeX, you need `\LaTeX`
  - The first command is always `\documentclass` defining the type of document (e.g., `article, book, report, letter`).
  - You can set several options. For example, to set size of text to 10 points and the letter paper size: `\documentclass[10pt,letterpaper]{article}`.

- After having declared the type of document, you can specify packages you want to use. The most useful are:
  - `\usepackage{color}`: use colors for text in your document.
  - `\usepackage{amsmath,amssymb}`: American Mathematical Society formats and commands for typesetting mathematics.
  - `\usepackage{fancyhdr}`: fancy headers and footers.
  - `\usepackage{graphicx}`: include figures in pdf, ps, eps, gif and jpeg.
  - `\usepackage{listings}`: typeset source code for various programming languages.

  - `\usepackage{rotating}`: rotate tables and figures.
  - `\usepackage{lineno}`: line numbers.

- Once you select the packages, you can start your document with `\begin{document}`, and end it with `\end{document}`.

4.6 LaTeX templates

There a lots of useful LaTeXtemplates out there. As an example of structure of a document, take the article template provided by the journal PNAS:

```
\documentclass{pnastwo}
\usepackage{amssymb,amsfonts,amsmath}
%% For PNAS Only:
\contributor{Submitted to Proceedings
of the National Academy of Sciences of the United States of America}
\url{www.pnas.org/cgi/doi/10.1073/pnas.0709640104}
\copyrightyear{2014}
\issuedate{Issue Date}
\volume{Volume}
\issuenumber{Issue Number}

\begin{document}
\title{My Title}
\author{Some Name \affil{1}{Imperial College London, UK} \and
Some O. Name\affil{2}{University of Exeter, Penryn, Corwall, UK}}
\maketitle
\begin{article}
\begin{abstract}
Mind blowing abstract.
\end{abstract}
\keywords{term1 | term2 | term3}

%% Main text of the paper
\dropcap{I}n this work, we show how \LaTeX can be used to typeset a PNAS paper. Lorem ↩
    ipsum dolor sit amet, consectetur adipiscing elit. Phasellus sodales consectetur ↩
    lobortis. Proin tincidunt eros dapibus ipsum faucibus sed rhoncus augue mollis. In↩
     lectus velit, interdum at adipiscing quis, imperdiet sed justo. Praesent commodo,↩
     mi iaculis tincidunt mollis, sapien lectus aliquam neque, ac faucibus arcu est eu↩
     sem. Ut non lacus lacus, eu suscipit odio. Aliquam erat volutpat. Vivamus dapibus↩
     pretium nunc, et placerat turpis bibendum mollis. Fusce eu mi ut nulla accumsan ↩
    viverra. In nulla tellus, ultrices ut venenatis nec, laoreet eget diam. ↩
    Pellentesque aliquam facilisis ultricies. Vestibulum sollicitudin leo non neque ↩
    vehicula a volutpat eros faucibus. Vestibulum nec lorem dui.

\begin{materials}
These are the materials and methods.
\end{materials}

\begin{acknowledgments}
-- text of acknowledgments here, including grant info --
\end{acknowledgments}

\end{article}
\end{document}
```

I have added some templates in the `CMEEMasteRepo` that you should have a look and play around with

4.7 Typesetting math

There are two ways to display math

1. First, one can produce inline mathematics (i.e., within the text).
2. Second, one can produce stand-alone, numbered equations and formulae.

For inline math, the "dollar" sign flanks the math to be typeset. For example, the code:

```
$\int_0^1 p^x (1-p)^y dp$
```

Becomes $\int_0^1 p^x(1-p)^y dp$

For numbered equations (almost always a great idea), LaTeX provides the `equation` environment:

```
\begin{equation}
\int_0^1 \left(\ln \left( \frac{1}{x} \right)
\right)^y dx = y!
\end{equation}
```

Becomes:

$$\int_0^1 \left(\ln\left(\frac{1}{x}\right)\right)^y dx = y! \tag{4.1}$$

4.8 A few more tips

The following tips might prove handy:

- LaTeX has a full set of symbols and operators (plenty of lists online)
- Long documents can be split into separate `.tex` documents and combined using `input`
- Long documents can be split into separate `.tex` documents and
- Figures can be included using the `graphicx` package
- You can use Mendeley to export and maintain `.bib` files
- You can redefine environments and commands in the preamble

4.8.1 Practicals

Test `CompileLaTeX.sh` with `FirstExample.tex` and bring it under verson control under`CMEECourseWork/Week1` in your repository. Make sure that `CompileLaTeX.sh` will work if I ran it from my computer using `FirstExample.tex` as an input.

4.9 Practicals wrap-up

Make sure you have your `Week 1` directory organized with `Data`, `Sandbox` and `Code` with the necessary files and this week's (functional!) scripts in there. Every script should run without errors on my computer. This includes the five solutions (single-line commands you came up with) in `UnixPrac1.txt`.

Commit and push everything by next Wednesday 5 PM.

4.10 Readings & Resources

- The Visual LaTeX FAQ: sometimes it is difficult to describe what you want to do! `http://get-software.net/info/visualFAQ/visualFAQ.pdf`
- Myriad online resources for LaTeX, including: `www.http://en.wikibooks.org/wiki/LaTeX/Introduction`, `www.ctan.org/tex-archive/info/lshort/english/` `http://ftp.uni-erlangen.de/mirrors/CTAN/info/lshort/english/lshort.pdf`
- Beautiful presentations in LaTeX: `http://tug.org/pracjourn/2005-2/miller/miller.pdf`
- Bibliographies in LaTeX: `http://schneider.ncifcrf.gov/latex.html`

Chapter 5

Basic Biological Computing in Python

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

Firstly, chapter 1's UNIX question?

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

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

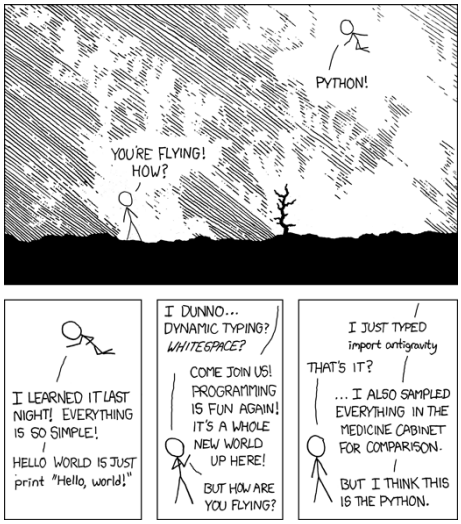
5.1 Outline of the `python` module

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

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

5.2 Why `python`?

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



www.xkcd.com

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

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

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

http://julialang.org/

Figure 5.2: `python` is pretty fast!

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

5.2.1 The Zen of python

Open a terminal and type

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

5.3 Installing `python`

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

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

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

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

Tip

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

5.4 Getting started with `python`

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

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

>>> 2 * 2 # Multiplication
4

>>> 2 / 2 # Integer division
1

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

>>> 2 / 2.
1.0

>>> 2 > 3
False

>>> 2 >= 2
True
```

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

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

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

5.4.1 `ipython`

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

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

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

You should now see (after some text):

```
In [ ]:
```

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

5.4.2 Magic commands

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

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

Try any of these now!

5.4.3 Determining an object's type

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

```
In [1]: a = 1

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

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

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

5.4.4 Configuring `ipython`

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

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

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

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

5.5 Python variables

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

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

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

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

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

In [ ]: a = 2. #Float

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

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

In [ ]: a = "Two" #String

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

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

In [10]: a = True #Boolean

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

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

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

5.5.1 `python` operators

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

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

5.5.2 Assigning and manipulating variables

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

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

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

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

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

In []: x = 5

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

 In []: y = 8

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

 In []: x = 'My string'

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

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

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

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

 In []: z = '88'

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

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

Tip

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

5.6 `python` data types and data structures

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

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

5.6.1 Lists

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

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

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

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

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

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

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

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

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

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

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

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

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

5.6.2 Tuples

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

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

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

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

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

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

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

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

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

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

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

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

5.6.3 Sets

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

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

 In []: b = set(a)

 In []: b
```

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

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

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

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

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

The key set operations in `python` are:

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

5.6.4 Dictionaries

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

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

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

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

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

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

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

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

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

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

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

So, in summary,

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

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

5.6.5 Copying mutable objects

Copying mutable objects can be tricky. Try this:

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

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

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

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

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

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

5.6.6 `python` with strings

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

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

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

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

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

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

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

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

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

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

5.7 Writing `python` code

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

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

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

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

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

5.8 python Input/Output

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

```
First Line
Second Line

Third Line

Fourth Line
```

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

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

# close the file
f.close()

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

f.close()

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

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

f.close()

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

import pickle

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

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

print another_dictionary
```

Note the following:

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

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

```
import csv

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

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

f.close()

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

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

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

Tip

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

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

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

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

Then,

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

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

5.8.1 Writing `python` functions (or modules)

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

```
#!/usr/bin/python

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

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

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

# constants can go here


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

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

Running your `python` code

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

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

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

Alternatively, you can use ipython:

```
$ ipython boilerplate.py
```

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

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

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

5.8.2 Components of the `python` function

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

The shebang

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

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

The Docstring

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

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

Internal Variables

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

Function definitions and "modules"

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

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

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

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

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

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

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

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

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

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

Then type

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

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

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

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

Now run it:

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

Now, try:

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

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

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

What on earth is `sys.argv`?

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

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

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

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

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

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

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

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

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

OK, finally, what about this bit:

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

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

5.8.3 Variable scope

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

```
## Try this first

_a_global = 10

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

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



## Now try this

_a_global = 10

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

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

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

5.9 Control statements

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

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

```
$
```

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

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

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

import sys

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

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

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

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

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

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

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

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

Now run the code:

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

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

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

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

5.9.1 Control flow exercises

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

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

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

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

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

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

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

def foo2(x, y):
```

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

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

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

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

foo5(10)
```

5.10 Loops

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

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

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

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

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

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

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

5.10.1 List comprehensions

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

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

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

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

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

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

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

```
print oaks_loops

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

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

5.10.2 Practicals

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

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

5.11 Functions, Modules, and code compartmentalization

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

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

5.11.1 Importing Modules

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

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

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

5.12 Python packages

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

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

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

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

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

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

5.12.1 Practicals

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

Align DNA sequences

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

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

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

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

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

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

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

5.13 Errors in your `python` code

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

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

Then, think about this:

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

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

5.13.1 Unit testing

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

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

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

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

Unit testing with `doctest`

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

```
#!/usr/bin/python

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

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

import sys
import doctest # Import the doctest module

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

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

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

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

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

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

## I SUPPRESSED THIS BLOCK: WHY?

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

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

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

Now type `run test_control_flow.py -v`:

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

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

Tip

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

5.13.2 Debugging

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

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

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

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

createabug(25)
```

Now run it:

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

ZeroDivisionError: float division by zero
```

OK, so let's `%pdb` it

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

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

ipdb>
```

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

Tip

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

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

So let's continue our debugging:

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

ipdb> q

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

5.13.3 Paranoid programming: debugging with breakpoints

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

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

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

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

5.13.4 Practicals

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

Missing oaks problem

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

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

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

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

5.14 Practicals wrap-up

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

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

5.15 Readings and Resources

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

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

Chapter 6

Advanced Biological Computing in Python

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

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

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

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

6.1 Regular expressions in `python`

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

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

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

www.xkcd.com/208/

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

6.1.1 Metacharacters vs. regular characters

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

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

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

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

6.1.2 regex elements

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

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

6.1.3 regex in python

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

Tip

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

strings for all but the simplest expressions.

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

```
import re

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

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

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

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

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

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

# this should print "None"
print match

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

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

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

```
MyStr = 'an example'

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

match.group(1)
'Samraat Pawar'

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

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

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

6.1.4 Some RegExercises

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

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

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

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

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

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

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

6.1.5 Important `re` functions

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

6.1.6 Practicals

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

Blackbirds problem

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

6.2 Numerical computing in `python`

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

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

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

Tip

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

So let's try `scipy`:

```
In []: import scipy

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

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

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

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

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

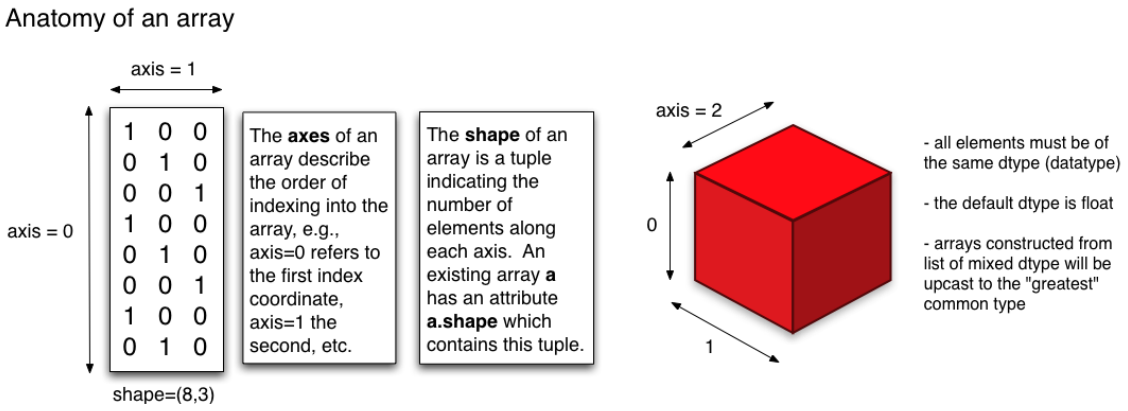


Figure 6.2: A graphical depiction of numpy/scipy arrays, which can have multiple dimensions (even greater than 3). From `http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/python/arrays.html`

You can also specify the data type of the array:

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

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

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

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

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

In []: x
```

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

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

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

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

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

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

Tip

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

You can also convert to and from Python lists:

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

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

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

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

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

6.2.1 Indexing and accessing arrays

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

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

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

And accessing particular elements:

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

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

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

Tip

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

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

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

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

6.2.2 Manipulating arrays

Manipulating `scipy` arrays is pretty straightforward.

Replacing, adding or deleting elements

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

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

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

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

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

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

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

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

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

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

And concatenation:

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

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

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

Flattening or reshaping arrays

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

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

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

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

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

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

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

6.2.3 Pre-allocating arrays

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

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

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

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

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

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

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

6.2.4 scipy matrices

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

Tip

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

Matrix-vector operations

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.5 Two useful `scipy` sub-packages

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

`scipy.stats`

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

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

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

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

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

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

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

`scipy.integrate` – the Lotka-Volterra model

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

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

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

The Lotka-Volterra model is:

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

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

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

6.3 The need for speed: Profiling in Python

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

6.3.1 Profiling

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

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

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

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

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

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

some_function(1000)
```

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

```
         54 function calls in 3.652 seconds

   Ordered by: internal time

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

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

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

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

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

some_function(1000)
```

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

```
         52 function calls in 2.153 seconds

   Ordered by: internal time

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

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

6.3.2 Quick profiling with `timeit`

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

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

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

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

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

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

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

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

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

my_list = range(1000)

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


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

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


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

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

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


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

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


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

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


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

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

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

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

6.3.3 Practicals

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

Lotka-Volterra model problem

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4 Networks in `python` (and R!)

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

- Metabolic networks
- Gene regulatory networks

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

Can you think of a few more examples from biology?

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

For this you need to first install the package:

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

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

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

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

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

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

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

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

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

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

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

##Plot using networkx:

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

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

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

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

6.4.1 Practical (Extra Credit)

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

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

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

6.5 Databases and `python`

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

6.5.1 Relational databases

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

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

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

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

Let's look at a simple example.

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

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

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

Site table:

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

Sample table:

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

Species table:

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

Individual table:

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

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

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

6.5.2 SQLite

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Now let's import the dataset:

```
sqlite> .mode csv

sqlite> .import TraitInfo.csv TraitInfo
```

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

```
sqlite> .tables

TraitInfo
```

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

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

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

Let's turn on some nicer formatting:

```
sqlite> .mode column

sqlite> .header ON

sqlite> SELECT * FROM TraitInfo LIMIT 5;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Let's try some more elaborate queries:

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

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

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

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

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

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

```
744
748
```

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

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

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

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

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

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


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

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

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

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

```
Resource Escape Response Probability
```

We can also order by any column:

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

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

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

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

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

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

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


sqlite> .import Consumer.csv Consumer

# Now we have two tables in our database:

sqlite> .tables
Consumer   TraitInfo


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

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

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

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

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

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

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


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

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

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

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

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

6.5.3 SQLite with python

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

```
# import the sqlite3 library
import sqlite3

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

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

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

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

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

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

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

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

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

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

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

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

# and commit
conn.commit()

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

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

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

```
import sqlite3

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

c = conn.cursor()

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

conn.commit()

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

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

conn.commit()

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

conn.close()
```

6.6 Using `python` to build workflows

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

6.6.1 Using `subprocess`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.6.2 Running `R`

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

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

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

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

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

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

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

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

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

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

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

6.6.3 Practicals

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

Using `os` problem 1

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

Using `os` problem 2

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

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

6.7 Practicals wrap-up

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

6.8 Readings and Resources

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

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

Chapter 7

Introduction to R

7.1 Outline of the the R module

You will use R a lot during the rest of your courses, your thesis dissertation, and very likely, your career. The `R` module aims to lay down the foundations for you to become comfortable with it.

