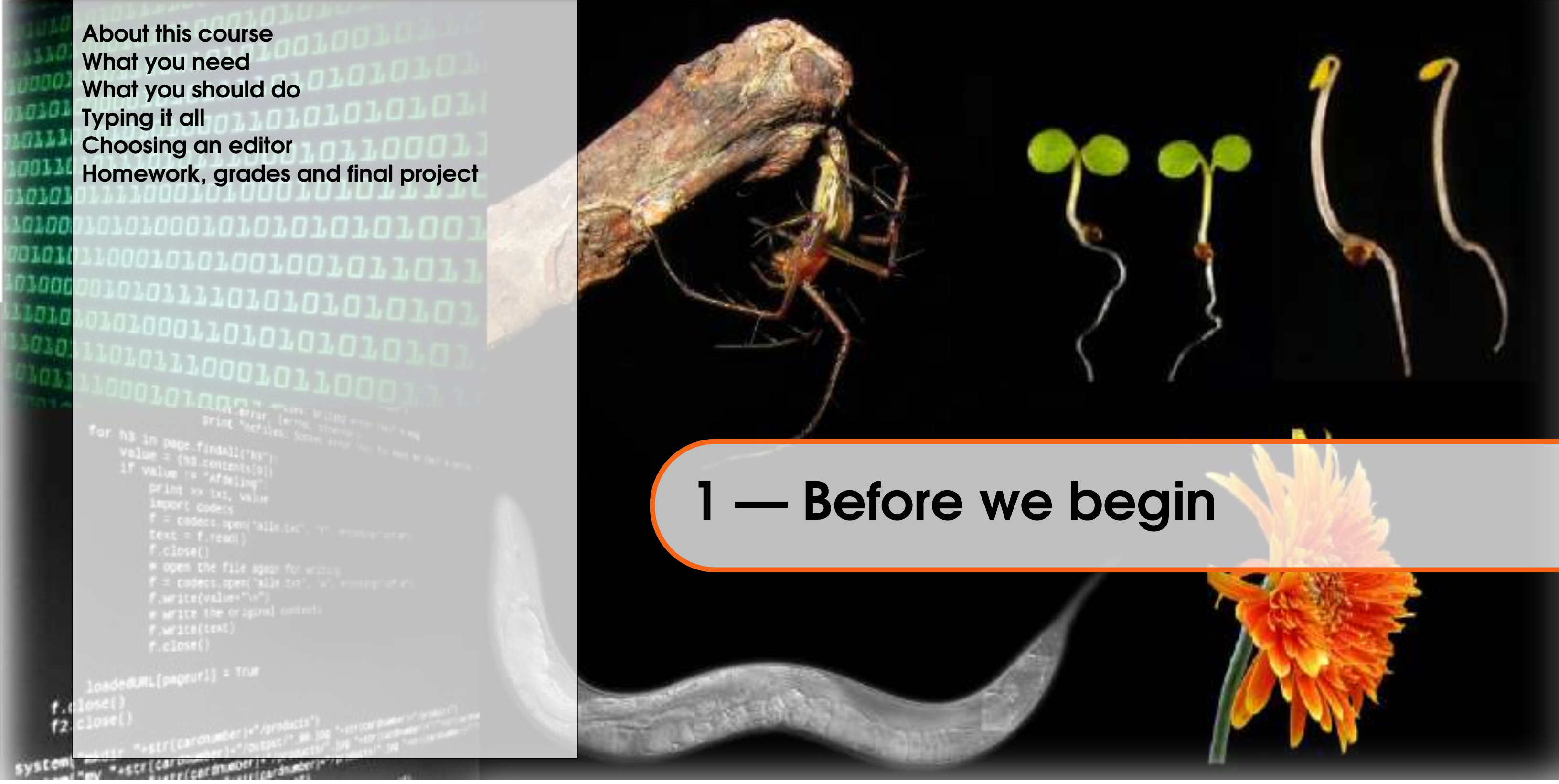
Introduction to Scientific Computing for Biologists – 2013/2014 –

Stefano Allesina

Contents



1 — Before we begin

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

1.1 About this course

This course is something I have been thinking about for a long time. Before I started my Ph.D., I worked for a year as a computer programmer for a small telephone company. This experience changed dramatically the way I approached scientific research. Learning how to program efficiently, how to organize code and data, how to automate analysis and how to collaborate with others has greatly improved my abilities, and has led me to attempt projects that would have been impossible to complete without these skills.

Throughout my career, I taught my fellow students and postdocs programming tricks, encouraged them to further their computational skills, and constantly pursued new and better ways of carrying out my research. When I started my laboratory at the University of Chicago, this interest has become a necessity. Managing many projects involving many collaborators is very difficult, if one does not organize the data, code, figures and papers in a logical and effective way.

Computing is a challenge for scientists, especially for those not trained in the so-called “hard sciences”. By definition, as scientists we are trying to do something no one has attempted before. As such, no off-the-shelf software is typically available for the analysis we want to perform. Hence the need for at least rudimentary programming skills.

Data is growing exponentially in size and quality. This data deluge requires better data organization and flow. Agencies and publishers are more and more often requiring scientists to deposit the data and the code for analysis prior to papers’ acceptance. Thus, the organization of code and data should be approached with the same effort we put in papers.

The choice of the material to cover reflects my own preferences and biases. We are going to use a Linux environment (or Mac OSX), and I will cover only the use of free software (i.e., open source software, which users can adapt and modify without charge). Throughout the text, I try to present possible alternatives, along with their advantages and disadvantages.

I will show how to automate repetitive tasks. Ideally, each project should be fully automated. This means that once we have gathered the data, the statistical analysis, the production of tables and figures, etc. should be automatic. This has three main advantages: a) repeating the same analysis for a different dataset is straightforward, b) the analysis is repeatable, c) small modifications of the workflow should be easy to implement. This third point is especially important for scientific work: reviewers will always ask you to implement slightly different approaches, and having the whole analyisis automated will save you enormous amounts of time.

What follows is a series of semi-independent chapters covering various aspects of scientific computing, from programming to data organization. The emphasis is always on the practice, rather than the theory. The examples and exercises are, whenever possible, biologically motivated.

I would greatly appreciate any feedback you might provide, e.g., What is missing? What is not clear? What should be explained in more detail? What could be removed? Also, if you have data or analysis that could suitable for the examples and exercises, please come and see me.

1.2 What you need

All that you need is a laptop running Linux (preferred) or MacOSX (acceptable) and an Internet connection. Throughout the class, I will illustrate the examples and exercises on a machine running Ubuntu (a popular version of Linux). If you have a Mac, using Mac OSX is fine: most commands will give you exactly the desired result, and I'll try to mention the subtle differences between the two.

Windows is not an option, as the required commands and approaches would be too different from those presented in class. If you have a Windows laptop, there are several options:

Dual-boot This is the preferred option. Basically, you will install Ubuntu along with Windows in two different partitions of the hard-disk. When you turn your computer on, you will be asked to choose which system to launch. To transform your laptop into a dual-boot machine Google "dual boot Ubuntu + Windows 7/XP" and follow the instructions that seem more recent and easy to follow. Many YouTube videos illustrate the procedure.

USB key You can install Ubuntu on a USB stick and launch it at startup. This solution is quite slow though, and I am not sure you can actually install software.

Virtual machine You can run Ubuntu as a normal Windows program. I have never tried this, but it seems quite straigthforward. WMware and Wubi seem quite popular.

Resuscitate an old laptop If you have an old laptop sitting around, you can probably resuscitate it and run Linux on it.

Borrow a Mac The Department of Ecology & Evolution has three Mac laptops that can be used for the class.

R Using Unix-like systems has a steep learning curve. However, the effort pays of really quickly. Try asking someone in my lab about their experience.

1.3 What you should do

Before the first class, please open a terminal and run the following commands, which will download the files containing the exercises and the homework.

To open a terminal, in Ubuntu `CTRL+ALT+T`. Then type:

```
$ cd ~
$ sudo apt-get install git
$ git clone https://bitbucket.org/AllesinaLab/introscicomp2014.git
```

In Mac, first go to `git-scm.com/downloads` and install git. Then open a terminal (in spotlight, type Terminal) and type:

```
$ cd ~
$ git clone https://bitbucket.org/AllesinaLab/introscicomp2014.git
```

1.4 Typing it all

I will provide printed notes for each chapter, containing several snippets of code. You will type it all during the class. I could have provided you with the source code in electronic format, and simply asked you to copy and paste it. However, typing the code makes you more aware of what we are doing, and certain things are better learned through your fingers than through your eyes.

1.5 Choosing an editor

You will need to edit code throughout the class. We will do so in a text editor (i.e., a program to edit text files and source code). If you already have an editor of choice, then use whatever you're comfortable with. If you don't, you have plenty of choices. The simplest to use are `gedit` (in Ubuntu) or `jedit` (Ubuntu and Mac). `Geany` and Sublime Text 2 seem also very popular.

If you want a more professional light-weight code editor, choose `vim` or `emacs`. These have much steeper learning curves, but they pay off in the long run. I use `emacs` for most of my code and paper writing, and I will use `emacs` in class.

If you want a full-blown IDE (Integrated Development Environment), which provides many advanced tools, use `Eclipse` in Ubuntu or MacOSX, or `XCode` in MacOSX. These are excellent programs, but a bit of an overkill for our class.

The internet is filled with geeks arguing over which editor is "the best". Even though such a thing exists, the advantages over the other editors are going to be pretty minor. So, pick one and stick with it!

1.6 Homework, grades and final project

Several exercises and readings conclude each chapter. You can either do these by yourself or collectively in the optional weekly session of the class. Because you are in graduate school, you should be self-motivated, and approach this homework seriously. Homework should be a great occasion to test and further your skills, rather than something you have to do because I said so. Contact me or the TA in case you have problems with the homework.

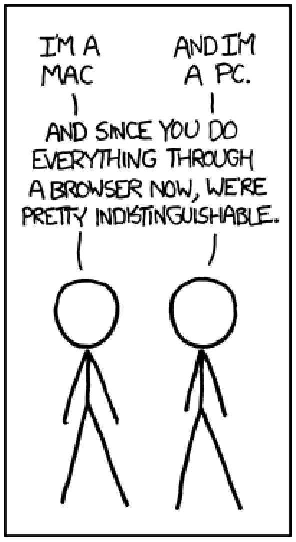
Because this is the second time I am teaching this class, I will not grade your homework or give you a final grade. The class is going to be pass/fail unless you absolutely need a grade (because of fellowships etc.).

During the Quarter, you should work on your Final Project. The idea is to take a real problem in the lab you are doing rotations, and solve it using the methods showcased in class. The problem can be anything, provided it satisfies at least three of the following:

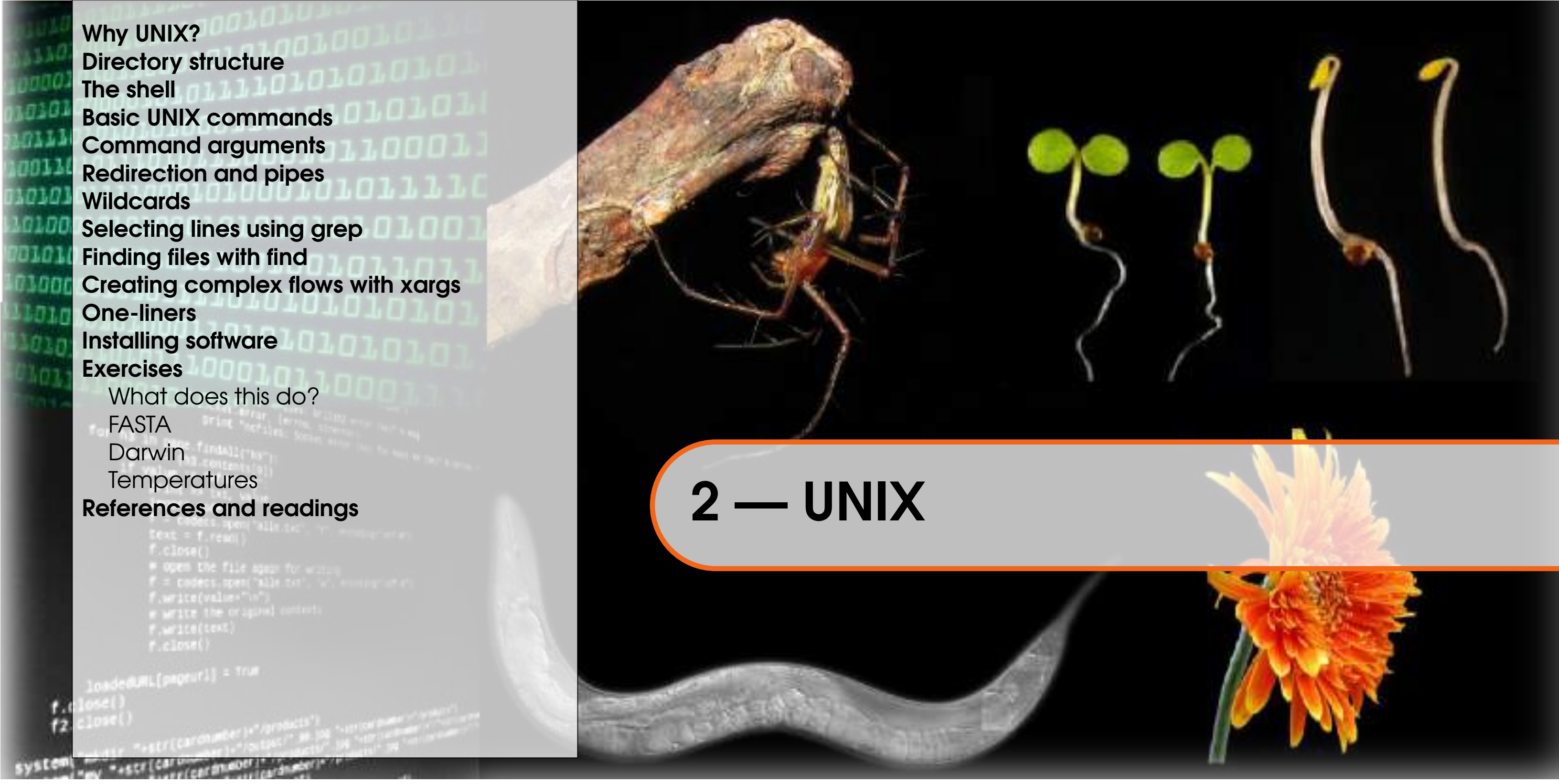
- Requires a large data set or a large number of simulations (n>5000).
- Requires programming in `python` or `R`.

- Requires running jobs on the computer cluster `beast.uchicago.edu`.
- Makes extensive use of regular expressions.
- Makes extensive use of scripting.
- Requires complex plots in `ggplot2`.

If you prefer, you can reproduce the analysis of a recent paper in your lab. If you run out of ideas, come and see me. Each project must be approved before you actually put time into it, and has to be completed by the end of the Quarter. A final report, written in LaTeX and detailing the problem, the solution and the code, must accompany the project.


I'M A MAC
AND I'M A PC.
AND SINCE YOU DO EVERYTHING THROUGH A BROWSER NOW, WE'RE PRETTY INDISTINGUISHABLE.

`http://xkcd.com/934/`



2 — UNIX

2.1 Why UNIX?

UNIX is an operating system developed in the 1970s by a group of AT&T programmers (notably Brian Kernighan and Dennis Ritchie, the fathers of the programming language `C`). It is multi-user and network-oriented by design. It is characterized by the use of plain text files for storing data and a strictly hierarchical file system.

There are many advantages to using a *nix operating system (UNIX, Linux, Mac OSX). First, UNIX is an operating system written by programmers for programmers. This means that UNIX is an ideal environment for developing your code and storing your data. Second, in UNIX hundreds of small programs are available to perform simple tasks. Third, these small programs can be stringed together efficiently: you do not need to write a "monolithic" program to perform menial tasks. Simply, string together a sequence of small programs. Fourth, text is the rule: almost anything in UNIX is a file, and a text file at that. This means that all of your projects will be portable to other machines, and can be read and written without the need for sophisticated (and expensive) proprietary software (how many times did your `.doc` or `.ppt` files not show up properly, because of a different software version?). Fifth, it is easy to automate tasks in UNIX: simply write the commands you would type in a text file and run it! Sixth, it is very stable, freely available (Linux) and robust.

There are disadvantages to UNIX too: most of the commands will be given through a "shell" (yes, the black screen seen in so many movies on hackers and computer geeks). This means a steeper learning curve. Also, some commercially available software does not run on Linux (but most does run on Mac OSX). Because you're in the driving seat, typing a few wrong commands can literally wipe out your hard disk. Because most of the software is open source, there is a great deal of documentation on-line. If you're off-line, however, it could be difficult to find answers.

Doug McIlroy summarized the philosophy behind UNIX as "Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface." Basically, UNIX is a catalog of small programs that do one thing each, but that can be stringed together to create complex flows. Most of the data is stored in flat text files, as this is the common currency among all programs. Another UNIX developer, Richard Gabriel,

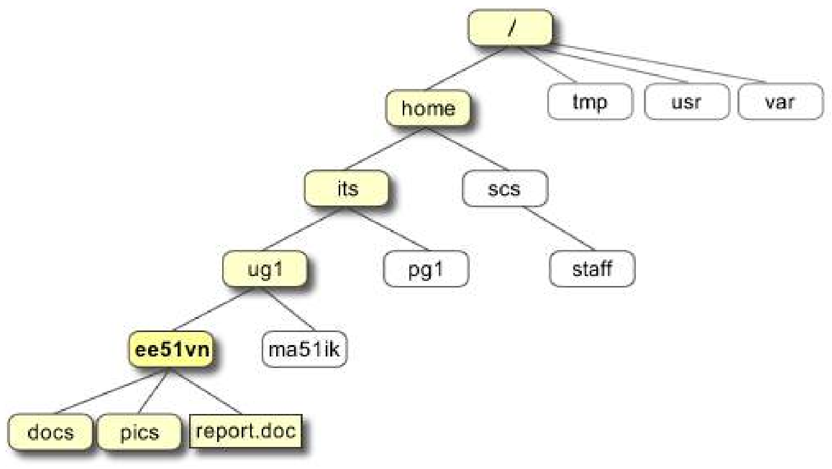


Figure 2.1: A possible UNIX directory structure, showing the "directory tree". At the top we find the / directory. In the home directory we have the users `its` and `scs`. Image taken from `www.ee.surrey.ac.uk/Teaching/Unix/unixintro.html`.

posited that in UNIX "worse is better", meaning that simplicity and portability are preferred over efficiency.

2.2 Directory structure

The directory structure of UNIX is organized hierarchically in a tree. Because we are all biologists, we can think of it as a phylogenetic tree. The common ancestor of all directories in UNIX is the / (slash) directory. From this node, several important directories branch:

`/bin`	Contains several basic programs.
`/etc`	Contains configuration files.
`/dev`	Contains the files connecting to devices (keyboard, mouse, screen, etc.).
`/home`	Contains the home directory of each user. This is where you will be typically working.
`/tmp`	Temporary files.

Each directory can contain sub-directories and files. When you navigate the system, you are in one directory, and can move deeper in the tree or upward typing some commands. Figure 2.1 illustrates these concepts.

2.3 The shell

In UNIX, you typically interface with the kernel (the core of the operating system) through a "shell": a text command processor (also called CLI, command line interface). You type commands in the shell, and the shell translates them for the kernel and shows you the results of your operation. We are going to use the `bash` shell, one of the most popular.

To launch a shell, in Ubuntu `Ctrl + Alt + t`, or open the dash (hold the `Meta` key) and type "Terminal". In Mac OSX in spotlight search for "Terminal". When you launch a shell, it automatically starts in your home directory `/home/yourname/`, also called $\sim$. For Mac users, the home directory is `/Users/yourname/`. An important trick is the use of the `Tab` key. When you press `Tab` in a shell, it will try to complete your command. Using this trick can save you a lot of typing! You can navigate the commands you 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` Clears the screen
`Ctrl + U` Clears the line before the cursor position
`Ctrl + K` Clear the line after the cursor
`Ctrl + C` Kill whatever you are running
`Ctrl + D` Exit the current shell
`Alt + F` Move cursor forward one word (in Mac, `Esc + F`)
`Alt + B` Move cursor backward one word (in Mac, `Esc + B`)

Mastering these and other keyboard shortcuts will save you a lot of time, so write these down and try to use them whenever you need to move in the shell: eventually, you will remember and automatically use most of them.

2.4 Basic UNIX commands

Here are some of the basic UNIX commands, along with a brief description.

`whoami` Display your user-name.
`pwd` Show the current directory.
`ls` List the files in the directory.
`cd [NAMEOFDIR]` Change directory. `cd ..` moves one directory up, `cd /` goes to the root, `cd ~` to your home directory.
`cp [FROM] [TO]` Copy a file.
`mv [FROM] [TO]` Move or rename a file.
`touch [FILENAME]` Create an empty file.
`echo "string to print"` Print a string.
`rm [TOREMOVE]` Remove a file.
`mkdir [DIRECT]` Create a directory.
`rmdir [DIRECT]` 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 the 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 (hit `q` to exit).[1]
`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 the current date.
`file [FILENAME]` Determine the type of a file.
`man [COMMAND]` Show help page of a command.
`passwd` Change your user password.
`chmod [FILENAME]` Change the permissions (who can read/write) of a file.

[1] Funny fact: there is a command called `more` that does the same thing, but with less flexibility. Clearly, `less` is `more`.

Now we are going to type a few examples:

```
$ cd IntroSciComp2014
$ cd UNIX
$ cd Sandbox
$ pwd
$ ls
$ touch TestFile
$ mv TestFile TestFile2
$ rm TestFile2
$ mkdir TestDirectory
$ touch TestDirectory/Test.txt
$ rmdir TestDirectory
$ rm TestDirectory/Test.txt
$ rmdir TestDirectory
```

2.5 Command arguments

Most programs in UNIX accept arguments that modify the program's behavior. For example, `ls -l` (`ls` "minus"`l`) lists the files in a longer format. Here are some of the most useful arguments.

`cp -r [DIR1] [DIR2]` Copy a directory recursively (i.e., including all the sub-directories 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 sub-directories 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).

2.6 Redirection and pipes

So far, we have sent the output of each program (e.g., `ls`) to the screen. However, it is easy to send the output to a file ("redirect"). This can be accomplished using the following commands:
> 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.

Some examples:

```
$ echo "My first line." > test.txt
$ cat test.txt
$ echo "My second line" >> test.txt
$ ls / >> ListRootDir.txt
$ cat ListRootDir.txt
```

Moreover, we can concatenate several commands using "pipes" (the symbol |). This is one of the most useful features of UNIX, and can do wonders. For example, we want to count how many files are in the root directory. We can do this by typing:

```
$ ls / | wc -l
```

We will see many examples of pipes throughout the class.

2.7 Wildcards

Many times, we want to select some particular files for manipulations. Instead of listing all the files, we can use wildcards to select several files based on their names. First, let's create a directory and populate it with some files:

```
$ 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 are going to 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).

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

2.8 Selecting lines using grep

`grep` is a command that matches strings in a file. It is based on regular expressions (much more on this later). We will now see some basic usage of `grep`. To have a test file, we are going to download a list of protected species from a website of the UN.

```
$ cd ~/IntroductionScientificComputing/UNIX/SandBox/
$ wget http://www.cep.unep.org/pubs/legislation/spawannxs.txt
$ head -n 50 spawannxs.txt
```

In Mac, the `wget` command is not available by default (it can be installed, though). Thus, I provided the in the folder `Data` as well. To copy the file, go to the `Sandbox` directory and type:

```
$ cp ../Data/spawannxs.txt .
$ head -n 50 spawannxs.txt
```

We want to see how many orchids are to be protected. To do so, type:

```
$ grep Orchidaceae spawannxs.txt
$ grep -c Orchidaceae spawannxs.txt
```

What about falcons? Typing Falco, we find:

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

Using the argument `-i` we make it 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 macaws of the genus “Ara”. We have a 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 the problem by specifying `-w` so that grep will match only full words:

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

If we want to show also the line(s) after the one matched, use `-A x`, where `x` is the 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 given pattern, use `-v`.

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

To match one of several string, use `grep "string1\|string2" file`. We can use grep on multiple files at a time! Simply, use wildcards to specify the file names.

2.9 Finding files with find

UNIX provides a command-line program for finding files, called, unsurprisingly, `find`. To test it, we are going to create some files.

```
$ mkdir TestFind
$ cd TestFind
$ mkdir -p Dir1/Dir11/Dir111
$ 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 we can use find to match particular files.

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

Using `-iname` ignores the case. You can use wildcards:

```
$ find . -name "*.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
```

Finding only directories:

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

You can use find to create complicated commands. For example, this command uses two pipes, sort and head to find the largest 10 files:

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

2.10 Creating complex flows with xargs

Many times you want to execute a command for each line returned by a program. For example, you have a list of webpages in the file `trout-web.txt`, and you'd like to download all those with ID = 2xx in the URL. Finding which ones you should download can be easily done with `grep`:

```
$ cat ../Data/trout-web.txt
$ grep "ID\=2..$" ../Data/trout-web.txt
```

The second command, as we will see later, simply takes all the strings terminating in `ID=2` followed by any two characters. We've seen that you can download a page with the command `wget`. However, the command does not accept a list: we would have to type each address separately. `xargs` solves the problem:

```
$ grep "ID\=2..$" ../Data/trout-web.txt | xargs wget
```

`xargs` is mostly used in combination with `ls`, `find` or `grep`. For example, add the extension `.backup` to all `.tex` files:

```
$ ls *.tex | xargs -I mv {} {}.backup
```

In this case, the flag `-I` states that the input lines should be taken one by one. The first set of parenthesis represent the actual file name passed to `xargs`, while the last command encodes the desired final file name.

Remove temporary files (useful when the list is very long):

```
$ ls *.tmp | xargs rm
```

Display the first two lines of files matching a pattern:

```
$ ls | grep "PaTteRn" | xargs -i head -n 2
```

Although the use of `xargs` is quite advanced, many common problems can be easily solved by using this command.

2.11 One-liners

- Append a string to an existing file

```
$ echo "my string" >> myfile
```

- Make a copy of a file. These commands do the same thing.

```
    $ cp myfile myfile_backup
    $ cp myfile{,_backup}
```

- Generate the alphabet:

```
    $ echo {a..z}
    # without spaces
    $ printf "%c" {a..z}
    # upper case
    $ echo {A..Z}
```

- All integers in a range:

```
    $ echo {1..100}
    $ seq 1 100
```

- Copy the same string several times

```
  $ echo mystring{,,,,}
```

- Print files in current directory and subdirectories in order of size

```
  $ du -a ./ | sort -n -r
```

- Erase command history

```
  $ rm ~/.bash_history
```

- Convert a number to any base

```
  $ bc <<< 'obase = 2; 256'
  $ bc <<< 'obase = 3; 256'
  $ bc <<< 'obase = 16; 256'
  $ bc <<< 'obase = 16; 255'
```

- Rename all file names to lower case

```
  $ for i in `ls -1`; do mv $i "${i,,}" ; done
```

- Extract and sort unique lines from several files

```
  $ uniq -u < (sort file1 file2 file3)
```

- Top 20 commands by use (!!!)

```
  $ history | sed 's/^ \+//;s/ _/ /' | cut -d' ' -f2- |
    awk '{ count[$0]++ } END { for (i in count) print count[i], i }'|
    sort -rn | head -20
```

2.12 Installing software

As a user, you can install software in your home directory. However, only the computer administrator (typically, you) can install software such that all users can see it. In the old days, this meant logging in as `root` (the administrator), and do whatever needed. In Ubuntu, it is sufficient to add `sudo` (Super User DO) in front of a command, and it will be run as if it were coming from `root`.

