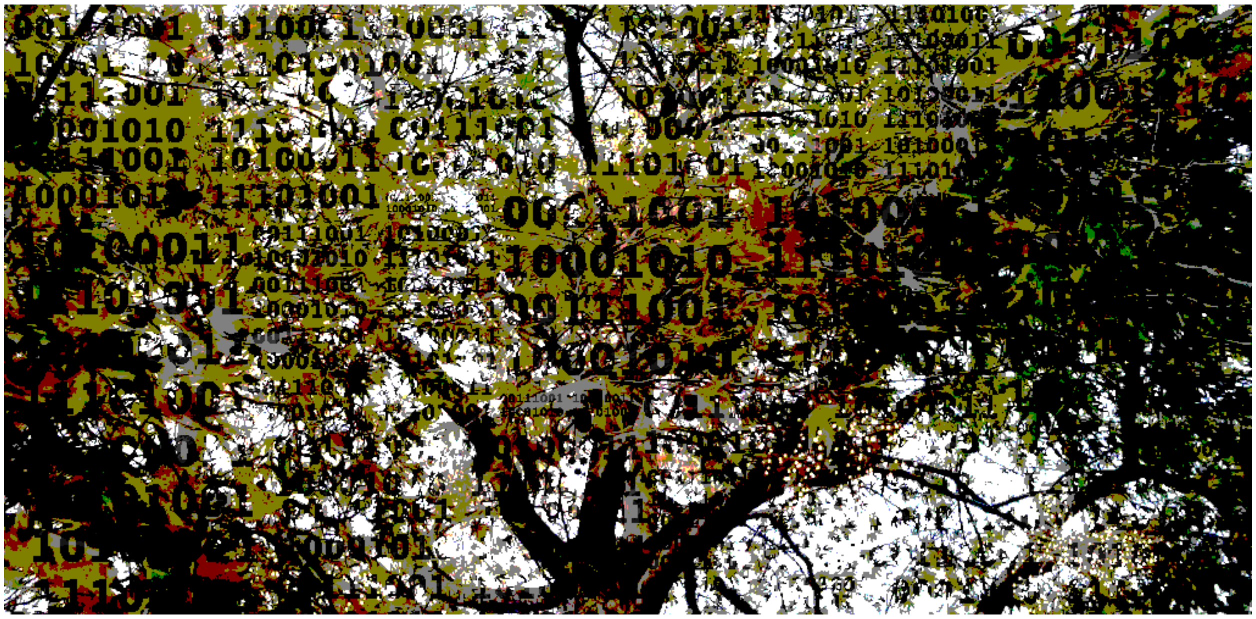


The Biologist's Guide to Computing

Release 0.9.1

Tjelvar S. G. Olsson

December 30, 2016

Contents

CHAPTER 1

Preface

1.1 Who should read this book

Biologists that need to use the command line. If you want to learn how to use the command line to install and run software this book is for you. The command line is used throughout the book and you will quickly gain familiarity with the most important commands. You will also learn how to install software and how to work on remote machines. The latter will be important if you want to run bioinformatics software on your institutes high performance cluster.

Biologists that want to create their own data analysis scripts. If you want to learn how to write your own data analysis scripts this book is also for you. The book starts off by explaining fundamental computing concepts, teaching you how to think like a computer. You will then learn how to use Python by creating a script to analyse the guanine-cytosine (GC) content of a bacterial genome. There is also a chapter on data visualisation that teaches you how to work with R. Furthermore, programming best practises are highlighted and explained throughout the book.

Biologists that want to ensure their data analysis is reproducible. If you want to ensure that your data analysis is reproducible this book is also for you. Early on you will learn how to use version control to track changes to your projects. Furthermore, the concept of using automation to ensure reproducibility is explored in detail.

No prior knowledge required. Important concepts and jargon are explained as they are introduced. No prior knowledge is required. Just a willingness to learn, experiment and have fun geeking out.

1.2 Licence

This work uses the Creative Commons (CC0 1.0)[1] licence. So you can copy, modify, distribute and perform the work, even for commercial purposes, without having to ask for permission.

1.3 Source code

The source[2] for this book is hosted on GitHub[3].

[1] http://creativecommons.org/publicdomain/zero/1.0/
[2] https://github.com/tjelvar-olsson/biologists-guide-to-computing
[3] https://github.com/

1.4 Feedback

This book is still a work in progress. I would really appreciate your feedback. Please send me an email[4] to let me know what you think. Alternatively you can contact me via Twitter @tjelvar_olsson[5], @bioguide2comp[6] or message me via the Facebook page[7].

If you want to receive updates about this book do sign up to the mailing list. You can find the sign up form on the website: biologistsguide2computing.com[8].

1.5 Acknowledgements

Thanks to Nadia Radzman, Sam Mugford and Anna Stavrinides for providing feedback on early versions of the initial chapters. Many thanks to Tyler McCleary for continued in depth feedback and suggestions for improvements. Thanks to Nick Pullen for feedback and discussions on the data visualisation chapter. Many thanks to Matthew Hartley for discussions and encouragement.

I'm also grateful for feedback from Lucy Liu and Morten Grøftehauge.

[4] tjelvar@biologistsguide2computing.com
[5] https://twitter.com/tjelvar_olsson
[6] https://twitter.com/bioguide2comp
[7] https://www.facebook.com/biologistsguide2computing
[8] http://biologistsguide2computing.com/

CHAPTER 2

Getting some motivation

2.1 Why you should care about computing and coding

Computers can be powerful allies. They are ideal for automating repetitive tasks. Furthermore, they can perform calculations and analysis that are too complex for the human brain to process.

These days many parts of the biological sciences are becoming more and more data driven. Technological advancements have led to a huge increase in the generation of biological data. Data analysis is required to extract biological insights from this data. To a large extent the rate limiting factor in generating insight is the lack of appropriate data analysis tools.

However, as a biologist you may prefer to work in the lab over sitting in front of the computer. You may even ask yourself if learning to program has any benefit to you whatsoever.

Similar to learning a new language there are several benefits to learning computer programming. Importantly it gives you a new perspective. In fact professional programmers are encouraged to learn a new programming language every year to allow them to experience new problem solving techniques.

More concretely, learning to program will help you get a better understanding of data analysis. It may even expand your mind to allow you to see new possibilities available to you.

Furthermore, by getting more familiar with computing your interactions with programmers and data analysis folk will improve. So even if you discover that you don't like programming learning to code will help you express your ideas to people that do.

2.2 What you can expect to learn here

Many books on computing assume that the reader has a basic knowledge of how a computer operates and the associated jargon. This book attempts not to make any such assumptions and important concepts are explained in How to think like a computer (page 5). Furthermore, important terms and their background are explained throughout in attempt to demystify computing jargon. This should stand you in good stead when faced with more advanced material on computing.

Use of the command line is required for many bioinformatics tools and access to high-performance computing clusters. The essence of the Unix command line, used in both Linux and Macs, is introduced in First steps towards automation (page 13). After which the command line is used persistently to interact with a variety of useful programs. At the end you should feel comfortable using the command line to interact with your computer and to automate tasks.

Data forms a central part of the biological sciences. However, dealing with large volumes of data can feel overwhelming. One way to regain some control is to ensure that there is structure to the data. In Structuring and storing data (page 23) important data structures and file formats for representing them are discussed. The chapter also highlights the value of using plain text files to store data. After reading this chapter you should have a good idea of how to store structured representations of your data.

Sooner or later one wishes that one had done a better job of keeping records of one's work. What changes were made when? What were the reasons behind the decisions to make those changes? These issues becomes even more accentuated when working on scripts and programs where tiny changes can have devastating effects. Because this is such an important issue in software development there are great tools for mitigating against it. In Keeping track of your work (page 31) the concept of version control is introduced using the popular open source tool Git. After working through the examples in this chapter you should feel comfortable using Git to keep track of your work.

Being able to create data analysis scripts is empowering. In Data analysis (page 43) the Python scripting language is introduced and we create a script for calculating the local GC content of the *Streptomyces coelicolor* A3(2) genome. After working through this chapter you will have a basic knowledge of how to use Python to read in data from a file, manipulate it and write out results to another file. The chapter also introduces more general programming concepts such as variables, functions and loops.

Data visualisation is an important part of scientific investigation. In Data visualisation (page 61) the statistical scripting language R is used to look at how one can explore data interactively from the command line and create scripts that make the generation of figures reproducible. After working through this chapter you will have a gained familiarity in working with R and an understanding of how to use R's ggplot2 package to create beautiful and informative figures.

The best science often involves collaboration. In Collaborating on projects (page 83) you will learn how to use make use of the powerful collaboration features of Git and you will learn how to back up and share your project using the free hosting services on GitHub.

Communication is an important aspect of science and one aspect of scientific communication is writing manuscripts. In Creating scientific documents (page 95) you will learn how lightweight text documents can be converted to beautiful scientific manuscripts with citations and bibliographies using Pandoc. You will also learn how you can use Pandoc to convert (almost) any document file format into any other.

After having spent weeks and months analysing data, generating figures and assembling all the information into a manuscript it can be devastating to find that one needs to start all over again because an updated data set has become available. However, this needn't be the case. If all your analysis, figure generation and manuscript assembly was automated you would just need to replace the raw data and press "Go". In Automation is your friend (page 103) you will learn how to achieve this state of bliss.

When tackling more complex data analysis one needs to spend more time thinking about the problem up front. In the Practical problem solving (page 107) chapter we will look at techniques for breaking problems into smaller and more manageable chunks. The chapter will also extend your familiarity with Python, and introduces the concepts of string manipulation and regular expressions.

At some point you may need to work on a remote computer. For many this may be when one needs access to the institute's high performance computing cluster. In Working remotely (page 129) you will learn how to use the command line to log in to a remote computer and how to copy data to and from the remote machine.

Installing software is not particularly exciting. However, it is a means to an end. In Managing your system (page 137) we go over various methods of installing software. The chapter also introduces some fundamental Unix-based systems administration concepts required to understand what is needed to install software successfully.