Specifically it is designed to deliver the following objectives:

- Giving you an introduction to R syntax and programming conventions, assuming you have never set your eyes on R or any other programming language before
- Teaching you principles of data processing and exploration (including visualization) using R
- Teaching you principles of clean and efficient programming using R
- Teaching you how to generate publication quality graphics in R — publication quality is thesis quality!
- Teaching you how to develop reproducible data analyses "work flows" so you (or anybody else) can run and re-run your analyses, graphics outputs and all, in R

7.2 What is `R`?

`R` is a freely available statistical software with strong programming capabilities widely used by professional scientists around the world. It was based on the commercial statistical software `S` by Robert Gentleman and Ross Ihaka. The first stable version appeared in 2000.

`R` was essentially designed for *programming* statistical analysis and data-mining. It has become the standard tool for data analysis and visualization in biology in matter of just 15 years or so. It is also increasingly being used for modelling in biology. This is because

1. `R` has many tried and tested packages to perform practically all statistical analysis
2. `R` has numerous packages for data-handling and processing
3. `R` has excellent graphing and visualization capabilities
4. `R` has good capabilities for efficient mathematical calculations, including matrix algebra

7.3 Why R?

There are many commercial statistical (minitab, SPSS, etc) software packages in the world that are mouse-driven, warm, and friendly, and have lots of statistical tests and plotting/graphing capabilities. Why not just use them? Here are some very good reasons:

- R is scriptable, so you can build a perfectly repeatable record of your analysis. This in itself has several advantages:
  - You can never replicate *exactly* the same analysis with all the same steps using a point-and-click approach/software. With R you can reproduce your full analysis for yourself (in the future!), your colleagues, your supervisor/employer, and any journal you might want to submit your work to.
  - You may need to rerun your analysis every time you get new data. Once you have it all in a R script, you can just rerun your analysis and go home!
  - You may need to tweak your analysis many times (new data, supervisor changes mind, you change mind, paper reviewers want you do something differently). Having the analysis recorded as script then allows you to do so by revising the relevant parts of your analysis with relatively little pain.
- R provides basically every statistical test you'll ever need and is constantly being improved – you can tailor your analyses rather than trying to use the more limited options each statistical software package can offer
- R can produce publication-quality graphics that can be re-produded with scripts – you won't get RSI mouse-clicking your way though graphing and re-graphing your data every time you change your analysis!
- R is freely available for all common computer operating systems – if you want a copy on your laptop, help yourself at the CRAN website.

Thus, being able to program in R means you can develop and automate your own data handling, statistical analysis, and graphing/plotting, a set of skills you are likely to need in many, if not most careers paths!

7.3.1 Would you ever need anything other than R?

Being able to program R means you can develop and automate your statistical analyses and the generation of figures into a reproducible work flow. *For most of you, using R as your only programming language will do the job!*

However, if your work also includes extensive numerical simulations, manipulation of very large matrices, bioinformatics, or includes relational database access and manipulations, you may be better-off *also* knowing another programming language that is more versatile and computationally efficient (like `python`, `perl` or `C`).

7.4 Installing R

If you are using a college computer, R will likely already be available. Otherwise you can install R on your own computer as follows:

On linux/ubuntu, run the following in terminal:

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

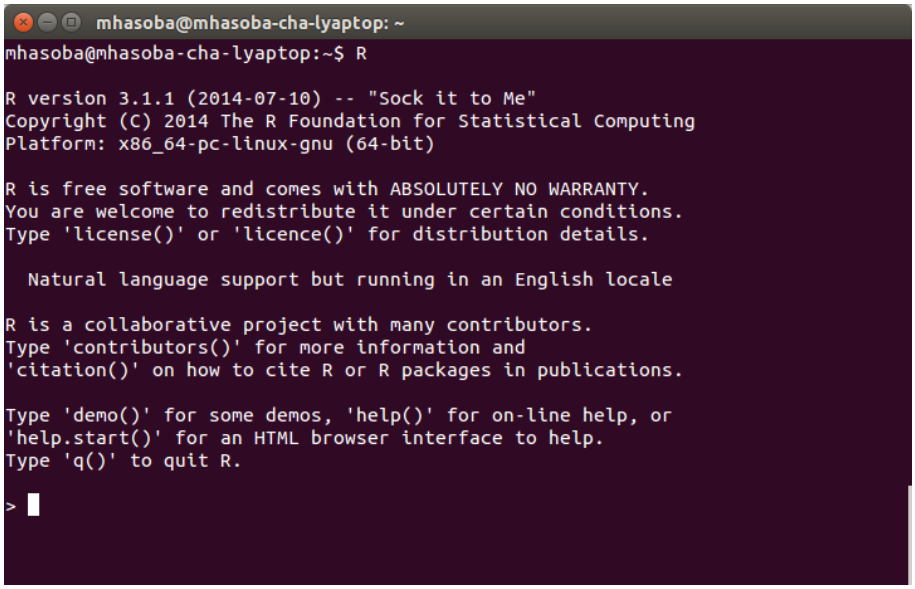
On Mac OS X, download and install from: `http://cran.r-project.org/bin/macosx/`

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

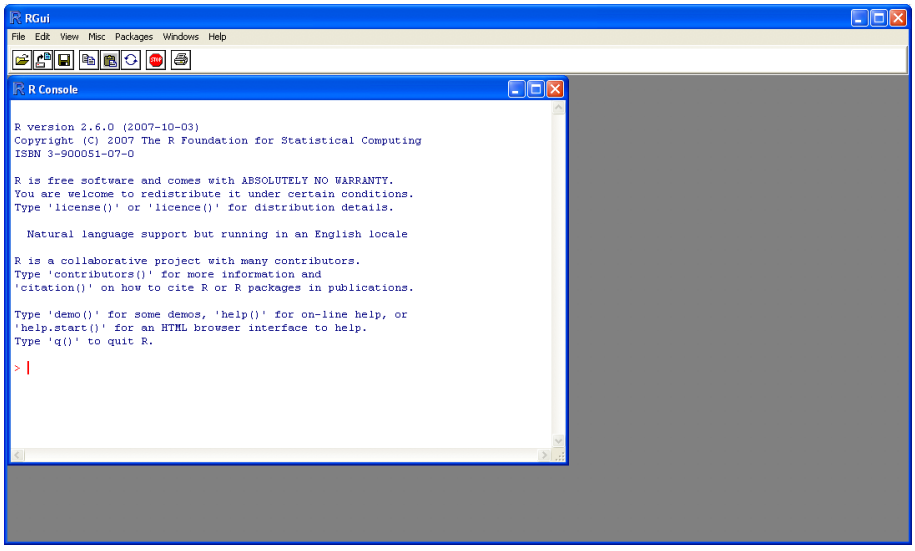
7.5 Getting started

Let's briefly look at the bare-bones `R` interface and command line interface (CLI), and then switch to a more Interactive Development Environment (IDE) such as RStudio or something else.

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):



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



7.5.1 Gooey IDEs!

You can also use an IDE (Interactive Development Environment) that offers useful features like syntax highlighting (google it!) and embedded data and plot viewing windows, such as RStudio, `geany`, `vim`, etc.

These IDEs come with graphic user interfaces (GUI's, or "gooeys") of differing levels of sophistication and shiny-ness.

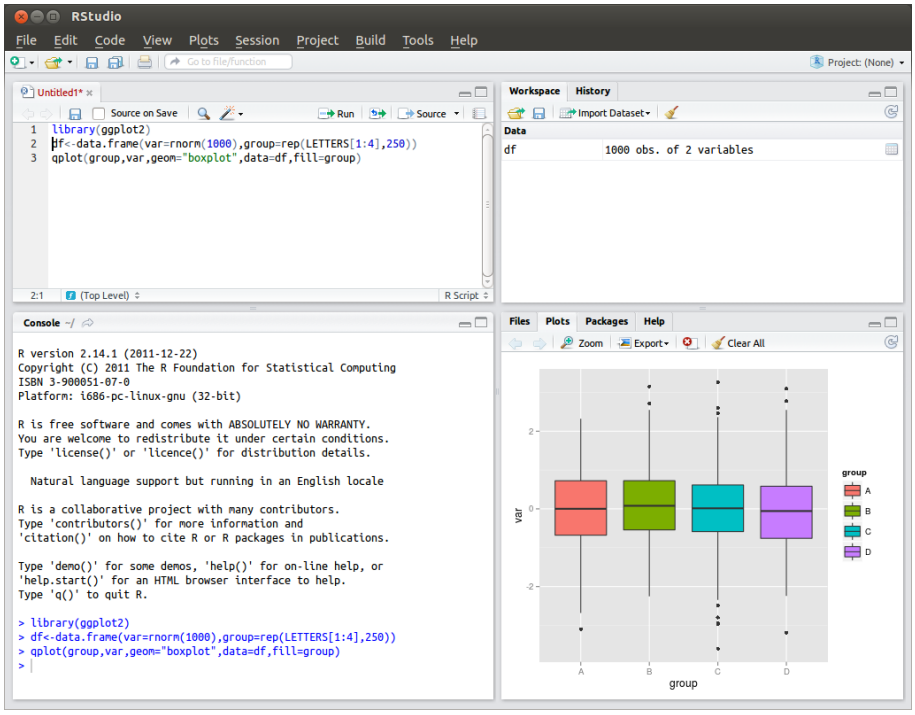
In fact, one of the reasons that R has become so popular is that there are some very nice, freely available GUI IDE's for it where you can use your mouse to do certain "stuff".

In particular, RStudio is very good.

A big advantage of something like RStudio is that you will get "syntax highlighting" wherein R language elements such as variables, commands, and brackets are differently colored. This is very handy and will make your R programming far more convenient and error-free. If you are using your own desktop/laptop, you can download it freely from `http://rstudio.org/` and install (versions are available for all computer platforms). If you are suing a college computer, it should already be available.

The R modules do not require you to use RStudio (everything will be command line based), but I would recommend that you to use it, or some other IDE like `geany` or `emacs`.

You may now launch RStudio (assuming it is available on your computer). Here is an example RStudio window:



You are still in the R environment, but with some useful frills and additional thrills. There are four main elements here (you can hide or resize any of them using the RStudio GUI options):

Console This is the same R terminal you saw above. In this week we will work mainly in this console

Source code This panel is the code editor with syntax highlighting and other handy features that we will explore soon. This is where you will write your scripts/programs, and save them. To save, you will use `Ctrl + S` and to execute it you will use `Ctrl + Shift + S`

Environment This panel lists all the variables you created (more on this later). It has another tab that shows you the history of the commands you typed.

Plots This panel shows you all the plots you drew. Other tabs in this panel allow you to access the list of packages you have loaded, and the help page for commands (just type `help(command_name)` in the Console) and packages.

In RStudio, you can feed your mouse habit, but I strongly suggest using the terminal/console, and resist, to the extent possible, the temptation to do everything using the shiny buttons you see (that is the dark side pulling, my young *padawan*).

7.6 Some R Basics

Gets get started with some R basics. You will be working by entering R commands interactively at the R user prompt (>). Up and down arrow keys scroll through your command history.

7.6.1 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"

7.6.2 R Warm-up

Like in any programming language, you will need to use "variables" to store information in a R session's workspace. Each Variable has a reserved location in your memory (RAM), and takes up "real estate" in it — that is when you create a variable you reserve some space in yur computer's memory.

Now, try assigning a few variables in the R and doing things to them:

```
> a <- 4  # store 4 as variable a
> 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) # build a vector with c (for "concatenate")
```

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*. You will be amazed by how difficult it is to read and understand what a certain R script does (or any other script, for that matter) without judicious comments — even scripts you yourself wrote not so long ago!

Tip

`c()` **(concatenate) is one of the most commonly used functions — it will appear again and again! (try** `?c`**).**

OK continuing our R warmup:

```
> 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
```

Tip

Thus, a "vector" is like a single column or row in a "spreadsheet". Multiple vectors can be combined to make a matrix (the full spreadsheet).

This is one of many ways R stores and processes data. More on R data types and objects below.

A single value (any kind) is a vector object of length 1 by default. That's why in the console you see `[1]` before any single-value output (e.g., type `8`, and you will see `[1] 8`).

```
> 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
```

7.6.3 Variable names and Tabbing

In R, you can name variables in the following way to keep track of related variables:

```
> wing.width.cm <- 1.2 #Using dot notation
```

```
> wing.length.cm <- c(4.7, 5.2, 4.8)
```

This can be handy; type:

```
> wing.
```

And then hit the `tab` key to reveal all variables in that category. This is nice — variable names should be as obvious as possible. However, they should not be over-long either! Good style and readability is more important than just convenient variable names.

7.6.4 Operators

The usual "operators" are available in R:

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

7.6.5 When things go wrong

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

- 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 or mismatched brackets (see next subsection)
- Do not mix double quotes and single quotes

Tip

When things are taking too long and the R command window seems frozen, try `Ctrl + C` to force an exit from whatever is going on.

7.6.6 Types of parentheses

R has specific uses for different types of parentheses that you need to get used to:

`f(3,4)`	call the function (or command) f, with the arguments 3 & 4.
`a + (b*c)`	to enforce order over which statements or calculations are executed. Here `(b*c)` is executed before adding to `a`; here is an alternative order: `(a + b)*c`
`{ expr1; expr2; ...exprn }`	group a set of expressions or commands into one compound expression. Value returned is value of last expression; used in building function, loops, and conditionals (more on these soon!).
`x[4]`	get the 4th element of the vector `x`.
`li[[3]]`	get the 3rd element of some list `li`, and return it. (compare with li[3], which returns a list with just the 3rd element inside). More on lists in next section

7.7 Variable Types

There are different kinds of data variable types in R, but you will basically need to know four for most of your work: integer, float (or "numeric", including real numbers), string (or "character", e.g., text), and Boolean ("logical"; `True` or `False`). Try this:

```
> v <- TRUE
> class(v)
[1] "logical"

> v <- 3.2
> class(v)
[1] "numeric"

> v <- 2L
> class(v)
[1] "integer"

> v <- "A string"
> class(v)
[1] "character"
```

Tip

`E` Notation
R will use `E` notation in outputs of statistical tests to display very large or small num-

bers. If you are not used to different representations of long numbers, the `E` notation might be confusing.
Try this:

```
> 1E4
[1] 10000
> 1e4
[1] 10000
> 5e-2
[1] 0.05
> 1E4^2
[1] 1e+08
> 1 / 3 / 1e8
[1] 3.333333e-09
```

Keep and eye out for `E` notation!

7.7.1 Type Conversion and Special Values

In the following examples, the `as.*` commands all convert a variable from one type to another:

```
> as.integer(3.1)
[1] 3
> as.numeric(4)
[1] 4
> as.roman(155)
[1] CLV
> as.character(155) # same as converting to string
[1] "155"
> as.logical(5) #what's happening here?!
[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
```

Tip

Beware of the difference between `NA` (`N`ot `A`vailable) and `NaN` (`N`ot `a` `N`umber)!
R will use `NA` to represent/identify missing values in data or outputs, while `NaN` represent nonsense values (e.g., 0/0) that cannot be represented as a number or some other data type.
See what R has to say about this: try `?is.nan`, `?is.na`, `?NA`, `?NaN` in the R commandline (one at a time!).
There are also `Inf` (Infinity, e.g., 1/0), and `NULL` (variable not set) value types. Look these up as well using `?` .

7.8 Data Structure types

R comes with different built-in structures for data storage and manipulation. Mastering these, and knowing which one to use when will help you write better, more efficient programs and also handle diverse datasets (numbers, counts, names, dates, etc).