2.13 Exercises

2.13.1 What does this do?

1. `ls [A-Z]*`
2. `cp *.sh /tmp`
3. `man pwd`
4. `cd ~`
5. `mkdir -p  /test1/test2/test3`
6. `cp -r /tmp .`
7. `ls | less`
8. `head -n 24 test.txt » abc.txt`
9. `echo "aaa" > aaa.txt`
10. `ps -u sallesina`
11. `top`
12. `ls | grep .sh | xargs mv /tmp`
13. `du -sk /home/* | sort -r -n | head -10`
14. `sort -r names.txt`

2.13.2 FASTA

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. For example, the first few lines of the file `407228326.fasta` are:

```
$ head -n3 407228326.fasta
>gi|407228326|dbj|AB690564.1| Homo sapiens x Mus musculus hybrid cell line ...
AAAAAAACAAAAAGATACATATATATGATATATCTGATATATGATATATATATGATATATCTGATATATG
ATATATATATGATATATCTGATATATGATATATATATGATATATCTGATATATGATATATATATGATATA
```

First, we are going to count how many lines are in each file:

```
$ wc -l *.fasta
     28 407228326.fasta
    127 407228412.fasta
  78104 E.coli.fasta
  78259 total
```

Thus, the `E. coli` genome is much larger than the other files. What if we want to have just the sequence? We need to skip the first line. To do so, we use the command `tail -n+2` which means print everything starting from the second line.

```
$ tail -n+2 407228326.fasta
AAAAAAACAAAAAGATACATATATATGATATATCTGATATATGATATATATATGATATATCTGATATATG
ATATATATATGATATATCTGATATATGATATATATATGATATATCTGATATATGATATATATATGATATA
TCTGATATATATACATATGGTATACATGAGATACATCATATGTATATATGGTATACATGAGATACATCAT
ATGTATATATGGTATACATGAGATACATCATATGTATATATGGTATACATGAGATACATCATATGTATAC
ATGAGATACATCATATGTATACATGAGATACATCATATGTATACATGAGATACATCATATGTATACATGA
GATACATCATATGTATACATGAGATACATCATATGTATACATGAGATACATCATATGTATACATGAGATA
CATCATATGTATACATGACATACATCATATGTATACATGACATACATCATATGTATACATGACATACATA
TGTATATATGATACATATATATCATGTATATATGATACATATGTACATATATATGTATATATGATACATA
TGTATCATATATATCAGGTATATATGATACATATGTATCATATATATCAGGTATATATGATACATATGTA
TCATATATATCAGGTATATATGATACATATGTATCATATATATCAGGTATATATGATACATATGTATCAT
ATATATCAGGTATATATGATACATATGTACATATATATCAGGTATATATGTACATATGTACATATATCAG
GTATATATGTACATATGTACATATATATCAGGTATATATGTACATATGTACATATATATCAGGTATMTAT
GTACATATGWTCATATATATCAGGTATATATGTACATATGTTCATATATATCATGTATATATGAACATAT
GTACGTATATATCATGTATATATGATACATATATACGTATATATCATGTATATATGATACATGTATACGT
ATATATCATGTATATATGATACATATATACGTATATCATGTATATATGACGTACGTCATATATATCATGT
GTATATATGATGTACGTCATATATATCAGATATCATATATACACATATATCCTATATGTGTATATATGAT
ATATGTATATATGATCTATATACATATATATCATATGTTCTATGATATATATGATATATATCATATCTGA
TATATATGATATATATCATATCTGATATATATGATATATATCATATCTGATATATATGATATATATCATA
TCTGATATATATGATATATATCATATCTGATATATATGATATATATCATATCTGATATATGATATATATC
ATATCTGATATATGATATATATCATATCTGATATATATGATATATATCATATCTGATATATATGATATAT
ATCATATCTGATATATATGATATATATCATATCTGATATATATGATATATATCATATCTGATATATATGA
TATATATCATATCTGATATATATGATATATATCATATCTGATATATATGATATATATCATATCTGATATA
TATGATATATATCATATCATATATGATATATGATATATATCATATCATATATGATATATGATATATATCA
TATCATATATGATATATGATATATATGATATATCATATCATATATGATATATGATATATATGATATATCA
TATCATATAATATATGATATGATATGATATGATATATATGATATATCATATCATATAATATATGATATAT
ATGATATATATCATATATCATATATATAATATATGATATATATGATATATATCATATATCATATCATATA
TCTTTTTTTTTTTTTT
```

Now we would like to count the sequence length. To do so, we count the number of characters in the file (skipping the first line) once we have removed the character “newline” that divides the lines. To remove the newline, we use the command `tr -d`, which deletes a given character.

```
$ tail -n+2 407228326.fasta | tr -d "\n"
AAAAAAACAAAAAGATACATATATATGATATATCTGATATATGATAT....
```

Finally, using the command `wc -c` we return the number of characters:

```
$ tail -n+2 407228326.fasta | tr -d "\n" | wc -c
1836

$ tail -n+2 407228412.fasta | tr -d "\n" | wc -c
8849

$ tail -n+2 E.coli.fasta | tr -d "\n" | wc -c
4686137
```

Now we want to count the matches of a particular sequence, “ATGC” in the genome of *E. coli*. We start by removing the first line and removing newline characters. Then, we use the command `grep` to print out all occurrences of the sequence. Finally, we count the number of occurrences.

```
# This shows the behavior of grep -o

$ tail -n+2 407228326.fasta | tr -d "\n" | grep -o "ATGTACATA"
ATGTACATA
ATGTACATA
ATGTACATA
ATGTACATA
ATGTACATA
ATGTACATA
ATGTACATA

$ tail -n+2 E.coli.fasta | tr -d "\n" | grep -o "ATGC" | wc -l
21968
```

Similarly, if we want to compute the AT/GC ratio, we need the following

```
$ tail -n+2 E.coli.fasta | tr -d "\n" | grep -o [A,T] | wc -l
2306454
$ tail -n+2 E.coli.fasta | tr -d "\n" | grep -o [G,C] | wc -l
2379681
```

2.13.3 Darwin

A copy of the book “On the Origin of Species” by Charles Darwin, taken from Project Gutenberg: `http://www.gutenberg.org/files/1228/old/otoos10.txt`
is in your `UNIX/Data` directory.

- Remove the first 264 lines and the lines from 13900 onward: these lines contain Project Gutenberg data and the subject index. You can accomplish this by redirecting the output provided by the commands `head` and `tail`. Make sure to read the documentations on these commands (`man head`) Save the results in the file `darwin.txt`.
- Count the occurrences in `darwin.txt` of the terms “evolution” or “evolved”, appearing as full words and ignoring case.
- Count the occurrences of “natural selection” and “Wallace”.

Save all the commands, in the right order, in a text file.

2.13.4 Temperatures

In the directory `Data/Temperatures` you find the daily maximum and minimum temperatures recorded by several stations in 1800-1803.

- Write a one-line command that extracts the maximum temperatures of each station for Jan 1st from all the files. The result should look like this:

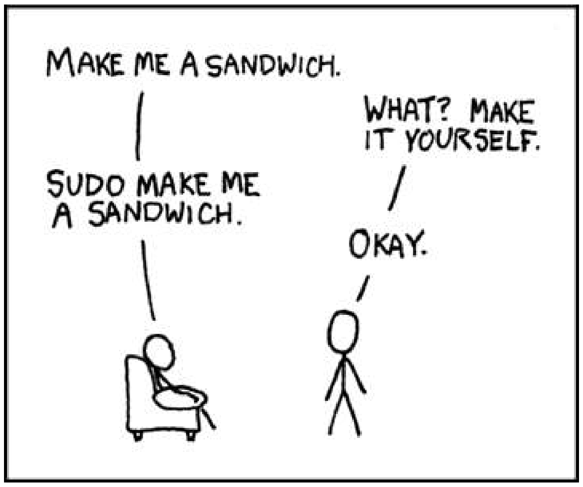
```
1800.csv:EZE00100082,18000101,TMAX,-86,,,E,
1800.csv:ITE00100554,18000101,TMAX,-75,,,E,
1801.csv:EZE00100082,18010101,TMAX,11,,,E,
1801.csv:ITE00100554,18010101,TMAX,84,,,E,
1802.csv:EZE00100082,18020101,TMAX,-20,,,E,
1802.csv:ITE00100554,18020101,TMAX,25,,,E,
1803.csv:EZE00100082,18030101,TMAX,32,,,E,
1803.csv:ITE00100554,18030101,TMAX,38,,,E,
```

To write the command, you must use "wildcards" in `grep`. For example, `180[0-3]` mathces `1800`, `1801`, `1802` and `1803`.

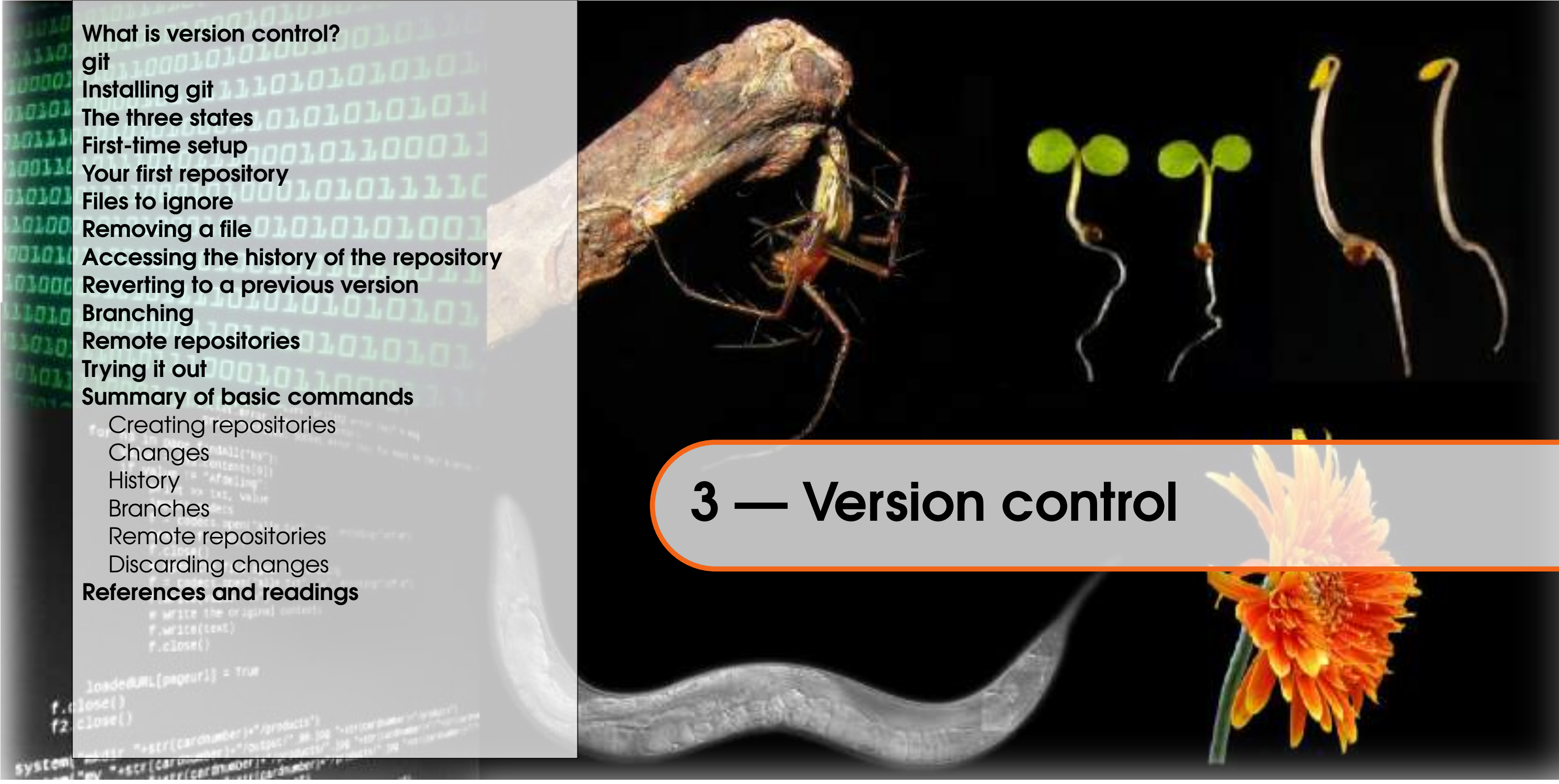
- Write a one-line command that finds the warmest temperature for a given day of 1800. The command should return `79` for February 5th and `225` for May 5th.
  To write the command, you will concatenate `grep` (to extract the max temperatures for each station on the day of interest), `cut` (to extract the right column from the result of grep), `sort` (to sort the results in ascending order), and `tail` (to return only the maximum temperature). It's going to be a quite long string of commands! Check the documentation for `cut` online or using `man cut`.

2.14 References and readings

There are very many UNIX tutorials out there. I really like the lectures you can find on
`http://software-carpentry.org/v4/shell/`
Either watch the video tutorials or read the pdfs of the presentations.
An impressive list of 687 UNIX commands can be found here (along with their man page):
`http://www.linuxdevcenter.com/cmd`
The library provides you with access to several e-books on UNIX, some specific to MacOSX or Ubuntu, and some more general. Browse the `lens.lib.uchicago.edu` website and find a good introductory book.



`http://xkcd.com/149/`



3 — Version control

3.1 What is version control?

A version control software records all the changes you make to a set of files and directories, so that you can access any previous version of the files. In this way, for each project you know who changed what, when and (possibly) why.

Because we are scientists, I will talk about research projects. Typically, you want to have a `Projects` directory in your home directory. Each directory inside the `Projects` directory is a research project, and is also a *separate repository*. In this way, it is easy to collaborate on a specific project without the need to search for the files to share: the whole project is contained in the directory (data, analysis, results and manuscripts). Also, this makes it very simple to backup important files. In my laboratory, the typical project directory contains the following sub-directories:

`Data` The data required for the project.
`Code` Code for the project.
`PDFs` Articles, manuals, etc.
`Figures` Scripts and pdfs of all figures.
`Paper` Manuscript(s).

Each directory contains other directories. For example, the `Paper` directory may contain a directory `Nature`, a directory `Science`, and a directory `PNAS`: one directory for each submission. In this way, you can access all the different versions of the manuscript, and this can greatly speed up resubmissions (which are unfortunately quite common in my laboratory). In the same way, the `Figures` directory contains a directory for each figure. Keeping projects organized is very important. In fact, there will be a time when you will have to juggle many projects at the same time, and knowing where to find each piece of the project will come extremely handy.

The main advantage of a version control system is that you can access all versions of the data, code, manuscript, etc. Hence, you can roll back to previous version when you discover a bug, find when the bug was introduced, etc. Moreover, you can back up all the versions very easily.

3.2 git

In this class, we are going to use `git`, which is probably the most popular version control system available today. Other possibilities are Concurrent Version System –`cvs` (quite old-fashioned), and Subversion – `svn` (very nice centralized system). While `svn` and `cvs` typically require you to set up a web-server, `git` does not. Also, the sites `github.com` and `bitbucket.org` will host your repositories for free. Another popular choice (very similar to `git`) is Mercurial.

R The website `bitbucket.org` will give you free private repositories for all your projects if you register with an academic email. That's good news, as you would have to pay good money for this service otherwise.

`git` was 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. This means that you do not rely as much on a centralized server, as each person working on a project has a complete copy of the whole history of the project.

We will explore the use of `git` for managing personal projects. This is not the common use of a version control system, which is thought for large-scale collaborations, but it is going to be useful to you even though we are going to explore just a few of its features.

3.3 Installing git

If you followed the instructions in Chapter 1, you should already have `git` installed in you machine. If not, in Ubuntu type this in a terminal:

```
$ sudo apt-get install git
```

In Max OS X:

```
http://code.google.com/p/git-osx-installer
```

or:

```
http://git-scm.com/downloads
```

I would also strongly suggest installing`gitk`, which is a nice visual interface for examining the status of a repository.

```
$ sudo apt-get install gitk
```

3.4 The three states

Each file in `git` can reside in one of three states: committed, modified and staged. A committed file resides in the repository, and thus is also "saved" in your local repository. A modified file is a file you have changed but have not committed to the repository. Staged files are in a limbo: they have been marked to be committed, but they haven't been committed yet.

All the files, metadata and versions are stored in the `.git` Directory. The Working Directory contains a version of the project. The files are extracted from the `.git` directory for you to modify. The Staging Area is a file containing information on what you marked to commit.

The typical work flow is:

1. Create the repository, add the files.
2. Modify the files.
3. Stage the files, adding a snapshot to the Staging Area.
4. Commit the changes, moving the files from the Staging Area to the repository.
5. Go back to 2.

The motto of git is “commit early, commit often”. Basically, each time you add a (possibly working) functionality to your project, or remove chunks of code, you want to commit the changes. Ideally, commits should be a “logical unit” that you can describe in a few words (e.g., “rewritten the code to use MCMC”). In fact, choosing a good moment to commit changes is quite important. Say for example, that your code has two bugs. Then, you should have two separate commits, one for each fixed bug. In this way, you can access the version without bugs, the one with one bug fixed and the original. You should do this because sometimes fixing a bug creates another bug, and thus it’s important to be able to access each bug-fix independently.

3.5 First-time setup

The first time you use `git`, you should do the following to customize your environment.

```
$ git config --global user.name "Stefano Allesina"
$ git config --global user.email "sallesina@uchicago.edu"
```

If you want to set up an editor to be used in git, type:

```
$ git config --global core.editor emacs
```

or whathever editor you want to use. Now check that the changes went through:

```
$ git config --list
```

3.6 Your first repository

We are going to create a new repository to test basic commands.

```
$ cd /tmp/
$ mkdir GitProjects
$ cd GitProjects
$ mkdir MyFirstRepo
$ cd MyFirstRepo
$ git init
Initialized empty Git repository in /tmp/GitProjects/MyFirstRepo/.git/
```

That’s it. Now you have your first repository. We can thus start modifying the repository. First, we create a file `README.txt` containing a brief description of the project. You can create the file using your text editor or simply `echo`.

```
$ echo "First repository for git" > README.txt

$ ls -all
```

```
total 16
drwxrwxr-x 3 sallesina sallesina 4096 2012-06-23 16:43 .
drwxrwxr-x 3 sallesina sallesina 4096 2012-06-23 16:39 ..
drwxrwxr-x 7 sallesina sallesina 4096 2012-06-23 16:44 .git
-rw-rw-r-- 1 sallesina sallesina   25 2012-06-23 16:43 README.txt
```

We have created the file, but we haven't added it to the repository yet. To do so, type:

```
$ git add README.txt

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached $<$file$>$..." to unstage)
#
# new file:   README.txt
#
```

Note that in Mac filenames are case-insensitive, while in Linux they are case sensitive. The `README` file has been staged. We can now commit the file to the repository:

```
$ git commit -am "Added README file."
master (root-commit) bdaaeb4] Added README file.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README.txt
```

As you can see, each commit reports which operations have been performed, as well as the message associated with the commit. Basically, once you add a file you make it "tracked": any modification will be recorded to be committed the next time you commit the changes to the repository. A file is not tracked until it has been added. To see what I mean, modify the file `README.txt`:

```
$ echo "Another line of text" >> README.txt

$ cat README.txt
First repository for git
Another line of text

$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add $<$file$>$..." to update what will be committed)
#   (use "git checkout -- $<$file$>$..." to discard changes)
#
# modified:   README.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

3.7 Files to ignore

Typically, you have some files you don't want to track. These might be log files, temporary files, or executables for the code. You can ignore entire classes of files by adding them to a special file called `.gitignore` to the main directory of your repository:

```
$ 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
```

3.8 Removing a file

Sometimes we want to remove a file. To do so, we use the command `git rm`. For example,

```
$ 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
```

3.9 Accessing the history of the repository

You often want to see when a particular change was introduced. You can do so by reading the log of the repository:

```
$ git log
commit 08b5c1c78c8181d4606d37594681fdcfca3149ec
Author: Stefano Allesina <sallesina@uchicago.edu>
Date:   Wed Sep 5 16:41:51 2012 -0500

    removed the file

commit 13f701775bce71998abe4dd1c48a4df8ed76c08b
Author: Stefano Allesina <sallesina@uchicago.edu>
Date:   Wed Sep 5 16:41:16 2012 -0500

    added a new file that we'll remove later

commit a228dd3d5b1921ef18c5efd926ef11ca47306ed5
Author: Stefano Allesina <sallesina@uchicago.edu>
Date:   Wed Sep 5 10:03:40 2012 -0500

    Added README file
```

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

3.10 Reverting to a previous version

Things can go horribly wrong with new the modifications, and you do not want to commit them, but rather, return to the pristine state before the changes. You can do so by:

```
$ git reset --hard
$ git commit -am "returned to previous state"
```

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 the version:

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

$ git log
commit c797824c9acbc59767a3931473aa3c53b6834aae
Author: Stefano Allesina <sallesina@uchicago.edu>
Date:   Wed Aug 22 16:59:02 2012 -0500

    modified the file

commit 8e99f2784fd0f90d21246685c36871ff2c65163e
Author: Stefano Allesina <sallesina@uchicago.edu>
Date:   Wed Aug 22 16:40:19 2012 -0500
```

```
    Added a file

$ git checkout 8e99f2
Note: checking out '8e99f2'.

You are in 'detached HEAD' state. You can look around,
make experimental changes and commit them, and you can
discard any commits you make in this state without
impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you
create, you may do so (now or later) by using -b with
the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 8e99f27... Added a file
```

Now you can look around. However, if you commit changes you create some sort of "alternate reality" (i.e., a branch). To go back to the future, type `git checkout master`.

3.11 Branching

Say you want to try something out, but you're not sure it will work well. For example, 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 the speed. You don't want to ruin your current document/code/data, though. What you need is branching. Branching creates an "alternate reality" in which you can experiment. If you like what you see, you can "merge" the branch into the project. If not, you can delete the branch and forget about it. To branch, type:

```
$ 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(-)
```

Now that you have your experimental branch, and that you did what you had to do, you can decide to merge the branch, by typing:

```
$git checkout master

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

$ cat README.txt
First Repository for git
Another Line of text
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 the changes, especially in the code, is quite painful).

3.12 Remote repositories

We have seen how to deal with local repositories, and only one user. Typically, repositories are hosted on a server, and each collaborator checks out a copy, modifies it and uploads the changes to the server. `git` will promptly report any conflicts (i.e., two users modifying the same file in different way) so that you can decide what to do and resolve the conflict.

If you have a server available, then it is quite easy to set up `git` for the whole laboratory. If you don't, the websites `github` or `bitbucket` will host your repositories for free or for fee (read carefully the restrictions).

3.13 Trying it out

Your `introscicomp2014` directory is a git repository. For your homework and exercises, try to commit changes, branch and test out the functionality of`git`. If using repositories becomes a habit, your scientific productivity will be greatly improved. I know it seems a bit weird at first, but I have to say I switched to version control in 2009 and I don't know how I managed before.

3.14 Summary of basic commands

3.14.1 Creating repositories

Creating a new repository for a given project:

```
$ cd NameOfTheDirectory
$ git init .
```

Cloning an existing repository:

```
$ git clone AddressOfTheRepository
```

3.14.2 Changes

Which files have changed?

```
$ git status
```

Stage all current changes for the next commit:

```
$ git add .
```

Commit all changes in tracked files:

```
$ git commit -am "A descriptive message"
```

3.14.3 History

Show all commits:

```
$ git log
```

Show all changes to a specific file:

```
$ git status -p MyFile
```

Who changed the file:

```
$ git blame -p MyFile
```

3.14.4 Branches

Show all existing branches:

```
$ git branch
```

N.B.: the asterisk marks which branch you're currently working on (i.e., the current `HEAD`).
Start working on a branch:

```
$ git checkout OneOfTheBranches
```

Create a branch:

```
$ git branch NameOfNewBranch
```

N.B.: The new branch will be rooted in the current `HEAD`, i.e., the branch you're working on when you create the branch.
Delete a branch:

```
$ git -d branch BranchToDelete
```

Merge a branch into the current `HEAD`:

```
$ git merge NameOfTheBranchToMerge
```

Manage conflicts:

```
$ git mergetool
```

See all changes, branches and logs using a visual interface:

```
$ gitk
```

3.14.5 Remote repositories

Download all changes in the remote repository and merge to current `HEAD`:

```
$ git pull
```

Send local changes to remote repository:

```
$ git push
```

3.14.6 Discarding changes

Discard all local changes:

```
$ git reset --hard HEAD
```

Discard changes to specific file:

```
$ git checkout HEAD NameOfTheFile
```

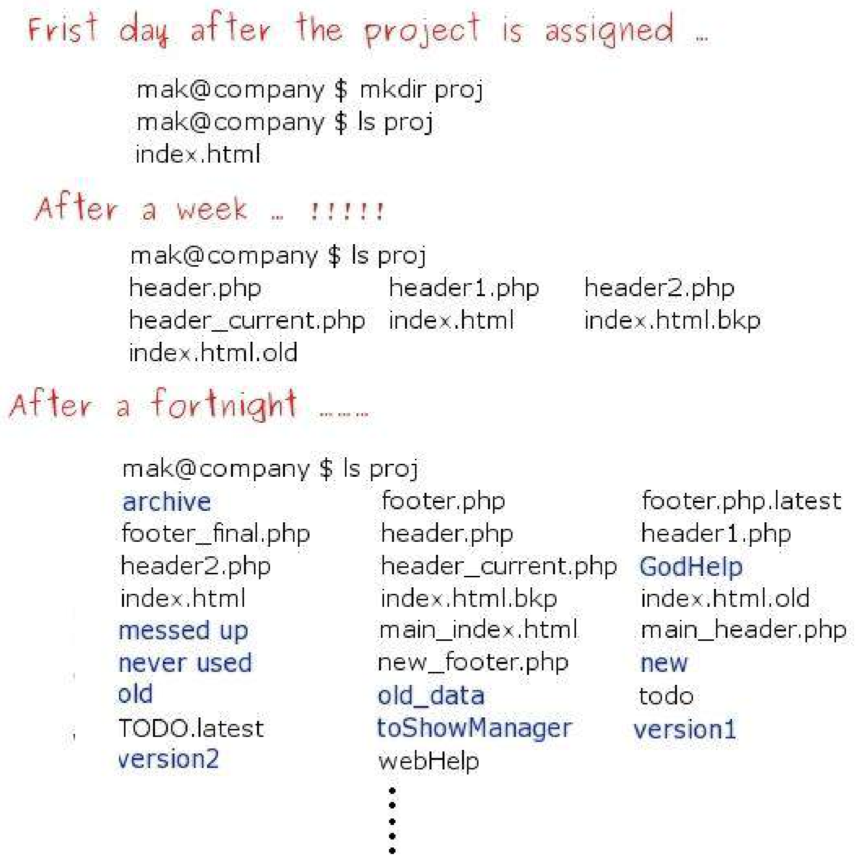
Revert a commit and produce new commit with opposite changes:

```
$ git revert NameOfCommit
```

3.15 References and readings

The website `git-scm.com/book` hosts a wonderful book on git. There are several tutorials on the internet, some of which very well done. I really like the approach taken by
`www.sbf5.com/ cduan/technical/git/`.

Other good resources:
`https://www.atlassian.com/git/tutorial`
`https://confluence.atlassian.com/display/BITBUCKET/Bitbucket+101`

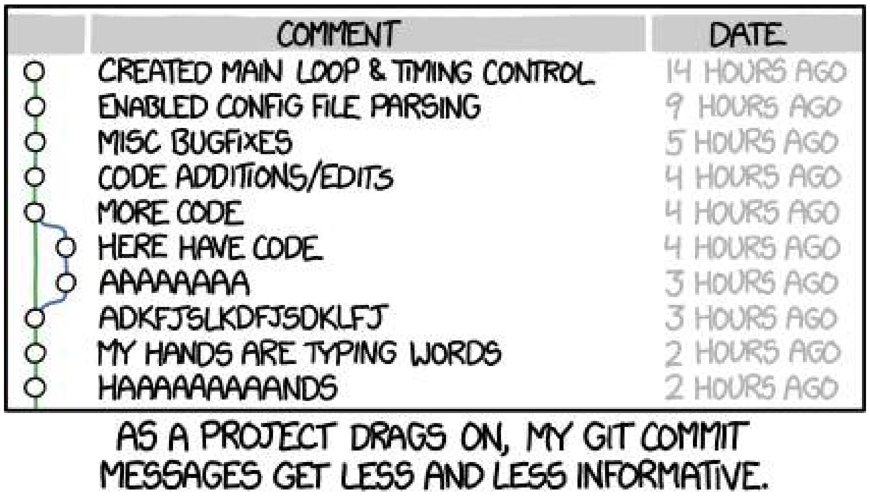


```
mak@company $ mkdir proj
mak@company $ ls proj
index.html
```

```
mak@company $ ls proj
header.php           header1.php    header2.php
header_current.php   index.html     index.html.bkp
index.html.old
```