Finally the book ends with Next steps (page 147), a short chapter giving some suggestions on how to continue building your knowledge of scientific computing.

CHAPTER 3

How to think like a computer

These days computers can perform complex task. However, at their core computers only have a relatively small set of basic capabilities. Information is stored as zeros and ones and Boolean logic is used to perform calculations.

This chapter will cover fundamental aspects of computing required to learn scripting and programming. It will introduce bits and bytes and how they can be used to represent numbers and characters (as in letters and symbols). The chapter will also give a brief overview of how a computer stores information and performs calculations.

3.1 Binary, bits and bytes

In computing a bit is the smallest unit of information and it can be either zero or one. Suppose that we wanted to represent a positive integer using zeros and ones. We can achieve this using using the base-2 numeral systems, a.k.a. binary.

Integer

An integer is a whole number, i.e. a number that can be expressed without a fractional component. In computing people often make a distinction between signed and unsigned integers. The former includes negative values and the latter does not.

Take for example the number 108. This number can be represented as two sets of hundreds and eight sets of ones (as well as zero sets of tens).

$$\begin{aligned}108 &= 100 + 0 + 8 \\ &= (1 * 100) + (0 * 10) + (8 * 1) \\ &= (1 * 10^2) + (0 * 10^1) + (8 * 10^0)\end{aligned}$$

The equations above illustrate the basis of the base-10 numeral system. Note that the base of the exponent in each term is ten. This means that one needs ten states to represent numbers. However, computers only have two states 0 and 1. To express an integer in binary we therefore need to formulate it so that the base of each exponent is 2.

$$\begin{aligned}108 &= 64 + 32 + 0 + 8 + 4 + 0 + 0 \\ &= (1 * 64) + (1 * 32) + (0 * 16) + (1 * 8) + (1 * 4) + (0 * 2) + (0 * 1) \\ &= (1 * 2^6) + (1 * 2^5) + (0 * 2^4) + (1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (0 * 2^0)\end{aligned}$$

Bit

In computing the term “Bit” is an abbreviation of the term “Binary digIT”.

So to express 108 as binary we could use the bits `1101100`.

Table 3.1: Representing the integer 108 in binary.

64	32	16	8	4	2	1
1	1	0	1	1	0	0

In computing bits are often grouped together and the most common grouping is to bundle 8-bits together, this is known as a byte. One byte can therefore have 256 unique states. An unsigned 8-bit integer, stored in a byte, can therefore represent any integer ranging from 0 to 255.

To recap, in computing a bit is the smallest unit of information and it can be either zero or one, and a byte is eight bits grouped together. Using several bits we can express integers using the binary numeral system.

3.2 Character encodings

If a computer only works in terms of zeros and ones how can one use it to read and write text? This is the subject of character encodings.

One of the most prevalent character encodings is ASCII (American Standard Code for Information Interchange), which was first published in 1963. It has its roots in telegraphic codes. Based on the English alphabet it encodes 128 characters into 7-bit integers (Fig. 3.1). These characters include: 0-9, a-z, A-Z, the space character, punctuation symbols and control codes for operating Teletype machines. Although many of the control characters are now obsolete some of them still form the basis of how we interpret text in computers. For example the “line feed” character, which originally moved the paper in the printer forward, is now used to represent new lines in text documents.

In going from Teletype machines to computers different operating systems had different interpretations on how new lines should be represented. One surviving convention, still in use in Windows, requires both the “line feed” and the “carriage return” characters. The former representing the advancement of paper in the printer and the latter the repositioning of the print-head. Another convention, used in Unix-like systems, is to simply use the line feed character to represent new lines. This can result in compatibility issues when opening Unix-like text files in some Windows programs. For example when one opens a multi-line file with Unix-based line endings in Notepad the text appears like one long line.

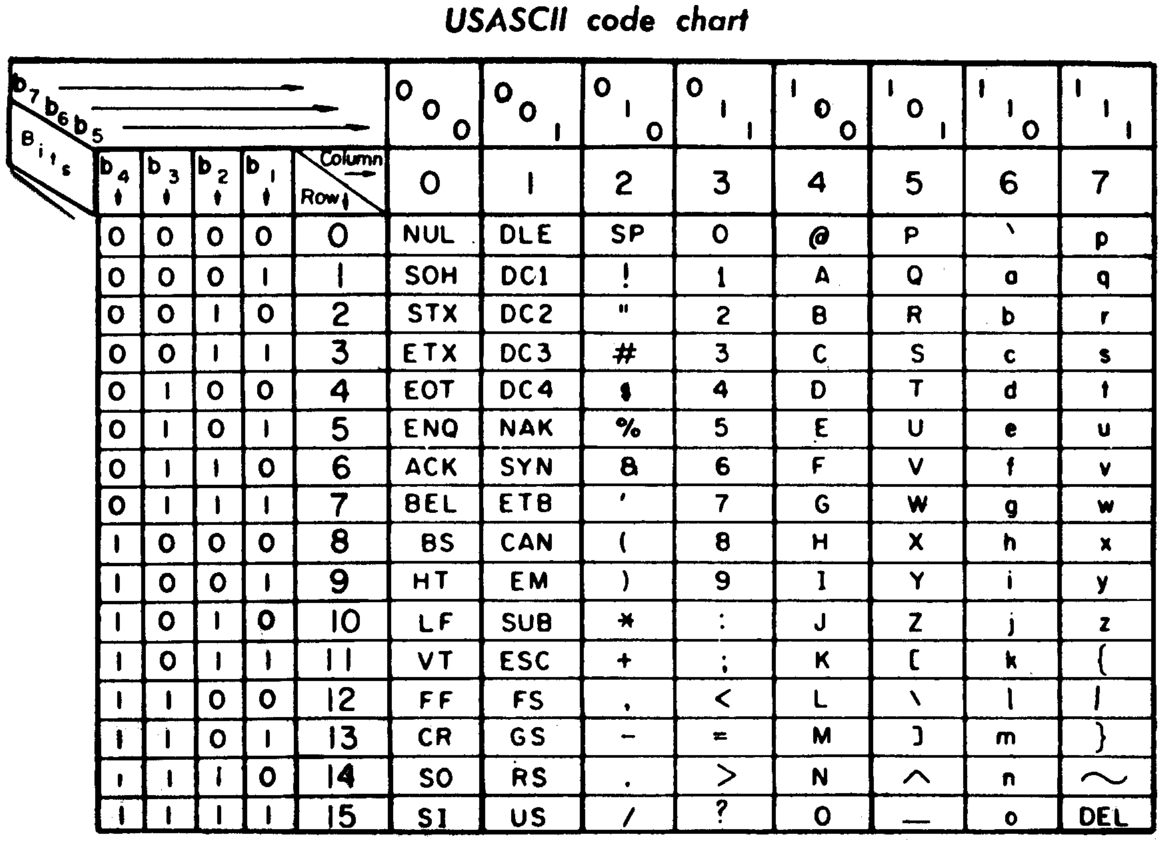
Many other character encodings are based on ASCII. A famous example is ISO-8859-1 (Latin 1), which uses an eighth bit to expand the ASCII character set to include the remaining characters from the Latin script.

These days the emergent standard is Unicode, which provides a standard encoding for dealing with most of the worlds writing systems. Unicode is backwards compatible with ASCII and ISO-8859-1 in that Unicode’s code points, the natural numbers used to assign unique characters, have the same values as those of the ASCII and ISO-8859-1 encodings.

Natural number

A natural number is a whole non-negative number. In computing such a number is commonly referred to as an unsigned integer.

[9] https://commons.wikimedia.org/wiki/File%3AASCII_Code_Chart-Quick_ref_card.png



USASCII code chart

Bits b_7 b_6 b_5					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Column / Row	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	−	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

Fig. 3.1: ASCII Code Chart, scanner copied from the material delivered with TermiNet 300 impact type printer with Keyboard, February 1972, General Electric Data communication Product Dept., Waynesboro VA. By Namazu-tron [Public domain], via Wikimedia Commons[9].

So in computers "text" is really just a convention that maps integers to characters, or more formally a character encoding.

3.3 Real numbers

So we have learnt that integers can be represented in computers with zeros and ones by making use of the binary numeral system. Let us now discuss how real numbers can be represented and some issues that one can come across when working with these.

Real numbers

Real numbers include rational numbers such as integers and fractions as well as irrational numbers such as the $\sqrt{2}$ and π.

One way of storing and working with real numbers is to use the fixed-point number representation. A fixed-point number is basically an integer that is scaled by an implicit factor. So if the implicit scaling factor was $1/10$ and the explicit integer was 127 then the real number represented would be 12.7.

The fixed-point number representation has the disadvantage that the range of numbers that can be represented is relatively small. Consider an unsigned 8-bit integer. It can represent the range 0 to 255. With an implicit scaling factor of $1/10$ it can represent the numbers 0.0 to 25.5 with a step size of 0.1. With an implicit scaling factor of $1/100$ it could represent the numbers 0.00 to 2.55 with a step size of 0.01.

The floating-point number was invented to work around the inherent small range limitation of fixed-point number representations. Floating-point numbers basically allow the decimal (radix) point to float. This means that numbers of differing orders of magnitude can be expressed using the same units. It is similar to scientific notation where the distance to the moon can be expressed as $3.844 * 10^8$ and the size of a typical bacterium can be expressed as $1.0 * 10^{-6}$. A consequence of this is that the numbers that can be expressed are not uniformly spaced, i.e. as the size of the exponent increases the step size between two representable numbers increases.

Floating point number

Floating point is a means to represent real numbers. The term refers to the fact that the radix point of the number is allowed to "float" relative to the significant digits of the number. For example the numbers 12.7^{-10}, 12.7 and 12.7^7 could all be expressed by floating the radix point around the significant digits 127.

Not all numbers can be represented precisely using floating-point numbers. Furthermore, arithmetic operations on floating-point numbers cannot truly represent arithmetic operations. This can lead to issues with accuracy. We can illustrate this using Python (we will get more details on scripting and Python in the Data analysis (page 43) chapter).

```
>>> 0.6 / 0.2
2.9999999999999996
```

In the above a rounding error of $4 * 10^{-16}$ has been introduced as a consequence of working with floating point representations of numbers.