7.8.1 Vectors

The Vector, which you first saw above, is a fundamental data object in R. Scalars (single data values) are treated as vector of length 1. *A vector is like a single column or row in a spreadsheet.* Now get back into R (if you somehow quit R using `q()` or something else), and try this:

```
> 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)
```

`R` vectors can only store data of a single type (e.g., all numeric or all character). If you try to combine different types, R will homogenize everything to the same data type. To see this, try the following:

```
> v1 <- c(0.02, TRUE, 1)
> v1 # TRUE gets converted to 1.00!
[1] 0.02 1.00 1.00
> v1 <- c(0.02, "Mary", 1)
> v1 # Everything gets converted to text!
[1] "0.02" "Mary" "1"
```

7.8.2 Matrices and arrays

A `R matrix` is a 2 dimensional vector (has both rows and columns). and an `R array` is can store data in more than two dimensions (e.g., a stack of 2-D matrices).

`R` has many functions to build and manipulate matrices and arrays. Try:

```
> mat1 <- matrix(1:25, 5, 5)
> mat1
     [,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
> mat1 <- matrix(1:25, 5, 5, byrow=TRUE)
> mat1
     [,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(mat1) #get the size of the matrix
[1] 5 5
```

Make an array consisting of two 5×5 matrices containing the integers 1–50:

```
> 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
```

Just like vectors, R matrices and arrays have to be of a homogeneous type, and R will do the same sort of type homogenization you saw for vectors above (try inserting a text value in `mat1` and see what happens).

7.8.3 Data frames

This is a very important data type in R. Unlike matrices and vectors, R data frames can store data in which each column contains a different data type (e.g., numbers, strings, boolean) or even a combination of data types, just like a standard spreadsheet. Indeed, the dataframe data type was built to emulate some of the convenient properties of spreadsheets. Many statistical and plotting functions and packages in R naturally use data frames. Let's build and manipulate a dataframe:

```
> 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 numbers from a uniform distribution
```

```
> 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
```

Unlike matrices, you can access the contents of data frames by naming the columns using a $ sign:

```
> MyDF$A.name
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[,1] #using numerical indexing instead
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[c("A.name","another")] # show two specific columns only
   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
```

You can check whether a particular object is a dataframe data structure with:

```
> class(MyDF)
[1] "data.frame"
```

You can check the structure of a dataframe with `str()`:

```
> str(MyDF)
'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 ...
```

You can print the column names and top few rows with `head()`,

```
> head(MyDF)
```

And the bottom few rows with `tail()`,

```
> tail(MyDF)
```

Tip

What is the "factor" data type in the data frame you created? You will see that R may mysteriously convert certain columns in dataframes to a factor type, which has "levels".
R has a special data type called 'factor'. It basically considers columns containing strings-only in a data frame to be a grouping variable. You can convert vectors to and from the `factor` class. Try the following:

```
> a <- as.factor(2)
> a
[1] 2
Levels: 2
> class(a)
[1] "factor"
```

7.8.4 Lists

A list is used to collect a group of data objects of different sizes and types (e.g., one whole data frame and one vector can both be in a single list). It is simply an ordered collection of objects (that can be variables). The outputs of many statistical functions in R are lists (e.g. linear model fitting using `lm()`), to return all relevant information in one output object. So you need to know how to unpack and manipulate lists.

```
> List1 <- list(species=c("Quercus robur","Fraxinus excelsior"), age=c(123, 84))
> List1
$species
[1] "Quercus robur"      "Fraxinus excelsior"

$age
[1] 123 84
```

You can access contents of a list item using number of the item instead of name:

```
> List1[[1]]
[1] "Quercus robur"      "Fraxinus excelsior"
> List1[[2]]
[1] 123 84
```

And you can access it using the name of the item in these two way:

```
> List1[["age"]]
[1] 123 84
> List1$age
[1] 123 84
```

You can build lists of lists too! Also, you have perhaps guessed by now that R dataframes are actually a kind of list.

Tip

If dataframes are so nice, why use R matrices at all? The problem is that dataframes can be too slow when large numbers of mathematical calculations or operations need to be performed. In such cases, you will need to convert a dataframe to a matrix. But for statistical analyses and plotting, *data frames will get the job done***.**

7.9 Creating and manipulating data structures

7.9.1 Creating 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.

7.9.2 Acessing parts of data stuctures – Indices and Indexing

Every element (entry) of a vector in R has an order: the first value, second, third, etc. To illustrate this, let's create a simple vector:

```
> MyVar <- c( 'a' , 'b' , 'c' , 'd' , 'e' )
```

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

```
> 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"
```

You can also manipulate data structures/objects 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
```

For matrices, you need to use both row and column indices:

```
> mat1 <- matrix(1:25, 5, 5, byrow=TRUE) #create a matrix
> mat1
     [,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
> mat1[1,2]
[1] 2
> mat1[1,2:4]
[1] 2 3 4
> mat1[1:2,2:4]
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    7    8    9
```

7.9.3 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!

7.9.4 Basic vector-matrix operations

```
> v <- c(0, 1, 2, 3, 4)
> v2 <- v*2 # multiply whole vector by 2
> v2
[1] 0 2 4 6 8
> v * v2 # element-wise product
[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
```

7.9.5 Strings and Pasting

It is important to know how to handle strings in R for two main reasons:

- To deal with text data, such at experimental treatment names
- To generate appropriate text labels and titles for figures

Let's try creating and manipulating strings:

```
> 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
```

As you can see above, both double and single quotes work, but I suggest that you use double quotes — this will allow you to define strings that contain a single quotes, which is often necessary.

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.

Tip

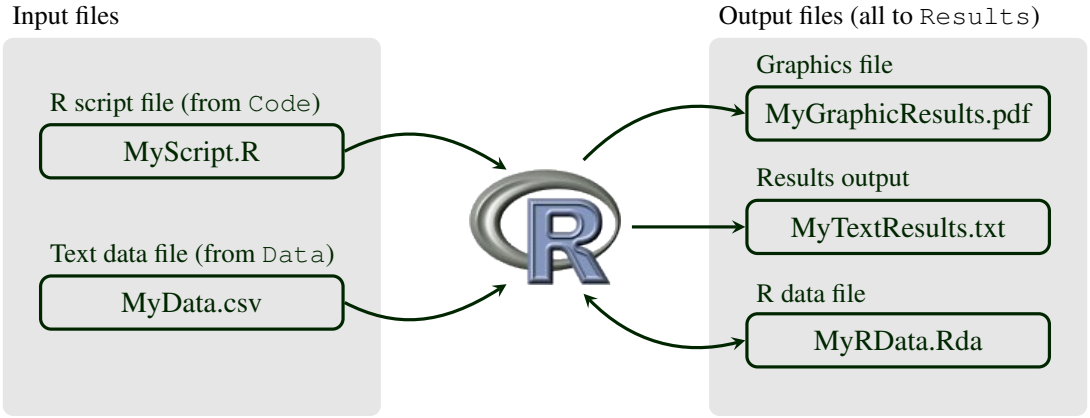
Data "structures" vs. "objects" in `R`**:**

You will often see the terms "object" and "data structure" thrown around this week and elsewhere. These two have a very distinct meaning in object-oriented programs (OOPs) like `R`. The main thing you need to keep in mind is that a data structrure is just a "dumb" container for data (e.g., a vector). An object, on the other hand can be a data structure, but also any other variables or functions in your R environment. R, being an OOP, needs to convert everything in the current environment to an object so that it knows what to do with each such entity — each object type has its own set of rules for operations and manipulations that R uses when interpreting your commands. If all this soudns like gobbledygook, don't worry too much!

7.10 Your analysis workflow

In using `R` for an analysis, you will likely use and create several files. 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" (more on this below). Also, you don't want to mix code files with data and results files. So to keep everything you will create separate directories for these as well.

Thus, your typical `R` analysis workflow will be:



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 RStudio in this class (more on this below). 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 single`Rdata` 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.

So let's build your `R` analysis project structure.

Do the following:

- ⋆ Create a sensibly names directory (e.g., `MyICRModule`, `Week3`, etc in an appropriate location on your computer. If you are using a college Windows computer, you will need to create it in your `H:` drive.
- ⋆ Create subdirectories within `this directory` called `Code`, `Data`, and `Results`

Tip

Avoid including spaces in your file or directory names, as this will often create problems when you share your file or directory with somebody else. Many software programs do not handle spaces in file/directory names well. Use underscores instead of spaces. For example, instead of `My IC R Module`, use `My_IC_R_Module`.

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

```
> dir.create("MyICRModule")
> dir.create("MyICRModule/Code")
> dir.create("MyICRModule/Data")
> dir.create("MyICRModule/Results")
```

7.10.1 The R Workspace and Working Directory

R has a "workspace" – a current working environment that includes any user-defined data structures objects (vectors, matrices, data frames, lists) as well as other objects (e.g., functions). At the end of an R session, the user can save an image of the current workspace that is automatically reloaded the next time R is started. Your workspace is saved in your "Working Directory", which has to be set manually.

[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.

So before we go any further, let's get sort out where your R "Working Directory" should be and how you should set it. R has a default location where it assumes your working directory is.

in Windows, it is `C:/Windows/system32` or similar.

in Mac, it is `/User/User Name` or similar.

In UNIX/Linux, it is where you are when you launch R.

To see where your current working directory is, at the R command prompt, type:

```
> getwd()
```

This tells you what the current working directory ("`wd`") is.

Now, set the working directory to be `MyICRModule/Code`. For example, if you created `MyICRModule` directly in your `H:\`, the you would use:

```
> setwd("H:/MyICRModule/Code")
> dir() #check what's in the current working directory
```

On your own computer, you can also change R's default to a particular working directory where you would like to start (easily done in RStudio).

In Linux, you can do this by editing the `Rprofile.site` site with `sudo geany /etc/R/Rprofile.site`. In that file, you would add your start-up parameters between the lines `.First <- function() cat("\n Welcome to R!\n\n")` and `.Last <- function() cat("\n Goodbye!\n\n")` — between these lines, insert `setwd("/home/YourName/YourDirectoryPath")`

In Windows and Macs, you can find the `Rprofile.site` file by searching for it. When I last checked, it used to be at `C:\Program Files\R\etc\Rprofile.site`

If you are using RStudio, you can change the default working directory by thorugh the RStudio "Options" dialog.

7.11 Importing and Exporting Data

We are now 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 (`csv`) text file or in a tab separated text file. Then, you can use the function `read.csv` (or `read.table`) to import your data. Now, lets get some data into your `Data` directory.

- ⋆ Go to the repository you downloaded from bitbucket and unzipped, and navigate to the `Data` directory.
- ⋆ Copy the file `trees.csv` into your own `Data` directory.
- ⋆ Now, try the following:

```
> MyData <- read.csv("../Data/trees.csv")
> ls() #Check that MyData has appeared
> 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) #another 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. Also, note the UNIX-like paths using forward slashes (Windows uses back slashes).

7.11.1 Relative paths!

The `../` in `read.csv("../Data/trees.csv")` above signifies a "relative" path. That is, you are asking R to load data that lies in a different directory (folder) relative your current location (in this case, you are in your `Code` directory). In other, more dorky words, `../Data/trees.txt` points to a file named `trees.txt` located in the "parent" of the current directory.

What is an absolute path?— one that specifies the whole path on your computer, say from `C:/` "upwards".

Using relative paths in in your R scripts and code will make your code computer independent and your life better! The relative path way should always be the way you load data in your analyses scripts — it will guarantee that your analysis works on every computer, not just your college computer.

Also, *AVOID putting a* `setwd` *command at the start of your R script*, as setting the working directory always requires an absolute directory path, which will differ across computers, platforms, and users. Let the end user sort out how to set the working directory. So to import data and export results, your script should *not* use absolute paths.

7.11.2 Writing out to and saving files

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
```

7.12 Writing R code

Typing in commands interactively in the R console 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.

- ⋆ Open a new text file, call it `basic_io.R`, and save it to your `Code` directory.
- ⋆ Write the above input-output commands in it:

```
#A simple R script to illustrate R input-output.
# Run line by line and check inputs outputs to understand what is
# happening

MyData <- read.csv("../Data/trees.csv", header = TRUE) # import with headers
```

```
write.csv(MyData, "../Results/MyData.csv") #write it out as a new file

write.table(MyData[1,], file = "../Results/MyData.csv",append=TRUE) # Append to it

write.csv(MyData, "../Results/MyData.csv", row.names=TRUE) # write row names

write.table(MyData, "../Results/MyData.csv", col.names=FALSE) # ignore column names
```

You will get a warning with `write.table(MyData[1,], file = "../Results/MyData.csv",apper` because R thinks it is silly that you are appending hearders to the file, which already has headers!

- ⋆ Place the cursor on the first line of code in the script file and run it by pressing the keyboard shortcut (PC: ctrl+R, Mac: command+enter, Linux: ctrl+enter if you are using `geany`).
- ⋆ Check after every line that you are getting the expected result.

7.12.1 Running R code

But even writing to a script file and running the code line-by-line or block-by-block is not your ultimate goal. What you would really like to do (believe me, you do!) is 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.

- ⋆ Try sourcing `basic_io.R`:

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

- ⋆ If you get errors, fix them!

Tip

Do not put a `source()` command inside the script file you are sourcing, as it is then trying to run itself again and again and that's just cruel!

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

Tip

The command `source()` has a `chdir` argument whose default value is FALSE. When set to TRUE, it will change the working directory to the directory of the file being sourced.

7.13 Writing R Functions

Like any other programming language, R lets you write your own functions. A function is a block of re-useable code that takes an input, does something with it (or to it!), and returns the result. All the "commands" that you have been using, such as `ls()`, `mean()`, `c()`, etc are basically functions. You will want to write your own function for eevery scenario where a particular, task or set of analysis steps need to be performed again and again.

The syntax for R functions is quite simple, with each function accepting "arguments" and "returning" a value.

Type the following into a script file called `boilerplate.R` and save it in your `Code` directory.

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

  # Statements involving Arg1, Arg2:
  print(paste("Argument", as.character(Arg1), "is a", class(Arg1))) # print Arg1's type
  print(paste("Argument", as.character(Arg2), "is a", class(Arg2))) # print Arg2's type

  return (c(Arg1, Arg2)) #this is optional, but very useful
}

MyFunction(1,2) #test the function
MyFunction("Riki","Tiki") #A different test
```

Note the curly brackets – these are necessary for R to know where the specification of the function starts and ends. Also, note the indentation. Not necessary (unlike Python), but recommended to make code more readable.

Now source the script

```
> source("boilerplate.R")
```

This will run the script, and also save your function `MyFunction` as an object into your workspace (try `ls()`, and you will see `MyFunction` appear in the list of objects).

Also try:

```
> class(MyFunction)
[1] "function"
```

So, yes, `MyFunction` is a function!

Now let's write an script containing a more useful function:

- ⋆ In your text editor type the following in a file called `TreeHeight.R`, and save it in your `Code` directory:

```
# 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)
```

⋆ Run `TreeHeight.R` block by block and check what each line is doing.
⋆ Now run it using `source`.
⋆ If you get errors, read the error mesages and fix them.