```
mak@company $ ls proj
archive            footer.php           footer.php.latest
footer_final.php   header.php           header1.php
header2.php        header_current.php   GodHelp
index.html         index.html.bkp       index.html.old
messed up          main_index.html      main_header.php
never used         new_footer.php       new
old                old_data             todo
TODO.latest        toShowManager        version1
version2           webHelp
                      ⋮
```

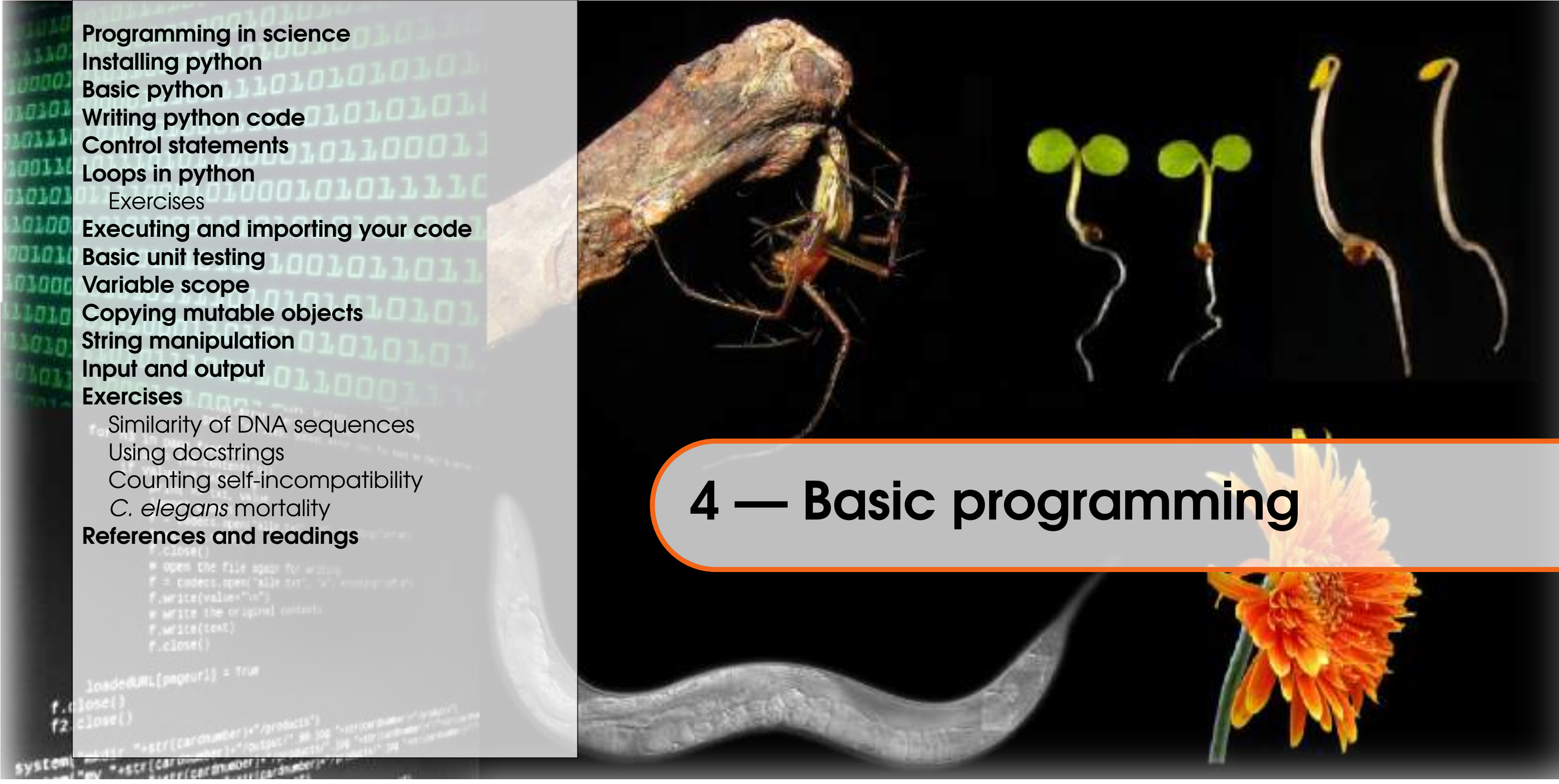
maktoons.blogspot.com/2009/06/if-dont-use-version-control-system.html



COMMENT	DATE
CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
ENABLED CONFIG FILE PARSING	9 HOURS AGO
MISC BUGFIXES	5 HOURS AGO
CODE ADDITIONS/EDITS	4 HOURS AGO
MORE CODE	4 HOURS AGO
HERE HAVE CODE	4 HOURS AGO
AAAAAAAA	3 HOURS AGO
ADKFJSLKDFJSDKLFJ	3 HOURS AGO
MY HANDS ARE TYPING WORDS	2 HOURS AGO
HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

http://xkcd.com/1296/



4 — Basic programming

4.1 Programming in science

As scientists, we are often attempting new analyses that have never been carried out before. As such, no software is typically available to perform exactly the analysis we are planning. Knowing how to program eases this problem as we can write our own software.

Several hundred programming languages are currently available (more than 2,000 historically!). Which ones should a biologist choose? Note the use of the plural: as for natural languages, knowing more than one language brings great advantages. In fact, each programming language has been developed with a certain intended audience and particular tasks in mind. For example, `COBOL`, a very old language, is still used in business applications, while `FORTRAN`, one of the very first programming languages, is a popular choice for writing numerical libraries for scientific analysis. On the other hand, programming an individual-based-model in `FORTRAN` or integrating differential equations in `COBOL` is like trying to empty lake Michigan with a spoon. The debate on which programming language is better for what application is long and very heated. For example, in 1975 E.W. Dijkstra (a great computer scientist, who invented very clever algorithms) stated that "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence." My suggestion is to learn at least three languages:

- A fast, compiled (or semi-compiled) procedural language. Procedural means that the main building blocks of each program are functions (or procedures), taking some input and returning some output. My preference is for `C`, especially because fantastic libraries for scientific computing are available (e.g., the GNU Scientific Library). Other possibilities are `FORTRAN`, `Go` (a new programming language launched by Google), etc. If you need objects, use `Objective-C`, `Java`, or `C++`.
  You can use this first language to do the heavy-lifting for your analysis (i.e., programs that require millions or billions of operations). In this class, we are not going to deal with compiled languages.
- A modern, easy-to-write, interpreted (or semi-compiled) language. My choice is `python`, as many packages are available, including many targeting the development of scientific

applications. python emphasizes readability, so that it is easy to write and to read the code (esp. compared to C). Other options are perl and Ruby.
You can use this second programming language for prototyping, double checking the results, data manipulation, and quick programming tasks. In this chapter we are going to see some of the basic functionalities of python.

- A mathematical/statistical software with programming capabilities, like R, Mathematica or MATLAB. A new language is Julia, which aims at combining the simplicity of R with the speed of C.
You can use this software to perform mathematical and statistical analysis, data manipulation and to draw figures for your papers. In this class we are going to use R as it is quite popular among ecologists and evolutionary biologists.

The website shootout.alioth.debian.org shows that if a program takes about 1 second to run in FORTRAN, it takes about 1.3s in C and 49.3s inpython. Does that mean that everybody should just use FORTRAN? No, unless they need to repeat the same operations many, many times. Otherwise, always consider the total time elapsed before the results are ready: programming + debugging + running. The total time could be much shorter in python.

In this and following chapters, I will introduce the basic syntax of python along with simple examples. I chose python because a) it's easy to teach how to program using python; b) python is a very well-thought language, with great string manipulation capabilities (e.g., for bioinformatics), web-interfaces (download data from the internet), and useful types (e.g., sets, dictionaries); c) it is freely available and well documented; and d) it is easy to adopt a good programming style while writing python code.

4.2 Installing python

To install python and ipython in Ubuntu:

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

Installation on Mac can be tricky: XCode ships its own version of python, but this could lead to problems trying to install ipython and scipy. Please install everything from their official pages:

```
Python (choose version 2.7.3):
http://www.python.org/download/

Scipy:
http://www.scipy.org/Installing_SciPy/Mac_OS_X

IPython:
http://ipython.org/download.html
(warning: install also readline)

to test that everything is working, in a terminal type:
$ ipython
In [1]: import scipy
check that there are no messages and then
CTRL-D CTRL-D to exit
```

4.3 Basic python

First, we are going to use `python` in "interactive mode" (i.e., as a shell). To do so, open a terminal and type `python`. Even better, if you installed it, type `ipython` (an enhanced shell for `python`). For example, type:

```
In [1]: 2 + 2
Out[1]: 4

In [2]: 2 * 2
Out[2]: 4

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

In [4]: 2 / 2.0
Out[4]: 1.0

In [5]: 2 / 2.
Out[5]: 1.0

In [6]: 2 > 3
Out[6]: False

In [7]: 2 >= 2
Out[7]: True

In [8]: 2 == 2
Out[8]: True

In [9]: 2 != 2
Out[9]: False

In [10]: 'hi ' + 'how are you?'
Out[10]: 'hi how are you?'

In [11]: "I'm fine, " + 'thank you'
Out[11]: "I'm fine, thank you"

In [12]: """Arnold's 6'2" tall."""
Out[12]: 'Arnold\'s 6\'2" tall.'
```

As we just saw, `python` has different types of variables, for example integers, floats (real numbers) and strings. In `python`, the following operators are defined:

`+` Addition
`-` Subtraction
`*` Multiplication

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

In any programming language, you typically manipulate variables. A variable is a box that can contain a value.

```
In [12]: x = 5

In [13]: x
Out[13]: 5

In [14]: x + 3
Out[14]: 8

In [15]: y = 8

In [16]: x + y
Out[16]: 13

In [17]: x = 'My string'

In [18]: x + ' is now longer'
Out[18]: 'My string is now longer'

In [19]: x + y
TypeError: cannot concatenate 'str' and 'int' objects
```

If sensible, we can convert a type into another:

```
In [20]: x
Out[20]: 'My string'

In [21]: y
Out[21]: 8

In [22]: x + str(y)
Out[22]: 'My string8'
```

```
In [23]: z = '88'

In [24]: x + z
Out[24]: 'My string88'

In [25]: y + int(z)
Out[25]: 96
```

In python, the type of a variable is determined when the program is running (dynamic typing), while this is not true for say C or FORTRAN.

You can initialize multiple variables at once:

```
In [1]: x, y, z = 3, 16, "abc"

In [2]: x
Out[2]: 3

In [3]: y
Out[3]: 16

In [4]: z
Out[4]: 'abc'
```

python provides several types of variables that can contain multiple values (or even variables). A list is simply a collection of ordered values:

```
# Anything starting with # is a comment

In [26]: MyList = [3,2.44,'green',True]

In [27]: MyList[1]
Out[27]: 2.44

In [28]: MyList[0] # NOTE: FIRST ELEMENT -> 0
Out[28]: 3

In [29]: MyList[3]
Out[29]: True

In [30]: MyList[4]
IndexError: list index out of range

In [31]: MyList[2] = 'blue'

In [32]: MyList
Out[32]: [3, 2.44, 'blue', True]
```

```
In [33]: MyList[0] = 'blue'

In [34]: MyList
Out[34]: ['blue', 2.44, 'blue', True]

In [35]: MyList.append('a new item')

In [36]: MyList
Out[36]: ['blue', 2.44, 'blue', True, 'a new item']

In [37]: MyList.sort()

In [38]: MyList
Out[38]: [True, 2.44, 'a new item', 'blue', 'blue']

In [39]: MyList.count('blue')
Out[39]: 2
```

This example shows that `python` is an object-oriented language: each list "inherits" some methods (the functions you call, for example `MyList.count()`) from an "ideal" list object. You can decide to use `python` as an object-oriented language (i.e., write objects) or as a procedural language (i.e., write functions) or as a mixture of the two. This flexibility is very convenient, as `python` can adapt to your project and not the other way around.

An interesting type in `python` is the dictionary: a set of values indexed by keys (in a vector, they would be indexed by numbers). Dictionaries are very useful when the variables do not have a natural order.

```
In [1]: GenomeSize={'Homo sapiens': 3200.0,
  'Escherichia coli': 4.6, 'Arabidopsis thaliana': 157.0}

In [2]: GenomeSize
Out[2]:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
 'Homo sapiens': 3200.0}

In [3]: GenomeSize['Arabidopsis thaliana']
Out[3]: 157.0

In [4]: GenomeSize['Saccharomyces cerevisiae'] = 12.1

In [5]: GenomeSize
Out[5]:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
 'Homo sapiens': 3200.0,
 'Saccharomyces cerevisiae': 12.1}

# ALREADY IN DICTIONARY!
In [6]: GenomeSize['Escherichia coli'] = 4.6

In [7]: GenomeSize
Out[7]:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
 'Homo sapiens': 3200.0,
 'Saccharomyces cerevisiae': 12.1}

In [8]: GenomeSize['Homo sapiens'] = 3201.1

In [9]: GenomeSize
Out[9]:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
 'Homo sapiens': 3201.1,
 'Saccharomyces cerevisiae': 12.1}
```

Note that if we add a new key, it will be added to the dictionary, while if we add a key that it is already present, nothing will happen. This makes it very useful when we are reading data that can contain duplicates.

Another interesting data type is the tuple. tuples contain sequences. For example, the connections in a food web could be seen as a list of tuples. Suppose we have three species, a, b and c. a is eaten by b and c, b is consumed by c and c is cannibalistic.

We can represent each feeding relation as a 2-tuple. For example,`('a','b')` means that b eats a.

```
In [12]: FoodWeb=[('a','b'),('a','c'),('b','c'),('c','c')]

In [13]: FoodWeb[0]
Out[13]: ('a', 'b')

In [14]: FoodWeb[0][0]
Out[14]: 'a'

# Note that tuples are "immutable"
In [15]: FoodWeb[0][0] = "bbb"
TypeError: 'tuple' object does not support item assignment

In [16]: FoodWeb[0] = ("bbb","ccc")

In [17]: FoodWeb[0]
Out[17]: ('bbb', 'ccc')
```

In this case, a list of tuples is more appropriate that a list of lists, as a list implicitly means that the order does not matter (and in fact, we can sort the list etc.), while in a tuple, the order matters.

If you are dealing with sets, you can transform a list into a set: a list with no duplicate elements, and with the possibility of union, intersection and difference operations.

```
In [1]: a = [5,6,7,7,7,8,9,9]

In [2]: b = set(a)

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

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

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

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

In [7]: b ^ c
Out[7]: set([3, 4, 7, 8, 9])
```

The operations are: Union | (or); Intersection & (and); symmetric difference (elements in set b but not in c and in c but not in b), ^; and so forth.

To summarize, if you have elements indexed by numbers use lists, if they are indexed by keys, use dictionaries, if they are ordered sequences, use tuples and if they are sets, use sets.

Try concatenating objects using +:

```
In [1]: a = [1, 2, 3]

In [2]: b = [4, 5]

In [3]: a + b
Out[3]: [1, 2, 3, 4, 5]

In [4]: a = (1, 2)

In [5]: b = (4, 6)

In [6]: a + b
Out[6]: (1, 2, 4, 6)

In [7]: z1 = {1: 'AAA', 2: 'BBB'}

In [8]: z2 = {3: 'CCC', 4: 'DDD'}
In [9]: z1 + z2
-------------------------------------------
TypeError  Traceback (most recent call last)

----> 1 z1 + z2

TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Similarly, try using `*`:

```
In [1]: "a" * 3
Out[1]: 'aaa'

In [2]: (1, 2, 4) * 2
Out[2]: (1, 2, 4, 1, 2, 4)

In [3]: [1, 3, 5] * 4
Out[3]: [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
```

4.4 Writing python code

For the rest of the chapter, we will write the code in an editor and test it in `python`. Before we do so, however, we can take a slight detour on best programming practices. We all learn programming calling variables `MickeyMouse`, `tmp3`, `RealPart`, etc. Moreover, in the excitement of wanting to see the results, we make undocumented changes to the code, rewrite functions or use some "hacks". Learning a new language is a great occasion to leave these bad habits behind, and start with best practices. Although initially painful, in a while it will become a second nature, saving us hundreds of hours in the long run, and making our code easier to read, share and debug. As such, I try to adhere to the PEP 8 `python` guideline (`python.org/dev/peps/pep-0008/`) and to present well-commented, easy to read `python` code. The main points are the following:

- Wrap lines so that they are less than 80 characters long. You can use parentheses () or signal that the line continues using a "backslash" \.
- Use 4 spaces for indentation, no tabs.
- Separate functions using a blank line.
- When possible, write comments on separate lines.
- Use docstrings to document **how to use the code**, and comments to explain why and how the code works.
- Naming conventions:
  - `_internal_global_variable`
  - `a_variable`
  - `SOME_CONSTANT`
  - `a_function`
  - Never call a variable `l` or `O` or `o`.
- Use spaces around operators and after commas:
  `a = func(x, y) + other(3, 4)`.

We start from a boilerplate for our code:

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

"""Descrption of this program

   you can use several lines"""

__author__ = 'Stefano Allesina (sallesina@uchicago.edu)'
__version__ = '0.0.1'

# imports
import sys
# constants
# functions

def main(argv):
    print 'This is a boilerplate'
    return 0

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

The first line (with the "shebang" `#!`) tells the computer where to look for `python`. The triple quotes start a "docstring", i.e., a comment that will be recognized by an automatic documentation program (more on that later). Then, the file contains information on the author and the version (note that each starts and ends with two underscore, signaling that these are "internal" variables. Never choose names starting with two underscores for your variables!). This is good in case you want to share your code, or to remember the state of a file (e.g., a version of 1.X.X would be a working, complete version). We then import the package `sys`, containing functions to interface our program with the operating system. This is followed by the main function. Note that all the content of the function is indented using 4 spaces. This is very important! In fact, that's how `python` determines which lines belong to which function. The same goes for the code under the `if` statement. The last few lines are a bit esoteric. The main idea is to set up the code such that when the program is executed from the command line, then the function `main` will be called.

Each file should be named using lowercase text divided by underscores if necessary. Go to `python/Sandbox`. Type the file and save it as `boilerplate.py`. In a terminal, type:

```
$ python boilerplate.py
$ python -i boilerplate.py
```

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

4.5 Control statements

A program is a sequence of operations. We can control the flow of operations using control statements. We are going to explore some of these commands in this more complex program:

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

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

__author__ = 'Stefano Allesina (sallesina@uchicago.edu)'
__version__ = '0.0.1'

import sys

def even_or_odd(x=0):
    """Find whether a number x is even or odd."""
    if x % 2 == 0:
        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:
        largest = 4
    elif x % 3 == 0:
        largest = 3
    elif x % 2 == 0:
        largest = 2
    else:
        return "No divisor found for %d!" % x
    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):
        if x % i == 0:
          print "%d is not a prime: %d is a divisor" % (x, i)
          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):
    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)
```

To begin typing the program, copy the boilerplate:

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

And open it with an editor. We are going to discuss the following:

- What is a function? A function is a piece of code that takes an (optional) input and returns an (optional) output obtained after performing some operations.
- Default values. By setting `x=0` we say that in case `x` is not specified, it should take value 0.
- Return values. Each function can return a value or a variable.
- Docstrings. The comments flanked by triple quotes are called docstrings, and they are considered a part of the running code (normal comments, on the other hand, are stripped). Hence, you can access your docstrings at run time.
- The conditional `if`. It is the easiest way to produce a branch in the code. `elif` means “else, if”, while `else` is called when all other conditions are not met.
- Printing formatted text. `print "%d %s %f %e" % (20,"30",0.0003,0.00003)` results in `20 30 0.000300 3.000000e-05`.
- `range()`. Returns a sequence of integers: `range(3)` → `0 1 2`, while `range(1,3)` → `1 2`.

4.6 Loops in python

There are two main ways of looping in `python`: a `for` loop repeats operations stepping through a list of values; a `while` loop cycles through the code until a condition is met.

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

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

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

print total

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

b = True
while b:
    print "infinite loop"
```

4.6.1 Exercises

A few exercises to make sure you understood these concepts:

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

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

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

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

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


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

def foo2(x, y):
    if x > y:
        return x
    return y

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

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

# This is a recursive function
# meaning that the function calls itself
```

```python
# read about it at
# en.wikipedia.org/wiki/Recursion_(computer_science)
def foo5(x):
    if x == 1:
        return 1
    return x * foo5(x - 1)
```

4.7 Executing and importing your code

If you want to execute a `python` script file from within a `python` shell, you can call

```
>>> execfile("filetoload.py")
```

In `ipython`, type `%run filetoload.py`. Most of the time, however, you will write a collection of functions in a `.py` file, and use them in other scripts. The file containing the functions is called a *module*. There are different ways of *importing* a module:

- `import my_module`. Import the module `my_module`. The functions in the module can be accessed as `my_module.one_of_my_functions()`.
- `from my_module import my_function`. In this way, you import only the function `my_function` contained in module `my_module`. The function can be accessed as if it were part of the main file: `my_function()`.
- `import my_module as mm`. Import the module `my_module` and call it `mm`. This is convenient only when the name of the module is very long. The functions in the module can be accessed as `mm.one_of_my_functions()`.
- `from my_module import *`. Import all the functions in module `my_module`. You should never use this.

The main problem to avoid is *namespace pollution*: i.e., by importing functions and variables you overwrite current functions and variables that have the same name. Using the first type of import avoids the problem entirely. For the same reason, use an initial _ in front of the global variables. This avoids importing their names when loading the package. If possible, avoid global variables entirely.

4.8 Basic unit testing

In a complex programming task, most of the time is spent debugging, not writing the code. Unit Testing is a way to write code that prevents common mistakes, and encourages writing solid, reliable and well-documented code. The idea is to write *independent* tests for the *smallest units* of code, i.e., whenever you write a function, you write a small piece of code to test it. Testing is then done automatically, so that whenever you modify the code, you can make sure all tests are still returning the right value.

We will use the simplest testing tool provided by python: `doctest`. Basically, simple tests for each function are embedded in the docstring describing the function. Besides testing the code, this also provides a great way of documenting the functions.

Let's copy the file `control_flow.py` into the file `test_control_flow.py`, and edit the first function.

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

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

__author__ = 'Stefano Allesina (sallesina@uchicago.edu)'
__version__ = '0.0.1'

import sys

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!'

    note that 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!'

    """
    if x % 2 == 0:
        return "%d is Even!" % x
    return "%d is Odd!" % x

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

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

To run the tests, in a terminal type:
`python -m doctest -v test_control_flow.py` You should get the following:

```
$ python -m doctest -v test_control_flow.py
```

```
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
2 items had no tests:
    test_control_flow
    test_control_flow.main
1 items passed all tests:
   4 tests in test_control_flow.even_or_odd
4 tests in 3 items.
4 passed and 0 failed.
```

For more complex testing, see the documentation of `doctest`, the package `nose` and the package `unittest`.

4.9 Variable scope

With *scope* we mean which variables are visible at a given point of the code. Global variables are visible inside and outside of functions. Local variables, on the other hand, are only accessible inside a function.

```python
## Try this first

_a_global = 10

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


a_function()
print "Outside the function", _a_global

## Now try this

_a_global = 10

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

a_function()
print "Outside the function", _a_global

## IF POSSIBLE, AVOID GLOBALS ALTOGETHER!
```

4.10 Copying mutable objects

You have to be especially careful when copying objects. In fact, when you type `b = a`, you are just creating a new “tag” (b) for the value already tagged by a. When dealing with immutable types (int, float, tuples), this is not a problem, however, copying lists can be tricky.

```
# First, try this:
a = [1, 2, 3]
b = a
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
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]]
```

4.11 String manipulation

python has great methods to manipulate strings. The most powerful methods are based on regular expressions, which we will introduce later on. The simplest functions are the following:

```
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'
```

4.12 Input and output

```
##############################
# USER INPUT
##############################
# To capture input from user:
input_user = raw_input()
print input_user

# Note that it is always a string:
x = raw_input("Enter an integer: ")
print "The square of %d is %d" % (x, x**2)

# We need to interpret it:
x = int(raw_input("Enter an integer: "))
print "The square of %d is %d" % (x, x**2)

##############################
# FILE INPUT
##############################
# Open a file for reading
f = open('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('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('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('testp.p','wb') ## note the b: accept binary files
pickle.dump(my_dictionary, f)
f.close()

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

print another_dictionary
```

The csv package allows to easily manipulate comma separated files:

```
import csv

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

csvread = csv.reader(f)
for row in csvread:
    print row
```

```python
    print "The species is", row[0]

f.close()

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

4.13 Exercises

4.13.1 Similarity of DNA sequences

We want to take two sequences of DNA, and align them such that they are as similar as possible. To do so, we start with the longest string as references, and try to position the shorter string in all possible positions. For each position, we count a "score" (i.e., number of bases matched perfectly over the number of bases tried).

First, we analyze the following code:

```python
# These are the two sequences to match
seq2 = "ATCGCCGGATTACGGG"
seq1 = "CAATTCGGAT"

# assign the longest sequence
# to s1, and the shortest to s2
# l1 is the length of the longest,
# l2 that of the shortest

l1 = len(seq1)
l2 = len(seq2)
if l1 >= l2:
    s1 = seq1
    s2 = seq2
else:
    s1 = seq2
    s2 = seq1
    l1, l2 = l2, l1 # swap the two lengths

```

```
# function that computes a score
# by returning the number of matches
# starting from arbitrary startpoint
def calculate_score(s1, s2, l1, l2, startpoint):
    # startpoint is the point at which we want to start
    matched = "" # contains string for alignement
    score = 0
    for i in range(l2):
        if (i + startpoint) < l1:
            # if it's matching the character
            if s1[i + startpoint] == s2[i]:
                matched = matched + "*"
                score = score + 1
            else:
                matched = matched + "-"

    # build some formatted output
    print "." * startpoint + matched
    print "." * startpoint + s2
    print s1
    print score
    print ""

    return score


calculate_score(s1, s2, l1, l2, 0)
calculate_score(s1, s2, l1, l2, 1)
calculate_score(s1, s2, l1, l2, 5)

# now try to find the best match (highest score)
my_best_align = None
my_best_score = -1

for i in range(l1):
    z = calculate_score(s1, s2, l1, l2, i)
    if z > my_best_score:
        my_best_align = "." * i + s2
        my_best_score = z

print my_best_align
print s1
print "Best score:", my_best_score
```

Now, modify the function such that, in case of the same number of matches, returns the one in which the highest proportion of pairs are matched. That is, if alignement 1 has 5 matches out of 9 bases, while alignement 2 has 5 matches out of 6 bases, alignement 2 is to be preferred.

4.13.2 Using docstrings

Write docstring tests for all the functions in `control_flow.py`.

4.13.3 Counting self-incompatibility

The directory `python/Data/GoldbergEtAl` contains the data from the recent Goldberg *et al.* (Science 2010). In the csv file there is a list of 356 plant species and a Status describing whether the species is self-incompatible (0), self-compatible (1) or more complex situations (2-5).

- Write a program that counts how many species are in each category of Status.
- Write another program that builds a table specifying how many species are in each Status for each genus (note that genus and species name are separated by an underscore in the first column).

4.13.4 *C. elegans* mortality

Read the paper by Morran *et al.* "Running with the Red Queen: Host-Parasite Coevolution Selects for Biparental Sex" (Science 333: 216–218, 2011). In the directory `python/Data/MorranEtAl` you find the data necessary to replicate Fig 2.

Take the data and compute the average mortality rate for the groups of organisms labeled from *a* to *r* in the graph. The README.txt file contains the description of the data. For example to compute the mean mortality rate for case *a* you would average among all replicates having i) Ancestor *marcescens* (Serratia = A), ii) obligately selfing worms (Mating type = COX), and iii) Ancestral populations (Worm Gen = An).

4.14 References and readings

- The Zen of python. Open a python shell and type `import this`.
- Code like a Pythonista: Idiomatic Python. Google it.
- Also good: the Google Python Style Guide.
- Browse the python tutorial `docs.python.org/tutorial/`.
- Chapters 7-13 of "Practical Computing for Biologists" by Haddock & Dunn teach how to use python for biological problems.
- `software-carpentry.org` has great lectures on python.
- `docs.python.org` contains the official documentation, which is great and very detailed.
- "Python Testing: Beginner's Guide", by D. Arbuckle is available online through the library.
- "Python cookbook", by A. Martelli *et al.* contains hundreds of solved problems (available online through the library).

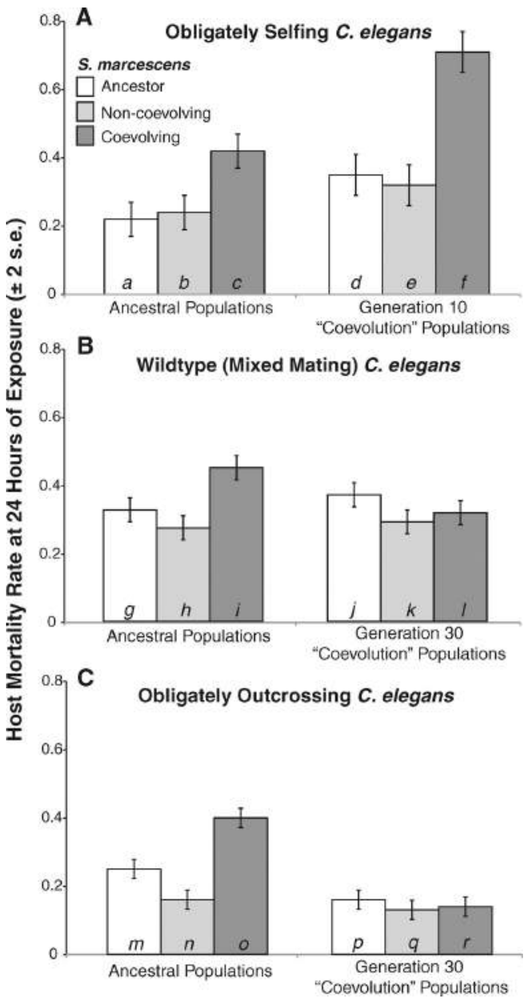
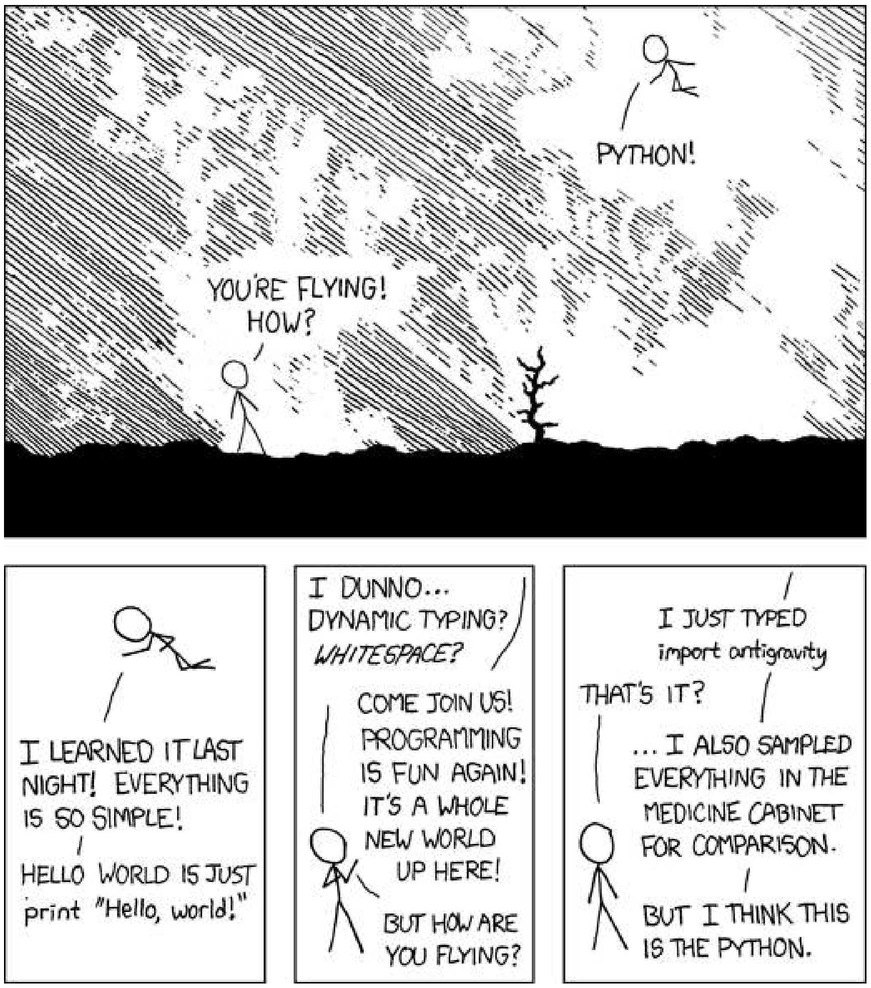
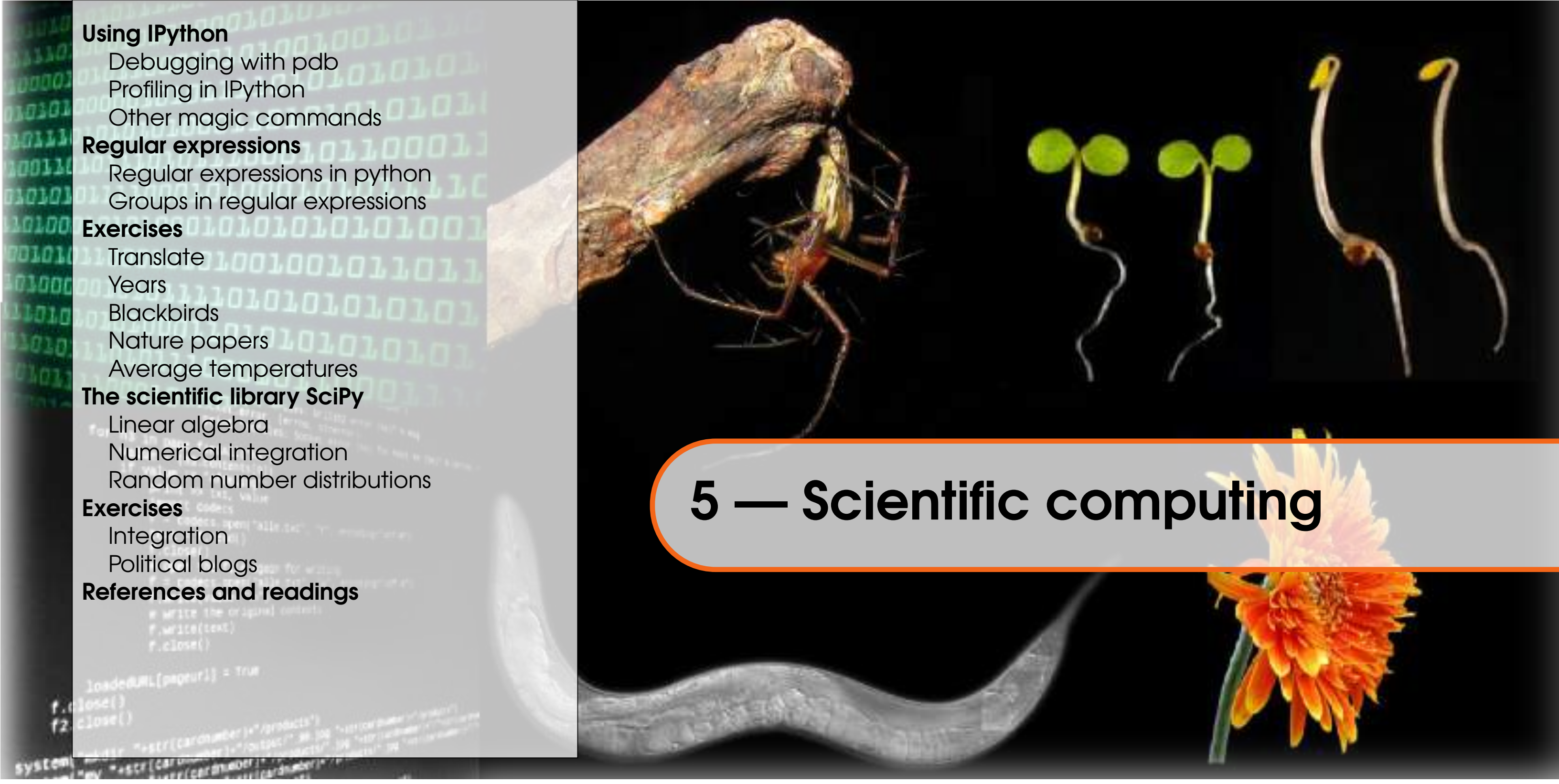


Figure 4.1: Morran *et al.* Figure 2. From their paper: "Coevolutionary dynamics of hosts and pathogens. We exposed hosts evolved under the coevolution treatment and their ancestral populations (before coevolution) to three pathogen populations: (i) an ancestor strain (ancestral to all S. marcescens used in this study), (ii) a noncoevolving strain (evolved without selection), and (iii) their respective coevolving strain (coevolving with the host population). We evaluated host mortality after 24 hours of exposure to the pathogens and present the means across the replicate host populations. (A) Three obligately selfing C. elegans populations persisted beyond 10 host generations in the coevolution treatment. These populations were assayed before extinction. (B) All five wild-type C. elegans populations in the coevolution treatment and their ancestors were assayed at the endpoint of the experiment (30 host generations). (C) All five obligately outcrossing C. elegans populations in the coevolution treatment and their ancestors were assayed at the endpoint of the experiment."