3.4 Boolean logic

Boolean logic is a mathematical formalism for describing logical relations. In Boolean logic things are either `True` or `False`. These truth values are often represented as 1 and 0 respectively. There are three basic operators `AND`, `OR` and `NOT` for working with truth values. These are sometimes referred to as logic gates.

x	y	x AND y	x OR y
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

x	NOT x
0	1
1	0

Using these axioms more complex logic gates can be built up. For example, by combining `NOT` and `AND` one can create what is commonly referred to as a `NAND` gate.

x	y	x AND y	NOT (x AND y)
0	0	0	1
1	0	0	1
0	1	0	1
1	1	1	0

Importantly one can use Boolean logic gates to implement integer arithmetic and memory. This combined with the fact that it is relatively easy to physically implement Boolean logic using relays led to the construction of the first computers.

Although you may not want to create your own computer, having a basic understanding of Boolean logic will help you when implementing algorithms. For example one often needs to make conditional logic statements along the lines of "`IF` the cell contains green fluorescent protein `AND` it is illuminated with light of wavelength 395 nm `THEN` it will emit fluorescence at 509 nm".

Note: Boolean logic is also used in Boolean networks, a formalism that can be used to describe gene regulatory networks at the protein expression level, i.e. mRNA and protein concentrations are not considered. The expression level of each gene is considered to be either on or off, a.k.a. 1 or 0. The *state* of the model is the set of Boolean values used to describe the gene network transcription levels at that point in time. Inhibition and activation are modeled by creating Boolean rules, which can be further combined using Boolean logic. By iteratively applying the Boolean rules the dynamics of the system can be evaluated over time. As time progresses the number of states of the network decreases as the system is driven towards a relatively small number of dynamic cycles and stable states, known as attractors. The attractors often correspond to specific differentiated states of cells in biological systems.

3.5 The microprocessor

A microprocessor executes machine instructions. Machine instructions tell the microprocessor what to do. At the most basic level there are three things that a microprocessor can do: perform mathematical operations, move data from one memory location to another, make decisions and jump to new sets of instructions based on those decisions.

The C programming language

C is a popular programming language designed by Dennis Ritchie in 1972. It is a low-level language, which means that it allows the programmer to work close to the hardware by providing direct access to the system's memory. One of the most famous C projects is the Linux kernel, which is a massive open source project with millions of lines of code and thousands of contributors.

Most programming languages provide some sort of abstraction layer so that the programmer does not need to think in terms of machine instructions. For example, the main purpose of a C compiler is to convert C source code into machine instructions.

When working with higher level languages, such as Python, one does not really need to worry about what happens at the microprocessor level.

However, knowing that a microprocessor can make decisions and jump to new sets of instructions can be useful when trying to understand concepts such as loops. A loop is essentially a set of machine instructions that end with a decision to jump back to the beginning of the same set of instructions.

Loops often include a condition for exiting the loop. If the condition for exiting the loop is not defined, or it cannot be reached, the loop will keep cycling forever in what is termed an "infinite loop".

Below is a basic C program illustrating a while loop. The loop terminates when the integer `i` is no longer less than 3.

```c
1  int main () {
2     int i = 0;
3     while( i < 3 ) {
4        i = i + 1;
5     }
6     return 0;
7  }
```

In more detail, the lines 1 and 7 define a small set of code that forms the *main* part of the program. Line 2 defines an integer, `i`, and sets it equal to 0. Lines 3 and 5 define a *while loop* to iterate as long as `i` is less than 3. Line 4 executes a command to change the value of `i`. Line 6 states that the main program should return an exit status 0.

What the exit status of a program?

The exit status of a program is a means for the program to state if it was terminated normally or abnormally. The exit status 0 means that the program ended normally. Any other exit status means that the program was terminated abnormally.

3.6 Computer memory

Computer memory comes in different forms with different characteristics. The hard drive of a computer is a type of memory where data can be stored permanently. RAM (Random Access Memory) is a type of memory where data is volatile, i.e. it is not retained when the machine reboots. A less well known type of memory is the registry, which resides in the CPU (Central Processing Unit). Being physically close to the CPU means that reading and writing of data to the registry is very fast. Other important characteristics of computer memory include the its size and cost. Below is a table summarising these characteristics.

Location	Speed	Size	Cost	Volatile
Registry	Very fast	Very small	Very expensive	Yes
RAM	Fast	Small/Moderate	Expensive	Yes
Hard disk	Slow	Very large	Cheap	No

One could liken these types of storage to different cooling devices in the lab. The minus 80 freezer is large and represents a way to store DNA plasmids and primers persistently. When preparing for a cloning experiment one may get samples of plasmids and primers and store them in the fridge in the lab for easy access. However, although the lab fridge provides easy access it does not present a permanent storage solution. Finally, when performing the cloning experiment one takes the samples out of the fridge and places them in an ice-bucket. The ice-bucket is a "working storage" solution to which one can have very fast access. In the example above the minus 80 freezer represents the hard disk, the lab fridge represents RAM and the ice bucket represents the registry.

If one is working with really large data sets the main bottleneck in the processing pipeline can be reading data from and writing data to memory. This is known as being *I/O* (input/output) bound.

3.7 Key concepts

- A bit is the smallest piece of data that can be stored in a computer, it can be set to either zero or one
- A byte is 8-bits
- An unsigned 8-bit integer can represent any of the integers between 0 and 255
- A character encoding maps characters to integers, common character encodings include ASCII and Unicode
- Real numbers tend to be handled using floating-point representation
- There are some inherent limitations when working with floating-point numbers which can lead to issues with accuracy
- Boolean logic is one of the main tools employed by the computer to do work and to store data
- A microprocessor executes machine instructions
- Machine instructions can tell the microprocessor to perform mathematical operations, move data around and to make decisions to jump to new sets of machine instructions
- The hard disk, RAM and the register are different types of memory with different characteristics
- Processes that spend most of their time reading and writing data are said to be *I/O* bound

CHAPTER 4

First steps towards automation

One of the first steps towards automating repetitive tasks is to become familiar with the command line. In this chapter we will accomplish this by using the command line to extract UniProt identifiers for all human proteins in Swiss-Prot.

More precisely we will be using a terminal emulator to run Bash. A terminal emulator is an application that gives you access to another program known as the shell. The shell allows you to interact with the operating system's services and programs. The most widely used shell and the default on Mac and most Linux based distributions is Bash.

You will get the most benefit from this chapter if you work through the example as you go along. You are therefore encouraged to open up a terminal now (Fig. 4.1)!

On Macs the command line is available through the Terminal application. There are a number of terminal emulators available for Linux. If you are using the Gnome-based desktop the default is likely to be the Gnome Terminal.

On Windows things are a little bit more complicated. Windows does not come bundled with Bash by default. I would recommend installing VirtualBox[10] and running a Linux distribution such as BioLinux[11] in it. VirtualBox is a so called *hypervisor* that lets you run a *virtual machine*. In this scenario you would run BioLinux as a virtual machine. For more information on how to run BioLinux as a virtual machine see the BioLinux installation notes[12].

The benefit of learning how to use Linux/Mac and Bash is that most bioinformatics software are developed on and designed to run on these platforms. Furthermore, the terminal and Bash provides an excellent way of creating analysis pipelines.

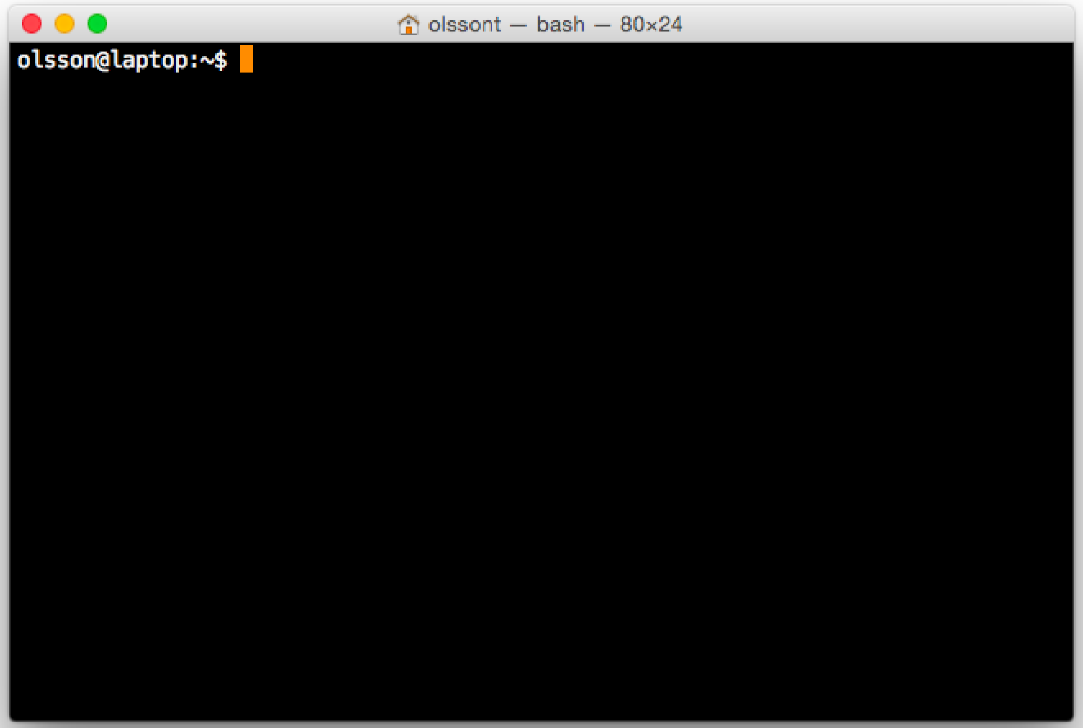
4.1 First things first, how to find help

Most of the commands on the command line have built in help that can be accessed by providing either the argument `-h` or `--help`. For example to access help for the `gunzip` command, which we will use later, enter the command below into the shell and press enter:

```
$ gunzip --help
```

More descriptive documentation can usually be found using the `man` (manual) command. For example to view the manual page of the `ls` command you can run the command below.