7.14 Practicals

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
```

2. Imagine you wanted to make the `TreeHeight.R` script more general so that it could be used for other datasets, not just `trees.csv`.

   - Write another R script called `get_TreeHeight.R` that takes a csv file name from the command line (e.g., `get_TreeHeight.R Trees.csv`) and outputs the result to a file just like `TreeHeight.R` above, but this time includes the input file name in the output file name as `InputFileName_treeheights.csv`. Note that you will have to strip the `.csv` or whatever the extension is from the filename, and also `../` etc., if you are using relative paths
     (Hint: Command-line parameters are accessible within the R running environment via `commandArgs()` — so help(commandArgs) might be your starting point.)
   - Write a Unix shell script called `run_get_TreeHeight.sh` that tests `get_TreeHeight.R` — you can include `trees.csv` as your example file.

7.15 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. Such statements are useful to include in functions and scripts because you may only want to do certain calculations or otehr

tasks, under certain conditions (e.g., `if` the dataset is from a particular year, do something different).

- ⋆ Type the following in a script file called `control.R` (save it in your `Code` directory)
- ⋆ Run `control.R` function block by block (not line by line!) and check what each line is doing.
- ⋆ Now run it using `source`.
- ⋆ If you get errors, fix them! (OK, I am going to stop saying this henceforth)

```
## 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)
}
```

7.16 Useful R Functions

There are a number of very useful functions available by default (in the "base packages"). Here are some particularly useful ones:

7.16.1 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

7.16.2 Strings

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

7.16.3 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

7.17 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.

In UNIX, you can use the terminal to install R packages:

```
$ sudo apt-get install r-cran-ggplot2 r-cran-plyr r-cran-reshape2
```

In Windows and Macs, you can install a package within R by using the `install.packages()` command.

Let's install the following packages:

```
> install.packages(c("ggplot2","plyr","reshape2"))
```

In UNIX, you will have to launch a `sudo R` session to get this to work.

You can also use the RStudio GUI to install packages using your mouse and menu.

7.18 Practicals wrap-up

1. Review and make sure you can run all the commands, code fragments, and named scripts we have built till now and get the expected outputs.
2. Annotate/comment your code lines as much and as often as necessary using #.
3. Keep all files organized in `code`, `data` and `results` directories.

`git add, commit` *and* `push` *all your code and data from this chapter to your git repository by the following Wednesday 5PM.*

7.19 Readings

- Check the readings under the R directory in the bitbucket master repository
- 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" (coming up in Chapter 8), "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 8

Data wrangling, exploration, and visualization in R

Before you do any fancy statistical analyses with data, you must clean, explore, and visualize it. This chapter aims at introducing you to R packages and commands that will allow you to build a computational pipeline/workflow for these critical steps of your data analysis.

We will start with some basic plotting and data exploration. We will then use the Pound Hill Data collected by students in the Silwood Field Course week to learn some rules and methods for data processing and storage. Finally, you will learn to generate publication-quality graphics using the `ggplot2` package.

8.1 Basic plotting and graphical data exploration

R can produce beautiful graphics without the time-consuming and fiddly methods that you might have used in Excel or equivalent. You should also make it a habit to quickly plot the data for exploratory analysis. So we are going to learn some basic plotting first.

8.1.1 Basic plotting commands

Here is a menu of basic R plotting commands (use `?commandname` to learn more about it):

`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

8.1.2 R graphics devices

In all that follows, you may often end up plotting multiple plots on the same graphics window without intending to do so, because R by default keeps plotting in the most recent plotting window that was opened. You can close a particular graphics window or "device" by using `dev.off()`, and all open devices/windows with `graphics.off()`. By default, `dev.off()` will close the most recent figure device that was opened. Note that there are invisible devices as well! That is if you are printing to pdf (coming up below), the device or graphics window will not be visible on your computer screen.

So let's try some simple plotting for data exploration. As a case study, we will use a dataset on Consumer-Resource (e.g., Predator-Prey) body mass ratios taken from the Ecological Archives of the ESA (Barnes *et al.* 2008, Ecology 89:881).

- ⋆ Go to the repository you downloaded from bitbucket and unzipped, and navigate to the `Data` directory.
- ⋆ Copy the file `EcolArchives-E089-51-D1.csv` into your own `Data` directory.
- ⋆ Now, do the following:

```
> MyDF <- read.csv("../Data/EcolArchives-E089-51-D1.csv")
> dim(MyDF) #check the size of the data frame you loaded
[1] 34931    15
```

Let's look at what the data contain (type `MyDF$` and hit the TAB key twice. This will give the following result:

```
> 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
```

In RStudio, you will instead see a drop-down list of all the column headers.

You can also use the `str()` and `head()` commands that you learned about in the Chapter 7.

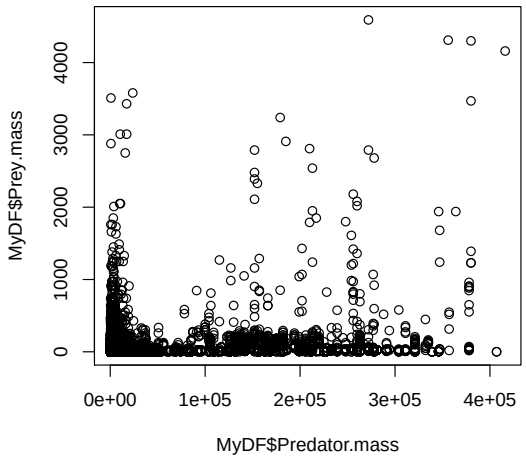
As you can see, these data contain predator-prey body size information. This is an interesting dataset because it is huge, and covers a wide range of body sizes of aquatic species involved in consumer-resource interactions — from unicells to whales. Analyzing this dataset should tell us a lot about what sizes of prey predators like to eat.

8.1.3 Scatter Plot

So let's start by plotting Predator mass vs. Prey mass:

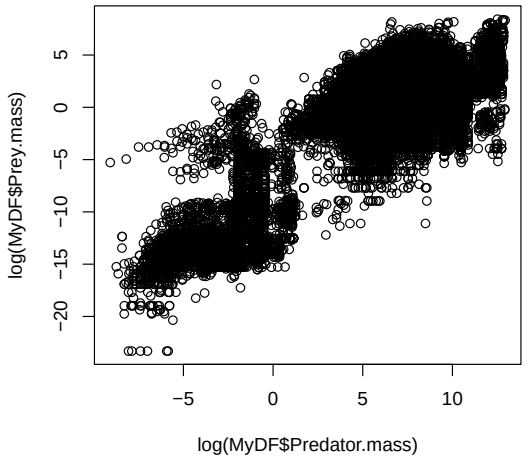
```
> plot(MyDF$Predator.mass,MyDF$Prey.mass)
```

Figure 8.1: A consumer-resource (predator-prey) interaction waiting to happen.



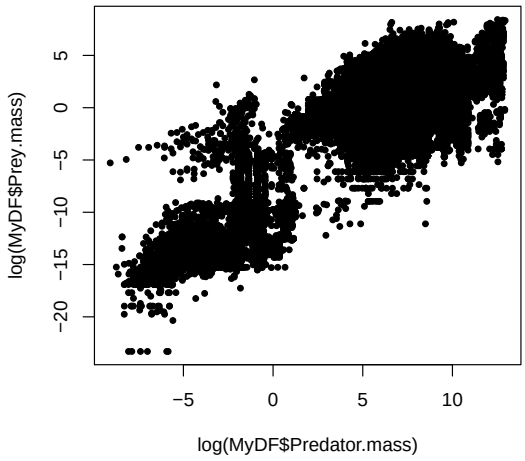
That doesn't look very nice! Let's try taking logarithms (why?).

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

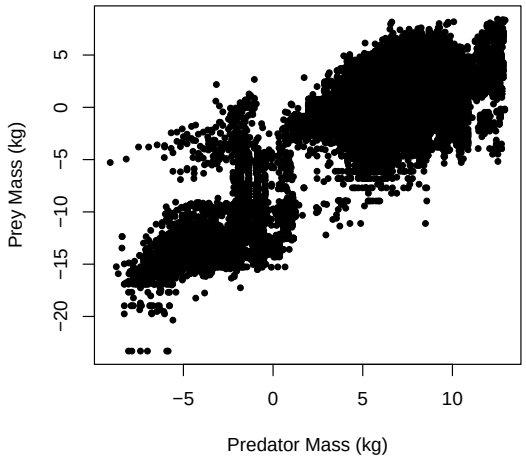


We can change almost any aspect of the resulting graph; let's change the symbols by specifying the `plot` `ch`aracters using `pch`:

```
> 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
```



Tip

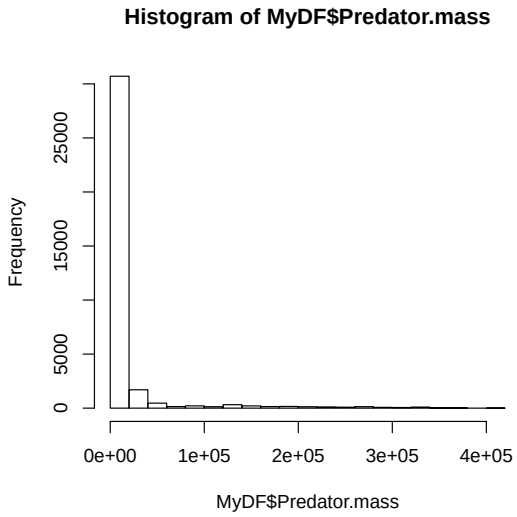
A really great summary of basic R graphical parameters can be found at `http://www.statmethods.net/advgraphs/parameters.html`

8.1.4 Histograms

Why did we have to take a logarithm to see the relationship between predator and prey size? Plotting histograms of the two classes (predator, prey) should be insightful, as we can then see the "marginal" distributions of the two variables.

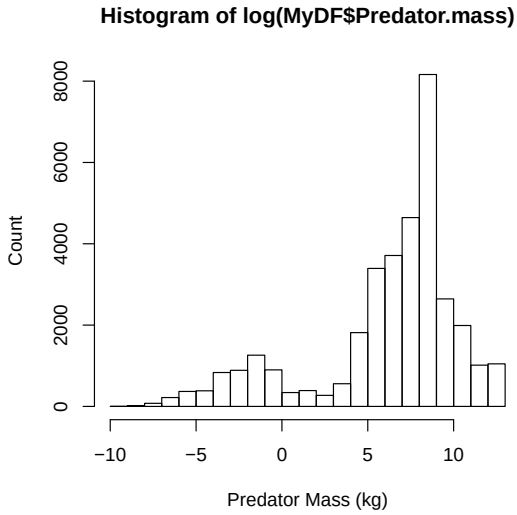
Let's first plot a histogram of predator body masses:

```
> hist(MyDF$Predator.mass)
```

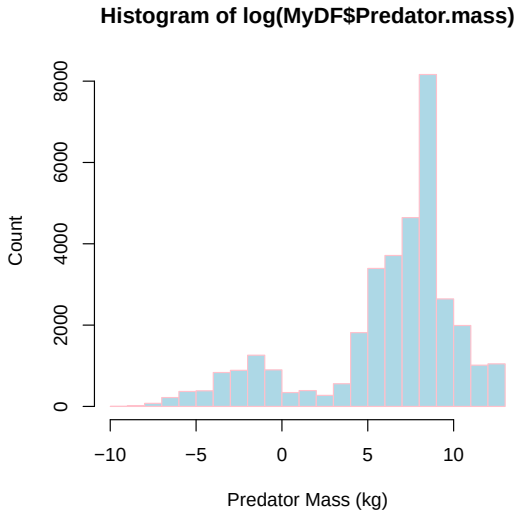


Clearly, the data are heavily skewed, with small body sized organisms dominating (that's a universal pattern on planet earth). Let's now take a logarithm and see if we can get a better idea of what the distribution of predator sizes looks like:

```
> hist(log(MyDF$Predator.mass),
    xlab = "Predator Mass (kg)", ylab = "Count") # include labels
```




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



So, taking a log really makes clearer what the distribution of body predator sizes looks like. *Try the same with prey body masses.*

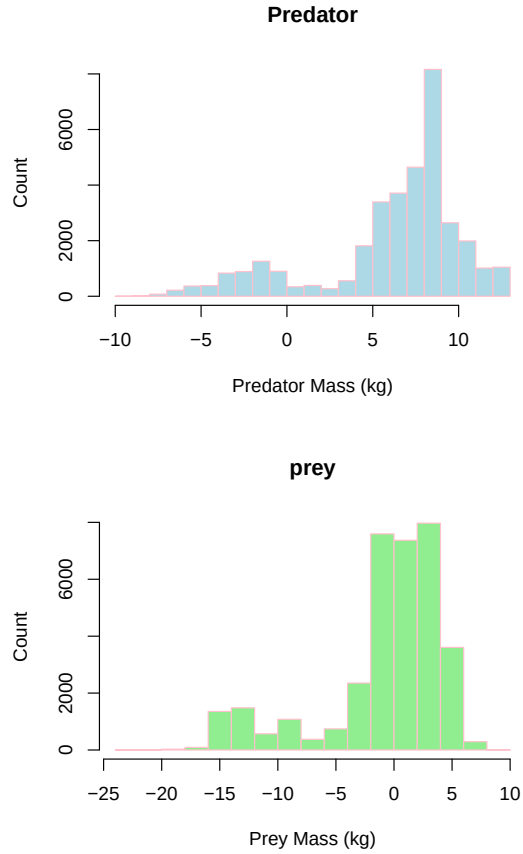
Exercise

We can do a lot of beautification and fine-tuning of your R plots! As an exercise, try adjusting the histogram bin widths to make them same for the predator and prey, and making the x and y labels larger and in boldface. To get started, look at the help documentation of `hist`.

8.1.5 Subplots

We can also plot both predator and prey body masses in different sub-plots using `par` so that we can compare them visually.

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



Another option for making multi-panel plots is the `layout` function.

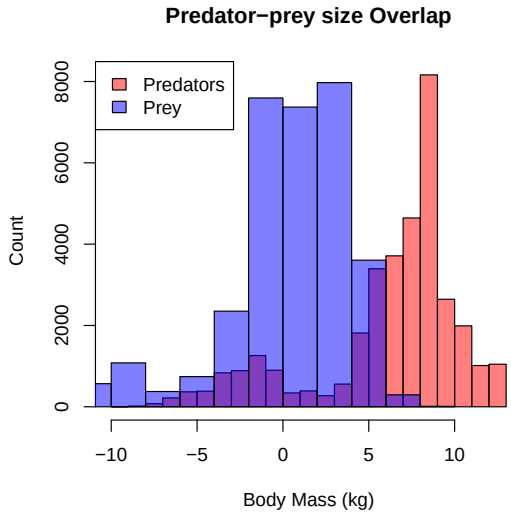
8.1.6 Overlaying plots

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

```
> hist(log(MyDF$Predator.mass), # Predator histogram
    xlab="Body Mass (kg)", ylab="Count",
    col = rgb(1, 0, 0, 0.5), # Note 'rgb', fourth value is transparency
    main = "Predator-prey size Overlap")
> hist(log(MyDF$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
```

Tip

Plot annotation with text can be done with either single or double quotes, i.e., 'Plot Title' or "Plot Title", respectively. But it is generally a good idea to use double quotes because sometimes you would like to use an apostrophe in your title or axis label strings.

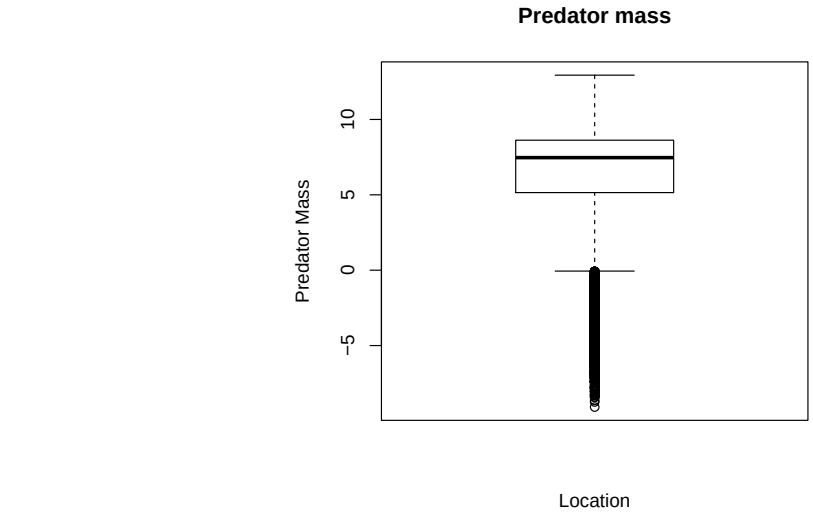


It would be nicer to have both the plots with the same bin sizes – try to do it

8.1.7 Boxplots

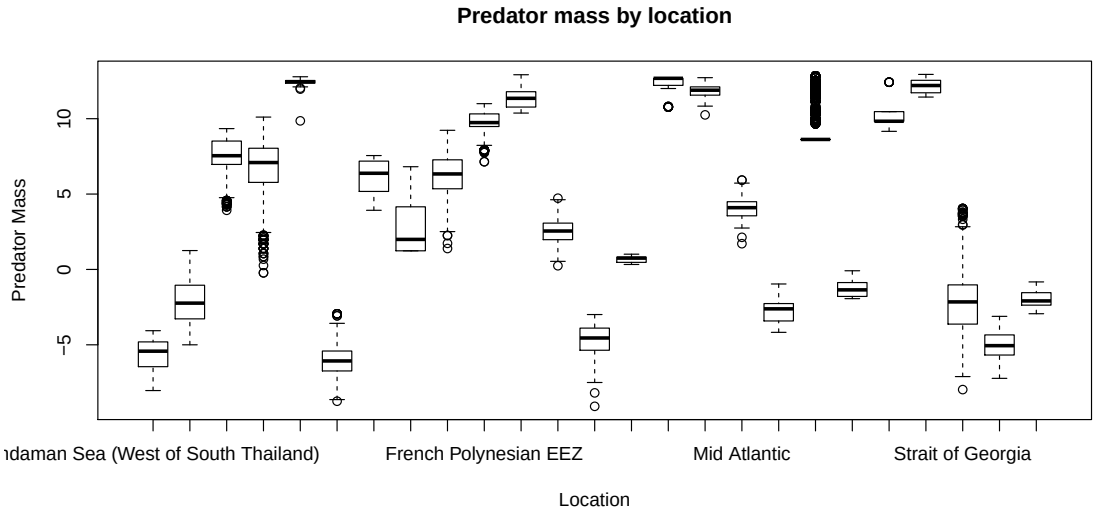
Now, let's try plotting boxplots instead of histograms. These are useful for getting a visual summary of the distribution of your data.