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

http://xkcd.com/353/



5 — Scientific computing

5.1 Using IPython

We're going to explore some of the nicest features of IPython. Mastering these features will make you a better and faster programmer.

First, try to TAB everything! Variable names, directories, etc. will be automatically completed, or you will see a list of the possibilities. Also, if you have an object (say a list) and hit TAB after a typing a dot, you can see the object's functions and attributes.

```
In [1]: alist = ['a', 'b', 'c']

In [2]: alist.
alist.append   alist.extend   alist.insert   alist.remove
alist.sort     alist.count    alist.index    alist.pop
alist.reverse

In [2]: adict = {'mickey': 100, 'mouse': 3.14}

In [3]: adict.
adict.clear       adict.items       adict.pop
adict.viewitems   adict.copy        adict.iteritems
adict.popitem     adict.viewkeys    adict.fromkeys
adict.iterkeys    adict.setdefault  adict.viewvalues
adict.get         adict.itervalues  adict.update
adict.has_key     adict.keys        adict.values
```

Another useful feature is the question mark:

```
In [3]: ?adict
Type:       dict
```

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

IPython features several "magic commands". These commands start with the percent sign %. The most important, when developing your code, is `%run myscript.py`. The command reads from the disk the python file `myscript.py`, loads the namespace and applies any change you might have made. Thus, when you're writing the code, you can test its behavior piece by piece. Moreover, if you type `%run -t myscript.py` you can measure the time it takes to run the script.

5.1.1 Debugging with pdb

pdb is the python debugger. If you call the magic command %pdb, the debugger is turned on. At any error in the code (technically, and uncaught exception), the debugger will start, letting you print variables values, examine the code and see what went wrong. Using a debugger is the best way to deal with bugs. Please do not litter your code with `print` statements trying to zero-in the bug.

For example, let's write a simple function that contains a division by zero:

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

createabug(25)
```

Save the file in the `python/Sandbox`. Now let's magic-run it!

```
In [1]: %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
```

Typically, we'd like to stop the code just before the error, and check why things went wrong. In this case, it is fairly obvious, but this is not the typical case. The debugger lets us see what each variable was doing when the bug happened. Let's turn the debugger on:

```
In [18]: %pdb
Automatic pdb calling has been turned ON

In [19]: %run debugme.py
[lots of text]
ZeroDivisionError: float division by zero
> createabug()
      3     z = 0.
----> 4     y = y/z
      5     return y

ipdb>
```

Note that now we're in the debugger shell, and not in IPython. To move within the debugger, we can perform the following operations:

`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.
`r` continue until the end of the function.

To continue our example:

```
ipdb> p x
25
ipdb> p y
390625
ipdb> p z
0.0
ipdb> p y/z
*** ZeroDivisionError: ZeroDivisionError
('float division by zero',)
ipdb> list
      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 [21]: %pdb
Automatic pdb calling has been turned OFF
```

Often, you want to launch the debugger at a given point of the code (to make sure everything is going well). Simply put this snippet of code wherever you want to start your debugging session, and then call the `%run` magic function.

```
import pdb; pdb.set_trace()
```

Similarly, running the code with the flag `%run -d` automatically starts a debugging session from the first line of your code. We don't have time for a complete session on debugging (maybe in a follow-up class?), but if you are serious about programming, please start using a debugger!

5.1.2 Profiling in IPython

Premature optimization is the root of all evil – Donald Knuth

If you're programming in python, speed is not your main concern. Do not let speed drive the design of your code. If you have a clean design, you can speed things up by for example rewriting your code (or parts of your code) in C. However, there are times in which knowing where your code spends most of its time is useful. Maybe "optimizing" just a few functions can yield much better performances. The key is to know where your code is spending its time, and this operation is called "profiling". IPython lets you profile your code by simply typing the magic command `%run -p`. Let's write a simple (and quite stupid) program:

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

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

```

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

some_function(1000)
```

Now running the magic command we find:

```
52 function calls in 5.940 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    4.117    4.117    5.934    5.934 profileme.py:1(a_useless_function) <======
     2    1.819    0.910    1.819    0.910 {range} <== *****
     1    0.004    0.004    0.005    0.005 profileme.py:8(a_less_useless_function) <=
     1    0.000    0.000    5.940    5.940 {execfile}
     1    0.000    0.000    5.940    5.940 profileme.py:15(some_function) <======
```

When you have to iterate over a large number of values, use xrange instead of range. The difference is that xrange does not create all the values before iteration, but rather creates them one at a time.

```
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)
```

```
50 function calls in 3.211 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    3.207    3.207    3.207    3.207 profileme2.py:1(a_useless_function)
     1    0.003    0.003    0.003    0.003 profileme2.py:8(a_less_useless_function)
     1    0.000    0.000    3.211    3.211 {execfile}
     1    0.000    0.000    3.211    3.211 interactiveshell.py:2270(safe_execfile)
     1    0.000    0.000    3.210    3.210 profileme2.py:15(some_function)
```

5.1.3 Other magic commands

`%who` Shows current namespace (all variables, modules and functions)
`%whos` Also display the type of each variable. Typing `%whos function` only displays functions etc.
`%pwd` Print working directory.
`%history` Print recent commands.
`%cpaste` Paste indented code into IPython. This is very useful when you want to paste just part of a function (that could have extra indentation).

Let's try the `%cpaste` function. First, create a file with the following code:

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

Now launch IPython and type:

```
In [1]: x = 11

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

In [3]: %cpaste
Pasting code; enter '--' alone on the line to stop
or use Ctrl-D.
:    for i in range(x):
:        print x
:--
11
11
11
11
11
11
11
11
11
```

```
11
11
```

5.2 Regular expressions

Regular expressions are tools to find a match of a particular pattern in a string of text. They can be used to search for particular patterns (e.g., DNA motifs in sequence data), navigate through sets of files, parse text files, extract information from `html` and `xml` files.

We are going to explore regular expressions in python, but regular expression packages are available for most programming languages. Also, by the end of this section it should be clear that the program `grep` we saw in the first chapter is in fact using regular expressions (grep: global regular expression print).

5.2.1 Regular expressions in python

The regular expression functions in python are in the module `re`. Thus, on the top of our program we should write `import re`. Typically, we want to build a scaffolding for the text we want to match. To do so, we will use particular "place-holders". The main ones are:

`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
`\w`	match a "word character" (alphanumeric + underscore)
`\d`	match a numeric character
`[atgc]`	match any character listed: a, t, c, g
`at|gc`	match at or gc
`[^atgc]`	any character not listed: any character but a, t, g, c
`?`	match zero or one time
`*`	match zero or more times
`+`	match one or more times
`{n}`	match exactly n times
`{n,}`	match at least n times
`{n,m}`	match at least n but not more than m times
`^`	match the beginning of a line
`$`	match the end of a line

In python, there are several regular expression functions. The simplest one is `re.search`, which search for a match of a given pattern in the string. It returns `None` if the pattern is not found, and a *match object* if the pattern is found.

For example, try typing:

```python
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 at 0x3223098>

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

# now search for the pattern
# "followed by some letters"
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
```

If we just want to know whether a pattern is matched, we can use an `if`:

```
str = 'an example'

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

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

Note that we always put `r` in front of our regular expression string! This is very important, as it says to python to consider the string in its "raw" form.

```
import re

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

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

```
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', 'once upon a time')
match.group() # 'once upon a '
## NOTE THAT *, ? and + are all "greedy":
## They try to match as much text as possible

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

match = re.search(r'\d*\.?\d*','1432.75+60.22i')
match.group() # '1432.75'

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

match = re.search(r'[ACGT]+', "the motif ATTCGT")
match.group() # 'ATTCGT'
```

5.2.2 Groups in regular expressions

One powerful feature of regular expression is the possibility of defining groups (i.e., pieces of the string we want to use). Simply surround the part of interest with round parentheses.

```
import re
"""
Goal:
scrape emails from the file
../Data/email_addresses_e_and_e.txt
containing the email addresses of
all the E&E faculty.

Note the following:
Multiple dots in the domain name:
justin.borevitz(at)anu(dot)edu(dot)au
reinitz(at)galton(dot)uchicago(dot)edu
```

```
Hyphen in the local part:
j-coyne(at)uchicago(dot)edu
"""

# read the file with email addresses
f = open('../Data/email_addresses_e_and_e.txt')
my_file_email = f.read()
f.close()

# substitute (at) with @
my_text = my_file_email.replace('(at)', '@')

# substitute (dot) with .
my_text = my_text.replace('(dot)', '.')

# now build a regular expression to grab
# emails
my_reg = r'email:\s?([A-Za-z\.\-_0-9]*@[A-Za-z\.\-_0-9]*\.\w{2,3})'
emails = re.findall(my_reg, my_text)

# A little more difficult: save the last names
# Note that each is followed by Ph.D.

my_reg2 = r'([A-Za-z-\']+),\sPh\.D\.'
lastnames = re.findall(my_reg2, my_text)

# now print them together
for i in range(len(emails)):
    print lastnames[i], emails[i]

# Exercise: now add the phone number!
```

Some more on the grouping:

```
str = "Stefano Allesina, sallesina@uchicago.edu,
Ecology & Evolution"

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

match..group()
'Stefano Allesina, sallesina@uchicago.edu, Ecology
& Evolution'

match.group(0)
'Stefano Allesina, sallesina@uchicago.edu, Ecology
& Evolution'
```

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

match.group(0)
'Stefano Allesina, sallesina@uchicago.edu, Ecology
& Evolution'

match.group(1)
'Stefano Allesina'

match.group(2)
'sallesina@uchicago.edu'

match.group(3)
'Ecology & Evolution'
```

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!).

5.3 Exercises

5.3.1 Translate

Translate the following regular expressions into regular English!

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

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

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

5.3.2 Years

Write a regular expression to match a date in format YYYYMMDD, making sure that only seemingly valid dates match (i.e., year greater than 1900, first digit in month either 0 or 1, first digit in day ≤ 3).

5.3.3 Blackbirds

Complete the code `blackbirds.py` that you find in the sandbox.

```python
import re

# Read the file
f = open('../Data/blackbirds.txt', 'r')
text = f.read()
f.close()

# remove \t\n and put a space in
text = text.replace('\t',' ')
text = text.replace('\n',' ')

# note that there are "strange characters" (these are accents and
# non-ascii symbols) because we don't care for them, let's transform
# to ASCII
text = text.decode('ascii', 'ignore')

## Now write a regular expression that captures
## the Kingdom, Philum and Species name for each species.

my_reg = ??????

# such that re.findall(my_reg, text) should return
#[(u'Animalia', u'Chordata', u'Euphagus carolinus'),
# (u'Animalia', u'Chordata', u'Euphagus cyanocephalus'),
# (u'Animalia', u'Chordata', u'Turdus boulboul'),
# (u'Animalia', u'Chordata', u'Agelaius assimilis')]
```

5.3.4 Nature papers

In the data folder you find a csv file with the 1,334 Letters and Articles published by Nature in 2011. Use the `csv` and `re` modules to count the number of authors for each paper. The minimum should be 1, the maximum 373, the median 6 and the mean about 10. Use `re.compile` to compile your regular expression before using (this will speed up performance). Use `re.findall` to find all authors of a paper. Remember to `decode` the authors list before applying the regular expression (so you don't have to deal with accents etc.)

5.3.5 Average temperatures

Find the average temperature in January from 1995 for each city in the folder `Data/AvgTemperature`.

A template of the code is in the `Sandbox`:

```python
import re


## Goal: find the average temperature
## in January for all the cities in
## Data/AvgTemperature

# The records look like:
# 1 1 1995 58.3\r\n <- used for new line

# First: build a regular expression that
# extracts the day, year and average temperature

myreg = re.compile(r'') # <- write the regular expression!

# Now let's test it on the Rome temperatures
filename = "../Data/AvgTemperature/IYROME.txt"
f = open(filename, 'r')
mytext = f.read()
f.close()

print re.findall(myreg, mytext)
# This should give you a list containing
# tuples:
# ('1', '1995', '58.3'),
# ('2', '1995', '46.5'),
# ('3', '1995', '41.6'),
# ....
# Now elaborate the list to get the average
# for each year

# finally, repeat for each city
```

5.4 The scientific library SciPy

SciPy provides many useful routines for scientific computing. The main functions deal with integration (integrals or differential equations), Fourier transforms, interpolation, linear algebra, special functions (such as Incomplete Gamma, Bessel functions and many others), and statistical functions. Numpy is a package that provides functions for numerical linear algebra. It has been embedded in SciPy and the two now use the same syntax. We are going to explore some of the main features.

5.4.1 Linear algebra

The main class for vectors and matrices is called `array`. However, there is also a class `matrix` that provides matrices and vectors for linear algebra (quite confusing!, we are going to use arrays only).

```
In [1]: import scipy

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

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

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

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

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

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

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

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

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


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

```
x.repeat        x.strides
x.clip          x.dump          x.mean
x.reshape       x.sum
x.compress      x.dumps         x.min
x.resize        x.swapaxes

In [11]: x.tolist()
Out[11]: [0.0, 1.0, 2.0, 3.0, 4.0]

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

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

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

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

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

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

In [20]: scipy.ones((4,2))
Out[20]:
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])

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

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

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

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

In [25]: m.fill(16)

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

We can perform the usual operations on arrays:

```
In [1]: import scipy

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

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

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

# NOTE: Rows first

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

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

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

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

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

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

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

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

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

To make full use of the linear algebra capabilities of scipy, we have to import the subpackage `linalg`.

```
In [1]: import scipy
In [2]: import scipy.linalg

In [3]: m = scipy.array([[4, 3], [8, 7]])
In [4]: b = scipy.array([1, 2])

In [5]: scipy.linalg.inv(m) # INVERSE
Out[5]:
array([[ 1.75, -0.75],
       [-2.  ,  1.  ]])

In [6]: scipy.linalg.det(m) # DETERMINANT
Out[6]: 4.0

In [7]: scipy.linalg.eig(m) # EIGENVALUES/VECTORS
Out[7]:
(array([  0.37652462+0.j,  10.62347538+0.j]),
 array([[-0.63772689, -0.41258634],
       [ 0.77026256, -0.9109185 ]]))

In [9]: scipy.linalg.eigvals(m) # ONLY VALUES
Out[9]: array([  0.37652462+0.j,  10.62347538+0.j])

In [10]: (scipy.linalg.eigvals(m)[0]).real # REAL PART
Out[10]: 0.37652461702020101

In [15]: scipy.linalg.norm(m) # FROBENIUS NORM
Out[15]: 11.74734012447073

In [17]: scipy.linalg.cholesky(m) # CHOLESKY DECOMP.
Out[17]:
array([[ 2.        ,  1.5       ],
       [ 0.        ,  2.17944947]])

In [18]: scipy.linalg.svd(m) # SINGULAR VALUES
Out[18]:
(array([[-0.42499684, -0.90519483],
       [-0.90519483,  0.42499684]]),
 array([ 11.74240011,   0.34064586]),
 array([[-0.76147516, -0.64819409],
       [-0.64819409,  0.76147516]]))

# Solve linear systems
```

```
In [33]: scipy.linalg.solve(m, b) ## m X x = b
Out[33]: array([ 0.25,  0.  ])
In [35]: scipy.dot(m, scipy.linalg.solve(m, b)) == b
Out[35]: array([ True,  True], dtype=bool)

# Building special matrices
# Block diagonal
In [19]: A = [[1, 2], [3, 4]]
In [20]: B = [[5, 6, 7], [8, 9, 10], [11, 12 ,13]]
In [22]: scipy.linalg.block_diag(A, B)
Out[22]:
array([[ 1,  2,  0,  0,  0],
       [ 3,  4,  0,  0,  0],
       [ 0,  0,  5,  6,  7],
       [ 0,  0,  8,  9, 10],
       [ 0,  0, 11, 12, 13]])

# Leslie matrix
In [23]: Fecundity = [0.1, 0.3, 7.2]
In [24]: Transition = [0.1, 0.2]
In [25]: scipy.linalg.leslie(Fecundity, Transition)
Out[25]:
array([[ 0.1,  0.3,  7.2],
       [ 0.1,  0. ,  0. ],
       [ 0. ,  0.2,  0. ]])

# Triangular matrix
In [30]: scipy.linalg.tri(6, dtype = float)
Out[30]:
array([[ 1.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.]])
```

5.4.2 Numerical integration

Say we want to integrate numerically:

$$\int_0^\infty \frac{8}{x^2+16}dx = \pi \tag{5.1}$$

we can perform the integration numerically using the function quad:

```
In [20]: import scipy.integrate

In [21]: def integrand(x):
    return 8./(x**2 + 16.)
   ....:

In [22]: scipy.integrate.quad(integrand, 0, scipy.inf)
Out[22]: (3.1415926535897936, 2.8707080344128833e-09)
```

The first number is the result, the second the estimated error.

Now, let's integrate a differential equation. We take the simple Lotka-Volterra system:

$$\begin{cases} \frac{dx}{dt} = x(1-x) - axy \\ \frac{dx}{dt} = -dy + axy \end{cases}$$

Which yields a single, globally stable equilibrium ($x^* = d/a$, $y^* = (a-d)/a^2$).

```
import scipy
import scipy.integrate

def dX_dt(Z,t):
    # Z contains [x, y]
    x = Z[0]
    y = Z[1]
    dxdt = x * (1. - x) - a * x * y
    dydt = - d * y + a * x * y
    return scipy.array([dxdt, dydt])

# the parameters
a = 0.5
d = 0.3

# now define time
# integrate from 0 to 100
# using 1000 points
t = scipy.linspace(0, 100, 1000)

# initial conditions
x0 = 0.2
y0 = 0.7
Z0 = [x0, y0]
Z, infodict = scipy.integrate.odeint(dX_dt, Z0, t, full_output=True)
print infodict['message']
print 'Final state', Z[-1]
# repeat with different initial conditions
x0 = 1.2
y0 = 9.7
Z0 = [x0, y0]
Z, infodict = scipy.integrate.odeint(dX_dt, Z0, t, full_output=True)
```

```
print infodict['message']
print 'Final state', Z[-1]
```

5.4.3 Random number distributions

For simulations, you often need to draw random numbers from specific distributions. Scipy lets you choose among many distributions with discrete or continuous support.

```
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])
```

5.5 Exercises

5.5.1 Integration

Integrate numerically:

$$\int_0^1 x^{10}(1-x)^2 dx = \frac{10!2!}{13!} \approx 0.0011655 \tag{5.2}$$

Starting from $x = 0.1$ and $x = 120$, and for time in $(0,500)$, integrate:

$$\frac{dx}{dt} = 0.1x(9-x) \tag{5.3}$$

which should converge to 9.

5.5.2 Political blogs

This exercise is quite difficult: start it soon!

This "political exercise" was prepared for election day in 2012. In the directory `Data/PoliticalBlogs2004` you find a file describing the hyper-links between political blogs during the election campaign of 2004. The file is in graphml format. Basically, each blog is identified by an id:

```
node [
   id 10
   label "ackackack.com"
   value 0
   source "BlogCatalog"
 ]
node [
   id 11
   label "adamtalib.blogspot.com"
   value 0
   source "Blogarama"
 ]
```

First, using regular expressions, create a dictionary mapping ids to names of blogs (e.g., `11: adamtalib.blogspot.com`). Once all the blogs have been introduced, the information on the links between blogs is encoded. For example,

```
 edge [
   source 440
   target 55
 ]
```

means that a link goes from the blog with id 440 to the blog with id 55. Use regular expressions to build a list containing all the connections.

Which blogs are the most influential? Find the PageRank of each blog in the network by implementing the power method and applying it to the political blog network. A good introduction

to PageRank is here:
`http://nlp.stanford.edu/IR-book/pdf/21link.pdf`

Use the "teleport probability" $\alpha = 0.05$, and implement the power method to compute the dominant eigenvector. Also, try to get the dominant eigenvector using the `linalg` function.

5.6 References and readings

For IPython, check out:
`http://ipython.org/ipython-doc/stable/interactive/tips.html`
`http://wiki.ipython.org/Cookbook`

Google code has a short class on regular expressions in python:
`http://code.google.com/edu/languages/ google-python-class/regular-expressions.html`

A beautiful quote from Jamie Zawinski:

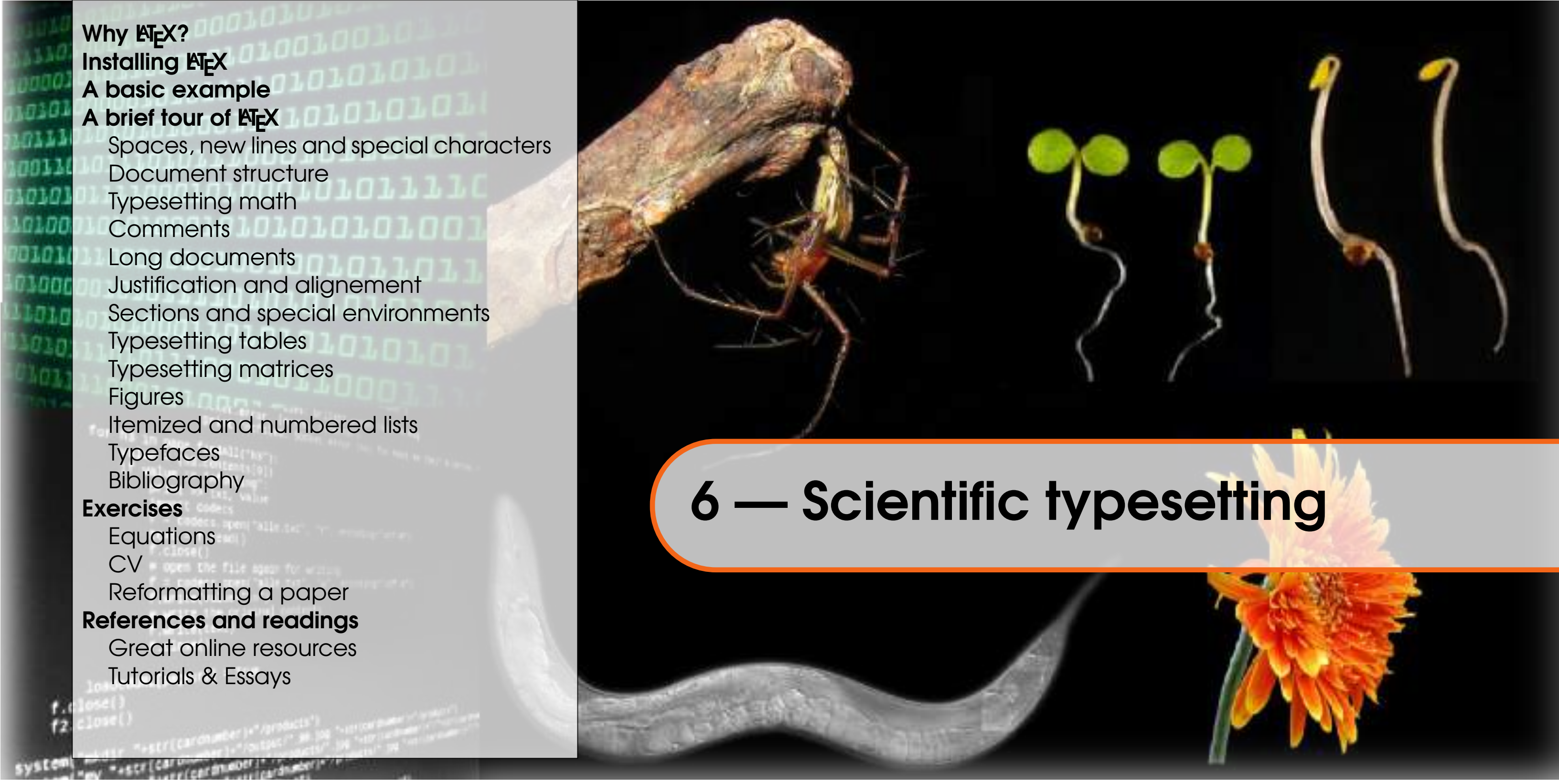
Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

The website `http://www.regular-expressions.info` is dedicated to the subject. It contains a really good list of canned solutions. The website `codinghorror.com` contains an interesting discussion on regular expressions (and hot sauce).

For SciPy, the best documentation is the official one:
`http://docs.scipy.org/doc/scipy/reference/`

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

http://xkcd.com/208/



6 — Scientific typesetting

6.1 Why LaTeX?

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

- The input is a text file. Hence, it is small and very portable. LaTeX compilers are freely available for all architectures. Therefore, you will obtain exactly the same result on any computer (this is not true for Word). Also, text files are great if you're using version control!
- LaTeX produces beautifully typeset documents. For example, these notes have been written in LaTeX. The mathematical formulas look much nicer than in other systems, and it is quite easy to create highly-complex mathematical expressions. In general, documents produced in LaTeX have a "professional look" that is difficult to obtain otherwise.
- LaTeX is very stable. The current version of LaTeX hasn't basically changed since 1994! This means that you will be able to access your text forever. In comparison, Microsoft issued 9 major versions of Word since 1994.
- LaTeX is free.
- LaTeX is a great choice for lengthy and complex documents (like your thesis).
- Many journals provide LaTeX templates, making the formatting of your manuscripts much quicker. It is very easy to move between two completely different styles. Bibliographic styles are also available: no need for EndNote or other bibliographic software. You can export bibliographies in BibTeX (the bibliographic style of LaTeX) from Google Scholar, CiteULike, Mendeley, Scopus, Web Of Science.
- On-line, freely available books and manuals are available. The number of users is such that any problem you might have has been already solved and the solution can be found with a

Google search.

Of course, there are also a number of disadvantages:

- It has a steeper learning curve.
- It is quite difficult to manage revisions with multiple authors (possibly, the most annoying problem) – but now there are cloud-based solutions!
- Typesetting tables is quite complex.
- Floating objects (figures, tables) are set automatically. Sometimes it is difficult to force them where they should belong.
- It is sometimes difficult to follow precisely the instructions of publishers if they are thought for Word (e.g., LaTeX by default adjusts the number of lines on a page to look pretty, but sometimes you need an exact number of lines per page).

6.2 Installing LaTeX

In Ubuntu, to install LaTeX along with many useful packages, type the following:

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

For Mac users, download from the site `tug.org/mactex/` a very large package containing LaTeX as well as any possible extension!

There are a number of WYSIWYG frontends to LaTeX, for example `Lyx` (`www.lyx.org`) and `TeXmacs` (`www.texmacs.org`). In Mac OSX, `TeXShop` works wonderfully, while in Windows one can install `MiKTeX`. In the rest of the chapter, we are going to use a text editor to compose simple `.tex` files.

6.3 A basic example

Open an editor and type the following in the file `FirstExample.tex`:

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

  \begin{abstract}
  Here we type the abstract of our paper.
  \end{abstract}

  \section{Introduction}
  And here the introduction.

  \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}
```