[10] https://www.virtualbox.org/
[11] http://environmentalomics.org/bio-linux-download/
[12] http://environmentalomics.org/bio-linux-installation/



```
olsson@laptop:~$
```

Fig. 4.1: Bash shell running in a Terminal application. The prompt displays the the name of the user (olsson), the name of the machine (laptop) and the current working directory (~). Note that the dollar symbol ($) is an indication that input is expected, i.e. the shell expects commands to be entered after it. The prompt displayed in the Bash shell can be customised. As such the prompt on your system may look different.

```
$ man ls
```

To get out of the "man-page" press the "q" key. Just like I encourage you to try out the examples outlined in this chapter, I also encourage you to examine the help and man-page documentation for the commands that we use to get a better understanding of what the commands do.

4.2 Creating a new directory for our project

First of all let us make sure that we are working in our home directory.

```
$ cd
```

The `cd` command, short for *change directory*, is used to move between directories. If called without a path to a directory it will move you into your home directory.

We can *print* out the name of the current *working directory* using the `pwd` command. Furthermore we can *list* the contents of a directory using the `ls` command.

```
$ pwd
/home/olsson
$ ls
```

Now that we know where we are and what files and directories are present let us create a new directory for our project. This is achieved using the `mkdir` command, short for *make directory*. After having created the directory move into it using the `cd` command.

```
$ mkdir first_step_towards_automation
$ cd first_step_towards_automation
```

Note: When using the command line one learns to avoid using white spaces in file and directory names. This is because white spaces are used to separate arguments. In the example above we used underscores instead of white spaces. However, one could just as well have used hyphens. This comes down to personal preference. It is possible to represent file names with spaces in them on the command line by using the backlash character (\) to "escape" the whitespace, for example `first\ steps\ towards\ automation` or by surrounding the text in quotes `"first steps towards automation"`.

4.3 Downloading the Swiss-Prot knowledge base

UniProt (Universal Protein Resource) is a comprehensive resource of protein sequences and annotations. The UniProt Knowledgebase (UniProtKB) consists of Swiss-Prot and TrEMBLE. Both are annotated. However, the procedure in which they are annotated differ. TrEMBLE uses an automatic annotation system, whereas the annotation in SwissProt is manual and includes a review process.

It is time to download the Swiss-Prot knowledge base from UniProt. We will use the `curl` program to do this. The `curl` command is a C program that allows us to stream data from URLs and FTP sites. By default the `curl` program writes the content of the URL to the *standard output stream*. To see this in action try running the command:

```
$ curl www.bbc.com
```

You should see a whole lot of HTML text appearing in your terminal window.

However, because we are going to download a large file we would like to write it to disk for future use. Many command line programs allow the user to specify additional options. In this particular case we can use the `--output` option to specify a file name that the output should be written to. To exemplify this let us download the BBC home page to a file named `bbc.html`.

```
$ curl --output bbc.html  www.bbc.com
```

Here we will use a URL shortened using bitly[13] to save on typing. The shortened URL contains a redirect to the relevant Swiss-Prot FASTA file hosted on the UniProt FTP site. To find out where the shortned URL redirects to run the command:

```
$ curl http://bit.ly/1l6SAKb
```

What is URL shortening?

URL shortening is a means to make URLs shorter whilst still directing the client to the desired page. It is achieved by using a redirect from a domain that is short, to the page with the longer URL. To view the HTTP redirect code, `301 Moved Permenantly`, you can use `curl`'s verbose option.

```
$ curl --verbose http://bit.ly/1l6SAKb
```

To allow the redirection to occur we need to use the `--location` option, which will redirect the request to the new location. Let us download the gzipped FASTA file from the UniProt FTP site:

```
$ curl --location --output uniprot_sprot.fasta.gz http://bit.ly/1l6SAKb
```

The downloaded file `uniprot_sprot.fasta.gz` has been compressed using the `gzip` protocol. We can extract it using the `gunzip` command. However, when extracted it more than doubles in size. So we will use the `--to-stdout` option to extract the content to the standard output stream whilst leaving the original file compressed.

Try running the command:

```
$ gunzip --to-stdout uniprot_sprot.fasta.gz
```

You should see a lot of FASTA lines printed to your terminal, or more formally the standard output stream.

What is a FASTA file?

FASTA is a simple file format for storing nucleotide or peptide sequences. It consists of a single-line description, starting with the greater than symbol (>), and a sequence which can be spread over several lines.

```
>TATA box
TATAAA
>Pribnow box
TATAAT
```

Options starting with two dashes, `--`, are known as long options. Many of these long options also have abbreviated "short" options. For example, the `-c` option of `gunzip` is equivalent to the `--to-stdout` option. Try running the command:

```
$ gunzip -c uniprot_sprot.fasta.gz
```

From now on the text will use the short `-c` option rather than the long `--to-stdout` option to save on typing.

[13] https://bitly.com/

Note: Remember that you can use the `--help` or `-h` option to get information on the meanings of the various options available to you.

Tab completion

Another way to save on typing in the terminal is to use tab completion. Start typing the command that you want to use and hit the tab key. For example, type in `gu` and then press the tab key. This will complete the command to `gunzip` without you having to do any more typing. If the letters typed into the terminal are not sufficient to specify a unique command it will list all of the possible options.
You can also use tab completion to fill in the names of existing files and directories.

4.4 Creating a work flow using pipes

Now it is time to introduce one of the greatest features of the command line: pipes! Pipes are a means to redirect the output from one command into another. The character used to represent a pipe is the vertical bar: `|`.

To illustrate the use of pipes we will redirect the output of the previous `gunzip` command to the word count program `wc`. Try running the command below:

```
$ gunzip -c uniprot_sprot.fasta.gz | wc
```

Re-using previous command

Rather than having to retype commands try using the "Up" and "Down" arrows to get access to previous commands.

It should give you three numbers, these are the line, word and character counts. To only see the line count one could use the `-l` option:

```
$ gunzip -c uniprot_sprot.fasta.gz | wc -l
```

Pipes are powerful because they allow a set of simple commands to be combined to perform tasks that are beyond the scope of any of the individual commands. This has led to a central Unix philosophy of having simple programs that do one task well and a rich ecosystem of such programs. The user is then free to combine these programs to create personalised tools to automate repetitive processing tasks.

Another powerful feature of pipes is that the program being piped to gets access to the output stream of data from the program piping data into the pipe as soon as it is available. This means that the processing of data can happen in parallel.

4.5 Examining files, without modifying them

Unix-based systems make a distinction between programs that are used for examining files, known as pagers, and programs that are used for editing files, known as text editors. The reason for making this distinction is to help prevent accidental changes to files when reading them.

To view the beginning of a file one can use the `head` command. Let us examine the first lines of the `uniprot_sprot.fasta.gz` file by pipeing the output of the `gunzip` command into `head`:

```
$ gunzip -c uniprot_sprot.fasta.gz | head
```

You should see something like the output below being written to the terminal window.

Listing 4.1: First ten lines of the `uniprot_sprot.fasta.gz` file. Note that the identifier lines have been truncated to only display the first 65 characters.

```
>sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R OS=Frog ...
MAFSAEDVLKEYDRRRRMEALLLSLYYPNDRKLLDYKEWSPPRVQVECPKAPVEWNNPPS
EKGLIVGHFSGIKYKGEKAQASEVDVNKMCCWVSKFKDAMRRYQGIQTCKIPGKVLSDLD
AKIKAYNLTVEGVEGFVRYSRVTKQHVAAFLKELRHSKQYENVNLIHYILTDKRVDIQHL
EKDLVKDFKALVESAHRMRQGHMINVKYILYQLLKKHGHGPDGPDILTVKTGSKGVLYDD
SFRKIYTDLGWKFTPL
>sp|Q6GZX3|002L_FRG3G Uncharacterized protein 002L OS=Frog virus ...
MSIIGATRLQNDKSDTYSAGPCYAGGCSAFTPRGTCGKDWDLGEQTCASGFCTSQPLCAR
IKKTQVCGLRYSSKGKDPLVSAEWDSRGAPYVRCTYDADLIDTQAQVDQFVSMFGESPSL
AERYCMRGVKNTAGELVSRVSSDADPAGGWCRKWYSAHRGPDQDAALGSFCIKNPGAADC
```

The beauty of the `head` command is that it allows you to quickly view the beginning of a file without having to read in the content of the entire file. The latter can present a real problem if working on "big data" files. In fact, this is also the beauty of pipes, which allows downstream programs to work on the stream of data without having to wait for it to be written to or read from disk.

By default the `head` command writes out the first ten lines. However, this can be modified using the `-n` option, for example to write out the first 20 lines:

```
$ gunzip -c uniprot_sprot.fasta.gz | head -n 20
```

Similarly, there is a `tail` command for displaying the tail end of a file, again ten lines by default.

```
$ gunzip -c uniprot_sprot.fasta.gz | tail
```

You may have noticed that the workflow above, to view the last ten lines, took a little longer to complete. That is because we needed to decompress the whole file before we could access the last ten lines of it.

To page through an entire file one can use the `less` command.

```
$ gunzip -c uniprot_sprot.fasta.gz | less
```

One can use the "Up" and "Down" arrows to navigate through the file using `less`. One can also use the "Space" key to move forward by an entire page, hence the term pager. To page back one page press the "b" key. When you are finished examining the file press "q" to quit `less`.

How am I supposed to be able to remember that `less` is a pager?

As you may have noticed, if one does not use a pager, the standard output is simply written to the terminal. This can be frustrating if the file is large and one wants to start reading at the top of the file and then page through it as one reads along. This is what pagers are for, moving through files one page at a time. One of the original pager programs was called `more`. It simply displayed one page of output at a time and when one wanted "more" output one simply pressed the space key. A usability issue with the `more` program was that it did not allow users to go back up a page. The `less` pager was therefore developed to work around this issue. It implemented reverse scrolling and a number of other additional features not present in `more`. However, `less` also implemented all the original features of the `more` program, resulting in the mnemonic "less is more".