```
> boxplot(log(MyDF$Predator.mass),
    xlab = "Location", ylab = "Predator Mass",
    main = "Predator mass")
```



Now let's see how many locations the data are from:

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



Note the tilde (~). This is to tell R to subdivide or categorize your analysis and plot by the "Factor" location. More on this later.

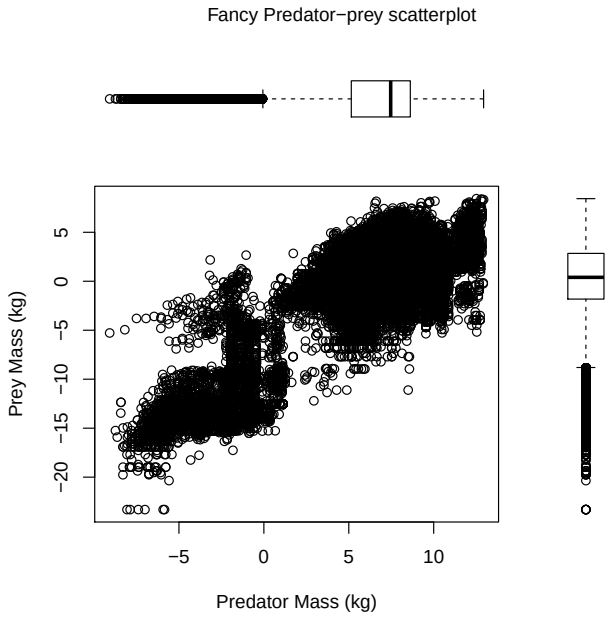
That's a lot of locations! You will need an appropriately wide plot to see all the boxplots adequately. Now let's try boxplots by feeding interaction type:

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

8.1.8 Combining plot types

It would be nice to see both the predator and prey marginal distributions as well as the scatterplot for an exploratory analysis. We can do this by adding boxplots of the marginal variables to the scatterplot.

```
> par(fig=c(0,0.8,0,0.8)) # specify figure size as proportion
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass),
    xlab = "Predator Mass (kg)", ylab = "Prey Mass (kg)") # Add labels
> par(fig=c(0,0.8,0.55,1), new=TRUE)
> boxplot(log(MyDF$Predator.mass), horizontal=TRUE, axes=FALSE)
> par(fig=c(0.65,1,0,0.8),new=TRUE)
> boxplot(log(MyDF$Prey.mass), axes=FALSE)
> mtext("Fancy Predator-prey scatterplot", side=3, outer=TRUE, line=-3)
```



To understand this plotting method, think of the full graph area as going from (0,0) in the lower left corner to (1,1) in the upper right corner. The format of the `fig=` parameter is a numerical vector of the form `c(x1, x2, y1, y2)`. The first `fig=` sets up the scatterplot going from 0 to 0.8 on the x axis and 0 to 0.8 on the y axis. The top boxplot goes from 0 to 0.8 on the x axis and 0.55 to 1 on the y axis. The right hand boxplot goes from 0.65 to 1 on the x axis and 0 to 0.8 on the y axis. You can experiment with these proportions to change the spacings between plots.

8.1.9 Lattice plots

You can also make lattice graphs to avoid the somewhat laborious `par()` approach above of getting multi-panel plots. For this, you will need to "load" a "library" that isnl't included by default when you run R:

```
> library(lattice)
```

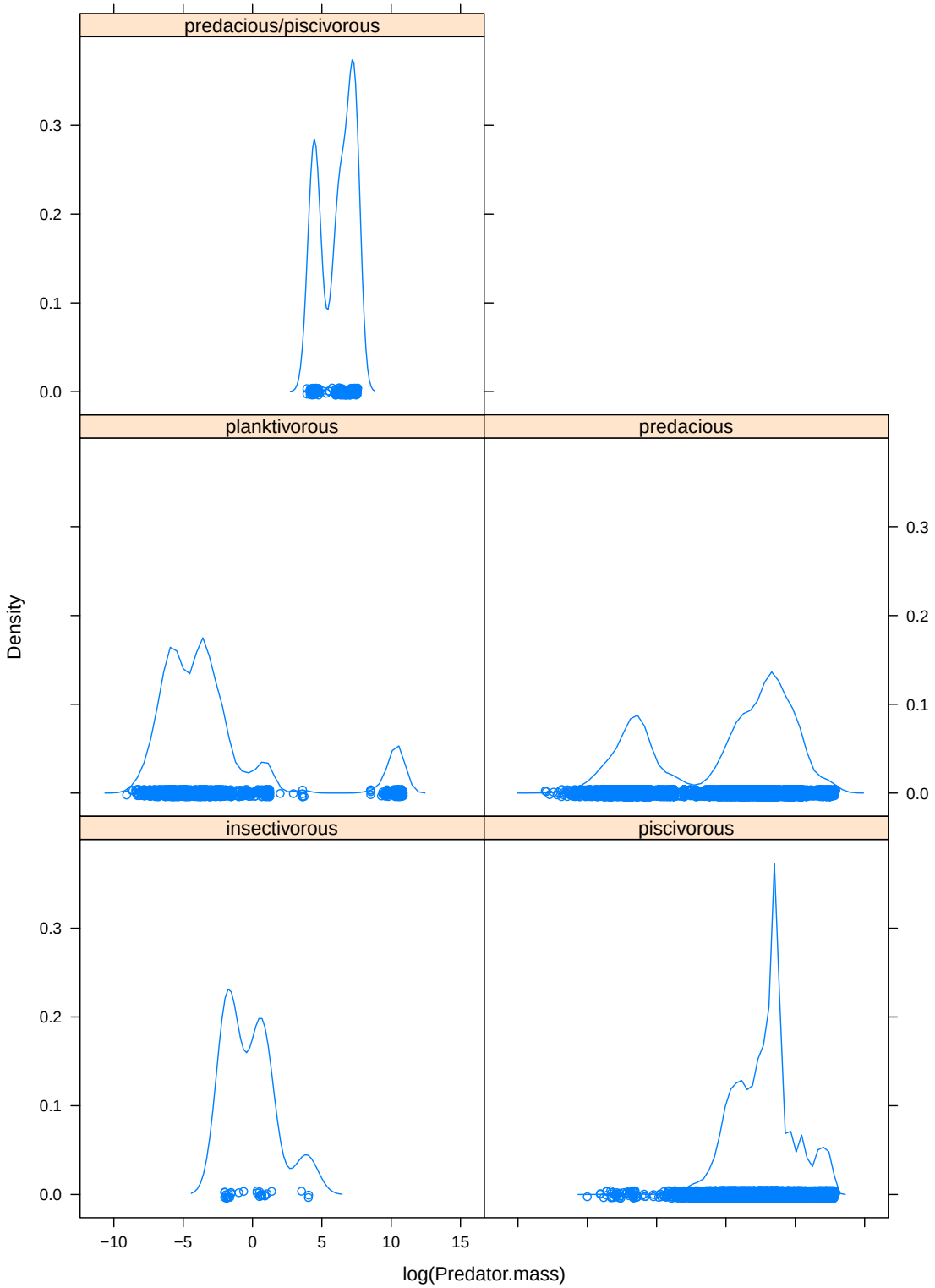


Figure 8.2: A `lattice` representation of the predator size data

A lattice plot of the above data for predator mass could look like Fig. 8.1.9 (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.

8.1.10 Saving your graphics

And you can also save the figure in a vector graphics format like a pdf. It is important to learn to do this, because you want to be able to save your plots in good resolution, and want to avoid the manual steps of clicking on the figure, doing "save as", etc. So let's save the figure as a PDF:

```
> pdf("../Results/Pred_Prey_Overlay.pdf", # Open blank pdf page
    11.7, 8.3) # These numbers are page dimensions in inches
> hist(log(MyDF$Predator.mass), # Plot predator histogram (note 'rgb')
    xlab="Body Mass (kg)", ylab="Count",
    col = rgb(1, 0, 0, 0.5),
    main = "Predator-Prey Size Overlap")
> hist(log(MyDF$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()
```

Tip

Always try to save results in a vector format, which can be scaled up to any size. For more on vector vs raster images/graphics, see: `https://en.wikipedia.org/wiki/Vector_graphics`.

Note that you are saving to the `Results` directory now. This should always be your workflow: store and retrieve data from a `Data` directory, keep your code and work from a `Code` directory, and save outputs to a `Results` directory.

You can also try other graphic output formats. For example, `png()` (a raster format) instead of `pdf()`. As always, look at the help documentation of each of these commands!

8.2 Practicals

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 and median log predator mass, prey mass, and predator-prey size ratio, *by feeding type*, 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!).
- ⋆ There are multiple ways to do this practical. The plotting and saving component is simple enough. For calculating the statistics by feeding type, you can either use the "loopy" way — first obtaining a list of feeding types (look up the `unique` or `levels` functions) and then loop over them, using `subset` to extract the dataset by feeding type at each iteration, or the R-savvy way, by using `tapply` and avoiding looping altogether (however, this isnlt covered till the next chapter, coming up later in the week).

8.3 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 datasets together. The `R` package `ggplot2` overcomes these issues, and produces truly high-quality, publication-ready graphics suitable for papers, theses and reports.

Tip

Currently, `ggplot2` cannot be used to create 3D graphs or mosaic plots. In any case, most of you won't be needing 3D plots! If you do, there are many ways to do 3D plots using other plotting packages in R. In particular, look up the `scatterplot3d` and `plot3D` packages.

`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. Specifically, according to this grammar, a statistical graphic is a "mapping" from data to aesthetic attributes (colour, shape, size; set using `aes`) of geometric objects (points, lines, bars; set using `geom`).

For more on the ideas underlying ggplot, see the book "ggplot2: Elegant Graphics for Data Analysis", by H. Wickham (in your Reading directory). Also, the website `ggplot2.org` a great resource.

`ggplot2` should already be available on the college computers. If you are using your own computer, look up the section on installing packages in Chapter 7.

ggplot can be used in two ways: with `qplot` (for `q`uick `plot`ting) and `ggplot` for full-blown, customizable plotting.

Tip

Note that `ggplot2` only accepts data in data frames.

8.3.1 Basic plotting with `qplot`

`qplot` can be used to quickly produce graphics for exploratory data analysis, and as a base for more complex graphics. It uses syntax that is closer to the standard R plotting commands.

We will use the same predator-prey body size dataset again – you will soon see how much nice the same types of plots you made above look when done with ggplot!.

Scatterplots

Let's start plotting the `Predator.mass` vs `Prey.mass`:

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

As before, 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)
```

Aesthetic mappings

These examples demonstrate a key difference between `qplot` and the standard `plot` command: When you want to assign colours, sizes or shapes to the points on your plot, using the `plot` command, it's your responsibility to convert (i.e., "map") a categorical variable in your data (e.g.,

type of feeding interaction in the above case) onto colors (or shapes) that `plot` knows how to use (e.g., by specifying "red", "blue", "green", etc).

ggplot does this mapping for you automatically, and also provides a legend! This makes it really easy to quickly include additional data (e.g., if a new feeding interaction type was added to the data) on the plot.

Instead of using ggplot's automatic mapping, if you want to manually set a color or a shape, you have to use `I()` (meaning "Identity"). To see this in practise, try the following:

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

You chose red, but ggplot used mapping to convert it to a particular shade of red. To set it manually to the real red, do this:

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

Similarly, for point size, compare these two:

```
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, size = 3) #with ggplot size mapping
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, size = I(3)) #no mapping
```

But for shape, ggplot doesn't have a continuous mapping because shapes are a discrete variable. To see this, compare these two:

```
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, shape = 3) #will give error
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, shape= I(3))
```

Setting transparency

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))
```

Here try using `alpha = .5` instead of `alpha = I(.5)` and see what happens.

Adding smoothers and regression lines

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 need to specify the method as being `lm`:

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

`lm` stands for `l`inear `m`odels (linear regression is a type of linear model). You will learn about linear models and fitting them to data (as you have done here) in the Stats in R week.

We can also add a "smoother" for each type of interaction:

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

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

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

Boxplots

Or we can draw a boxplot of the data (note the `geom` argument, which stands for `geom`etry):

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

Histograms and density plots

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.

Multi-faceted plots

An alternative way of displaying data belonging to different classes is using "faceting". We did this using `lattice()` previously, but ggplot does a much nicer job:

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

The `~.` (the space is not important) notation tells ggplot whether to do the faceting by row or by column. So if you want a by-column configuration, switch ~ and `.`, and also swap the position of the `.~`:

```
> qplot(log(Prey.mass/Predator.mass),
```

```
    facets =  .~ Type.of.feeding.interaction,
    data = MyDF, geom =  "density")
```

You can also facet by a combination of categories (this is going to be a big plot!):

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

And you can also change the order of the combination:

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

Logarithmic axes

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")
```

Plot annotations

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()
```

Saving your plots

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()
```

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

8.3.2 Some more important ggplot options

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

8.3.3 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")
```

8.3.4 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`, which allows you to use "layering". Layering is the mechanism by which additional data elements are added to a plot. Each layer can come from a different dataset and have a different aesthetic mapping, allowing us to create plots that could not be generated using `qplot()`, which permits only a single dataset and a single set of aesthetic mappings.

For a `ggplot` plotting command, we need at least:

- The data to be plotted, in a data frame;
- Aesthetics mappings, specifying which variables we want to plot, and how;
- The `geom`, defining the geometry for representing the data;
- (Optionally) some `stat` that transforms the data or performs statistics using the data.

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

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

Here we have created a graphics object `p` to which we can add layers and other plot elements.

Now try to plot the graph:

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

That is because we are yet to specify a geometry — only then can we 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 + theme(legend.position = "none")
```

We will not look at some case studies to see some useful ways in which you can use ggplot.

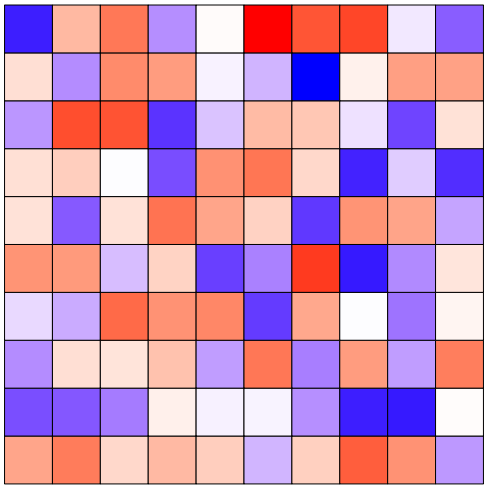


Figure 8.3: Random matrix with values sampled from uniform distribution.

8.3.5 Case study 1: plotting a matrix

Here 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 8.3. Because we want to plot a matrix, and `ggplot2` accepts only dataframes, we use the package `reshape2` that can "melt" a matrix into a dataframe:

```
require(ggplot2)
require(reshape2)

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

> Melt <- melt(M)

> Melt[1:4,]
  Var1 Var2     value
1    1    1 0.0698925
2    2    1 0.6333296
3    3    1 0.8990120
4    4    1 0.8425578

> ggplot(Melt, aes(Var1, Var2, fill = value)) + geom_tile()

# adding a black line dividing cells
> p <- ggplot(Melt, aes(Var1, Var2, fill = value))
> p <- p + geom_tile(colour = "black")

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

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

8.3.6 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 8.4).