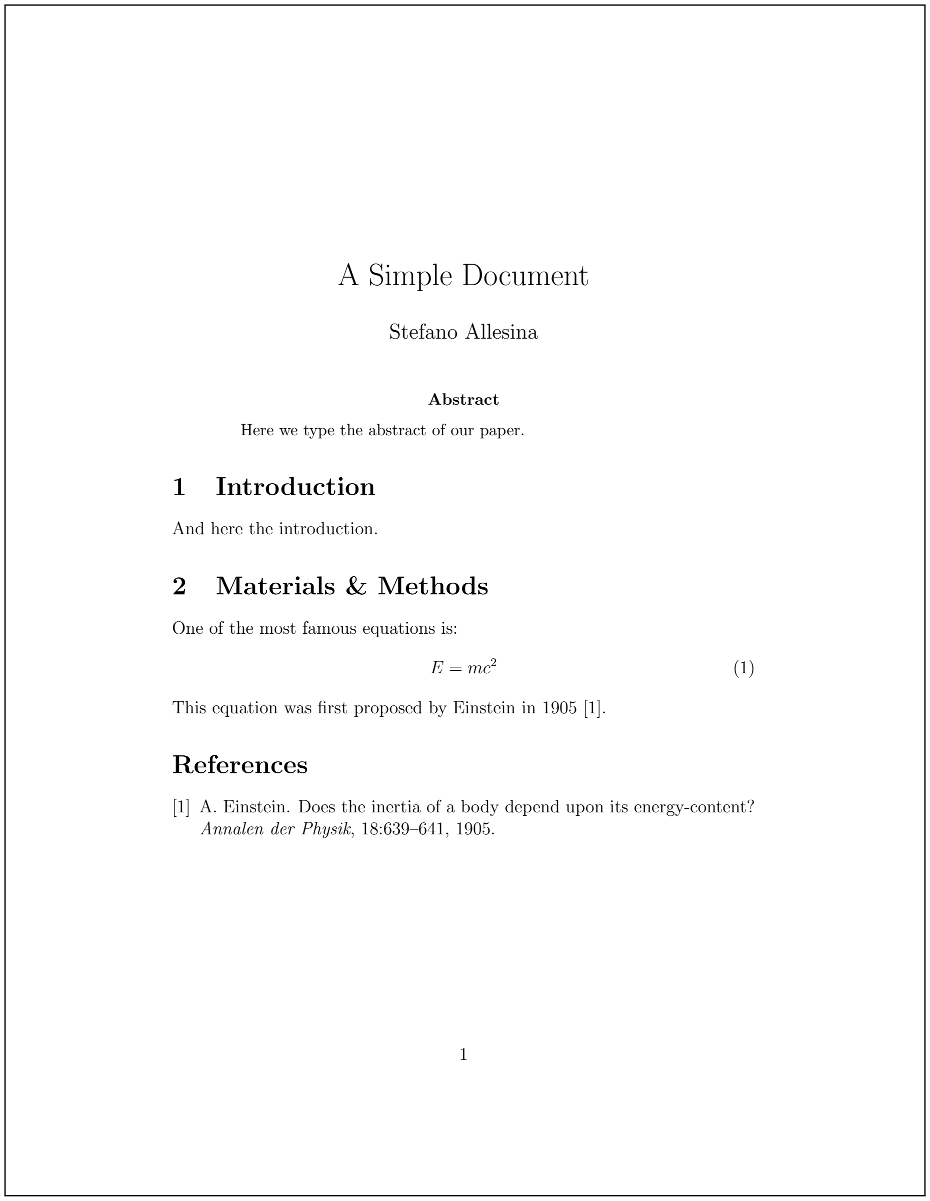
Now, let's get a citation for Einstein's paper. In Google Scholar, type "does the energy of a body einstein 1905". The paper should be the one on the top. Clicking "Cite" → "Import into BibTeX" should show the following text, that you will save in the file `FirstBiblio.bib`.

```
@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, go to the right directory and type:

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

This should produce the file `FirstExample.pdf` that you can open using, for example, Adobe Acrobat Reader (`evince` is available in Ubuntu, `Preview` in Mac OSX).


A Simple Document

Stefano Allesina

Abstract

Here we type the abstract of our paper.

1 Introduction

And here the introduction.

2 Materials & Methods

One of the most famous equations is:

$$E = mc^2 \tag{1}$$

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

6.4 A brief tour of LaTeX

6.4.1 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 characters, you have to add a "backslash" in front, e.g., \$ produces $.

6.4.2 Document structure

Each LaTeX command starts with \. For example, to typeset LaTeX you have to enter `\LaTeX`. The first command is always `\documentclass` defining the type of document you want to write. Examples are article, book, report, letter.

You can set several options. For example, to set the size of the text to 10 points and the letter paper size, type \documentclass[10pt,letterpaper]{article}.

After having declared the type of document, you can specify the 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, png, and jpeg.
- \usepackage{listings}: typeset source code for various programming languages.
- \usepackage{rotating}: rotate tables and figures.
- \usepackage{lineno}: line numbers.

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

As an example of structure of a document, take the article template provided by the journal Proceedings of the National Academy of Sciences USA (PNAS).

```latex
\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{2008}
\issuedate{Issue Date}
\volume{Volume}
\issuenumber{Issue Number}

\begin{document}
\title{My Title}
\author{Stefano Allesina\affil{1}{University of Chicago, Chicago, IL} \and
Luca E. Allesina\affil{2}{University of Chicago Laboratory School -- ↩
    Nursery 3, Chicago, IL}}
\maketitle
\begin{article}
\begin{abstract}
Here goes our wonderful 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}
```

6.4.3 Typesetting math

There are two ways to display mathematical contents. First, one can produce inline mathematics (i.e., within the text). Second, one can produce stand-alone, numbered equations and formulae. For inline mathematics, the “dollar” sign flanks the mathematics to be typeset. For example, the code:

```
The integral $\int_0^1 p^x (1-p)^y dp$ corresponds
to the defintion of a Beta function $\mathcal B(x+1, y+1)
= \Gamma(x+1)\Gamma(y+1)/\Gamma(x+y+2)$
which, in case of integer $x$ and $y$, assumes
the familiar form $x!y!/(x+y+1)!.$
```

Becomes:

The integral $\int_0^1 p^x (1-p)^y dp$ corresponds to the defintion of a Beta function $\mathcal B(x+1, y+1) = \Gamma(x+1)\Gamma(y+1)/\Gamma(x+y+2)$ which, in case of integer $x$ and $y$, assumes the familiar form $x!y!/(x+y+1)!$

For standalone formulas, there are different options. First, one can use the special symbols \[and \]. For example, this code:

```
The most beautiful mathematical result ever produced is Euler's
identity:

\[
e^{\pi i} + 1 = 0
\]
```

Becomes:
The most beautiful mathematical result ever produced is Euler's identity:

$$e^{\pi i} + 1 = 0$$

If we want to have numbered equations (almost always a great idea), LaTeX provides the `equation` environment:

```
A tricky integral

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

Becomes:
A tricky integral

$$\int_0^1 \left(\ln \left(\frac{1}{x} \right) \right)^y dx = y! \tag{6.1}$$

LaTeX has a full set of mathematical symbols and operators:

```
You can typeset limits and summations, multiple subscripts and
superscripts

\begin{equation}
\lim_{x \to \infty}, \sum_{i=1}^{\infty},
x^{y^{z^{2k}}}, x_{y_{z^2}}
\end{equation}

\ldots Greek letters

$\alpha, \beta, \rho, \Phi,
\lambda^\Gamma, \Sigma, \Theta$

\ldots fractions

\[a = \frac{b c^2}{\frac{x}{y \frac{k}{\ln x}}}\]

\ldots square roots, arrows etc.
\[
\sqrt{x},\, \sqrt[3]{x},\,
\overline{abcd}\underline{xy},\,
\widehat{abc}\overrightarrow{ky},\,
\downarrow \uparrow \rightarrow \Leftarrow,\,
\]

\ldots summations, integrals \ldots
\[\sum, \prod, \int, \iint, \iiiint, \bigcup\]

\ldots special functions
\[\cos, \exp, \min, \log, \tanh\]

\ldots and operators
\[\times, \pm, \neq, \leq,
\supset, \in, \propto\]

\ldots and so on.
```

Yelding:
You can typeset limits and summations, multiple subscripts and superscripts

$$\lim_{x\to\infty}, \sum_{i=1}^{\infty}, x^{y^{z^{2k}}}, x_{y_{z^2}} \tag{6.2}$$

…Greek letters
$\alpha, \beta, \rho, \Phi, \lambda^{\Gamma}, \Sigma, \Theta$
…fractions

$$a = \frac{bc^2}{\frac{x}{y^{\frac{k}{\ln x}}}}$$

…square roots, arrows etc.

$$\sqrt{x}, \sqrt[3]{x}, \overline{abcd\underline{xy}}, \widehat{abc}\overrightarrow{ky}, \downarrow\uparrow\rightarrow\Leftarrow$$

…summations, integrals …

$$\sum, \prod, \int, \iint, \iiiint, \bigcup$$

…special functions

$$\cos, \exp, \min, \log, \tanh$$

…and operators

$$\times, \pm, \neq, \leq, \supset, \in, \propto$$

…and so on.

6.4.4 Comments

Anything following a percentage sign % is considered a comment. To typeset the percentage sign, use the backslash \%.

6.4.5 Long documents

If you are planning to write a long document, it makes sense to split it into semi-independent files. Build a master file containing the document type, the basic setting etc. and then for example create a LaTeX file for each chapter. Finally, include (or exclude) each chapter using the \input command:

```
\documentclass{book}

\begin{document}

\title{\textbf{A simple book}}
\author{My Name}

\maketitle

\tableofcontents

\input{chapter1}
\input{chapter2}
\input{chapter3}
\input{chapter4}

\end{document}
```

6.4.6 Justification and alignement

To break a line, use \\ (this will not start a new paragraph). To start a new page use \newpage. To include all floating objects (e.g., figures and tables) before starting the new page use \clearpage. To make sure the new page starts on an odd-numbered page (e.g., chapters), use \cleardoublepage.

To align the text on the left, use \begin{flushleft} and \end{flushleft}. Similarly, \begin{flushright} and \end{flushright} produce right-justified text, while \begin{center} and \end{center} will center the text on the page.

6.4.7 Sections and special environments

```
Most \LaTeX documents respond to the following:

\section{A Section}
\subsection{aaa}
\subsubsection{bbb}
\paragraph{ccc}
\subparagraph{ddd}

Additionally, \texttt{book} and \texttt{report} use
```

```
\chapter{my chapter}

To produce a table of contents, type

\tableofcontents

And to produce the title page

\maketitle

To produce a footnote, use
\footnote{text in footnote}

The abstract is typically
\begin{abstract}
here's the abstract
\end{abstract}
```

6.4.8 Typesetting tables

The basic environment to typeset tables is called `tabular`:

```
\begin{tabular}{|l|r|c|}
l & stands for & left justified \\
r & stands for & right justified \\
c & stands for & centered \\
$\vert$ & produces & vertical lines\\
\hline
\textbackslash hline & produces & horizontal lines
\end{tabular}
\vspace{24pt}

How to typeset mutiple-column cells:

\vspace{0.1cm}
\begin{tabular}{|c|c|}
\hline
  element one & element two \\
\hline
  \multicolumn{2}{|c|}{spans multiple columns}\\
\hline
\end{tabular}

\vspace{0.1in}
And multiple-lines cells:
```

```
\begin{tabular}{|c|c|p{3cm}|}
\hline
short one & short one & very very very very very long one\\
\hline
\end{tabular}
\vspace{0.5cm}
```

Produces:

l	stands for	left justified
r	stands for	right justified
c	stands for	centered
|	produces	vertical lines
\hline	produces	horizontal lines

How to typeset mutiple-column cells:

element one	element two
spans multiple columns	

And multiple-lines cells:

short one	short one	very very very very very long one

In papers and other documents, however, you will typically include tables. These are “floating” bodies. Floating means that the object cannot be broken across pages, and thus LaTeX will try to place it where it looks “pretty”. Each table starts with the command `\begin{table}` followed by a specifier that determines its position. You can use:

Specifier	Place the floating table
h	Here. Try to place the table where specified.
t	At the Top of a page.
b	At the Bottom of a page.
p	In a separate page.
!	Try to force the positioning (e.g., !h).

Each table can contain a caption specified by the `\caption{my caption for the table}`. For example, the following:

```
\begin{table}[h]
  \begin{center}
    \begin{tabular}{lc}
      a small & table \\
      \hline
      1 & 2 \\
      3 & 4\\
    \end{tabular}
    \caption{The caption of my small table.}
  \end{center}
\end{table}
```

Produces a small table and tries to place it here. Note that LaTeX automatically numbers the table according to the document specification (e.g., using the chapter number etc.).

a small	table
1	2
3	4

Table 6.1: The caption of my small table.

6.4.9 Typesetting matrices

Matrices use the `\begin{array}` environment, that is similar to the `tabular`:

```
\begin{equation}
  A = \left[
    \begin{array}{ccc}
      \alpha & \beta & \gamma\\
      a & b & c\\
      \mathfrak a & \mathfrak b & \mathfrak c
    \end{array}
    \right]
\end{equation}

\begin{equation*}
  B = \left|
  \begin{array}{cc}
    a^2 & bac \\
    \frac{g}{f} & fa^3\sqrt{b}
  \end{array}
  \right|
\end{equation*}


\begin{equation}
  C = \left.
  \left(
    \begin{array}{ccc}
      \alpha & \beta & \gamma\\
      a & b & c\\
      \mathfrak a & \mathfrak b & \mathfrak c
    \end{array}
    \right)
  \right|_{a = a_0}
\end{equation}
```

$$A = \begin{bmatrix} \alpha & \beta & \gamma \\ a & b & c \\ \mathfrak{a} & \mathfrak{b} & \mathfrak{c} \end{bmatrix} \tag{6.3}$$

$$B = \begin{vmatrix} a^2 & bac \\ \frac{g}{f} & fa^3\sqrt{b} \end{vmatrix}$$

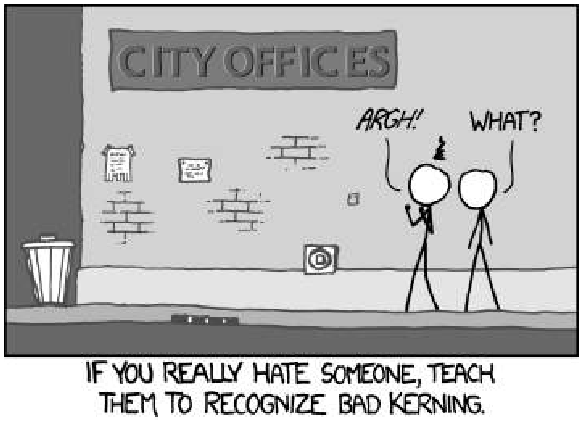
$$C = \left.\begin{pmatrix} \alpha & \beta & \gamma \\ a & b & c \\ \mathfrak{a} & \mathfrak{b} & \mathfrak{c} \end{pmatrix}\right|_{a=a_0} \tag{6.4}$$

6.4.10 Figures

Figures are also floating bodies. They can be included using the `graphicx` package. Depending on the installation, you might be able include only some formats. If you followed the instructions above, you should be able to use `.pdf`, `.ps`, `.eps`, `.jpg`, `.png` among others. If you used a more restrictive installation, you might be allowed to include only `.pdf`, `.ps and .eps`.

```
\begin{center}
\includegraphics[width = 0.5\linewidth]{xkcd/kerning.png}
\end{center}
```

Will import the picture into the file:


CITY OFFICES
ARGH!
WHAT?
IF YOU REALLY HATE SOMEONE, TEACH THEM TO RECOGNIZE BAD KERNING.

Also figures support captions and are automatically numbered:

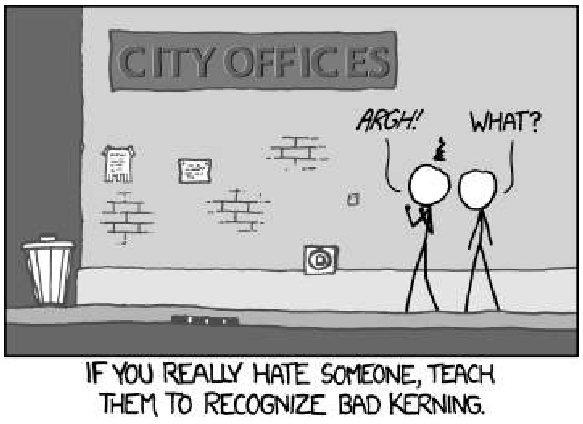
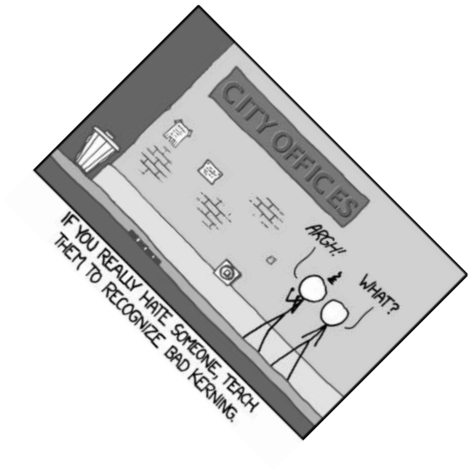

CITY OFFICES
ARGH!
WHAT?
IF YOU REALLY HATE SOMEONE, TEACH THEM TO RECOGNIZE BAD KERNING.

Figure 6.1: The caption of the figure.

```
\begin{figure}
  \begin{center}
    \includegraphics[width=0.5\linewidth]{xkcd/kerning.png}
  \end{center}
  \caption{The caption of the figure.}
\end{figure}
```

Besides `width`, other parameters that can be set are: `height`, `scale`, `angle`. More options let you clip and trim the figure (see the manual). For example:

```
\begin{center}
  \includegraphics[scale=0.4, angle=-45]{xkcd/kerning.png}
\end{center}
```


CITY OFFICES
ARGH!
WHAT?
IF YOU REALLY HATE SOMEONE, TEACH THEM TO RECOGNIZE BAD KERNING.

6.4.11 Itemized and numbered lists

The following code:

```latex
\begin{itemize}
\item First item.
\item Second item.
\item[$\star$] Third item.
\item[a] Fourth item.
\end{itemize}

\begin{enumerate}
\item First.
\item Second.
\item Third.
\end{enumerate}

\begin{description}
\item[First] A short description of First.
\item[Second] A short description of Second.
\item[Third] A short description of Third.
\end{description}
```

Produces:

- First item.
- Second item.

$\star$ Third item.

a Fourth item.

1. First.
2. Second.
3. Third.

First A short description of First.
Second A short description of Second.
Third A short description of Third.

6.4.12 Typefaces

The following code:

```latex
\begin{itemize}
\item \texttt{Typewriter like} also {\tt (but obsolete)}.
\item \textbf{Bold face} also {\bf (but obsolete)}.
\item \textit{Italics} also {\it (but obsolete)}.
\item \textsc{Small Caps} also {\sc (but obsolete)}.
\item \tiny tiny.
\item \footnotesize footnotesize.
\item \small small.
\item \large large.
\item \Large Large.
\item \normalsize normalsize.
```

```
\end{itemize}
```

Produces:

- `Typewriter like` also `(but obsolete)`.
- **Bold face** also **(but obsolete)**.
- *Italics* also *(but obsolete)*.
- SMALL CAPS also (BUT OBSOLETE).
- tiny.
- footnotesize.
- small.
- large.
- Large.
- normalsize.

Note that the size of the text (i.e., what is considered the `normalsize`) is governed by the initial choice in `documentclass`. The other sizes are changed proportionally.

6.4.13 Bibliography

There are many ways to include references and citations to your document. The best option is to use BibTeX to manage bibliographies. This requires three steps:

- Build a flat-file database containing your bibliography. The file is typically generated automatically from Web of Science, Scopus, Mendeley, Papers, EndNote, etc.
- Choose a BibTeX style for your references. Files specifying the citation style are available for most journals.
- Cite the references in your text.

Because this is easier done than said, we're going to explore the features using the templates provided by PLoS and Elsevier.

6.5 Exercises

6.5.1 Equations

Open a new LaTeX document, and typeset the following equations (make sure to include the packages `amsmath` and `amssymb`):

$$a = \sigma\sqrt{SC}(1 - \mathbb{E}^2(|X|)/\sigma^2)$$

$$\sum_{j=1}^{S} M_{ij} \approx -d + (S-1)\mathbb{E}(M_{ij})_{i\neq j} = -d + (S-1)\cdot C \cdot \mathbb{E}(|X|)$$

$$L(A|q,G) = \prod_{i=1}^{S}\prod_{j=1}^{S} p_{i,j}^{A_{i,j}}(1-p_{i,j})^{1-A_{i,j}} = \prod_{k=1}^{\gamma}\prod_{l=1}^{\gamma} q_{k,l}^{L_{k,l}}(1-q_{k,l})^{Z_{k,l}} \tag{6.5}$$

$$P(M_1|A) = \frac{P(A|M_1)P(M_1)}{P(A)} \tag{6.6}$$

$$P(A|M_i) = \mathscr{M}_i = \int P(A|M_i,\Theta_i)P(\Theta_i|M_i)d\Theta_i \tag{6.7}$$

$$\mathscr{M}_{Group|G} = \int_0^1 \int_0^1 \cdots \int_0^1 \prod_i^{\gamma} \prod_j^{\gamma} p_{i,j}^{L_{i,j}} (1-p_{i,j})^{Z_{i,j}} dp_{i,j}$$

6.5.2 CV

Build your CV in LaTeX using the `moderncv` style provided in the repository.

6.5.3 Reformatting a paper

In the folder `NestednessEigenvalues` you find a manuscript of Staniczenko *et al.* formatted for Nature Communications. Reformat it in the style of PLoS Computational Biology. (Optional: download the syle files and format it for PNAS).

6.6 References and readings

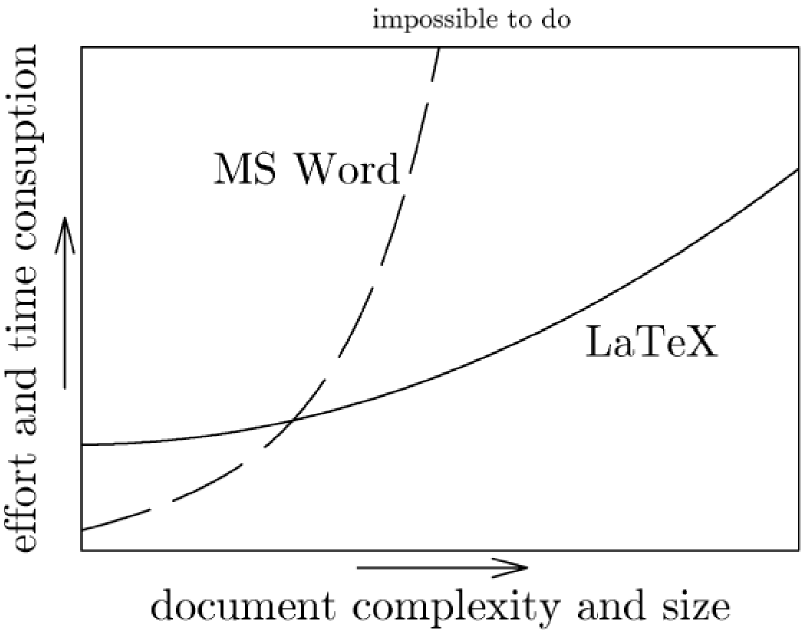
6.6.1 Great online resources

The Visual LaTeX FAQ: sometimes it is difficult to describe what you want to do!
`http://get-software.net/info/visualFAQ/visualFAQ.pdf`
Drawing formulas by hand and converting them into LaTeX
`http://webdemo.visionobjects.com`
Online equation editors
`http://www.sciweavers.org/free-online-latex-equation-editor`
`http://rogercortesi.com/eqn/`
From (semi-)natural language to LaTeX
`http://www.math.missouri.edu/~stephen/naturalmath/`
University of Chicago Dissertation Template
`https://wiki.uchicago.edu/display/DissertationTemplate`
Other disseration templates:
`http://latexforhumans.wordpress.com/`
Cloud-based collaborations using LaTeX
`https://www.writelatex.com/`
`https://www.sharelatex.com`

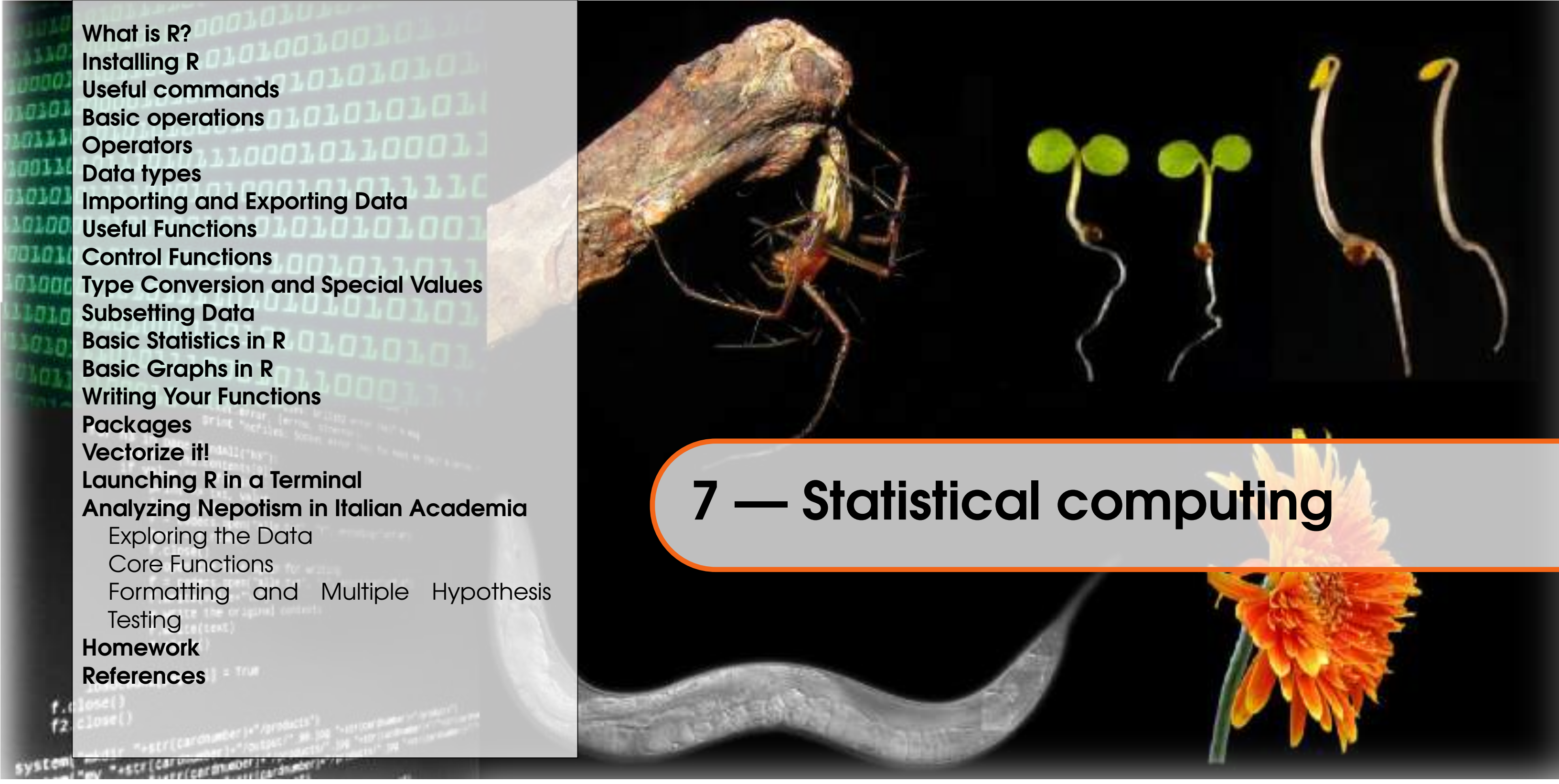
6.6.2 Tutorials & Essays

Minimal introduction to LaTeX.
`orion.math.iastate.edu/burkardt/latex_intro/latex_intro.html`
A (Not So) Short Introduction to LaTeX.
`www.ctan.org/tex-archive/info/lshort/english/`
The Beauty of LaTeX.
`http://nitens.org/taraborelli/latex`
Word vs. LaTeX:
`http://www.streamtoolsonline.com/word-v-latex.html`

http://3monththesis.com/writing-a-thesis-word-or-latex/
http://www.andy-roberts.net/writing/latex/benefits
Beautiful presentations in LaTeX:
www.stat.berkeley.edu/~luis/seminar/BeamerPres110128_Miki.pdf
Bibliographies for biological journals
www.lecb.ncifcrf.gov/~toms/latex.html



http://www.pinteric.com/miktex.html



7 — Statistical computing

7.1 What is R?

R is a freely available statistical software with strong programming capabilities. R is becoming very popular in ecology and other branches of biology. This is due to several factors: i) many packages are available to perform all sorts of statistical and mathematical analysis; ii) it can produce beautiful graphics; iii) it has a very good support for matrix-algebra. Although we can technically program R, the programming environment is not the greatest: especially the way types are managed is problematic (e.g., often your matrix will become a vector if it has only one column/row...). Nevertheless, being able to program R means you can automate the statistical analysis and the generation of figures.