4.6 Finding FASTA identifier lines corresponding to human proteins

Now that we have an idea of what the file looks like it is time to extract the FASTA identifiers that correspond to human proteins.

A powerful command for finding lines of interest in text is the `grep` program, which can be used to search for strings and patterns. Let us use it to search for the string "Homo":

```
$ gunzip -c uniprot_sprot.fasta.gz | grep Homo | less
```

To make the match more visible we can add the `--color=always` option, which will highlight the matched string as red.

```
$ gunzip -c uniprot_sprot.fasta.gz | grep --color=always Homo | less
```

If you scroll through the matches you will notice that we have some false positives. We can highlight these by performing another `grep` command that finds lines that do not contain the string "sapiens", using the `--invert-match` option or the equivalent `-v` short option.

```
$ gunzip -c uniprot_sprot.fasta.gz | grep Homo | grep -v sapiens
```

To make the search more specific we can search for the string "OS=Homo sapiens". To do this we need to surround the search pattern by quotes, which tells the shell that the two parts separated by a white space should be treated as one argument.

```
$ gunzip -c uniprot_sprot.fasta.gz | grep "OS=Homo sapiens"
```

To work out how many lines were matched we can pipe the output of `grep` to the `wc` command.

```
$ gunzip -c uniprot_sprot.fasta.gz | grep "OS=Homo sapiens" | wc -l
```

4.7 Extracting the UniProt identifiers

Below are the first three lines identified using the `grep` command.

Listing 4.2: First three lines of the `uniprot_sprot.fasta.gz` file identified using the `grep` command. Note that the lines have been truncated to only display the first 65 characters.

```
>sp|P31946|1433B_HUMAN 14-3-3 protein beta/alpha OS=Homo sapiens ...
>sp|P62258|1433E_HUMAN 14-3-3 protein epsilon OS=Homo sapiens GN=...
>sp|Q04917|1433F_HUMAN 14-3-3 protein eta OS=Homo sapiens GN=YWHA...
```

Now that we can identify description lines corresponding to human proteins we want to extract the UniProt identifiers from them. In this instance we will use the command `cut` to chop the line into smaller fragments, based on a delimiter character, and print out the relevant fragment. The delimiter we are going to use is the vertical bar ("|"). This has got nothing to do with pipeing, it is simply the character surrounding the UniProt identifier. By splitting the line by "|" the UniProt id will be available in the second fragment.

The command below makes use of the backslash character (\) at the end of the first line. This tells bash that the command continues on the next line. You can use this syntax in your scripts and in the terminal. Alternatively, you can simply include the content from both lines below in a single line, omitting the `\`.

```
$ gunzip -c uniprot_sprot.fasta.gz | grep "OS=Homo sapiens" \
| cut -d '|' -f 2
```

In the above the `-d` option specifies the delimiter to use to split the line, in this instance the pipe symbol (`|`). The `-f 2` option specifies that we want to extract the second field.

Note: Remember to try out these commands on your computer to see the actual output of the commands.

4.8 Using redirection to create an output file

By default the output from commands are written to the standard output stream. Earlier we saw that we could use the pipes to redirect the output to another command. However, it is also possible to redirect the output to a file, i.e. save the output to a file. This is achieved using the greater than symbol (>). You can use the idea of an arrow as a mnemonic, the output is going from the command into the file as indicated by the arrow.

```
$ gunzip -c uniprot_sprot.fasta.gz | grep "OS=Homo sapiens" \
| cut -d '|' -f 2 > human_uniprot_ids.txt
```

Now if you run the `ls` command you will see the file `human_uniprot_ids.txt` in the directory and you can view its contents using `less`:

```
$ less human_uniprot_ids.txt
```

The < redirection command

There is a third type of redirection <. This type of redirection is so common that it is often made implicit. The two commands below, for example, are equivalent.

```
$ wc -l < human_uniprot_ids.txt
   20197
$ wc -l human_uniprot_ids.txt
   20197 human_uniprot_ids.txt
```

Well done! You have just extracted the UniProt identifiers for all human proteins in Swiss-Prot. Have a cup of tea and a biscuit.

The remainder of this chapter will go over some more useful commands for working on the command line and reiterate some of the key take home messages.

4.9 Viewing the command history

Okay, so you have had a relaxing cup of tea and your head is no longer buzzing from information overload. However, you have also forgotten how you managed to extract those UniProt identifiers.

Not to worry. You can view the history of your previous commands using `history`:

```
$ history
```

Note that each command has a history number associated with it. You can use the number in the history to rerun a previous command without having to retype it. For example to rerun command number 597 you would type in:

```
$ !597
```

Note that the exclamation mark (`!`) in the above is required.

4.10 Clearing the terminal window

After having run the `history` command the terminal window is full of information. However, you find it distracting to have all those commands staring at you whilst you are trying to think.

To clear the screen of output one can use the `clear` command:

```
$ clear
```

Sometimes, for example if you try to view a binary file using a pager, your shell can start displaying garbage. In these cases it may help to run the `reset` command.

```
$ reset
```

In general it is advisable to use `clear` as it only clears the terminal screen whereas `reset` reinitialises the terminal.

4.11 Copying and renaming files

You want to store a copy of your `human_uniprot_id.txt` file in a backup directory.

For this exercise let us start by creating a backup directory.

```
$ mkdir backup
```

Now we can copy the file into the backup directory using the `cp` command.

```
$ cp human_uniprot_id.txt backup/
```

The command above uses the original name of the file. However, we could have given it a different name, for example including the date.

```
$ cp human_uniprot_id.txt backup/human_uniprot_id_2015-11-10.txt
```

Finally, suppose that one wanted to rename the original file to use hyphens rather than under scores. To to this one would use the `mv` command, mnemonic *move*.

```
$ mv human_uniprot_id.txt human-uniprot-id.txt
```

4.12 Removing files and directories

Having experimented with the command line we want to clean up by removing unwanted files and directories.

One can remove files using the `rm` command:

```
$ rm backup/human_uniprot_id.txt
```

Empty directories can be removed using the `rmdir` command:

```
$ mkdir empty
$ rmdir empty
```

To remove directories with files in them one can use the `rm` command with the recursive option:

```
$ rm -r backup
```

Warning: Think twice before deleting files, they will be deleted permanently. When using `rm` there is no such thing as recycle bin from which the files can be recovered.

4.13 Key concepts

- The command line is an excellent tool for automating repetitive tasks
- A terminal application provides access to a shell
- A shell allows you to interact with the operating system's services and programs
- The most commonly used shell is Bash
- Pipes can be used to combine different programs into more complicated work flows
- In general it is better to create small tools that do one thing well
- Think twice before deleting files
- Use the `-help` option to understand a command and its options

CHAPTER 5

Structuring and storing data

The biological sciences are becoming more and more data driven. However, dealing with large volumes of data can feel overwhelming. One way to regain some control is to ensure that there is some structure to the data. This chapter introduces important data structures and file formats for representing them.

5.1 Data structures

In How to think like a computer (page 5) we introduced some basic data types, including integers, floating point numbers, characters and strings. However, most of the time we are not interested in individual instances of integers or floating point values, we want to analyse lots of data points, often of a heterogeneous nature.

We need some way of structuring our data. In computing there are several well defined data structures for accomplishing this. Three data structures of particular interest are lists, dictionaries and graphs. Lists and dictionaries are generally useful and graphs are of particular interest in biology and chemistry.

A list, also known as an array, is basically a collection of elements in a specific order. The most common use case is to simply go through all the elements in a list and do something with them. For example if you had a stack of Petri dishes you might go through them, one-by-one, and only keep the ones that had single colonies suitable for picking. This is similar to what we did with the FASTA file in First steps towards automation (page 13), where we only retained the lines contain the expression "OS=Homo sapiens".

Because lists have an order associated with them they can also be used to implement different types of queuing behaviour such as "first-in-first-out" and "first-in-last-out". Furthermore, individual elements can usually be accessed through their numerical indices. This also means that you can sort lists. For example you could stack your Petri dishes based on the number of colonies in each one. Below is some pseudo code illustrating how a bunch of petri dish identifiers could be stored as a list.

```
petri_dishes = ["wt_control",
                "wt_treated",
                "mutant1_control",
                "mutant1_treated",
                "mutant2_control",
                "mutant2_treated"]
```

In Python one can access the third element (`"mutant1_control"`) using the syntax `petri_dishes[2]`. Note that index used is 2, rather than 3. This is because Python uses zero-based indexing, i.e. the first element of the list would be accessed using index zero, the third element therefore has index two.

Zero vs one-based indexing

In many programming languages (C, C++, JavaScript, Perl, Python, to name a few) lists use a zero-based indexing scheme. In other words the first element of the list is accessed using the index 0, the second using index 1, etc. However, be warned, this is by no means a universal rule as many languages use a one-based indexing scheme, example include Awk, Fortran, Mathematica, Matlab.

A dictionary, also known as a map or an associative array, is an unordered collection of elements that can be accessed through their associated identifiers. In other words each entry in a dictionary has a key, the identifier, and a value. For example, suppose that you needed to store information about the speed of various animals. In this case it would make a lot of sense to have a system where you could look up the speed information based on the animal's name.

```
animal_speed = {"cheeta": 120,
                "ostrich": 70,
                "cat": 30,
                "antelope": 88}
```

In Python one could look up the speed of an ostrich using the syntax `animal_speed["ostrich"]`.

It is worth noting that it is possible to create nested data structures. For example, think of a spreadsheet. The spreadsheet could be represented as a list containing lists. In other words the elements of the outer list would represent the rows in the spreadsheet and the elements in an inner list would represent values of cells at different columns for a specific row.

Let's illustrate this using the table below.