```
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)) +
  theme(legend.position = "none")

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

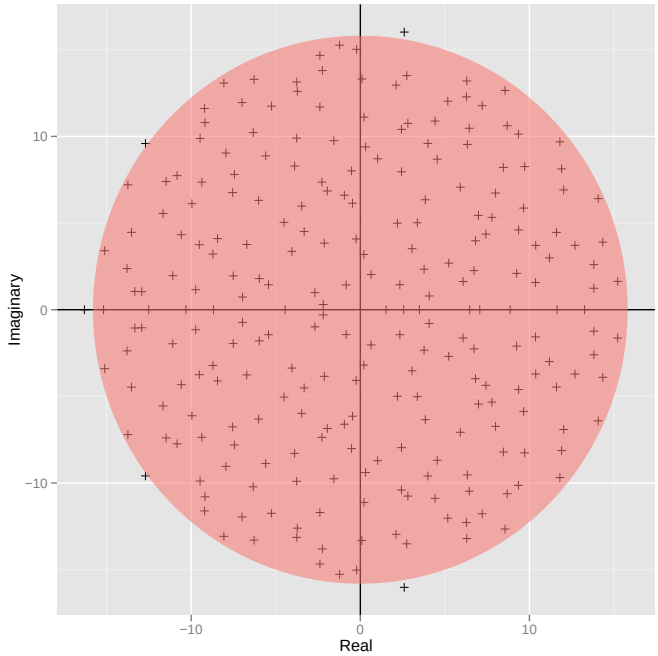


Figure 8.4: Girko's circular law.

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

8.3.7 Case study 3: annotating plots

In the plot in Figure 8.5, we use the geometry "text" to annotate the plot.

```
require(ggplot2)

a <- read.table("../Data/Results.txt", 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)
```

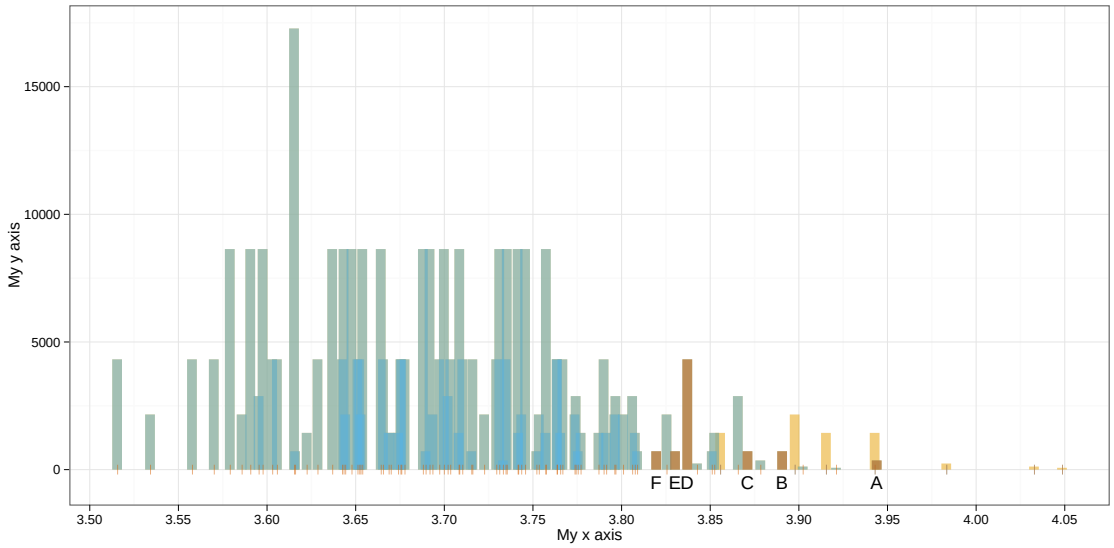


Figure 8.5: Overlay of three lineranges and a text geometry.

```
# 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() +
  theme(legend.position = "none")

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

8.3.8 Case study 4: mathematical display

```
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") +
  theme(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("../Results/MyLinReg.pdf")
print(p)
dev.off()
```

In Figure 8.6, you can see the mathematical annotation on the axis and in the plot area itself.

8.4 Data wrangling and exploration

You are likely to spend far more time than you think dredging through your data manually, checking it, editing it, and reformatting it to make it useful for the actual data exploration and statistical analysis. For example, you may need to:

- Identify the variables vs observations within the data — somebody else might have recorded the data, or you might have collected the data some time back!
- Fill in zeros (true measured/observed absences)
- Identify and add a value (e.g., `-999999`) to denote missing observations
- Derive or calculate new variables from the raw observations (e.g., convert measurements to SI units; kilograms, meters, seconds, etc.)
- Reshape/reformat your data into a layout that works best for analysis (e.g., for `R` itself) — e.g., from Wide to long data format for repeated (across sites, plots, plates, chambers, etc) measures data
- Merge multiple datasets together into a single data sheet

And this is far from an exhaustive list! Doing so many different things to your raw data is both

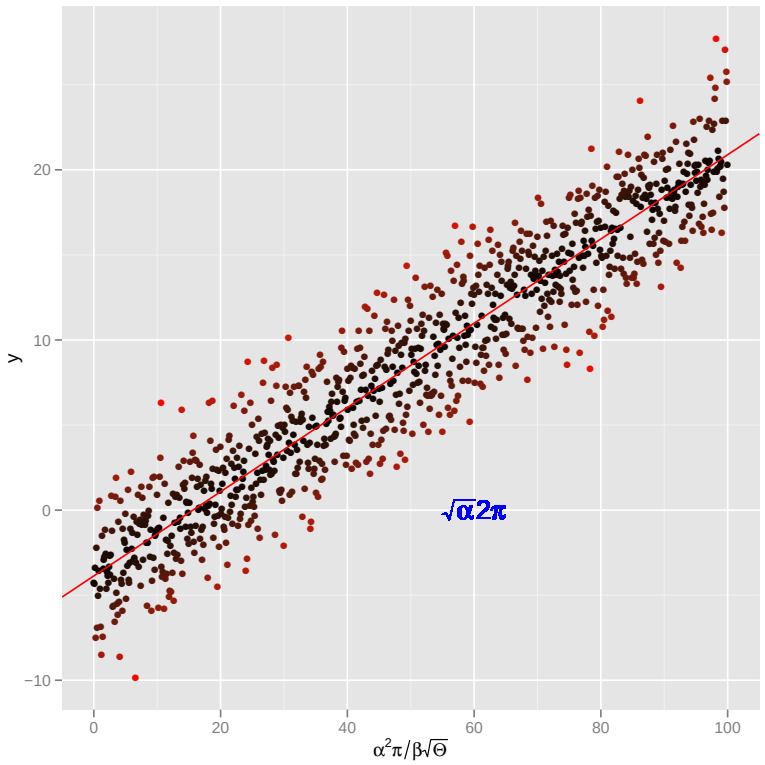


Figure 8.6: Linear regression with colors expressing residuals and mathematical annotations.

Figure 8.7: An illustration of a (metaphorical) datum being wrangled into submission.

time-consuming and risky. Why risky? Because to err is very human, and every new, tired mouse-click and/or keyboard-stab has a high probability of being incorrect!

8.4.1 Some data wrangling principles

So you would like to a record of the data wrangling process (so that it is repeatable and even reversible), and automate it to the extent possible. To this end, here are some rules and steps:

- Store data in universally-readable, nonproprietary software formats (e.g., `.csv`)
- Use plain ASCII text for your file names, variable/field/column names, and data values — make sure the "text encoding" is correct and standard (e.g., `UTF-8`)
- Keep a metadata file for each unique dataset (agian, in non-proprietary format)
- Don't (over-)modify your raw data by hand — use scripts instead — keep a copy of the data as they were recorded.
- Use meaningful names for your data and files and field (column) names
- When you add data, try not to add columns (widening the format); rather, design your tables/datasheets so that you add only rows (lengthening the format) — and convert "wide format data" to "long format data" using scripts, not by hand!
- All cells within a data column should contain only one type of information (i.e., either text (character), numeric, etc.).
- Ultimately, consider creating a relational database for your data (out of our scope here — see next chapter).

This is not an exhaustive list — see the Borer et al (2009) paper in your readings list.

We will use the Pound Hill data collected by students in the Silwood Field Course week for understanding/illustrating some of these principles.

To start with, we need to import the `raw` data file.

⋆ Go to the bitbucket repository and navigate to the `Data` directory.
⋆ Copy the file `PoundHillData.csv` and `PoundHillMetaData.csv` files into your own R module's `Data` directory.
⋆ Now load the data in R:

```
> MyData <- as.matrix(read.csv("../Data/PoundHillData.csv",header = F,
stringsAsFactors = F))
> MyMetaData <- read.csv("../Data/PoundHillMetaData.csv",header = T,
sep=";", stringsAsFactors = F)
```

Note that:

- Loading the data `as.matrix()`, and setting The `header` and `stringsAsFactors` guarantees that the data are imported "as is" so that you can wrangle them. Otherwise `read.csv` will convert the first row to column headers, convert everything to factors, etc. Note that all the data will be converted to character class in matrix here because at least some of the entries are already character class .

- For the metadata loading, the `header` is set to true because we do have metadata headers (`FieldName` and `Description`), and I have used semicolon (`;`) as delimiter because there are commas in one of the field descriptions.
- I have avoided spaces in the columns headers (so "FieldName" instead of "Field Name") — please avoid spaces in field or column names as much a possible as R will replace each space in a column header with a dot, which may be confusing.

Tip

In `R`, you can use `F` and `T` for boolean `FALSE` and `TRUE` respectively. Try:

```
> a <- T
> a
[1] TRUE
```

We won't do anything with the metadata file in this session except inspect the information it contains.

Keep a metadata file for each unique dataset

Data wrangling really begins immediately after data collection. You may collect data of different kinds (e.g., diversity, biomass, tree girth), etc. Keep the original spreadsheet well documented using a "metadata" file that describes the data (you would hopefully have written the first version of this even before you started collecting the data!). The minimum information needed to make a metadata file useful is a description of each of the *fields* — the column or row headers under which the information is stored in your data/spreadsheet. Here is the metadata file for the Pound Hill dataset:

Field/Column Name	Description
Cultivation	Cultivation treatments applied in three months: october, may, march
Block	Treatment blocks ids: a–d
Plot	Plot ids under each treatment : 1–12
Quadrat	Sampling quadrats (25×50 cm each) per plot: Q1–Q6
Species data	Number of individuals (count) per quadrat

Ideally, you would also like to add more information about the data, such as the measurement units of each type of observation. Here, there is just one type of observation: Number of individuals of species per sample (plot), which is a count (integer, or `int` class).

Don't (over-)modify your raw data by hand

When the dataset is large (e.g., 1000's of rows), cleaning and exploring it can get tricky, and you are very likely to make many mistakes. You should record all the steps you used to process it with an R script rather than risking a manual and basically irreproducible processing. Most importantly

avoid or minimize editing your raw data file. Let's see how we can modify the data using `R`. In fact, we should now start to keep a record of what we are doing to the data. This is illustrated in a code data file available on the bitbucket repository.

- ⋆ Go to the bitbucket repository and navigate to the `Code` directory.
- ⋆ Copy the script `DataWrang.R` into your own R module's `Code` directory and open it.

Go through the script carefully line by line, and make sure you understand what's going on. Read the comments — add to them if you want. One of the examples of data modification that you must avoind doing by hand, is illustrated in the script: filling in zeros.

Convert wide format data to long format using scripts

You typically record data in the field or experiments using a "wide" format, where a subject's (e.g., habitat, plot, treatment, species etc) repeated responses or observations (e.g., species count, biomass, etc) will be in a single row, and each response in a separate column. The raw Pound Hill data were recorded in precisely this way. However, the wide format is not ideal for data analysis — instead you need the data in a "long" format, where each row is one observation point per subject. So each subject will have data in multiple rows. Any measures/variables/observations that don't change across the subjects will have the same value in all the rows.

You can switch between wide and formats using `melt()` and `dcast()` from the `reshape2` package, as well as `gather()` and spread() from the `tidyr` package, which is a newer interface to the `reshape2` package. We will use `reshape2` in `DataWrang.R`. You can find out more about `tidyr` by googling it (some cheat-sheets and tutorials are also available online)!

8.4.2 Data exploration

Once you have wrangled the Pound Hill data to long format, you are ready to go! You may want to start by examining the basic properties of the data, such as the number of tree species (41) in the dataset, number of quadrats (replicates) per plot and cultivation treatment, etc.

The first plot you can try is a histogram of abundances of species, grouped by different factors. For example, you can look at distributions of species' abundances grouped by `Cultivation`).

8.4.3 Practicals

In this practical, you will write script that draws and saves a pdf file of Fig. 8.8, 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 `R` 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 the following figure 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 the figure 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.)

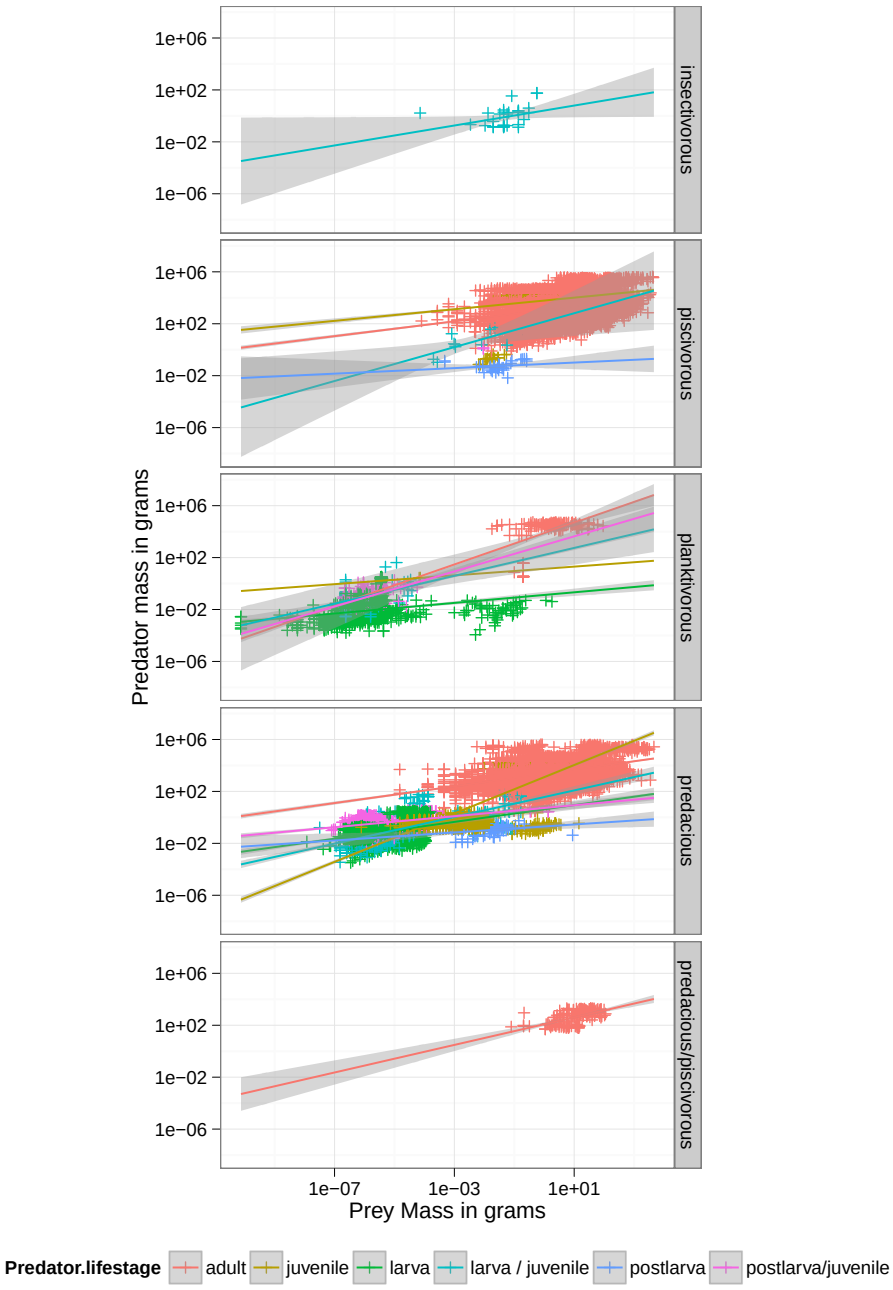


Figure 8.8: Write a script that generates this figure

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 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.
- You can also use the `plyr` function instead of looping (coming up in Chapter 9), and the `ggplot` command instead of `qplot`.

Extra Credit: Do the same as above, but the analysis this time should be separate by the dataset's `Location` field. Call it `PP_Regress_loc.R`. No need to generate plots for this (just the analysis results to a `*.csv` file), as a combination of `Type.of.feeding.interaction`, `Predator.lifestage`, and `Location` will be far to busy (faceting by three variables is one step too far)!

8.5 Practicals wrap-up

1. Review and make sure you can run all the commands, code fragments, and named scripts we have built till now and get the expected outputs.
2. Annotate/comment your code lines as much and as often as necessary using #.
3. Keep all code, data and results files organized in you R module directory

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

8.6 Readings

Check out `DataDataData!`, `Visualization` and `R` under `Readings` on the bitbucket master repository

- Brian McGill's "Ten commandments for good data management"; `https://dynamicecology.wordpress.com/2016/08/22/ten-commandments-for-good-data-management/`
- This paper covers similar ground (look in your readings directory): Borer, E. T., Seabloom, E. W., Jones, M. B., & Schildhauer, M. (2009). Some Simple Guidelines for Effective Data Management. Bulletin of the Ecological Society of America, 90(2), 205–214. `http://doi.org/10.1016/B978-1-4160-6654-5.00014-6`
- `http://www.theanalysisfactor.com/wide-and-long-data/`
- Rolandi et al. "A Brief Guide to Designing Effective Figures for the Scientific Paper", doi:10.1002/adma.201102518

- The classic Tufte `www.edwardtufte.com/tufte/books_vdqi` (btw, check out what Tufte thinks of PowerPoint!)
  Available in the Central Library, I have also added extracts and a related book in pdf on the master repository
- 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
- Effective scientific illustrations: `www.labtimes.org/labtimes/issues/lt2008/lt05/lt_2008_05_52_53.pdf`
- `https://web.archive.org/web/20120310121708/http://addictedtor.free.fr/graphiques/thumbs.php`

Chapter 9

Advanced topics in R

In this chapter, you will learn some additional topics in R to

- Make data wrangling, analyses, and simulations more efficient using vectorization and tools such as `plyr`
- Use some advanced tools for control flows and looping
- Generate random numbers for statistical simulations and looping
- Find and fix errors in R code using debugging
- Become aware of some additional tools and topics in R (accessing databases, building your own packages, etc.).

9.1 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) – this can slow code execution and operations down. Let's see an example that illustrates this point.

⋆ Type (save in `Code`) as `Vectorize1.R` the following script, and run it (it sums all elements of a matrix):