7.2 Installing R

Ubuntu:

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

Mac OS X: download from

```
http://cran.r-project.org/bin/macosx/
```

7.3 Useful commands

ls() list all the variables in the work space
rm() remove variable(s), e.g., `rm(list=ls())`
getwd() get current working directory
setwd() set working directory to ...
q() quit R

?Command show the documentation of *Command*
??Keyword search the all the packages/functions with *Keyword*, “fuzzy search”

7.4 Basic operations

We can use R in interactive mode, as a calculator. To launch R, simply type R in a terminal.

```
> a <- 4  ## assignment
> a
[1] 4
> a*a  ## product
[1] 16
> a_squared <- a*a
> sqrt(a_squared) ## square root
[1] 4
> v <- c(0, 1, 2, 3, 4) ## c: concatenate
> v
[1] 0 1 2 3 4
> is.vector(v) ## check if it's a vector
[1] TRUE
> mean(v) ## mean
[1] 2
> var(v) ## variance
[1] 2.5
> median(v) ## median
[1] 2
> sum(v) ## sum all elements
[1] 10
> prod(v + 1) ## multiply
[1] 120
> length(v) ## length of vector
[1] 5
> 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
> v2 = v
> v2 = v2*2 ## whole-vector operation
> v2
[1] 0 2 4 6 8
> v * v2 ## product element-wise
[1]  0  2  8 18 32
```

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

7.5 Operators

The usual operators are available (slight differences from `python`):
+ Addition
- Subtraction
***** Multiplication
/ Division
^ Power
%% Modulo
%/% Integer division
== Equals
!= Differs
> Greater
>= Greater or equal
& Logical and
| Logical or
! Logical not

7.6 Data types

As `python`, `R` comes with interesting data-types. Mastering these will help you write better and more compact programs.
Vectors

```
> 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)
```

Matrices and arrays

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

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

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

, , 2
```

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

Data frames

This is a very important data type that is peculiar to R. It is great for storing your data. Basically, it's a two-dimensional table in which each column can contain a different data type (e.g., numbers, strings, boolean). You can think of a dataframe as a spreadsheet. Data frames are great for plotting your data, performing regressions and such.

```
> Col1 <- 1:10
> Col1
 [1]  1  2  3  4  5  6  7  8  9 10
> Col2 <- LETTERS[1:10]
> Col2
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
> Col3 <- runif(10) ## 10 random num. from uniform
> Col3
 [1] 0.29109 0.91495 0.64962 0.95503 0.26589 0.02482 0.59718
 [8] 0.99134 0.98786 0.86168
> MyDF <- data.frame(Col1, Col2, Col3)
> MyDF
   Col1 Col2      Col3
1     1    A 0.2910981
2     2    B 0.9149558
3     3    C 0.6496248
4     4    D 0.9550331
5     5    E 0.2658936
6     6    F 0.0248217
7     7    G 0.5971868
8     8    H 0.9913407
9     9    I 0.9878679
10   10    J 0.8616854
> names(MyDF) <- c("A.name", "another", "another.one")
> MyDF
   A.name another another.one
1       1       A   0.2910981
2       2       B   0.9149558
3       3       C   0.6496248
4       4       D   0.9550331
5       5       E   0.2658936
6       6       F   0.0248217
7       7       G   0.5971868
```

```
8       8       H   0.9913407
9       9       I   0.9878679
10     10       J   0.8616854
> MyDF$A.name
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[,1]
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[c("A.name","another")]
   A.name another
1       1       A
2       2       B
3       3       C
4       4       D
5       5       E
6       6       F
7       7       G
8       8       H
9       9       I
10     10       J
> class(MyDF)
[1] "data.frame"
> str(MyDF)
'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 ...
```

Lists

A list is simply an ordered collection of objects (that can be other variables). You can build lists of lists too.

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

$ages
[1] 42 77 13 91

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

7.7 Importing and Exporting Data

R provides several functions for importing data. By far, the best option is to have your data in a comma separated value text file or in a tab separated file. Then, you can use the functionread.csv (or read.table) to import your data.

```
> read.csv("MyFile.csv")
> read.csv("MyFile.csv", header = TRUE) # file has header
> read.csv("MyFile.csv", sep = ';') # separated by ;
> read.csv("MyFile.csv", skip = 5) # skip the first 5 lines
```

Similarly, you can save your data frames using write.table or write.csv. Suppose you want to save the dataframe MyDF:

```
> write.csv(MyDF, "MyFile.csv")
> write.csv("MyFile.csv", append=TRUE) # append at the end
> read.csv("MyFile.csv", row.names=TRUE) # write row names
> read.csv("MyFile.csv", col.names=FALSE)
  # do not write col names
```

7.8 Useful Functions

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

Random number distributions

rnorm(10, m=0, sd=1) Draw 10 random numbers from a Gaussian distribution 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)

7.9 Control Functions

Also in R, you can write “if, then, else” statements, “for” and “while” loops. However, loops are painfully slow in R, so use them sparingly.

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

## On the same line
if (z <= 0.25) print ("Less than a quarter")

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

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

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

7.10 Type Conversion and Special Values

Beware of the difference between NA (Not Available) and NaN (Not a Number). Also, a quite annoying feature: there are different functions to check whether a number is finite, infinite, not a number. These can be confusing, so double check your code!

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

7.11 Subsetting Data

Often, you will need to get only part of a matrix, dataframe, etc. Here are ways of subsetting your data:

```
> M <- matrix(1:9, 3, 3)
> M
     [,1] [,2] [,3]
[1,]    1    4    7
```

```
[2,]    2    5    8
[3,]    3    6    9
> M[1,1]
[1] 1
> M[1, ]
[1] 1 4 7
> M[1:2, ]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
> M[c(1, 3), 2]
[1] 4 6
> M[-3,]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
> M[-3,-2]
     [,1] [,2]
[1,]    1    7
[2,]    2    8
> attach(iris)
> iris
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2 setosa
2           4.9         3.0          1.4         0.2 setosa
3           4.7         3.2          1.3         0.2 setosa
4           4.6         3.1          1.5         0.2 setosa
5           5.0         3.6          1.4         0.2 setosa
6           5.4         3.9          1.7         0.4 setosa
7           4.6         3.4          1.4         0.3 setosa
8           5.0         3.4          1.5         0.2 setosa
9           4.4         2.9          1.4         0.2 setosa
10          4.9         3.1          1.5         0.1 setosa
> iris$Sepal.Width[1:5]
[1] 3.5 3.0 3.2 3.1 3.6
> iris[1:3, c("Sepal.Width","Petal.Width")]
  Sepal.Width Petal.Width
1         3.5         0.2
2         3.0         0.2
3         3.2         0.2
> iris[(Sepal.Width < 3.5) & (Sepal.Length > 7.2),]
 Sepal.Length Sepal.Width Petal.Length Petal.Width Species
106      7.6         3.0          6.6         2.1 virginica
108      7.3         2.9          6.3         1.8 virginica
119      7.7         2.6          6.9         2.3 virginica
123      7.7         2.8          6.7         2.0 virginica
```

```
131        7.4          2.8          6.1          1.9 virginica
136        7.7          3.0          6.1          2.3 virginica
```

7.12 Basic Statistics in R

R contains several functions to perform basic (and advanced) statistics. I am not going to cover any statistics here, but rather list some basic commands to show you that statistical analysis in R is easy to perform.

```
> attach(iris)
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length  Petal.Width   
 Min.   :4.300   Min.   :2.000   Min.   :1.000 Min.   :0.100  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600 1st Qu.:0.300  
 Median :5.800   Median :3.000   Median :4.350 Median :1.300  
 Mean   :5.843   Mean   :3.057   Mean   :3.758 Mean   :1.199  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100 3rd Qu.:1.800  
 Max.   :7.900   Max.   :4.400   Max.   :6.900 Max.   :2.500  
       Species  
 setosa    :50  
 versicolor:50  
 virginica :50  
> table(Species)
Species
    setosa versicolor  virginica 
        50         50         50 

> table(Species, Petal.Width)
            Petal.Width
Species      0.1 0.2 0.3 0.4 0.5 0.6  1 1.1 1.2 1.3 1.4 ...
  setosa       5  29   7   7   1   1  0   0   0   0   0 ...
  versicolor   0   0   0   0   0   0  7   3   5  13   7 ...
  virginica    0   0   0   0   0   0  0   0   0   0   1 ...
            Petal.Width
Species      2.1 2.2 2.3 2.4 2.5
  setosa       0   0   0   0   0
  versicolor   0   0   0   0   0
  virginica    6   3   8   3   3
> t.test(Sepal.Width[which(Species == "setosa")],
         Sepal.Width[which(Species == "versicolor")])

Welch Two Sample t-test

data:  Sepal.Width[which(Species == "setosa")] and
       Sepal.Width[which(Species == "versicolor")]
t = 9.455, df = 94.698, p-value = 2.484e-15
```

```
alternative hypothesis: true difference in means is
not equal to 0
95 percent confidence interval:
 0.5198348 0.7961652
sample estimates:
mean of x mean of y
    3.428     2.770
> summary(lm(Sepal.Width ~ Sepal.Length))

Call:
lm(formula = Sepal.Width ~ Sepal.Length)

Residuals:
    Min      1Q  Median      3Q     Max
-1.1095 -0.2454 -0.0167  0.2763  1.3338

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   3.41895    0.25356   13.48   <2e-16 ***
Sepal.Length -0.06188    0.04297   -1.44    0.152
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4343 on 148 degrees of freedom
Multiple R-squared: 0.01382,Adjusted R-squared: 0.007159
F-statistic: 2.074 on 1 and 148 DF,  p-value: 0.1519
```

7.13 Basic Graphs in R

R can produce beautiful graphics. The syntax, however, is not great, and sometimes producing exactly the graph you have in mind can be very laborious. We will see in the next Chapter how to use `ggplot2`, which makes the syntax more consistent and allows the rapid creation of publication-grade graphics.

However, in many cases you just want to quickly plot the data for exploratory analysis and such. Here are the basic plotting commands:

`hist(mydata)`	Histogram.
`barplot(mydata)`	Bar plot.
`plot(y~x)`	Scatterplot.
`points(y1~x1)`	Add another series of points.
`boxplot(y~x)`	Boxplot.

7.14 Writing Your Functions

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

```r
MyFunction <- function(Arg1, Arg2){
  ## statements
  return (ReturnValue)
}

## Computes the Position^th Fibonacci
## number 1 1 2 3 5 8 13 ...
Fibonacci <- function(Position){
  if (Position < 3){
    return (1)
  }
  i <- 1
  j <- 1
  s <- 0
  for (k in (Position - 2):1){
    s <- i + j
    i <- j
    j <- s
  }
  return (j)
}

## Computes the Factorial of a number
## using recursion
Factorial <- function(N){
  if (N == 1){
    return (N)
  }
  return(N * Factorial(N - 1))
}

## Computes Coefficient of Variation
CoefVar <- function(vec){
  return(sd(vec) / mean(vec))
}
```

7.15 Packages

The main strength of `R` is that users can easily build packages and share them through the website CRAN. There are packages to do most statistical and mathematical analysis you might conceive, so check this out before re-inventing the wheel!

Visit `cran.r-project.org` and go to Packages to see a list and a brief description. To install a package, within `R` type `install.packages()` and choose the package to install.

7.16 Vectorize it!

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

Hence, in R you should try to avoid cycles as you would avoid bad diseases. In practical terms, sometimes it is much easier to throw in a for loop, and then optimize the code to avoid the loop if the running time is not satisfactory.

R has several functions that can operate on entire vectors and matrices. For example, consider this code for summing all elements of a matrix:

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

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

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

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

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

```r
## 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)
```

```r
## By column
ColMeans <- apply(M, 2, mean)
print (ColMeans)

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

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

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

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

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

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

7.17 Launching R in a Terminal

Most times, you want to run the final analysis without opening `R` in interactive mode. You can do so by typing

```
R CMD BATCH MyFile.R
```

This will create an `.Rout` file containing all the output.

7.18 Analyzing Nepotism in Italian Academia

To see how easy it is to perform analysis in `R`, we are going to replicate the results of my recent paper on nepotism in Italian academia (Allesina, PLoS One, 2011). I choose this example because it requires a large dataset, but the analysis is simple and can be used to illustrate many `R` commands.

Cases of nepotism have been documented in Italian universities Professors were found to engage in illegal practices to have relatives hired as academics in their institution or within the same discipline at other universities. Because the prevalence of nepotism in Italian academia is largely unknown, I recently proposed a statistical method to estimate rates of nepotism based on the distribution of last names of Italian academics. My results confirmed the findings obtained with more complex techniques, namely that a) some disciplines are more likely to display nepotistic tendencies, and b) a latitudinal pattern exists, showing a higher probability of nepotism in the south of the country and in Sicily.

The method is simple and requires minimal data, only a list of the names of all professors in Italy, and their corresponding discipline. For a given discipline, count the number of academics N and the number of unique last names L. Then draw a random sample from the list of all Italian professors, taking N professors without repetition. Finally, compute the probability p that a number of unique last names $L' \leq L$ is found in a sample. If the probability is very low, i.e., it is difficult to find a sample with a number of last names smaller than that empirically observed for the discipline, then the scarcity of last names in the discipline cannot be explained by unbiased (i.e., random) hiring processes.

7.18.1 Exploring the Data

```r
## Read the csv file
MyData <- read.csv("../Data/ITA-Names.csv")

## Show the first few records
MyData[1:5,]

## Number of records
dim(MyData)[1]

## Institutions
sort(unique(MyData[,"Institution"]))
length(unique(MyData[,"Institution"]))

## Last Names
sort(unique(MyData[,"Last"]))
length(unique(MyData[,"Last"]))

## Most Common Last Names
TableLast <- table(MyData[,"Last"])
sort(TableLast, decreasing = TRUE)[1:15]

## Disciplines
sort(unique(MyData[,"Discipline"]))
length(unique(MyData[,"Discipline"]))
sort(table(MyData[,"Discipline"]))

```

```r
## Males and Females
sort(table(MyData[,"Gender"]))
```

7.18.2 Core Functions

```r
## Sample Size names from NamesSet and
## returns the number of unique names
## found in the sample
GetOneSample <- function(NamesSet,Size){
  return(length(unique(sample(NamesSet,Size))))
}

## Takes a set of names in a discipline,
## and returns:
## 1) Number of professors in the discipline
## 2) Number of unique last names
## 3) Expected number of last names
## 4) p-value
## Input:
## AllNames - Set of all the names
## DiscNames - Names in the discipline
## NReps - Number of replicates for
## computing p-value
GetResultsNames <- function(AllNames, DiscNames, NReps){
  NProfs <- length(DiscNames)
  NNames <- length(unique(DiscNames))
  Expected <- 0
  p.value <- 0.0
  ## Using for loop!
  for (i in 1:NReps){
    ## Get one sample
    MyRes <- GetOneSample(AllNames, NProfs)
    ## Add to the expectation
    Expected <- Expected + MyRes
    ## Add to p-value
    if (MyRes <= NNames){
      p.value <- p.value + 1.0
    }
  }
  ## Normalize
  Expected <- Expected / NReps
  p.value <- p.value / NReps
  return( c(NProfs, NNames, Expected, p.value) )
}
```

```

## Test
MyData <- read.csv("../Data/ITA-Names.csv")
## Get only Medicine (MED)
MyDiscNames <- MyData[MyData[,"Discipline"] == "MED", "Last"]
## Compute stats
print(system.time(z <- GetResultsNames(MyData[, "Last"], MyDiscNames, 100)↩
    ))
print(system.time(z <- GetResultsNames(MyData[, "Last"], MyDiscNames, 500)↩
    ))
print(system.time(z <- GetResultsNames(MyData[, "Last"], MyDiscNames, ↩
    1000)))
```

But wait, we're using a for loop! Is it better to use `replicate` in this case?

```
## Sample Size names from NamesSet and
## returns the number of unique names
## found in the sample
GetOneSample <- function(NamesSet,Size){
  return(length(unique(sample(NamesSet,Size))))
}

## Takes a set of names in a discipline,
## and returns:
## 1) Number of professors in the discipline
## 2) Number of unique last names
## 3) Expected number of last names
## 4) p-value
## Input:
## AllNames - Set of all the names
## DiscNames - Names in the discipline
## NReps - Number of replicates for
## computing p-value
GetResultsNames2 <- function(AllNames, DiscNames, NReps){
  NProfs <- length(DiscNames)
  NNames <- length(unique(DiscNames))
  ## Using replicate
  MyRes <- replicate(NReps, GetOneSample(AllNames, NProfs))
  ## Normalize
  Expected <- sum(MyRes) / NReps
  p.value <- sum(MyRes <= NNames) / NReps
  return( c(NProfs, NNames, Expected, p.value) )
}

## Test
```

```
MyData <- read.csv("../Data/ITA-Names.csv")
## Get only Medicine (MED)
MyDiscNames <- MyData[MyData[,"Discipline"] == "MED", "Last"]
## Compute stats
print(system.time(z <- GetResultsNames2(MyData[, "Last"], MyDiscNames, ↩
    100)))
print(system.time(z <- GetResultsNames2(MyData[, "Last"], MyDiscNames, ↩
    500)))
print(system.time(z <- GetResultsNames2(MyData[, "Last"], MyDiscNames, ↩
    1000)))
```

Replicate is not better because R has to reserve a large chunk of memory when the number of replicates is high. Also, most of the time is spent in the sample step. We can thus use the for loop and write the complete program:

```
## Sample Size names from NamesSet and
## returns the number of unique names
## found in the sample
GetOneSample <- function(NamesSet,Size){
  return(length(unique(sample(NamesSet,Size))))
}

## Takes a set of names in a discipline,
## and returns:
## 1) Number of professors in the discipline
## 2) Number of unique last names
## 3) Expected number of last names
## 4) p-value
## Input:
## AllNames - Set of all the names
## DiscNames - Names in the discipline
## NReps - Number of replicates for
## computing p-value
GetResultsNames <- function(AllNames, DiscNames, NReps){
  NProfs <- length(DiscNames)
  NNames <- length(unique(DiscNames))
  Expected <- 0
  p.value <- 0.0
  ## Using for loop!
  for (i in 1:NReps){
    ## Get one sample
    MyRes <- GetOneSample(AllNames, NProfs)
    ## Add to the expectation
    Expected <- Expected + MyRes
    ## Add to p-value
    if (MyRes <= NNames){
      p.value <- p.value + 1.0
    }
  }
  ## Normalize
  Expected <- Expected / NReps
  p.value <- p.value / NReps
  return( c(NProfs, NNames, Expected, p.value) )
}

NReps <- 2500
## Get the data
MyData <- read.csv("../Data/ITA-Names.csv")

## Perform the analysis
AllDisc <- sort(unique(MyData[,"Discipline"]))
NDisc <- length(AllDisc)
Results <- matrix(0, 0, 4) ## matrix with 0 rows and 4 columns
colnames(Results) <- c("People", "Last Names", "Expected", "p.value")
AllNames <- MyData[, "Last"]
for (DD in AllDisc){
  DiscNames <- MyData[MyData[,"Discipline"] == DD , "Last"]
  z <- GetResultsNames(AllNames, DiscNames, NReps)
  print (c(DD, z))
  ## attach to the results
  Results <- rbind(Results, z)
}
## set row names
rownames(Results) <- AllDisc

## Save the results in a csv file
write.table(Results, "Results-LastNames.csv")
```

7.18.3 Formatting and Multiple Hypothesis Testing

When we are performing Monte Carlo simulations, we only approximate the true p-value. Clearly, the more simulations we run, the better the appoximation. Typically, the number of significant digits scales with the logarithm of the number of simulations. Hence, if we perform 10,000 simulations we get two significant digits, to get three digits we should perform a million simulations, etc. We want to approximate the results to the right number of digits and write < 0.01 whenever our approximation is 0 and > 0.99 whenever it is exactly 1.

Moreover, we are performing multiple tests on the same data, and we should correct using Bonferroni or other corrections. Instead of doing that, we compute a q-value, expressing the probability that a result is a false positive. To do so, we install the package `qvalue`.

```
> require(qvalue)
> qvalue(runif(10), pi0.method = "bootstrap")
$call
qvalue(p = runif(10), pi0.method = "bootstrap")

$pi0
[1] 0.75

$qvalues
 [1] 0.7003129 0.70031 0.70031 0.70422 0.329 0.15700 0.15700
 [8] 0.7003129 0.3291876 0.7003129

$pvalues
 [1] 0.77749086 0.819268 0.672216 0.938960 0.175566 0.041868
 [7] 0.02756145 0.84037544 0.15549211 0.58071018

$lambda
 [1] 0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45
[11] 0.50 0.55 0.60 0.65 0.70 0.75 0.80 0.85 0.90

attr(,"class")
[1] "qvalue"
```

We want to add "**" to the results with q-value < 0.05 and p-value < 0.05. That is, the highly significant results have low probability of being obtained at random (low p-value) and low probability of being false positives (low q-value).

Finally, we want to sort the results by p-value. We can accomplish all this with the following code:

```
require(qvalue)

## Set to two significant digits
## Add ** if highly significant
GetSignif <- function(pvalues){
  ## compute q-values
  z <- qvalue(pvalues, pi0.meth="bootstrap")
  ## Highly Significant: pvalue<=0.05 qvalue
  HighlySig <- (z$qvalues <= 0.05) & (z$pvalues <= 0.05)
  ## Round to two digits
  newpv <- round(pvalues,2)
  NumPV <- length(newpv)
  ## Transform to strings
  MyStrings <- format(newpv,digits=2,scientific=FALSE)
  ## Set small and large values
  MyStrings[newpv < 0.01] <- "<0.01"
  MyStrings[newpv > 0.99] <- ">0.99"
  ## Add significant
  for (i in 1:NumPV){
    if (HighlySig[i]){
      MyStrings[i] <- paste(MyStrings[i], "**", sep = "")
    }
  }
  return(MyStrings)
}

MyTab <- read.table("Results-LastNames.csv", header =TRUE)
## Sort by p-value
MyTab <- MyTab[sort(MyTab[,"p.value"], index.return = TRUE)$ix, ]
## Format 2 significant digits
MyTab[ , 4] <- GetSignif(MyTab[ , 4])
## Write the table
write.table(MyTab, "FormattedTable.csv")
```

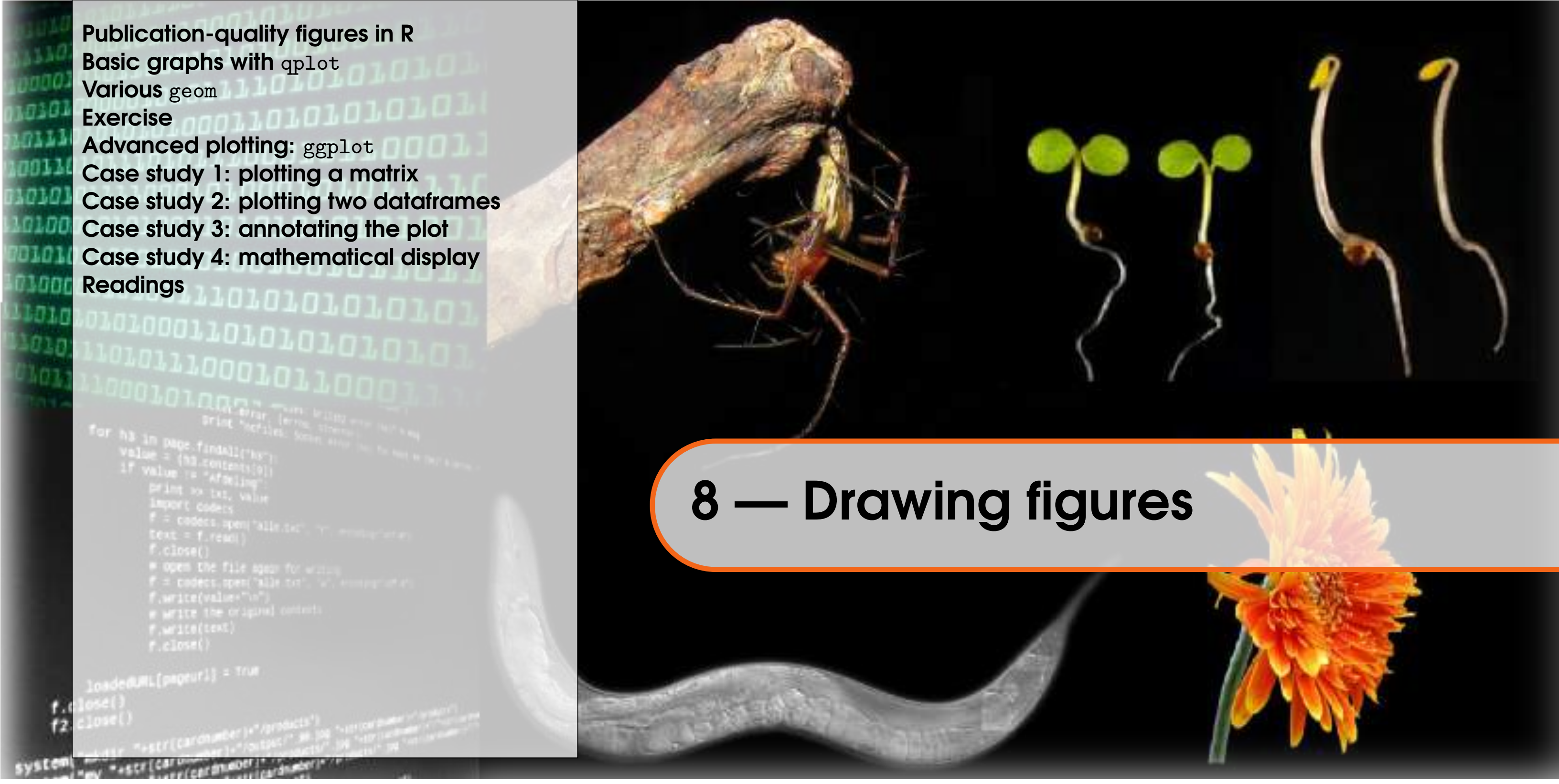
7.19 Homework

- Repeat the analysis using first names instead of last names. Are there disciplines with fewer first names than expected? Why would that be?
- Write a program that computes the fraction of women in each discipline: do disciplines with few women have too few last names?
- Write a program that analyzes the last names using only the male professors. Does this account for the results?
- Write a `python` program that analyzes the last names. Is the program faster? (You can test the time it takes using `time [command to execute]`. With my computer, `R` takes about 19 minutes to run the analysis of last names using 5000 replicates. The same is accomplished in `python` in 3.5 minutes.

7.20 References

There are many good references for `R`.

- Springer publishes the Use R! series (the yellow books), which is really good. I would suggest "A Beginner's Guide to R", "R by Example", "Numerical Ecology With R", "ggplot2" (we'll see this in another chapter), "A Primer of Ecology with R", "Nonlinear Regression with R", "Analysis of Phylogenetics and Evolution with R". They are all excellent.
- Ben Bolker recently published "Ecological Models and Data in R".
- For more focus on dynamical models, Soetaert & Herman. 2009 published "A practical guide to ecological modelling: using R as a simulation platform". I reviewed this book for Ecology ("Learning R the Practical Way", Ecology 90: 2335-2336).
- There are excellent websites: besides cran (containing all sorts of guides and manuals, you should check out www.statmethods.net, gallery.r-enthusiasts.com and rwiki.sciviews.org.



8 — Drawing figures

8.1 Publication-quality figures in R

`R` can produce beautiful graphics, but it takes a lot of work to obtain the desired result. This is because the starting point is pretty much a "bare" plot, and adding features commonly required for publication-grade figures (legends, statistics, regressions, etc.) can be quite involved. Moreover, it is very difficult to switch from one representation of the data to another (i.e., from boxplots to scatterplots), or to plot several dataset together.

In this Chapter we introduce the `R` package `ggplot2`, which can be used to produce high-quality graphics for papers, theses and reports. `ggplot2` differs from other approaches as it attempts to provide a "grammar" for graphics in which each layer is the equivalent of a verb, subject etc. and a plot is the equivalent of a sentence. All graphs start with a layer showing the data, other layers and commands are added to modify the plot.

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

```
> install.packages("ggplot2")
> install.packages("reshape")
```

8.2 Basic graphs with `qplot`

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

In `ggplot2`, it is very important to use data frames to store the data. As an example, we will use a dataset on predator-prey body mass ratio taken from the Ecological Archives of the ESA (Barnes *et al.*, Ecology 89:881 - 2008). I simplified the dataset for your convenience. Go in the `R/Sandbox` directory and launch `R`.

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

We can start plotting the `Predator.mass` vs `Prey.mass`.

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

This graph is very basic. Let's transform everything in logarithms:

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

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

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

The same as above, but changing the shape:

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

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

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

Because there are so many points, we can make them semi-transparent:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  colour = Type.of.feeding.interaction, alpha = I(1/5))
```

Now add a smoother to the points.

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

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

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

We can add a smoother for each type of interaction:

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

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

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

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

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

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

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

Or we can draw a boxplot of the data:

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

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)
```

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)
```

Using colour instead of fill draws only the edge of the curve:

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

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

An alternative way of displaying data belonging to different classes is the so-called “faceting”. A simple example:

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

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

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

Let’s add a title and labels:

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

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

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

Finally, let’s save a pdf file of the figure.

```
> pdf("MyFirst-ggplot2-Figure.pdf")
> print(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())
> dev.off()
```

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

Other important options to keep in mind:

xlim limits for x axis: xlim = c(0,12).

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

8.3 Various geom

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")
```

8.4 Exercise

Write a script called `Prob-9-1.R` that draws a pdf file of Figure 8.4.

8.5 Advanced plotting: `ggplot`

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

- The data to be plotted, in a data frame;
- Aesthetics mappings, specifying which variables we want to plot, and how;
- The `geom`, defining how to draw the data;
- (Optionally) some `stat` that transform the data or perform statistics using the data.