Time	Height	Weight
0	0	0
5	2	5
10	7	12
15	8	20

The table, excluding the header, could be represented using a list of lists.

```
table = [[0,  0,  0],
         [5,  2,  5],
         [10, 7, 12],
         [15, 8, 20]]
```

A graph, sometimes known as a tree, is a data structure that links nodes together via edges (Fig. 5.1). This should sound relatively familiar to you as it is the basic concept behind phylogenetic trees. Each node represents a species and the edges represent the inferred evolutionary relationships between the species. Graphs are also used to represent 2D connectivities of molecules. Because of their general utility lists and dictionaries are built-in to many high level programming languages such as Python and JavaScript. However, data structures for graphs are generally not.

5.2 Data persistence

Suppose that your program has generated a phylogenetic tree and it has used this tree to determine that scientists and baboons are more closely related than expected. Result! At this point the program is about to finish. What should happen to the phylogenetic tree? Should it be discarded or should it be stored for future use? If you want to store data for future use you need to save the data to disk.

[14] https://commons.wikimedia.org/wiki/File:CollapsedtreeLabels-simplified.svg
[15] https://commons.wikimedia.org/wiki/File:Aspirin-skeletal.svg

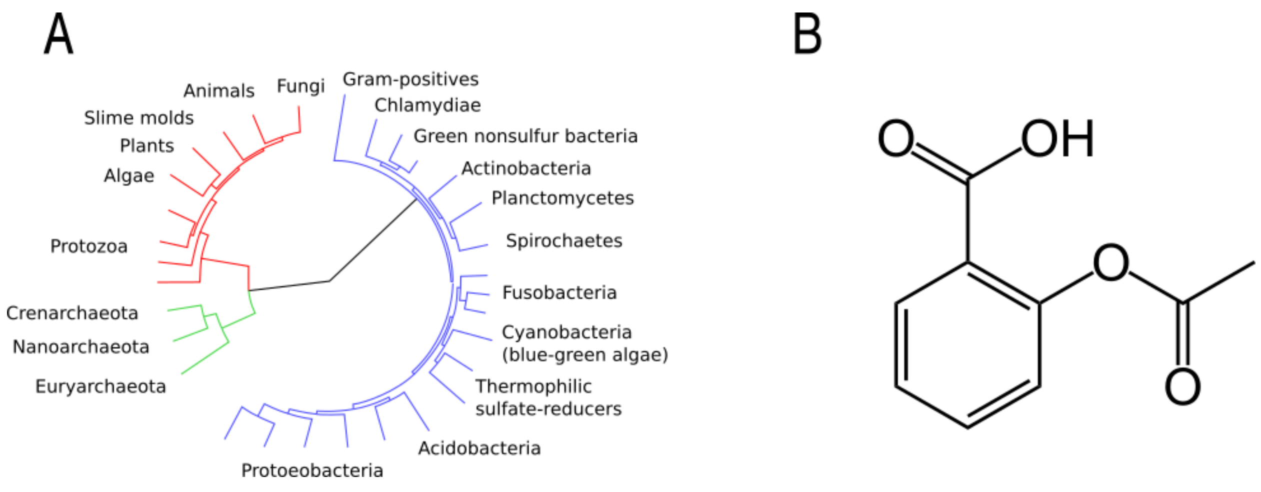


Fig. 5.1: Two examples of graphs: a phylogentic tree (A) and the chemical structure of Aspirin (B). Original images via Wikimeda Commons [Public domain] A[14] and B[15].

When you want to save data to disk you have a choice: you can save the data in a binary format that only your program understands or you can save the data as a plain text file. Storing your data in a binary format has advantages in that the resulting files will be smaller and they will load quicker than a plain text file. However, in the next section you will find out why you should (almost) always store your data as plain text files.

5.3 The beauty of plain text files

Plain text files have several advantages over binary format files. First of all you can open and edit them on any computer. The operating system does not matter as ASCII and Unicode are universal standards. With the slight caveat that you may occasionally have to deal with converting between Windows and Unix line endings (as discussed earlier in How to think like a computer (page 5)).

Furthermore, they are easy to use. You can simply open them in your text editor of choice and start typing away.

Some software companies try to employ a lock-in strategy where their software produces files in a proprietary, binary format. Meaning that you need access to the software in order to open the files produced using it. This is not great from the users point of view. It makes it difficult to use other tools to further analyse the data. It also makes it hard to share data with people that do not have a licence to the software in question. Making use of plain text files and software that can output data in plain text works around this problem.

Finally, there is a rich ecosystem of tools available for working with plain text files. Apart from text editors, there are all of the Unix command line tools. We looked at some of these in First steps towards automation (page 13). In the next chapter, Keeping track of your work (page 31), we will look at a tool called `git` that can be used to track changes to plain text files.

5.4 Useful plain text file formats

There are many well established file formats for representing data in plain text. These have arisen to solve different types of problems.

Plain text files are commonly used to store notes, for example the minutes of a meeting or a list of ideas. When writing these types of documents one wants to be able to make use of headers, bullet points etc. A popular file format for creating such documents is markdown[16]. Markdown (MD) provides a simple way to add formatting such as headers and bullet lists by providing a set of rules of for how certain plain text constructs should be converted to HTML and other document formats.

```
# Level 1 header

## Level 2 header

Here is some text in a paragraph.
It is possible to *emphasize words with italic*.
It is also possible to **strongly emphansize words in bold**.

- First item in a bullet list
- Second item in a bullet list

1. First item in a numbered list
2. Second item in a numbered list

[Link to BBC website](www.bbc.com)

![example image](path/to/example/image.png)
```

Hopefully the example above is self explanatory. For more information have a look at the official markdown syntax page[17].

Markdown specific text editors

There are many markdown specific text editors available. For Mac users a good option is Mou[r], for Linux (and Windows) users an option is MDCharm[s].

[r] http://25.io/mou/
[s] http://www.mdcharm.com/

Another scenario is to record tabular data, for example the results of a scientific experiment. In other words the type of data you would want to store in a spreadsheet. Comma Separated Value (CSV) files are ideally suited for this. This file format is relatively basic, values are simply separated by commas and the file can optionally start with a header. It is worth noting that you can include a comma in a value by surrounding it by double quotes. Below is an example of a three column CSV file containing a header and two data rows.

```
Last name,First name(s),Age
Smith,Alice,34
Smith,"Bob, Carter",56
```

Another scenario, when coding, is the ability to store richer data structures, such as lists or dictionaries, possibly nested within each other. There are two popular file formats for doing this JavaScript Object Notation[20] (JSON) and YAML Ain't Markup a Language[21] (YAML).

[16] https://daringfireball.net/projects/markdown/
[17] https://daringfireball.net/projects/markdown/syntax
[20] http://www.json.org/
[21] http://www.yaml.org/

Recursive acronyms

You may ask yourself why the full name of YAML includes the word YAML. This is because programmers are fond of *recursion*, procedures whose implementation call themselves. YAML is a so called recursive acronym, i.e. the acronym "calls" itself. Other famous recursive acronyms include GNU (GNU's Not Unix), curl (C URL Request Library) and Fiji (Fiji Is Just ImageJ).

JSON was designed to be easy for machines to generate and parse and is used extensively in web applications as it can be directly converted to JavaScript objects. Below is an example of JSON representing a list of scientific discoveries, where each discovery contains a set of key value pairs.

```
[
  {
    "year": 1653,
    "scientist": "Robert Hooke",
    "experiment": "light microscopy",
    "discovery": "cells"
  },
  {
    "year": 1944,
    "scientist": "Barbara McClintock",
    "experiment": "breeding maize plants for colour",
    "discovery": "jumping genes"
  }
]
```

What does it mean to "parse" a file?

Parsing basically means reading a piece of text and resolving it into syntactic parts. For example an English grammar parser might resolve the text "The large cat sat on the red hat" into nouns (cat, hat) and verbs (sat) and adjectives (large, red). In the context of parsing JSON and YAML files we are resolving the text into types such as integers and strings as well as higher level data structures such as lists and dictionaries.

YAML is similar to JSON in that it is a data serialisation standard. However, it places more focus on being human readable. Below is the same data structure represented using YAML.

```
---
  -
    year: 1653
    scientist: "Robert Hooke"
    experiment: "light microscopy"
    discovery: "cells"
  -
    year: 1944
    scientist: "Barbara McClintock"
    experiment: "breeding maize plants for colour"
    discovery: "jumping genes"
```

A nice feature of YAML is the ability to add comments to the data giving further explanation to the reader. These comments are ignored by programs parsing the files.

```
---
  # TODO: include an entry for Anton van Leeuwenhoek here.
  -
    year: 1653
    scientist: "Robert Hooke"
    experiment: "light microscopy"
    discovery: "cells"
  -
    year: 1944
    scientist: "Barbara McClintock"
    experiment: "breeding maize plants for colour"
    discovery: "jumping genes"
```

Comments

Comments are a common feature of most programming languages. They allow the programmer to explain the intention of the code and to make generic notes for future reference.
Comments begin with a program-specific character, or sequence of characters, in the example above the hash (#) symbol. In some languages comments require a closing sequence as well, for example a comment in HTML begins with `<!--` and ends with `-->`.

As scientist's we sometimes need to be able to work with graph data, for example phylogenetic trees and molecules. These often have their own domain specific plain text file formats. For example the Newick format[22] is commonly used to store phylogenetic trees and there are a multitude of file formats for representing molecules including the SMILES[23], and Molfile[24] file formats.

A neat file format for storing and visualising generic graph data is the DOT language[25]. Plain text files written in the DOT language can be visualised using the software Graphviz[26].

Some figures are well suited for being stored as plain text files. This is the case when all the content of the graphic can be described as mathematical functions. These are so called vector graphics and the standard file format for storing them as plain text files is Scalable Vector Graphics[27]. A neat feature of these types of images is that they can be arbitrarily scaled to different sizes without losing any resolution. Hence the word "scalable" in their name.