```
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)))
```

Note the `system.time` R function — it calculates how much time your code takes.

Both `SumAllElements()` and `sum()` approaches are correct, and will give you the right answer. However, the inbuilt function `sum()` is 100 times faster than the other, because it is uses vectorization that avoids the amount of looping that `SumAllElements()` uses!

Tip

In R, even if you should try to avoid loops, in practice, it is often much easier to throw in a `for` loop, and *then* "optimize" the code to avoid the loop if the running time is not satisfactory. Therefore, it won't hurt you to become really familiar with loops and looping as you learned in Chapter 7

Fortunately, R has several functions that can operate on entire vectors and matrices without requiring looping (Vectorization). That is, vectorizing a computer program means you write it such that as many operations as possible are applied to whole data structure (vectors, matrices, dataframes, lists, etc) at one go, instead of its individual elements.

Let's learn about some important R functions that allow vectorization in the following sections.

9.1.1 The `*apply` family of functions

There are a family of functions called `*apply` in R that vectorize your code for you. These functions are described in the help files (e.g. `?apply`).

For example, `apply` can be used when you want to apply a function to the rows or columns of a matrix (and higher-dimensional analogues – remember arrays!). This is not generally advisable for data frames as it will first need to coerce the data frame to a matrix first.

- ⋆ Type the following in a script file called `apply1.R`, save it to your `Code` directory, and run it:

```
## 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)
```

That was using apply on some of R's inbulit functions. You can use apply to define your own functions. Let's try it.

⋆ Type the following in a script file called `apply2.R`, save it to your `Code` directory, and run it:

```
SomeOperation <- function(v){ # (What does this function do?)
  if (sum(v) > 0){
    return (v * 100)
  }
  return (v)
}

M <- matrix(rnorm(100), 10, 10)
print (apply(M, 1, SomeOperation))
```

There are many other methods: `lapply`, `sapply`, `eapply`, etc. Each is best for a given data type. For example, `lapply` is best for R lists. Have a look at `http://stackoverflow.com/questions/3505701/r-grouping-functions-sapply-vs-lapply-vs-apply-vs-tapply-vs-by-vs-aggrega` for some guidelines.

The `tapply` function

We will look at `tapply`, which is particularly useful because it allows you to apply a function to subsets of a vector in a dataframe, with the subsets defined by some other vector in the same dataframe, usually a factor (this could be useful for your Chapter 8 pound hill data analysis, for example!).

This makes it a bit of a different member of the `*apply` family. Try this:

```
> x <- 1:20 # a vector

# A factor (of the same length) defining groups:
> y <- factor(rep(letters[1:5], each = 4))

# Add up the values in x within each subgroup defined by y:
> tapply(x, y, sum)
 a  b  c  d  e
10 26 42 58 74
```

9.1.2 Using `by`

You can also do something similar to `tapply` with the `by` function, i.e., apply a function to a dataframe using some factor to define the subsets. Try this:

```
## 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)
```

9.1.3 Using `replicate`

The `replicate` function is useful to avoid a loop for function that typically involves random number generation (more on this below). For example:

```
> print(replicate(10, runif(5)))
```

This generates a 10×5 matrix of uniformly distributed random numbers.

9.1.4 Using `plyr` and `ddply`

The `plyr` package combines the functionality of the `*apply` family, into a few handy functions. Look up `http://plyr.had.co.nz/`.

In particular, `ddply` is very useful, because for each subset of a data frame, it applies a function and then combines results into another data frame (very useful for Practical 8.4.3 in 8!). In other words, "ddply " means: take a data frame, split it up, do something to it, and return a data frame.

Look up `http://seananderson.ca/2013/12/01/plyr.html` and `https://www.r-bloggers.com/transforming-subsets-of-data-in-r-with-by-ddply-and-data-table` for examples. There you will also see a comparison of speed of `ddply` vs `by` at the latter web page— `ddply` is actually slower than other vectorized methods, as it trades-off compactness of use for some of the speed of vectorization! Indeed, overall functions in `plyr` can be slow if you are working with very large datasets that involve a lot of subsetting (analyses by many groups or grouping variables).

9.1.5 And then came `dplyr`...

So if you think this is the end of the options you have for data wrangling/vectorization, think again. There is the newest kid on the block — `dplyr` the next iteration of `plyr` — which addresses the speed issues in the latter. You will have to install it as you did `plyr` in Chapter 7: using `sudo apt get install` in Linux or `install.packages()` across all platforms. Let's have a quick look at `dplyr`:

```
require(dplyr)
attach(iris)
dplyr::tbl_df(iris) #like head(), but nicer!
dplyr::glimpse(iris) #like str(), but nicer!
utils::View(iris) #same as fix()!
dplyr::filter(iris, Sepal.Length > 7) #like subset(), but nicer!
dplyr::slice(iris, 10:15) # something new!
```

We are not going to try out `dplyr` in the `R` module in any further detail. It has many many handy functionalities. Have a look at `https://blog.rstudio.org/2014/01/17/introducing-dplyr` and this cheatsheet: `https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf` (also available in the Readings directory under `DataDataData!` in the course bitbucket repository.

Tip

The base `*apply` functions remain useful and worth knowing even if you do get into `plyr` or better still, `dplyr`

9.2 Some more control flow tools

Let's look at some more control tools, which we first learned about in Chapter 7.

9.2.1 `break`ing out of loops

Often it is useful (or necessary) to `break` out of a loop when some condition is met. Use `break` (like in pretty 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
    }
}
```

9.2.2 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`):

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

This script checks if a number is odd using the "modulo" operation and prints if it is.

Tip

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.

9.2.3 Practicals

A vectorization challenge

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:

$$N_{t+1} = N_t e^{r\left(1-\frac{N_t}{k}\right)} \tag{9.1}$$

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 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.

You will be marked on this one based upon how much faster your solution is compared to mine!

9.3 Generating Random Numbers

You will probably need to generate random numbers at some point in your journey towards becoming a proper data analyst or quantitative biologist.

R has many routines for generating random samples from various probability distributions — we have already used `runif()`, `rnorm()`. There are a number of random number distributions that you can sample or generate random numbers from:

`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)

9.3.1 "Seeding" random number generators

Before proceeding further, note that 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.

So 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).

Now, 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.

To try out how sampling works, type the following into `sample.R` and save in `Code`:

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

x <- rnorm(50) #Generate your population
doit <- function(x){
    x <- sample(x, replace = TRUE)
    if(length(unique(x)) > 30) { #only take mean if sample was sufficient
         print(paste("Mean of this sample was:", as.character(mean(x))))
         }
    }

## Run 100 iterations using vectorization:
result <- lapply(1:100, function(i) doit(x))

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

9.4 Errors and Debugging

9.4.1 "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).

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!

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`. Modify `sample.R` as follows the following into `try.R` and save in `Code` (what does this script do?):

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

x <- rnorm(50) #Generate your population
doit <- function(x){
    x <- sample(x, replace = TRUE)
    if(length(unique(x)) > 30) {#only take mean if sample was sufficient
         print(paste("Mean of this sample was:", as.character(mean(x))))
        }
    else {
        stop("Couldn't calculate mean: 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`.

9.4.2 Debugging

Once you have found an error, you would like to fix it. This is called debugging. Here are 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.

9.5 Launching/Running R in batch mode

Often, you may want to run the final analysis without opening R in interactive mode. 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, it's 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"
```

9.6 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`.

9.6.1 SQlite with R

Just like python, R can also be used to acccess, update and manage SQLite databases (this script file is in your Code directory):

```
#install the sqlite package
install.packages('sqldf')

# To load the packages
library(sqldf)

# The command below opens a connection to the database.
#If the database does not yet exist, one is created in the working directory of R.
db <- dbConnect(SQLite(), dbname='Test.sqlite')

# Now let's enter some data to the table
# Using the db connection to our database, the data are entered using SQL queries
# The next command just create the table
dbSendQuery(conn = db,
            "CREATE TABLE Consumer
       (OriginalID TEXT,
        ConKingdom TEXT,
        ConPhylum TEXT,
        ConSpecies TEXT)")

# Once the table is created, we can enter the data.
#INSERT specifies where the data is entered (here the School table).
#VALUES contains the data

 dbSendQuery(conn = db,
         "INSERT INTO Consumer
         VALUES (1, 'Animalia', 'Arthropoda', 'Chaoborus trivittatus')")
 dbSendQuery(conn = db,
         "INSERT INTO Consumer
         VALUES (2, 'Animalia', 'Arthropoda', 'Chaoborus americanus')")
 dbSendQuery(conn = db,
         "INSERT INTO Consumer
         VALUES (3, 'Animalia', 'Chordata', 'Stizostedion vitreum')")


# Once we have our table, we can query the results using:

dbGetQuery(db, "SELECT * FROM Consumer")
dbGetQuery(db, "SELECT * FROM Consumer WHERE ConPhylum='Chordata'")


# Tables can be also imported from csv files.
# As example, let's use the Biotraits dataset.
# The easiest way is to read the csv files into R as data frames.
# Then the data frames are imported into the database.

Resource <- read.csv("../Data/Resource.csv")  # Read csv files into R

# Import data frames into database
 dbWriteTable(conn = db, name = "Resource", value = Resource, row.names = FALSE)

# Check that the data have been correctly imported into the School table.
 dbListTables(db)                  # The tables in the database
 dbListFields(db,"Resource")       # The columns in a table
 dbReadTable(db, "Resource")    # The data in a table

# Before leaving RSQLite, there is a bit of tidying-up to do.
# The connection to the database is closed, and as precaution
# the three data frames are removed from RâĂŹs environment.
 dbDisconnect(db)             # Close connection
 rm(list = c("Resource"))    # Remove data frames
```

. This assumes that you are already Familiar with the Databases in python section of 6.

9.7 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 Students!

9.8 Sweave and knitr

Sweave and knitr are tools that allows you to write your Dissertation Report or some other document 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 document, which can be written using LaTeX, LyX, HTML, or Markdown. The report can be automatically updated if data or analysis change, which allows for truly reproducible research. Check out `http://www.statistik.lmu.de/~leisch/Sweave` and `http://yihui.name/knitr/`

9.9 Practicals

1. Autocorrelation in weather (this Practical may make more sense once you have done the R and Stats week where you will learn about correlation coefficients and p-values).
   - ⋆ 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 of one year significantly correlated with the next year (successive years), *across the years*. That us you will be calculating the correlation between $n-1$ pairs of years, where n is the total number of years. However, you can't use the standard p-value calculated for a correlation coefficient (using `R`'s `cor()` function – see below) because measurements of climatic variables in successive time-points in a time series (successive seconds, minutes, hours, months, years, etc.) are *not independent*.
   - ⋆ Therefore,
     (a) Compute the appropriate correlation coefficient between successive years and store it (look at the help file for `cor()`
     (b) Then repeat this calculation 10000 times by
     – randomly permuting the time series (Hint: you can use the `sample` function that we learned about in this Chapter — read the help file for this function and experiment with it),

– then computing the correlation coefficient for each randomly permuted year sequence and storing it

- ⋆ Then calculate what fraction of the correlation coefficients from step 2 were greater than that from step 1 (this is your approximate p-value).
- ⋆ *How do you interpret these results? Why? Present your results and their interpretation in a pdf document written in LaTeX(please include the the document's source code as well).*

2. Mapping (Extra Credit)

   Your project may not really need GIS, but you may still like/need to do some mapping. You 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 gold mine 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 bitbucket git repository — have a look at the database field headers and contents.
   - ⋆ What you need is latitude and longitude 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:
     (a) Loads the maps package
     (b) Loads the GPDD data
     (c) Creates a world map (use the map function, read its help file, also google examples using
     (d) maps)
     (e) Superimposes on the map all the locations from which we have data in the GPDD dataframe
     (f) 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? — include your answer as a comment at the end of your R script.*

9.10 R Module Wrap up

9.10.1 Some comments and suggestions

Thanks for enduring through the week! Learning to program in R or any other language, especially if it's your first-ever effort to learn programming, demands perseverance. Y'all have shown admirable quantities of this necessary quality. Keep going! I believe most if not all 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 Practicals will become available by 15th Nov.

9.11 Practicals wrap-up

1. Review and make sure you can run all the commands, code fragments, and named scripts we have built till now and get the expected outputs.
2. Annotate/comment your code lines as much and as often as necessary using #.
3. Keep all code, data and results files organized in the `Week3/` directory

`git add, commit` *and* `push` *all your code and data from this chapter to your git repository by Wednesday, Nov 2, 5PM.*

9.12 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 10

High Performance Computing

These instructions also apply, with suitable modifications, for R scripts.

10.1 Preparing scripts for running on the HPC

The script you will run needs a sha-bang (telling it what shell to run, usually bash), you need to allocate resources to PBS (such as walltime, number of processors, and memory[1], using the `#PBS` directive), and tell it what Python script to run. The bash script could look something like this:

```
#!/bin/bash

#lines declaring parameters to request from HPC:

## tell the batch manager to limit the walltime for the job to given hh:mm:ss
#PBS -l walltime=06:30:00

## tell the batch manager to use 1 node with 1 cpu (total 1*1 cpus) and 4000mb of ↩
    memory per node
#PBS -l select=1:ncpus=1:mem=4000mb
## *NOTE: serial jobs do not require a number of cpus*

## Name your job (optional, but can be convenient)
#PBS -N Py_test_1

## setup to get an email when scripts starts and ends (or aborts)
#PBS -m abe
## Look up man qsub for what the options a,b,e do

## Specify email address (multiple addresses can be set; look up man qsub)
#PBS -M your.email@imperial.ac.uk

# Load python as engine; default is 2.7.3 change version from 2.7.3 if
## needed (python 3 is supported)
module load python/2.7.3

# general tools
module load intel-suite
## Intel math kernel must be loaded at run time for compiling etc.

echo "Python is about to run"
```

[1] Most of the cx1 nodes have multiple cores, so there's no fixed memory assigned to each core. If you use more memory than your request on your #PBS directive, your job is likely to be terminated. If you request more memory than is available, the job will remain queued until sufficient memory is free for the job to run

```
python < $WORK/TestPyHPC/MyHPCScript.py
## tells the batch manager to execute MyHPCScript.py in
## TestPyHPC using python

# mv the output file result*
echo "Moving output files"
mv result* $WORK/TestPyHPC/output/

echo "Python has finished running"
```

Or, you can do something like this to move all files one-by-one to avoid exceeding memory allocation (`*.p` indicates that you used `pickle` to dump results):

```
for f in *.p; do
    echo "Processing $f..."
    mv $f $WORK/TestPyHPC/output/
done
```

NOTE: PBS allows you to submit jobs using a Python (instead of shell) script as well. Look up the qsub manual (`man qsub`) in the HPC terminal, or visit `https://gist.github.com/nobias/5b2373258e595e5242d5`

Your HPC enabled Python code could look like this:

```
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 02 16:20:48 2015

@author: Samraat Pawar

"""
import os # need this to get environment variables

i = int(os.getenv("PBS_ARRAY_INDEX"))

do_simulation(i)

def do_simulation():
    ...
    return ...
```

10.2 Copying scripts from your computer to the HPC server

Secure copy bash script file to `$HOME` on HPC server following `$ scp source host:destination` structure, e.g.:

```
$ scp script.sh user@login.cx1.hpc.ic.ac.uk:/home/user/whatever/script.sh
```

10.3 Running scripts on the HPC

Open a secure shell (ssh):

```
$ ssh user@login.cx1.hpc.ic.ac.uk
```

Check for available modules:

```
$ module avail
```

Your job then needs to be queued using `qsub` (PBS):

```
  $ qsub -j eo script.sh
```

where `-j eo` is an option to join both output and error into one file. Running the script will produce a job output (anything that is printed in the shell terminal (e.g. `echo`)), and an error file (related to whether the script was successful or not), in the form of {scriptname}.o{job id} and {scriptname}.e{jobid}.

The `qstat` command provides information on the job being submitted (which queue (short, medium, long), status, etc.) as well as information on all queues available (-q, -Q).

Chapter 11

The computing Miniproject

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" here you ask and answer a scientific question in biology (potentially involving multiple sub-questions/hypotheses). The miniproject will give you an opportunity to perform a "dry run" of executing your actual dissertation project.

11.1 Objectives

The main, generic question you will address is: *What mathematical model best fits an empirical dataset?*

You may think of this as testing a set of alternative hypotheses — arguably every alternative hypothesis is nothing but an alternative model to describe an observed phenomenon.

You may choose any dataset and set of alternative models, provided that it is feasible! The project must satisfy the following criteria:

1. It should employ all the biological computing tools you have learned so far: shell (bash) scripting, git, LaTeX, R, and Python. Using these tools, you will build a workflow that starts with the data and ends with a written report (in LaTeX).

2. *Atleast* two different models (hypotheses) must be fitted to the data. The models should be fitted and selected using an appropriate method (e.g., non-linear least squares for model fitting and the Akaike Information Criterion for mdoel selection, respectively). You will be given a primer on model fitting before you start on your Miniproject.

3. The project should be fully reproducible — a single Python script called `run_MiniProject.py` should glue the workflow together and run it. I should need to run just this script to get everything to work, from data processing to model fitting to plotting (e.g., in `R`) to compilation of the LaTeXwritten report.

If you are unable to find a dataset and/or problem that you would like to tackle, you may use the "TPC problem" given below.

11.2 The Report

The report should,

- be written in LaTeX using the `article` document class, in 11pt (any font will do, within reason!)
- be double-spaced, with *continuous* line numbers
- have a title, author name with affiliation and wordcount (next point) on a separate title page.
- The report body itself should have an introduction with objectives of the study, and appropriate additional sections such as methods, data, results, discussion, etc.
- must contain $\leq$4000 words *excluding the contents of the title page, but including the references* — there should be a wordcount at the beginning of the document.
- all references should be properly cited using bibtex.

For the writeup, you probably should look at the *general* (*not* word count, formatting etc) dissertation writing guidelines given in the Silwood Masters Student Guidebook.

11.3 Patching together your workflow components

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

11.4 Submission

Commit and push all your work to your bitbucket repository using a directory called `CMEEMiniProject` at the same level as the `Week1`, `Week2` etc. directories, by the Miniproject deadline given in your CMEE course guidebook.

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).

11.4.1 Marking criteria

I will be looking for the following while assessing your submission:

- A well-organized project where code, results, data, etc., are easy to locate, inspect, and use programatically
- A python or bash script called `run_MiniProject.py` or `run_MiniProject.sh` respectively, that runs the project
- A report that contains all the components indicated above in "The Report" subsection — I will be looking for some original thought and synthesis in the Intro and Discussion
- Quality of the presentation of the graphics and tables in your report, as well as any plots showing model fits to the data.

Here again, the marking criteria you may refer to is the summative marking criteria given in Appendix A titled "MARKING CRITERIA for EXAMS and ESSAYS and COURSEWORK".

Equal weightage will be given to the code+workflow and writeup components.

11.5 A Candidate Problem: Fitting TPCs

One choice you have is to use a large dataset that we can provide to address the following question: *How well does a mechanistic model based upon biochemical principles fit a dataset of thermal responses of individual fitness in plants?*. This is actually part of the biotraits database that you were introduced to in the Python and Databases Section.

This is currently a "hot" (no pun intended!) topic in biology, with both ecological and evolutionary consequences, as we discussed in the modelling 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.)!

11.5.1 The Data

The dataset is called `GrowthRespPhotoData.csv` and contains hundreds of "thermal responses" for growth, respiration and photosynthesis in plants (both aquatic and terrestrial). These were collected through lab experiments across the world, and compiled by various people over the years.

11.5.2 The Models

There are multiple models that might best describe these data.

The Schoolfield model (paper is in `Readings` directory) is a mechanistic option (11.5.2) that is based upon thermodynamics and enzyme kinetics:

$$B = \frac{B_0 e^{\frac{-E}{k}(\frac{1}{T} - \frac{1}{283.15})}}{1 + e^{\frac{E_l}{k}(\frac{1}{T_l} - \frac{1}{T})} + e^{\frac{E_h}{k}(\frac{1}{T_h} - \frac{1}{T})}} \tag{11.1}$$

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_l is the enzyme's low-temperarure de-activation energy (eV) which controls the behavior of the enzyme (and the curve) at very low temperatures, and T_l is the at which the enzyme is 50% low-temperature deactivated. E_h is the enzyme's high-temperature de-activation energy (eV) which controls the behavior of the enzyme (and the curve) at very high temperatures, and T_h is the at which the enzyme is 50% high-temperature deactivated. E is the activation energy (eV) which controls the rise of the curve up to the peak in the "normal operating range" for the enzyme (below the peak of the curve and above T_h).

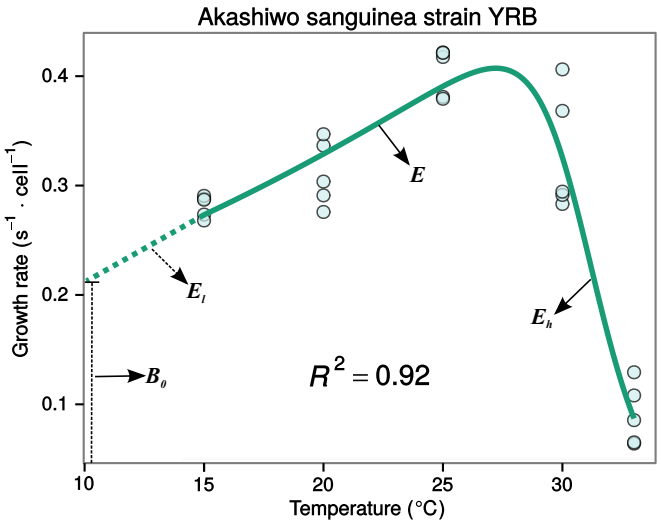


Figure 11.1: Example of a unimodal thermal response curve for a biological trait with the Schoolfield model fitted (Equation 11.1).

In many cases, a simplified Schoolfield model would be more appropriate for thermal response data, because low temperature inactivation is weak, or is undetectable in the data because low-temperature measurements were not made.

$$B = \frac{B_0 e^{\frac{-E}{k}(\frac{1}{T} - \frac{1}{283.15})}}{1 + e^{\frac{E_h}{k}(\frac{1}{T_h} - \frac{1}{T})}} \tag{11.2}$$

In other cases, a different simplified Schoolfield model would be more appropriate, because high temperature inactivation was not detectable in the data because measurements were not made at sufficiently high temperatures:

$$B = \frac{B_0 e^{\frac{-E}{k}(\frac{1}{T} - \frac{1}{283.15})}}{1 + e^{\frac{E_l}{k}(\frac{1}{T_l} - \frac{1}{T})}} \tag{11.3}$$

In addition, there are phenomenological alternatives. These include the general cubic polynomial model:

$$B = B_0 + B_1 T + B_2 T^2 + B_3 T^3 \tag{11.4}$$

with the parameters B_0, B_1, B_2 and B_3 not having any mechanistic underpinnings. Note that the cubic model has the same number of parameters as the the reduced Schoolfield model **??**. The parameter (T) of the cubic model (Equation 11.4) are in °C.

All the above parameters and equations are in SI units.

11.5.3 Fitting models to the data

You will use Nonlinear Least Squares (NLLS) to fit the alternative models above (eqns 11.1 – 11.4) to data, followed by model selection with AIC and BIC (also known as the Schwartz Criterion — read the Johnson and Omland paper).

11.5.4 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 Python or 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 (how will you get these?)
    - 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` - use `sudo` mode. In addition to the lmfit example in class, you may want to look for others online.
    - Will use the `try` construct because not all runs will converge. Recall the `try` example from R
    - The more thermal response curves you are able to fit, the better — that is part of the challenge!
    - 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. LaTeX source code that generates your report.

5. A Python script (saved in `Code`) called `run_MiniProject.py` that runs 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 how the `python lmfit` module works. Kartik 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 Sam Thompson if you need help on this.

11.6 Readings and Resources

- Levins, R. 1966 The strategy of model building in population biology. Am. Sci. 54, 421–431.
- Johnson, J. B. & Omland, K. S. 2004 Model selection in ecology and evolution. Trends Ecol. Evol. 19, 101–108.
- For the suggested fitting TPCs project: Papers in the `Temperature_response_papers` directory, but especially
  Schoolfield, R. M., Sharpe, P. J. & Magnuson, C. E. 1981 Non-linear regression of biological temperature-dependent rate models based on absolute reaction-rate theory. J. Theor. Biol. 88, 719–31.

Appendices

Appendix A

Computing Coursework Assessment Criteria

Here is the marking scheme I will use:

1. You would get 100 pts if,
   (a) All the in-class scripts[1] were in place (in the code directory in the respective week's directory) and functional when run on my computer
   (b) All the assigned practicals / problems are complete and functional, and give the right answers
   (c) The scripts are all up to the the mark in terms of internal documentation and commenting
   (d) There is a neat `readme` file detailing all the scripts and what they do week 1 onwards.
2. For every in-class script that gives a syntax error, 5 pts deducted, and for every script that gives an error because of wrong path (e.g., absolute) assignment, 2 pts deducted.
3. For every missing script or assigned practical/problem, 10 pts deducted
4. For every assigned practical/problem, 5 pts deducted for wrong answer if applicable (that is, script runs without error, but gives wrong numerical or text output).
5. For every missing `readme` file, 1 pt deducted.
6. For every extra, non-script file in `Code` directory, 0.5 pt deducted.
7. For every *valid* script file in `Code` directory lacking an appropriate extension, 0.5 pt deducted.
8. For every result of a code/script run not saved to a separate `Results` directory, 1 pt deducted.
9. For every extra-credit question completed, 2.5 pts added.

[1] *in-class scripts* are the ones that were give to you to practice with, which you only had to reproduce without error, while *assigned practicals / problems* are the assignments/problems you have been given that involve the writing of new scripts or the modification of existing ones.

From the points left after implementing the above criteria, I will exercise my judgement to deduct furtjer marks if the weekly directory structure is disorganized, the code inadequately commented or insufficiently documented, or the written components of practicals are not up to the mark.

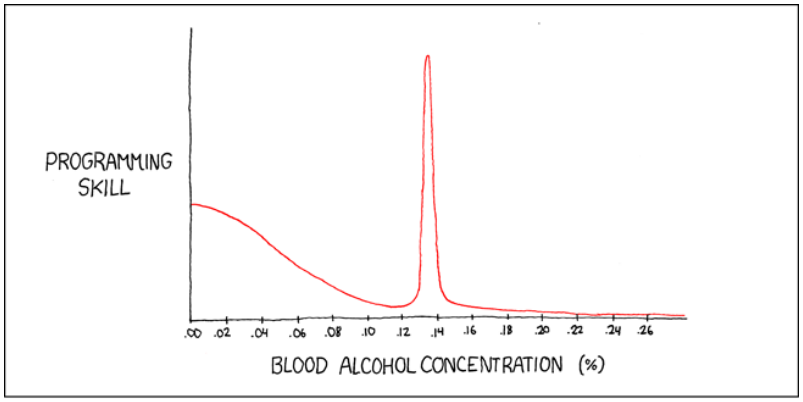
You will get feedback if these issues needed to be addressed in the final written assessment. The final marks will be based upon these weekly points and a coursework marking criteria (See below). I will up- or down-weigh the contribution of each week to the overall marks based upon the difficulty level.

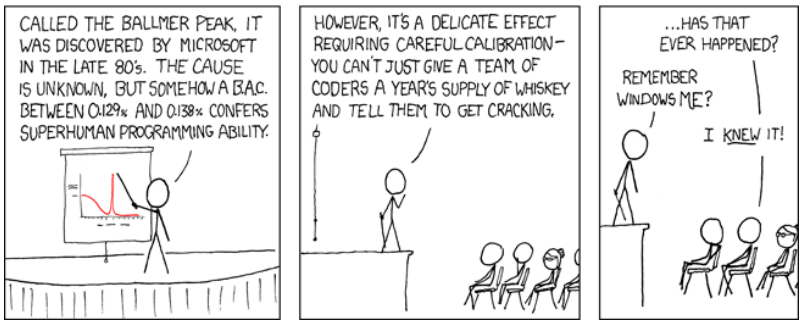
MARKING CRITERIA for EXAMS and ESSAYS and COURSEWORK

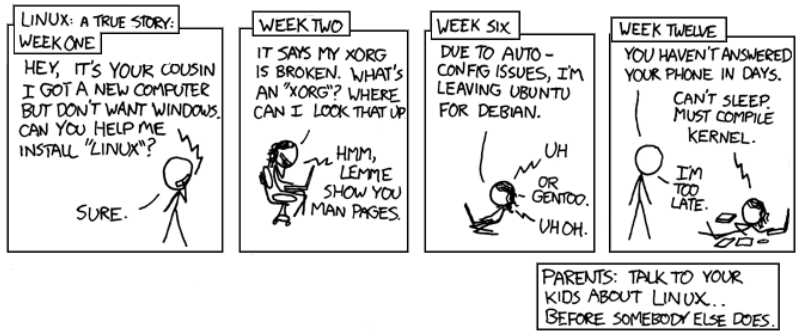
The following criteria are the basis on which the Department assesses both exam answers and coursework.

Literal Grade	Criteria (Problem type answers are marked on a semi-absolute scale)
A*	**Exceptional** Answer is an exceptionally well presented exposition of the subject, showing: (i) command of the relevant concepts and facts, (ii) a high critical or analytical ability**, (iii) originality, and (iv) evidence of substantial outside reading (where applicable). Numeric marks available 100, 95, 90, 85.
A	**Excellent** Answer is a very well presented exposition of the subject, showing many of the above features, but falling short in one or two of them. Numeric marks available 80, 76, 72.
B	**Very Good to Good** Answer (i) shows a clear grasp of the relevant concepts and facts, (ii) gives an accurate account of the relevant taught material *(as exemplified in the model answer),*and (iii) shows evidence of some outside reading *or* of critical or analytical ability**. Numeric marks available 68, 65, 62.
C	**Adequate** Answer: (i) shows a grasp of the basic concepts and facts, (ii) gives a mainly accurate account of at least half of the relevant taught material *(as exemplified in the model answer),*and (iii) does not go beyond that, *or* goes beyond that but is marred by significant errors. Numeric marks available 58, 55, 52.
F	**Unsatisfactory** Answer: 1.shows only a weak grasp of the basic concepts and facts, and is marred by major errors or brevity; numeric marks available 48, 45, 42; 2.shows a confused understanding of the question; is too inaccurate, too irrelevant, or too brief to indicate more than a vague understanding of the question; 35, 30, 25; 3.includes at most one to four sentences or facts that are correct and relevant to the question; numeric marks available 20, 15, 10, 5; 4.contains nothing correct that is relevant to the question; numeric mark 0.

** *Analytical* = assessing a hypothesis or statement by breaking it down into its elements and examining their inter-relationships and contribution to the whole; cf. *Critical* = judging a hypothesis or conclusion by examining the validity of the evidence adduced for it.







"I SPEND A LOT OF TIME ON THIS TASK. I SHOULD WRITE A PROGRAM AUTOMATING IT!"