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

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

Now try to plot the graph:

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

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

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

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

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

Let's remove the legend:

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

8.6 Case study 1: plotting a matrix

In this section we will plot a matrix of random values taken from a normal distribution $\mathscr{U}[0,1]$. Our goal is to produce the plot in Figure 8.2. Because we want to plot a matrix, and `ggplot2` accepts only dataframes, we use the package `reshape` that can "melt" a matrix into a dataframe:

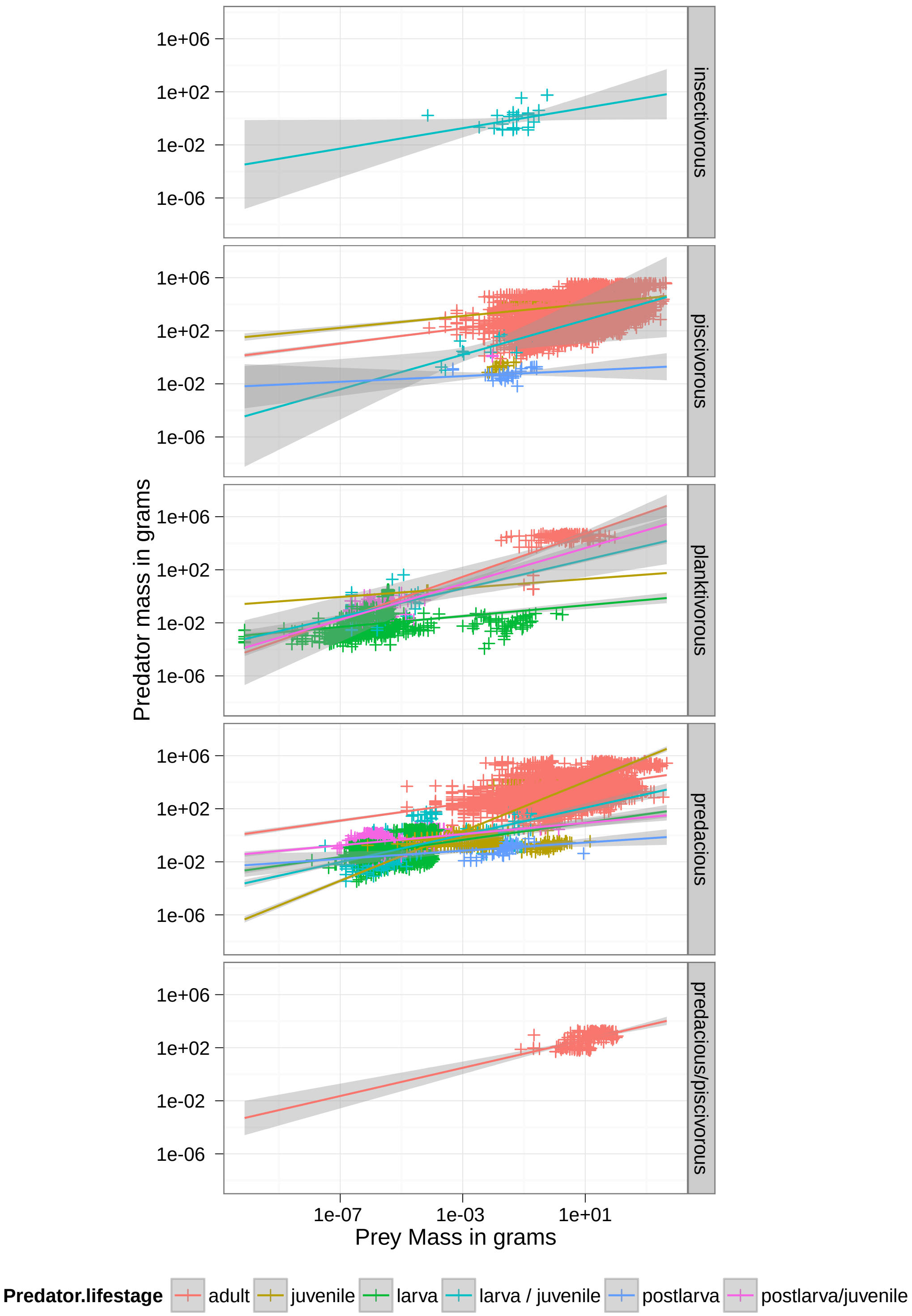


Figure 8.1: Write a script that generates this figure.

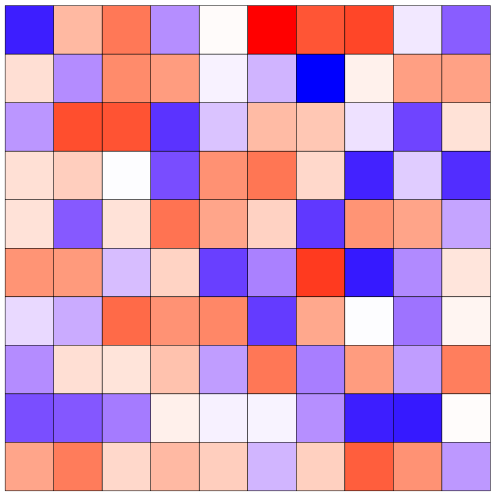


Figure 8.2: Random matrix with values sampled from uniform distribution.

```
require(ggplot2)
require(reshape)

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

> M <- GenerateMatrix(10)

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

> Melt <- melt(M)

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

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

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

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

# removing all the rest
> q <- q + theme(plot.background = element_blank(),
                 panel.grid.major = element_blank(),
                 panel.grid.minor = element_blank(),
                 panel.border = element_blank(),
                 panel.background = element_blank(),
                 axis.title.x = element_blank(),
                 axis.title.y = element_blank(),
                 axis.text = element_blank(),
                 axis.text.x = element_blank(),
                 axis.text.y = element_blank(),
                 axis.ticks = 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"))
```

8.7 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 8.3).

```
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)
```

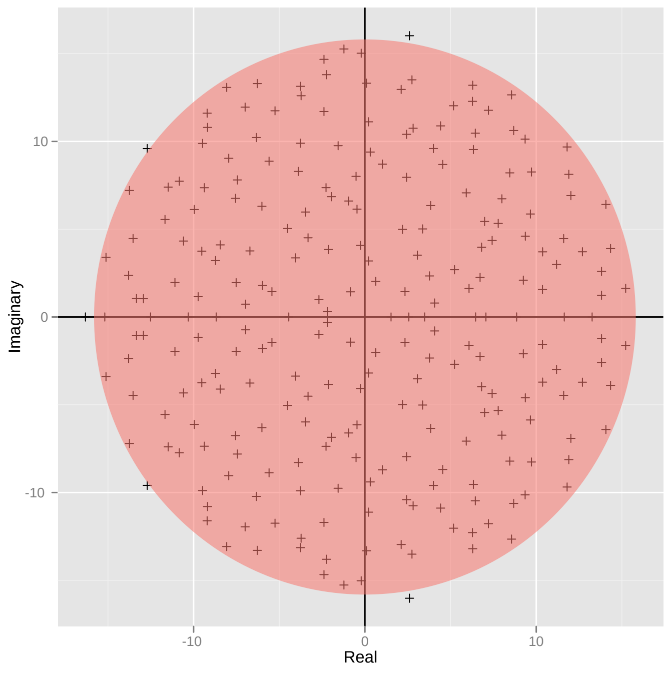


Figure 8.3: Girko's circular law.

```
  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:
```

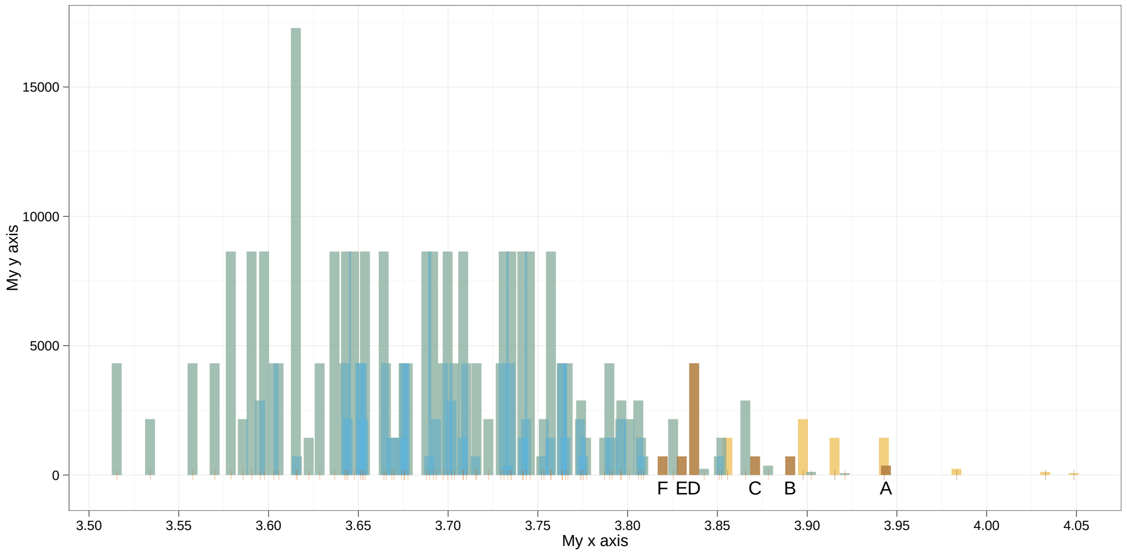


Figure 8.4: Overlay of three lineranges and a text geometry.

```
# 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(intercept = 0))
p <- p + geom_vline(aes(intercept = 0))

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

8.8 Case study 3: annotating the plot

In the plot in Figure 8.4, we use the geometry "text" to annotate the plot.

```r
require(ggplot2)

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

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

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

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

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

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

```

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


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

8.9 Case study 4: mathematical display

In Figure 8.5, you can see the mathematical annotation of the axis and on the plot.

```
require(ggplot2)

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

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

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

# plot the data
p <- ggplot(my_data, aes(x = x, y = y,
                          colour = abs(my_lm$residual))
            ) +
  geom_point() +
  scale_colour_gradient(low = "black", high = "red") +
  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],
```

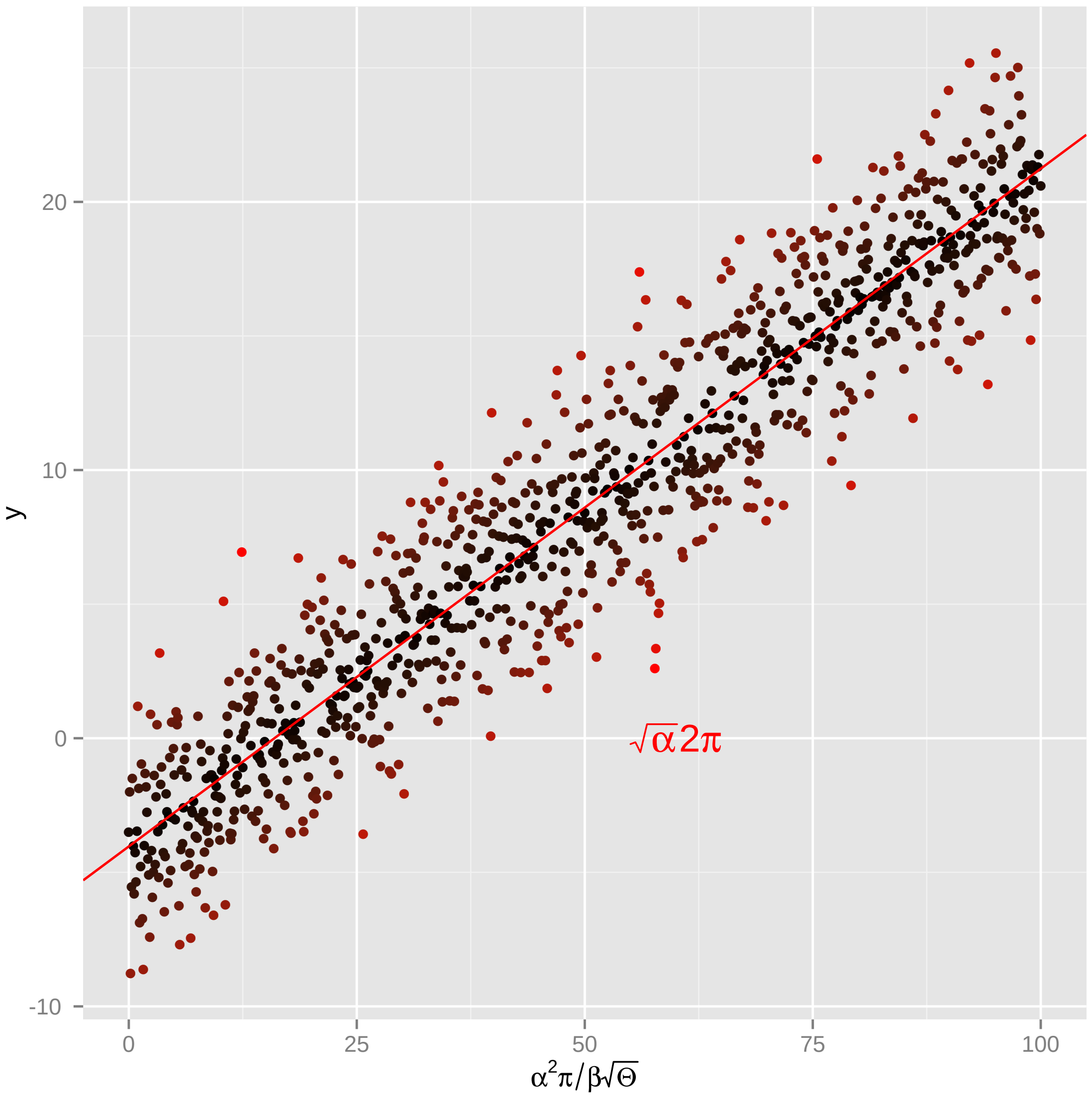
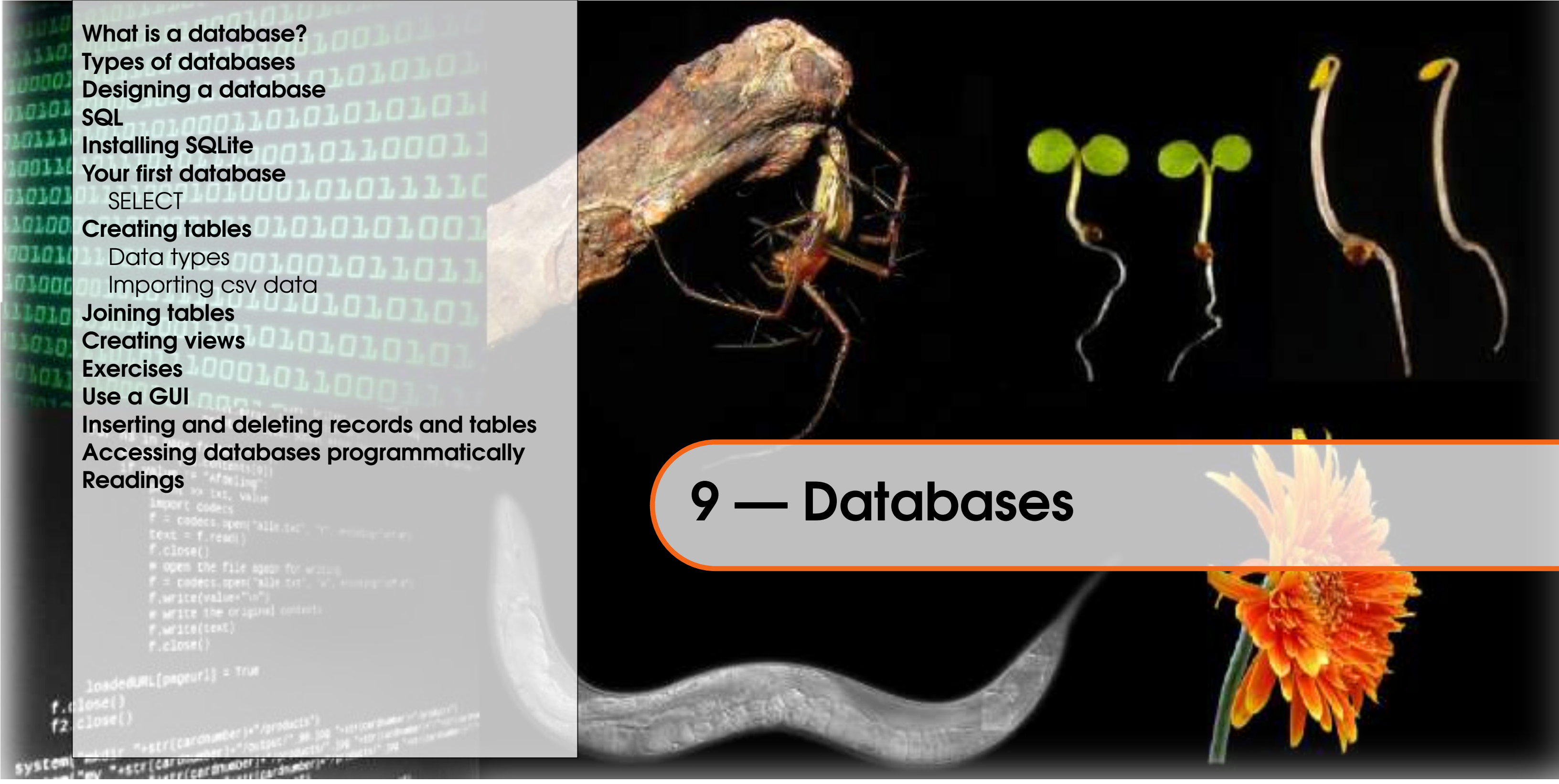


Figure 8.5: Linear regression with colors expressing residuals and mathematical annotations.

```
  slope = my_lm$coefficients[2][1],
  colour = "red")
# throw some math on the plot
p <- p + annotate("text", x = 60, y = 0,
                label = "sqrt(alpha) * 2* pi",
                parse = TRUE, size = 6,
                colour = "red")

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

8.10 Readings

- The classic Tufte `edwardtufte.com/tufte/books_vdqi` (btw, check out what Tufte thinks of PowerPoint!)
- Rolandi `et al.` "A Brief Guide to Designing Effective Figures for the Scientific Paper", doi:10.1002/adma.201102518
- 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
- `nicefigure.org`
- `labtimes.org/labtimes/issues/lt2008/lt05/lt_2008_05_52_53.pdf`
- `gallery.r-enthusiasts.com`
- `stackoverflow.com/questions/12675147/how-can-we-make-xkcd-style-graphs-in-r`



9 — Databases

9.1 What is a database?

If you have lots of data, you should consider storing it into a database. Throughout the class, we stuck to the notion that text is king. However, consider that when you save your data in a text format, each letter/digit typically requires 1 byte (8 bits). As such, the string “128” requires 3 bytes of storage. However, in a binary format you could store a number between 0 and 16777215 (2^{24} - 1) in three bytes. This means that storing the data in a binary format saves you much space. Moreover, for real numbers, you do not loose precision by “printing” the binary format to a text file. Another advantage is that you can divide the data in multiple tables, avoiding redundancies. For example, say that your data is collected at three sampling sites. Then, for each row of a text file you should specify the sampling site. However, you could just specify a number and have the description of the sampling site in another table. Finally, if you have many rows in your text file, the type of sequential access we used for our python code is not going to be efficient: you need to be able to instantly access any row regardless of its position.

A *relational* database is a collection of interrelated tables. The columns are usually called *fields*, while the rows are the *records*. Each field typically contains only one data type (e.g., integers, floats, strings), while each record is a “data point” composed of different values, one for each field. You can think of the record as a python tuple. Some fields are special, and are called *keys*. The *primary key* uniquely defines a record in a table (e.g., each row is identified by a unique number). Some fields (and typically all the keys) are indexed, to allow fast retrieval. *Foreign keys* are keys in a table that are primary keys in another table. They define relationships between the tables.

9.2 Types of databases

There are many software packages to manage databases. The most common open-source choices for large databases are PostgreSQL and MySQL. The most popular commercial database is Oracle. These are highly complex and powerful systems. They typically store the data in a server, and multiple users can interact with them (with different permissions) through clients. They can be set up such that the data can be accessed through the internet.

Although the logic is the same for all databases, the installation of these systems in quite complex (unless you want to install them on your machine for personal use). Therefore, we are going to explore SQLite, which is one of the simplest database available. It is typically designed to store application data. For example, it is used in MacOSX, Firefox, Acrobat Reader, iTunes, Skype and the iPhone, among many others. It is a self-contained, serverless, zero-configuration database engine.

9.3 Designing a database

Database design is a complex task. We want to split the data into logically consistent tables. The guiding principle should be *normalization*: minimize redundancy and dependency.

The easiest way to grasp the concept is to show a bad example and then fix it. Imagine you are collecting body sizes of species across different field sites. You might want to organize a table with the following rows.

One table:

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 the day of the sampling.
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.

However, it would be better 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 field is the primary key. Note that the last table contains three foreign keys, signaling that each individual is associated with one species, one sampling day and one sampling site. This creates one-to-many relationships (one site, many individuals). The structure of a table is called its *schema*.

These are three rules of thumb to create normalized databases:

1. Do not define duplicate fields within a table: each field should contain unique information.
2. Each field in a table should depend on the primary key. This means that a table should contain a logically consistent set of data (e.g., do not mix the sampling site and the sampling date).
3. Tables cannot contain duplicate information. If two tables need a common field, the field should go in a third, separate table.

9.4 SQL

Most modern relational databases let you interface with tables, indices etc. using a common language: SQL, or Structured Query Language. In fact, there are many SQL "dialects", but the simple commands we are going to use will work with all databases.

9.5 Installing SQLite

In Ubuntu, type in a shell:

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

Also, make sure that you have the right package for python:

```
$ ipython
In [1]: import sqlite3
```

Finally, install a GUI for SQLite3:

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

In Mac OS X, SQLite3 should be pre-installed in the OS. Type "sqlite3" in a terminal to check if SQLite successfully launches.

9.6 Your first database

For our first test, we are going to create a database of wood density. The data comes from the Ecology Letters paper by Chave *et al.* (2009). Go to your `SQL/Sandbox` directory and in a terminal type:

Note for Ubuntu: many programs install a version of sqlite3. In case you don't have readline support (i.e., arrows are not working), try to launch `/usr/bin/sqlite3`.

```
$ sqlite3 wood.db
SQLite version 3.7.5
Enter ".help" for instructions
Enter SQL statements terminated with a ";"

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

sqlite> .mode csv

sqlite> .import ../Data/WoodDatabase/wood.csv woodtb
```

Now we have created the table `woodtb` and we have populated it with the following fields:

- Number: primary key – a sequential integer number for each record.
- Family: family for the species, as text (string).
- Binomial: Latin binomial for the species.
- WoodDensity: wood density in grams per cubic centimeter. A real number.
- Region: name of the region.
- Reference number: ID of the corresponding paper.

We then imported the data from a csv into the table. We can list all the tables:

```
sqlite> .tables

woodtb
```

We are now going to run our first *Query*:

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

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

To turn on some nicer formatting:

```
sqlite> .mode column

sqlite> .header ON
sqlite> SELECT * FROM woodtb LIMIT 5;

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

9.6.1 SELECT

The main statement to select records from a table is SELECT.

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

sqlite> SELECT DISTINCT Region FROM woodtb;

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

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

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

sqlite> SELECT COUNT (*) FROM woodtb;

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

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

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

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

COUNT(DISTINCT Famil
--------------------
191

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

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

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

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


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

Number                Family      Binomial       WoodDensity
--------------------  ----------  -------------  -----------
```

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

The structure of `SELECT`:
(*Note: the characters are case **in**sensitive.*)

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

```
sqlite> SELECT Number FROM woodtb LIMIT 5;

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

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

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

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

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

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

```
sqlite> SELECT DISTINCT Region
   ...> FROM woodtb
   ...> WHERE Region LIKE "_ndia";

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

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

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

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

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


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

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

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

```
sqlite> SELECT DISTINCT Region
   ...> FROM woodtb
   ...> WHERE Region GLOB "africa*";

sqlite>
```

We can order by any column:

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

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

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

Binomial              Family
--------------------  -----------
Avicennia alba        Acanthaceae
Avicennia alba        Acanthaceae
Avicennia alba        Acanthaceae
Avicennia germinans   Acanthaceae
Avicennia germinans   Acanthaceae
```

We can use functions within our queries:
- *abs(X)* Absolute value.
- *length(X)* Number of characters.
- *lower(X)* Lowercase.
- *max(X, Y, ...)* Maximum.
- *min(X, Y, ...)* Minimum.
- *round(X,Y)* Round to Y digits.
- *upper(X)* Uppercase.

Other functions work on aggregate data:
- *avg(X)* Average.
- *count(X)* Count.
- *sum(X)* Sum.

Some examples with functions:

```
sqlite> SELECT ROUND(WoodDensity) FROM woodtb LIMIT 3;
```

```
ROUND(WoodDensity)
--------------------
1.0
1.0
1.0

sqlite> SELECT Family, AVG(WoodDensity)
   ...> FROM woodtb
   ...> GROUP BY Family;

Family                AVG(WoodDensity)
--------------------  -----------------
Acanthaceae           0.670722222222222
Achariaceae           0.605288888888889
Actinidiaceae         0.398666666666667
Adoxaceae             0.446333333333333
Akaniaceae            0.561
...

sqlite> SELECT COUNT(DISTINCT Region) AS Regs FROM woodtb;

Regs
--------------------
17

sqlite> SELECT COUNT(DISTINCT Binomial) AS Spp FROM woodtb;

Spp
--------------------
8412

sqlite> SELECT Region, COUNT(DISTINCT Binomial) AS Spp
   ...> FROM woodtb GROUP BY Region;

Region                Spp
--------------------  ----------
Africa (extratropica  320
Africa (tropical)     619
Australia             446
Australia/PNG (tropi  915
Central America (tro  329
....

sqlite> SELECT MAX(LENGTH(Binomial)) FROM woodtb;
```

```
MAX(LENGTH(Binomial)
--------------------
33

sqlite> SELECT MIN(LENGTH(Binomial)) FROM woodtb;

MIN(LENGTH(Binomial)
--------------------
9

sqlite> SELECT AVG(LENGTH(Binomial)) FROM woodtb;

AVG(LENGTH(Binomial)
--------------------
18.7308719941705
```

You can filter the results of a `GROUP BY` using the clause `HAVING`:

```
sqlite> SELECT Region, AVG(WoodDensity)
   ...> FROM woodtb
   ...> GROUP BY Region
   ...> ORDER BY AVG(WoodDensity);

Region                AVG(WoodDensity)
--------------------  -----------------
Europe                0.524896103896104
NorthAmerica          0.537050691244239
China                 0.54089801980198
South-East Asia       0.559223744292238
....

sqlite> SELECT Region, AVG(WoodDensity)
   ...> FROM woodtb
   ...> GROUP BY Region
   ...> HAVING AVG(WoodDensity) > 0.6
   ...> ORDER BY AVG(WoodDensity);

Region                AVG(WoodDensity)
--------------------  ----------------
Oceania               0.6044
South America (tropi  0.63174844571975
Australia/PNG (tropi  0.63592179487179
Africa (extratropica  0.64845584045584
India                 0.65175432525951
....
```

```
# USE "AS"!

sqlite> SELECT Region, AVG(WoodDensity) AS AWD
   ...> FROM woodtb
   ...> GROUP BY Region
   ...> HAVING AWD > 0.7
   ...> ORDER BY AWD;

Region                AWD
--------------------  -----------------
South America (extra  0.714744623655914
Australia             0.724740412979351
```

9.7 Creating tables

In order to build some examples in which we can JOIN tables, we are going to build a larger database of ecological data. The data, on species composition and abundance of mammalian communities, comes from the paper by Thibault *et al.* Ecology 2011.

9.7.1 Data types

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 bitmaps or pdfs).

9.7.2 Importing csv data

SQLite3 has limited functions to import csv data. The basic syntax is the following:

```
sqlite> create table test (id integer, name type, ...);

sqlite> .separator ","

sqlite> .import no_yes.csv test
```

However, beware of:

- 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.

We are going to run a script that creates the tables.

```
/*
Create the table communities
The fields are:

IDcommunity: integer, primary key
IDsite: integer
year: real
IDspecies: text
presence: integer
abundance: real
mass: real
*/

CREATE TABLE communities (
    IDcommunity INTEGER PRIMARY KEY,
    IDsite INTEGER,
    year REAL,
    IDspecies TEXT,
    presence INTEGER,
    abundance REAL,
    mass REAL);

/* Now populate the table from the csv */
.separator ","
.import ../Data/MCDB/communities.csv communities

/*
Create the table references
The fields are:
IDref: text, pk
refer: text
authors: text
pubyear: integer
title: text
source: text
*/

CREATE TABLE reference (
    IDref TEXT,
    refer TEXT,
    authors TEXT,
    pubyear INTEGER,
    title TEXT,
    source TEXT);

/* Now populate the table from the csv */
```

```
.separator ","
.import ../Data/MCDB/references.csv reference


/*
Create the table species
The fields are:
IDspecies text, pk
family text
genus text
sp text
splevel integer
*/

CREATE TABLE species (
    IDspecies TEXT PRIMARY KEY,
    family TEXT,
    genus TEXT,
    sp TEXT,
    splevel INTEGER);


/* Now populate the table from the csv */
.separator ","
.import ../Data/MCDB/species.csv species

/*
Create the table sites
The fields are:
IDsite integer, pk
IDreference text
location text
country text
state text
latitude real
longitude real
*/

CREATE TABLE sites (
    IDsite INTEGER PRIMARY KEY,
    IDreference TEXT,
    location TEXT,
    country TEXT,
    state TEXT,
    latitude REAL,
    longitude REAL);
```

```
/* Now populate the table from the csv */
.separator ","
.import ../Data/MCDB/sites.csv sites
```

I have created the script such that if you run in the `Sandbox`:

```
$ sqlite3 MCDB.db < ../Data/MCDB/script_to_build_database.sql
```

the database will be automatically created and populated.

9.8 Joining tables

When you have multiple tables, you typically want to join them using a common field (foreign key in one table, primary key in the other). For example, in the `communities` table we have a field `IDspecies`, which identifies the species whose information is stored in the `species` table.