However, there is another type of graphic that is ill suited to being represented as plain text files. These are so called raster images. In raster images the graphic is represented as a grid where each grid point is set to a certain intensity. Common file formats include PNG, JPEG, and GIF. If you are dealing with photographs or microscopy images the raw data will be recorded in raster form. The reason for storing these types of images as binary blobs, rather than plain text files, is that it saves a significant amount of disk space. Furthermore, image viewers can load these binary blobs much quicker than they could load the same amount of information stored as a plain text file.

However, suppose you needed to generate a figure as a raster image, say for example a scatter plot. Then you should consider writing a script to generate the figure. The instructions for generating the figure, i.e. the script, can then be stored as a plain text file. This concept will be explored in Data visualisation (page 61).

5.5 Tidy data

In the Data visualisation (page 61) chapter we will make use of the ggplot2 package. This requires data to be structured as Tidy Data[28], where each variable is a column and each observation is a row and each type of observational unit forms a table.

[22] https://en.wikipedia.org/wiki/Newick_format
[23] https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system
[24] https://en.wikipedia.org/wiki/Chemical_table_file
[25] http://www.graphviz.org/content/dot-language
[26] http://www.graphviz.org/
[27] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
[28] http://vita.had.co.nz/papers/tidy-data.pdf

Take for example the table below.

	Control	Heat shock
Wild type	3	15
Mutant	5	16

This data would be classified as "messy" because each row contains two observations, i.e. the control and the heat shock experiments. To reorganise the data so that it becomes tidy we need to "melt" or stack it.

Variant	Experiment	Result
Wild type	Control	3
Wild type	Heat shock	15
Mutant	Control	5
Mutant	Heat shock	16

The benefit of structuring your data in a tidy fashion is that it makes it easier to work with when you want to visualise and analyse it.

5.6 Find a good text editor and learn how to use it

A key step to boost your productivity is to find a text editor that suits you, and learning how to make the most of it.

Popular text editors include Sublime Text[29], Geany[30] and Atom[31]. I would recommend trying out at least two of them and doing some of your own research into text editors. Editing text files will be a core activity throughout the rest of this book and you want to be working with a text editor that makes you happy!

If you enjoy working on the command line I would highly recommend experimenting with command line text editors. Popular choices include nano[32], emacs[33] and vim[34]. The former is easy to learn, whereas the latter two give much more power, but are more difficult to learn.

Vim is great!

Personally, I use `vim` for everything. It is one of a few editors that is installed by default on most Unix based system. Furthermore, it is extremely powerful and allows you to do everything using the keyboard. I like this because using the mouse for extended periods of time makes my index finger hurt. If you have half an hour to spare I highly recommend that you try running the `vimtutor` command in a terminal.

5.7 Key concepts

- Lists, also known as arrays, are ordered collections of elements
- Dictionaries, also known as maps and associative arrays, are unordered collections of key-value pairs
- Graphs, sometimes known as trees, links nodes via edges and are of relevance to phylogenetic trees and molecular representations

[29] http://www.sublimetext.com/
[30] http://www.geany.org/Main/HomePage
[31] https://atom.io/
[32] http://www.nano-editor.org/
[33] https://www.gnu.org/software/emacs/
[34] http://www.vim.org/

- In computing persistence refers to data outliving the program that generated it
- If you want any data structures that you have generated to persist you need to write them to disk
- Saving your data as plain text files is almost always preferable to saving it as a binary *blob*
- There are a number of useful plain text file formats for you to make use of
- Don't invent your own file format
- Structuring your data in a "tidy" fashion will make it easier to analyse and visualise
- Learn how to make the most out of your text editor of choice

CHAPTER 6

Keeping track of your work

You are probably used to keeping a record of your experimental work in either physical or electronic lab notebooks. These provide a valuable audit trail and form part of the process that makes your research reproducible.

However, keeping track of ones work when developing data analysis scripts present different types of challenges from keeping track of laboratory work.

When developing scripts it is often beneficial to build them up one step at a time, adding functionality as one goes along. However, as scripts become more complex one often accidentally breaks them whilst trying to add more functionality. To make things worse it can often be difficult to remember what changes were introduced since the last working state of the code.

Because this is a problem that software engineers have been faced with for a long time there are now some excellent tools for dealing with it. The solution is to use a version control system. Here we will use Git, one of the most popular version control systems, to keep track of our work. In its most basic form you can think of Git as a tool for providing you with an infinite undo-button.

6.1 What is Git?

Git is a version control system. That means that it allows you to track changes made to files over time by taking snapshots. Furthermore it allows you to view differences between snapshots and revert back to previous snapshots. In fact Git can do a lot more than this, but this is all we need to know for this chapter. We will go into more detail on the collaborative aspects of Git in the Collaborating on projects (page 83) chapter.

Git tracks snapshots of files in what is called a repository. You can have many Git repositories. On my computer I have a directory named `projects` that currently contains around fifty sub-directories representing the projects that I have been working on recently. Each one of these sub-directories is a separate Git repository.

Git can track some or all of the files in a repository. The files can be located in the top level directory or in sub-directories. However, Git cannot track any files that live in parent directories of the base Git repository, e.g. none of my Git repositories can track files located in the parent `projects` directory.

In a Git repository files can be in one of four states: untracked, staged, unmodified and modified. When a file is initially created it is in an untracked state, meaning that it is not yet under version control. To start tracking the file one adds it to Git, and the state of the file then changes to staged. This means that the file is staged to be included in the next snapshot. Multiple files can be in a staged state when one takes a snapshot. To take a snapshot one commits all the files that are in the so called “staging area”. The state of the files then changes from staged to unmodified, see figure Fig. 6.1.

Any subsequent editing of the files under version control would result in their state changing from unmodified to modified. When one is happy with the edits made one would then add the files to the staging area and their state would change from modified to staged. At that point one is ready to take another snapshot by committing the staged edits. The state of the files then, again, change from staged to unmodified. And the cycle continues (Fig. 6.1).

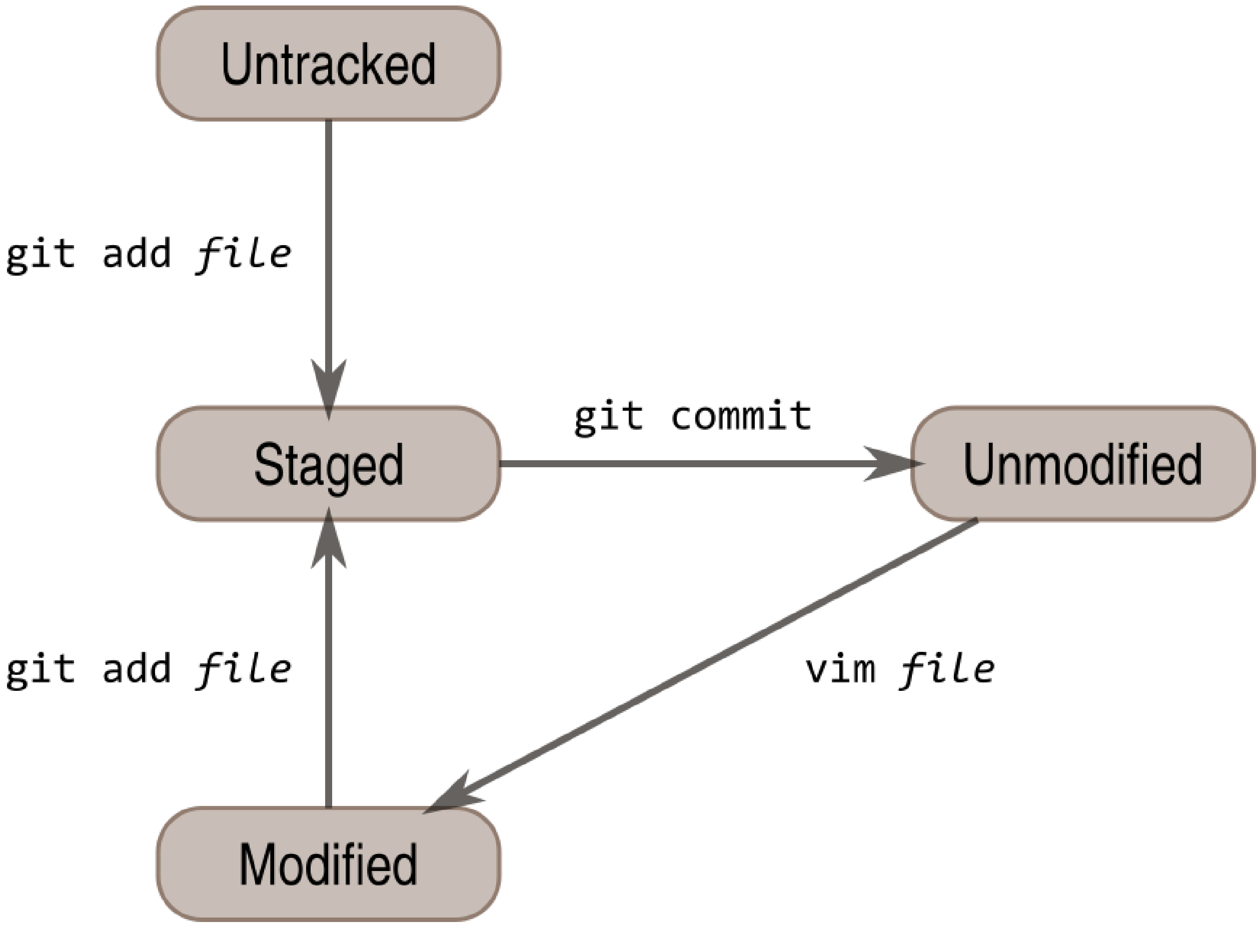


Fig. 6.1: Diagram illustrating the states a file can have in a Git repository. The commands `git add` and `git commit` are the key components to creating a snapshot. The `vim` command symbolises editing the file using the vim text editor, but any program altering the content of the file will result in the file being marked as *modified* by Git.

6.2 First time configuration of Git

Git may or may not be installed by default on your system. To find out if it is try running the command below.

```
$ git --version
```

If Git is installed you will see output along the lines of the below.

```
git version 2.6.3
```

If Git is not installed you will see a `command not found` message. In this case you will need to install it. If you do not know how to do this you can find general instructions on how to install software in Managing your system (page 137).