```
$ /usr/bin/sqlite3 MCDB.db
SQLite version 3.7.7 2011-06-23 19:49:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"

sqlite> .mode column

sqlite> .header ON

sqlite> SELECT * FROM communities LIMIT 2;

IDcommunity  IDsite      year        IDspecies   presence
-----------  ----------  ----------  ----------  ----------
1            1008        2002.0      CHPE        0
2            1008        2002.0      CHSX        0

sqlite> SELECT * FROM species LIMIT 2;

IDspecies   family       genus       sp          splevel
----------  -----------  ----------  ----------  ----------
ABBE        Abrocomidae  Abrocoma    bennettii   1
ABLO        Cricetidae   Abrothrix   longipilis  1

sqlite> SELECT IDcommunity, year, species.IDspecies,
   ...> family FROM
   ...> communities INNER JOIN species
   ...> ON communities.IDspecies = species.IDspecies
   ...> LIMIT 2;

IDcommunity  year        IDspecies   family
```

```
-----------  ----------  ----------  ------------
1            2002.0      CHPE        Heteromyidae
2            2002.0      CHSX        Heteromyidae
```

The INNER JOIN, the most common and useful type of join between tables, works as the intersection between two sets: we will take all the records from community whose IDspecies is present in the species table, and pair them.

```
sqlite> SELECT IDcommunity, refer, country FROM
   ...> communities INNER JOIN sites ON
   ...> communities.IDsite = sites.IDsite
   ...> INNER JOIN reference ON
   ...> sites.IDreference = reference.IDref
   ...> ORDER BY country LIMIT 25;

IDcommunity  refer       country
-----------  ----------  ----------
2738         BioOne      Argentina
2739         BioOne      Argentina
2740         BioOne      Argentina
2741         BioOne      Argentina
2742         BioOne      Argentina
2743         BioOne      Argentina
2744         BioOne      Argentina
```

9.9 Creating views

Because we do not want to type all of the queries above more than once, we can create a view: a virtual table that is generated at runtime. In this way, we can keep tables joined in the view, while having them materially divided in the database (and thus removing redundancy). We can query the view as we would query a table.

For example, this code would connect all the tables using inner joins:

```sql
SELECT * FROM
communities cc
INNER JOIN sites ss
ON cc.IDsite = ss.IDsite
INNER JOIN species sp
ON cc.IDspecies = sp.IDspecies
INNER JOIN reference rr
ON ss.IDreference = rr.IDref;
```

To create a view simply precede the code by CREATE VIEW bigtb AS:

```sql
CREATE VIEW bigtb AS
SELECT * FROM
```

```sql
communities cc
INNER JOIN sites ss
ON cc.IDsite = ss.IDsite
INNER JOIN species sp
ON cc.IDspecies = sp.IDspecies
INNER JOIN reference rr
ON ss.IDreference = rr.IDref;
```

```
sqlite> .tables

bigtb         communities  reference    sites         species

sqlite> SELECT COUNT(DISTINCT country) FROM bigtb;

COUNT(DISTINCT country)
-----------------------
40

sqlite> SELECT country, COUNT(DISTINCT family)
   ...> FROM bigtb GROUP BY
   ...> country ORDER BY country;

country     COUNT(DISTINCT family)
----------  ----------------------
Argentina   6
Australia   7
Benin       3
Bolivia     3
Brazil      18
Canada      10
Central Af  2
....
```

9.10 Exercises

Count the number of genuses for each degree of longitude (use round). Notice that we found an error!

```
LON           COUNT(DISTINCT genus)
------------  ---------------------
-114968494.0  9
-134.0        8
-133.0        4
-132.0        8
-131.0        5
-128.0        4
```

```
....
```

Count the number of species for each country and state. Note the descending order!

```
country     state       NUMSP
----------  ----------  ----------
USA         CA          77
USA         NM          75
Brazil      NULL        62
USA         AZ          55
USA         MN          54
....
```

9.11 Use a GUI

By now, you should know that I have a very nasty allergy to Graphical User Interfaces. However, to build complex queries they can help quite a bit. There are very many GUIs for SQLite. Some popular, cross-platform ones are `Sqliteman` and the Firefox extension `SQLite Manager` and `SQLite Studio`.

9.12 Inserting and deleting records and tables

Insert a record into a table:

```
INSERT INTO table_name
VALUES (value1, value2, value3,...)
```

Insert a record and set only some columns:

```
INSERT INTO table_name (column1, column2, column3,...)
VALUES (value1, value2, value3,...)
```

To change the value of one or more records:

```
UPDATE table_name
SET column1=value, column2=value2,...
WHERE some_column=some_value
```

To delete one or more records:

```
DELETE FROM table_name
WHERE some_column=some_value
```

To completely remove a table:

```
DROP TABLE table_name
```

9.13 Accessing databases programmatically

It is very easy to access, update and manage a database from python. In fact, this is what you should do to manage complex databases:

```python
# import the sqlite3 library
import sqlite3

# create a connection to the database
conn = sqlite3.connect('example.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)''')

# 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()
```

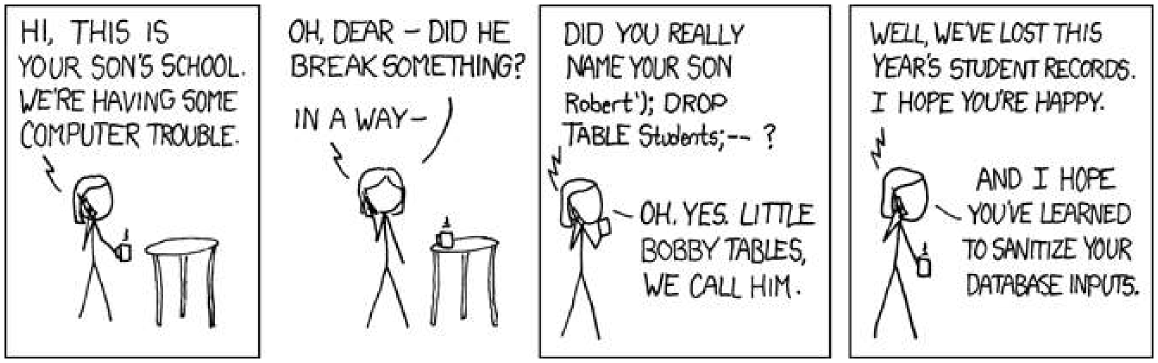
```python
# insert many records at once:
# create a list of tuples
manyrecs = [(5, 'goofy'),
           (6, 'donald'),
           (7, 'duck')]

# 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()
```

A cool trick is that you can create a database in memory, i.e. without accessing the disk. In this way you can use SQL within your programs!

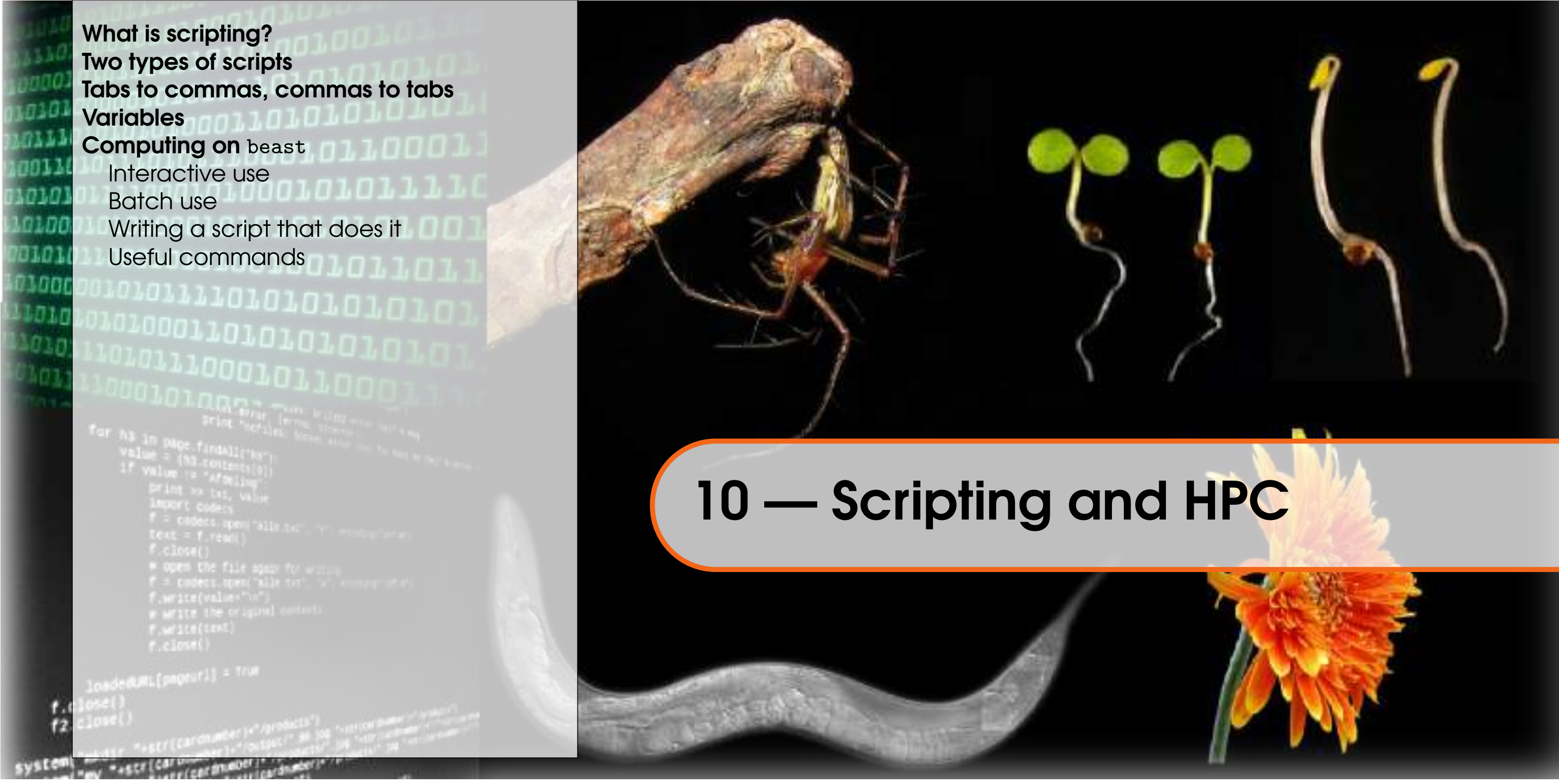
```python
import sqlite3

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()
```

9.14 Readings

The library has an electronic copy of “The Definitive Guide to SQLite”, which, as the name implies, is a pretty complete guide to SQLite.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

`http://xkcd.com/327/`



10 — Scripting and HPC

10.1 What is scripting?

Instead of typing all the 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 flows. In fact, most data manipulation can be handled by scripts without the need of writing a actual program. Scripts can be used to automate repetitive tasks, to do simple data manipulation or to perform maintenance of your computer (e.g., backup).

10.2 Two types of scripts

There are two ways of running a script, say`myscript.sh`. The first is to call the interpreter bash to run the file:

```
$ bash myscript.sh
```

Otherwise, we can make the script executable, and execute the script:

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

I typically choose one or the other, depending on the task the script performs:

- A script that does something specific in a given project: I will have an informative name for the script, leaving the script in the `Code` directory of the project. The script is not executable, and will be called using `bash` + name of the script. i.e., the script is a program like the others I wrote for the project.
- A script that does something generic (e.g., a shortcut for commands I have to perform several times), whose existence is not essential for a project (i.e., you can share the project without sharing the script). I use an informative name without the final `.sh`. I save the script in `sallesina/bin/` and make it executable.

Let's prepare the bin directory to house our generic scripts:

```
$ mkdir ~/bin
$ PATH=$PATH:$HOME/bin; export PATH
```

The last command tells UNIX to look in the directory bin for commands, and adds it to the list of directories that are searched.

10.3 Tabs to commas, commas to tabs

As an example, I often have to transform comma-separated files to tab-separated files and vice-versa (in C, it is much easier to read tab or space separated files than csv). There is a program called tr that can be used to delete or substitute characters. For example:

```
$ 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
```

We write small scripts to handle the conversion of files. We start from a boilerplate:

```bash
#!/bin/bash
# Author: Stefano Allesina sallesina@uchicago.edu
# Script: boilerplate.sh
# Desc: simple boilerplate for scripts
# Arguments: none
# Date: Sept 2012

echo "This is a script"

exit
```

A script to substitute all tabs with commas:

```bash
#!/bin/bash
# Author: Stefano Allesina sallesina@uchicago.edu
# 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: Dec 2012

echo "Creating a comma delimited version of $1"

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

exit
```

And, similarly, a script to convert csv into space delimited:

```
#!/bin/bash
# Author: Stefano Allesina sallesina@uchicago.edu
# Script: csvtospace.sh
# Desc: substitute the commas in the file with spaces
# saves the output into a .space file
# Arguments: 1-> comma delimited file
# Date: Dec 2012

echo "Creating a space delimited version of $1"

cat $1 | tr -s "," " " >> $1.space

exit
```

10.4 Variables

There are three ways to assign values to variables:

- Explicit declaration: `MYVAR=myvalue`.
- Reading from the user: `read MYVAR`
- Command substitution:

  ```
  MYVAR=$( (ls | wc -l) )
  ```

Note that in the first case, no spaces can be around the “=” sign! This is quite annoying, if you’re used to other languages. In general, the syntax of bash scripting is quite clunky. Here are some examples of assignments:

```
#!/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
```

Note the hideous quote marks before `expr`. I find the fastidious syntax of bash really annoying, and thus I rarely use it to do serious work (it is much faster to write the script in python!). Anyway, for example open an editor and save a new file `MyFirstScript.sh` containing the following:

```
#!/bin/bash

echo "Hello $USER!"
echo
```

To run, go in the right directory, and type `bash MyFirstScript.sh`.

Now, we are going to write a script that counts the lines in a file (much like `wc -l`). To do so, we will pass to the script a command-line argument.

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

To launch, type `bash CountLines.sh AFileName`. We stored the result from the command `wc -l` in a variable. This is a container that we can use to store values (in this case, the number of lines). To print a variable, we use the $ sign. The special variable `$1` is the first argument we passed to the program CountLines. Similarly, the second argument would be `$2` etc.

To understand this better, let's write a program that concatenates two files.

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

And a program that compiles our LaTeX files and cleans up the temporary files (this is a script I really use often!).

```
#!/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
```

10.5 Computing on `beast`

E&E and OBA students can get access to `beast.uchicago.edu`. Students in other department can access other computer clusters. If you have a user and a password, you can get a terminal by connecting to the machine:

```
$ ssh sallesina@beast.uchicago.edu
######################################################################
######################################################################

               Welcome to the Ecology & Evolution Grid

          Unauthorized use of this machine is prohibited.

   This is a University machine intended for University purposes.

The University reserves the right to monitor its use as necessary to
        ensure its stability, availability, and security.
######################################################################
######################################################################

sallesina@beast$
```

A computer cluster is often organized as follow: there are several machines (*nodes*), each with several processors (*cores*), connected with each other using high-throughput wiring (e.g., infiniband). Users can access only one of the nodes, the so-called *head node*. The head node runs specific software to *queue* programs and *schedule* their execution.

The idea is that users submit their program as *jobs*, which are then queued. When its turn arrives, a job is then sent from the head node to one of the computing nodes, it is executed and then the output is made available to the user. Contrary to what you experience when you launch a program locally, the execution is not immediate, but could be deferred (sometimes for several days). Some users find this very annoying, but in fact it is a sign that the cluster is operating at capacity, i.e., that it is working as it should. The scheduler enforces fair use policies, to guarantee that no-one can take over the grid. Some documentation is available at *beast.uchicago.edu*.

10.5.1 Interactive use

The simplest use of the cluster is to ask for an interactive node. This basically gives you access to a computer to run your programs interactively, as you would do on your PC. To launch an interactive session, type:

```
$ sallesina@beast$ qsub -I
qsub: waiting for job 268469.beast-net to start
```

The wait can be pretty long, depending on how busy the cluster is. However, this is the only acceptable way to use the cluster interactively: **Never run your programs on the head node!** In fact, this could bring the cluster down for everybody. The only programs you should run on the head node are scripts to submit jobs.

10.5.2 Batch use

You typically want to submit your programs as jobs and queue them. Eventually, they will all run. For each job, you have to write a bash script with this structure:

```
#!/bin/bash
#PBS -N myname
#PBS -q batch
#PBS -d .
#PBS -j oe
#PBS -S /bin/bash
python myprogram.py 123 mickey 456 mouse
```

The meaning is the following:

- *PBS -N myname* Choose a name for your job. The shorter and more informative, the better.
- *PBS -q batch* Choose a queue for your job. Beast has two queues: batch, which has access to more resources, and can run very long jobs, and short, which has fewer nodes and can run shorter jobs only (up to a week of computing).
- *PBS -d .* Use the current directory as the working directory.
- *PBS -j oe* Concatenate error and output messages for you to read.
- *PBS -S /bin/bash* Use bash for running the script.
- *python myprogram.py...* The actual command to be run: for R programs use `Rscript MyFile.R`. For bash scripts use `bash MyScript.sh`, etc.

Once you have written your job files, you can submit them using the `qsub` command:

```
$ sallesina@beast$ qsub jobname.sh
```

You can set the priority: set low priorities and be nice when you are submitting many jobs:

```
sallesina@beast$ qsub -p -100 jobname.sh
```

Priority goes roughly from -1000 to +1000. The default is zero, which works fine if you are submitting few jobs.

You can set the queue also when submitting:

```
sallesina@beast$ qsub -q batch jobname.sh

sallesina@beast$ qsub -q short jobname.sh
```

10.5.3 Writing a script that does it

I rarely write the job script or submit jobs by hand. Rather, I write a small python, R, or bash program that does it for me.

```
#!/usr/bin/python
#######################################################
## THINGS TO MODIFY
DataFile = "Test35_household_12364_21.txt"
NR = 12364 # Number of rows
NC = 21 # Number of columns

MinLinks = 56 # Minimum number of links to use
MaxLinks = 70 # Maximum number of links to use

NumberSteps = '1000000' # Note: use a string!

AndOr = 'both' # (possible values: 'and', 'or', 'both')
MinSeed = 1 # smallest seed to use
MaxSeed = 5 # largest seed to use
## END THINGS TO MODIFY
#########################################################
import os
import sys
import time

# Build scripts
for L in range(MinLinks, MaxLinks + 1):
    for S in range(MinSeed, MaxSeed + 1):
        # Choose a name for the job
        JNAME = 'H' + DataFile[0:5] + '-' + str(L) + '-' + str(S)
        # Open the file for writing
        f = open(JNAME + ".sh", "w")
        # Build the line to run the program
        launchstrand = "python FindHierarchy-Dictionary.py %s %d %s And %d ↩
            MaxLik None" %(DataFile, L, NumberSteps, S)
        launchstror = "python FindHierarchy-Dictionary.py %s %d %s Or %d ↩
            MaxLik None" %(DataFile, L, NumberSteps, S)
        if AndOr == 'both':
```

```
        f.write("#!/bin/bash\n#PBS -N %s \n#PBS -d . \n#PBS -j oe \n#↩
            PBS -S /bin/bash \n%s\n%s\n" % (JNAME, launchstrand, ↩
            launchstror))
    elif AndOr == 'and':
        f.write("#!/bin/bash\n#PBS -N %s \n#PBS -d . \n#PBS -j oe \n#↩
            PBS -S /bin/bash \n%s\n" % (JNAME, launchstrand))
    elif AndOr == 'or':
        f.write("#!/bin/bash\n#PBS -N %s \n#PBS -d . \n#PBS -j oe \n#↩
            PBS -S /bin/bash \n%s\n" % (JNAME, launchstror))
    # Close the file
    f.close()

    # Now submit the script
    os.system('qsub %s' % JNAME + ".sh")
    # And wait 3 seconds to let the queue handle the submission
    time.sleep(1)
```

10.5.4 Useful commands

To check the status of all jobs, use qstat:

```
sallesina@beast$ qstat  | head
Job id                    Name             User
------------------------- ---------------- ---------------
200164.beast-net           STDIN            ccchen
200347.beast-net           script1.sh       xuzhang
201696.beast-net           mymaxbatch.sh    egoldwyn
201930.beast-net           STDIN            ccchen
201965.beast-net           STDIN            ccchen
236303.beast-net           mymaxbatch.sh    egoldwyn
257614.beast-net           STDIN            cgmeyer
.....
```

Only your jobs:

```
sallesina@beast$ qstat  -u mjsmith | head

beast-net:

Job ID                 Username Queue    Jobname
-------------------- -------- -------- ----------------
268142.beast-net      mjsmith  batch    HCarne-176-3
268143.beast-net      mjsmith  batch    HCarne-177-1
268144.beast-net      mjsmith  batch    HCarne-177-2
268145.beast-net      mjsmith  batch    HCarne-177-3
268146.beast-net      mjsmith  batch    HCarne-178-1
```

Count how many jobs are running or queued:

```
sallesina@beast$ qstat | wc -l
697
sallesina@beast$ qstat -u mjsmith | wc -l
159
sallesina@beast$ qstat -r | wc -l
205
sallesina@beast$ qstat -ru mjsmith  | wc -l
27
```

To delete specific jobs from the queue, use `qdel`

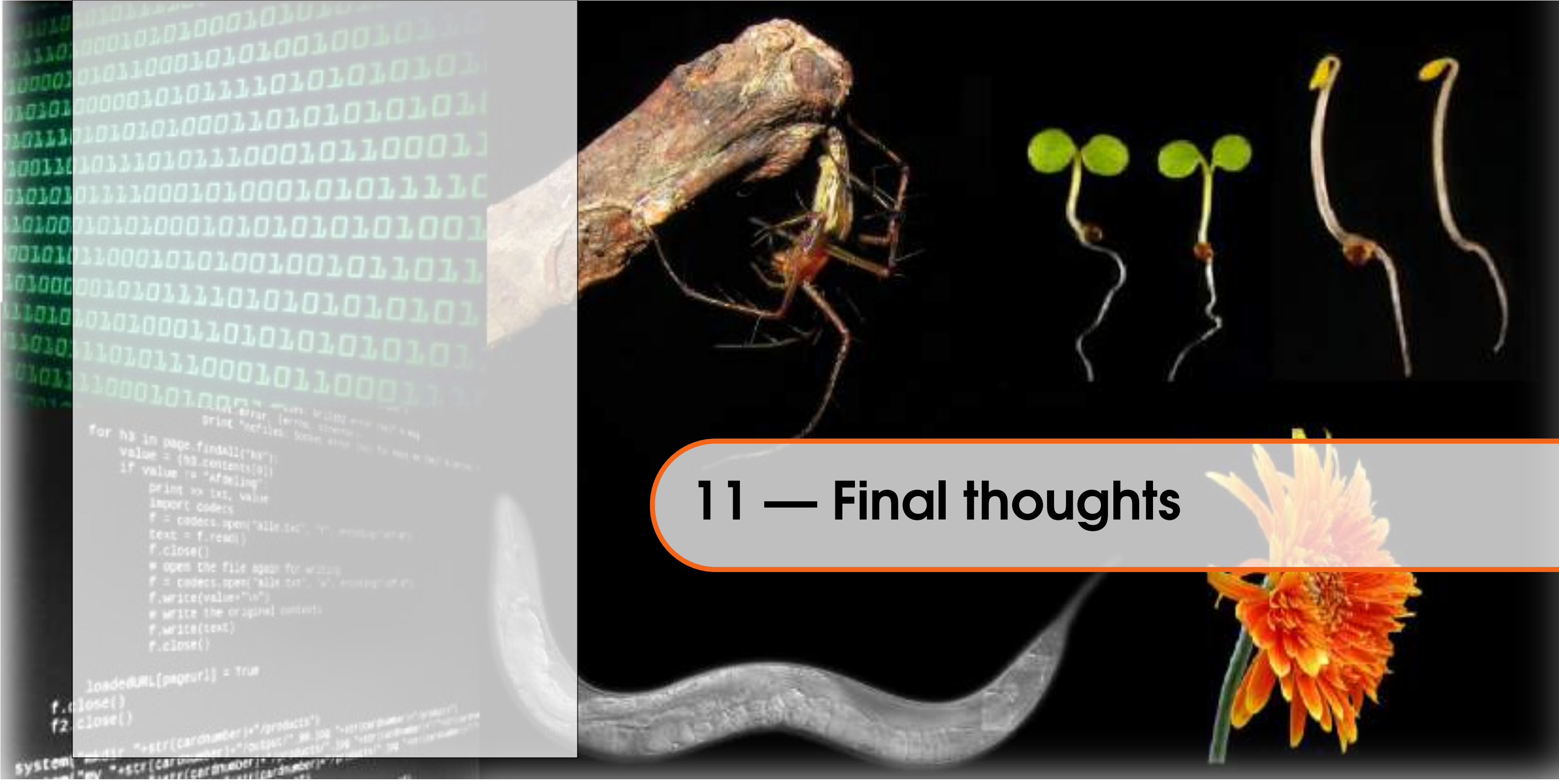
```
sallesina@beast$ qdel 268142 # This is the job ID from qstat

sallesina@beast$ qdel all # This will delete all your jobs
```

To check the status of the nodes:

```
sallesina@beast$ pbsnodes
minion0
     state = free
     np = 8
     properties = short
     ntype = cluster
     jobs = 0/268459.beast-net, 1/268460.beast-net
....
```

Check the status of the queues:

```
allesina@beast$ qstat -q

server: beast

Queue            Memory CPU Time Walltime Node  Run Que Lm  State
---------------- ------ -------- -------- ----  --- --- --  -----
short              --      --       --      --   13   0 --   E R
asreml             --      --       --      --    0   0 --   E R
batch              --      --       --      --  420 549 --   E R
OBA                --      --       --      --    9  48 --   E R
test               --      --       --      --    0   0 --   E R
                                               ----- -----
                                                 442   597
```



11 — Final thoughts

We have seen many tools and techniques that can make your life as a scientist easier (or more complicated!). You might have fell in love with some of the topics, and found others dull and boring. All of them respond to specific needs, and have been developed with certain problems in mind. The most powerful tool you now have is the possibility to integrate all these techniques to solve your scientific problems in a robust, reliable and easy-to-automate way. Some advice:

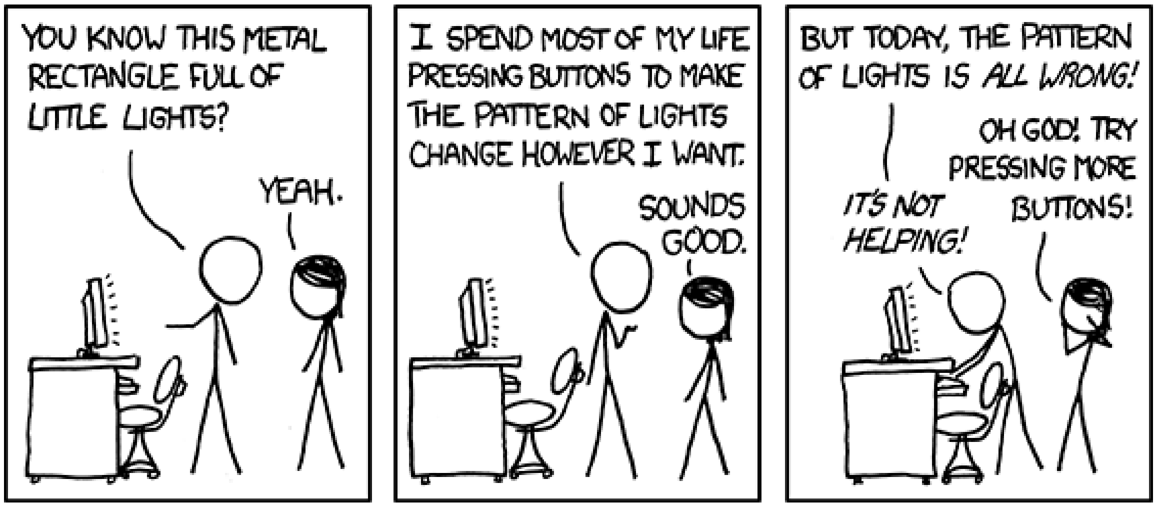
- **Divide a project into sub-projects that are easy to solve.** Never rely on a single program to carry out the whole analysis from start to finish. Rather, divide it into logical steps, and tackle each step separately. This will help you recycle lots of code, besides being a more efficient way of working.
- **Choose the right tool.** You could be tempted to do everything in `python`, or `C`, or `R`. Don't! Think about what is the best way of doing something. Do not be afraid of learning new things. Think medium-to-long term (while you're in graduate school, you can afford it!).
- **Automate.** Computers err in systematic ways, while you don't. It is easier to fix systematic mistakes than random ones. If it takes you X hours to do it by hand, and 3X hours to automate it, then automate it.
- **Document and organize.** Keep your pipeline streamlined, well-documented and well-organized. It can take months to get reviews back, and you don't want to forget what you did, how or why.
- **Do not hack/fix your code.** Do not make any extemporaneous change: you are almost surely introducing bugs.
- **Plan ahead, do little at a time.** Coding, and especially debugging, is quite boring. Find a good time to do it, and don't do it for too many consecutive hours.
- **Find a friend.** If you can convince someone, have them sit next to you while programming and explain them step-by-step what you are doing and why.

Please provide any feedback you might have: this will make the class a better experience for your colleagues next year. What did you find useful? What was too difficult or boring? Also, if you have some data/programs which would make nice exercises, let me know and I will include them in the next version.

Thank you all for the wonderful time in class, and good computing!

Chicago, IL
January 2014

Stefano Allesina
sallesina@uchicago.edu


YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS *ALL WRONG!*
OH GOD! TRY PRESSING MORE BUTTONS!
IT'S NOT HELPING!

`http://xkcd.com/722/`