Although we won't go into it in this chapter, Git is fundamentally a collaboration tool that helps people work on projects together. This means that we need to give Git some information about us for it to be able to keep track of who has done what, specifically our name and email address.

```
$ git config --global user.name "Tjelvar Olsson"
$ git config --global user.email "tjelvar@biologistsguide2computing.com"
```

We will look at the collaboration aspect of git in the Collaborating on projects (page 83) chapter.

6.3 Initialise the project

The first thing to do is to initialise the project using the `git init` command. If run with no argument it will set up tracking of files in the current working directory. If given an argument, such as `protein-count`, git will create a new directory with this name and set up tracking of files within it.

```
git init protein-count
cd protein-count/
```

What does `init` mean?

In computing the term `init` is often used to abbreviate the word "initialise", i.e a one time event that results in the creation of a new entity.

Use your editor of choice and create the markdown file `README.md` and add the content below to it.

```
# Protein count

Count the number of proteins of particular species
in a SwissProt FASTA file.
```

As mentioned, files in a Git repository, the project directory, can be in one of four states: untracked, unmodified, modified and staged. To view the state one can use the command `git status`.

```
$ git status
```

The command below produces the output below.

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

This tells us that the `README.md` file is untracked, in other words it is not yet under version control in Git. However, we would like to track it, so we add it to the Git repository using the `git add` command.

```
$ git add README.md
```

Let's see how this affected the status of the repository.

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
```

This stages the `README.md` file to be committed. To commit the current snapshot of the project to the Git repository we use the `git commit` command.

```
$ git commit -m "Added readme file."
[master (root-commit) e1dc880] Added readme file.
 1 file changed, 12 insertions(+)
 create mode 100644 README.md
```

In the above the `-m` option allows us to specify a commit message on the command line. The commit message should describe the change that your are committing. It will be useful when you review the project at a later date. It also helps communicate your change to any collaborators working on the same project.

Again, let's see how this affected the status of the repository.

```
$ git status
On branch master
nothing to commit, working directory clean
```

That's all you need to know to get started with Git. Start by initialising a project using `git init`. Then use `git add` and `git commit` iteratively to stage and commit snapshots of your project to version control.

6.4 Create a script for downloading the SwissProt FASTA file

We will now convert the command we used to download the SwissProt FASTA file from First steps towards automation (page 13) into a script. To add some organisation we will put this script in a directory named `scripts`. We will also create a directory named `data` for storing the downloaded file. By specifying more than one argument to the `mkdir` command one can create multiple directories.

```
$ mkdir scripts data
```

Using your favorite text editor enter the text below into the file `scripts/get_data.bash`.

```
#!/bin/bash

curl --location --output data/uniprot_sprot.fasta.gz http://bit.ly/1l6SAKb
```

The only difference between this script and the command we entered on the command line is the first line `#!/bin/bash`. This is a special construct, called the shebang, and is used to specify the shell to use when executing the content of the file.

However, in order to be able to execute the file, i.e. run it as a program, it needs to have "execute permissions". One can view the current set of permissions of a file by using `ls -l` (mnemonics `ls` list, `-l` long).

```
$ ls -l scripts/get_data.bash
-rw-r--r--  1 olssont  1340193827  88 29 Nov 10:45 scripts/get_data.bash
```

Note the first ten characters, the first specifies the file type and the remaining nine relate to the permissions of the file, see Fig. 6.2. There are three modes that can be turned on or off: read (`r`), write (`w`) and execute (`x`). Furthermore, these can be specified for the owner (`u`), group (`g`) and all users (`a` or `o`). The nine characters above state that the owner has read and write permissions on the file `rw-`, whereas both the group and all other users only have permission to read the file `r--`.

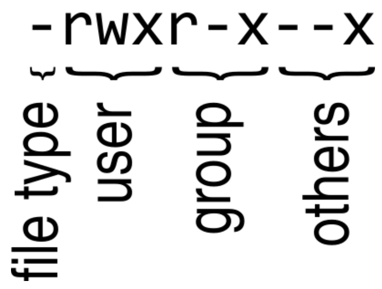


Fig. 6.2: Figure illustrating how the file permissions string should be interpreted. In the above the user has read, write and execute permissions. The members of the group have read and exectue permissions. All other users only have execute permissions on the file. In this case the file type character `-` indicates that the file is a regular/executable file. Other file type characters include `d` and `l` which are used to represent directories and symbolic links respectively.

What is a symbolic link?

The legend of Fig. 6.2 mentioned symbolic links. A symbolic link is a special type of file that points at another file. These can for example be used to create references to canonical representations of your data.

Let's take a moment to expand on the concept of groups in Unix-like operating systems. A user can be part of several groups. However, a file can only belong to one group. For example a PhD student could be part of the groups `famous-project-leader-group` and `awesome-institute-group`. In this hypothetical scenario the default group for the PhD student is the `famous-project-leader-group`. Any files that the student generates would therefore be assigned the group `famous-project-leader-group`. If the student wanted to make a file more widely accessible throughout the institute they could change the file's group to `awesome-institute-group`.

Let us give the file execute permissions. This is achieved using the `chmod` command, mnemonic "change file modes". The `chmod` command can be invoked in a number of different ways. Here we use the symbolic mode to specify that the user and the group (`ug`) should be given execute permissions (`+x`) on the `scripts/get_data.bash` file.

```
$ chmod ug+x scripts/get_data.bash
$ ls -l scripts/get_data.bash
-rwxr-xr--  1 olssont  1340193827  88 29 Nov 10:45 scripts/get_data.bash
```

Let us test the script by running it.

```
$ ./scripts/get_data.bash
$ ls data/
uniprot_sprot.fasta.gz
```

The file was downloaded to the `data` directory, success!

This is a good time to add the script to version control.

```
$ git add scripts/get_data.bash
$ git commit -m "Added script for downloading SwissProt FASTA file."
[master f80731e] Added script for downloading SwissProt FASTA file.
 1 file changed, 3 insertions(+)
 create mode 100755 scripts/get_data.bash
```

Let us check the status of our project.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        data/

nothing added to commit but untracked files present (use "git add" to track)
```

Git is telling us that there are files in the `data` directory that are currently not being tracked. However, in this project the data directory will contain files downloaded from a canonical resource and as the download script is in version control we do not need or want to track the files in this directory.

It is possible to tell Git to ignore files. Using your text editor of choice create the file `.gitignore` and add the content below to it.

Hidden files

On Unix-like systems dot-files, files starting with a ".", are treated as hidden files. These files are usually used to store configuration settings. The `~/.bashrc` file, for example, is used to configure your Bash shell environment. To list hidden files use `ls -a`.

```
data/*
```

In Bash the `*` symbol represents a wild card pattern that can match any string. The `*` symbol can be used in the same fashion in the `.gitignore` file. As such the line we added to our `.gitignore` file tells Git to ignore all files in the `data` directory.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

Git now ignores the content of the `data` directory and tells us that the `.gitignore` file is untracked. Let us add this file.

```
$ git add .gitignore
$ git commit -m "Added gitignore file."
$ git status
On branch master
nothing to commit, working directory clean
```

6.5 Improve script for downloading SwissProt FASTA file

However, the current setup has got an issue in terms of reproducibility. Depending on when the SwissProt FASTA file was downloaded one may obtain different results. It would therefore be useful to include the date of access in the file name. This can be achieved using the `date` command.

```
$ date
Thu 26 Nov 2015 09:20:32 GMT
```

The `date` command can be configured to create custom output formats using the `+` symbol followed by a string template specifying the desired format. In the below `%Y`, `%m` and `%d` will be replaced by the year, month and day respectively.

```
$ date +'%Y-%m-%d'
2015-11-26
```

To get the output of the `date` command into the file name one can use Bash's concept of command substitution. Command substitution makes it possible to evaluate the result of a command within a string. To see this in action we can use the `echo` command, which simply prints out the string that it is given.

```
$ echo "Today it is $(date +'%d')th"
Today it is 26th
```

It is time to introduce the concept of a variable. A variable is basically a means of storing a piece of information using a descriptive name. In bash one can assign a variable using the equals character (`=`). Below we create a variable named `PRIBNOV_BOX` and assign it the value `TATAAT`.

```
$ PRIBNOV_BOX="TATAAT"
```

The value of the variable can then be accessed by prefixing the variable name with the dollar character (`$`).

```
$ echo $PRIBNOV_BOX
TATAAT
```

Don't Repeat Yourself

The use of variables is a key concept in programming. It allows programmers to avoid having to repeat themselves. This is important as repetition increases the chances of introducing errors. Suppose, for example that you had a scaling factor of 1.35611 that you used at ten different places in your script. That presents ten opportunities for typing in the wrong number. Further, suppose that you, later on, needed to change the scaling factor. That presents ten opportunities for forgetting to update a value and another ten opportunities for mistyping the value. In this case it would have been better to create a variable named `scaling_factor` and use that variable in the ten places in your script. That way they are guaranteed to be the same value and you only need to edit one line if you need to change the value. In programming avoiding repetition is important enough to warrant it's own acronym *DRY* (Don't Repeat Yourself).

We now have all the information we need to improve the script. Edit the `script/get_data.bash` file to look like the below.

```
#!/bin/bash

FNAME="data/uniprot_sprot.$(date +'%Y-%m-%d').fasta.gz"
curl --location --output $FNAME http://bit.ly/1l6SAKb
```

Let's try running the script.

```
$ ./scripts/get_data.bash
```

Now we can check that the script has produced an appropriately named file.

```
$ ls data/
uniprot_sprot.2015-11-26.fasta.gz uniprot_sprot.fasta.gz
```

We have added a piece of functionality and have tested that it works as expected. This is a good time to commit our changes to Git. However, before we do that let us examine the changes to the project since the last commit using the `git diff` command.

```
$ git diff
diff --git a/scripts/get_data.bash b/scripts/get_data.bash
index d8e9bda..338d82c 100755
--- a/scripts/get_data.bash
+++ b/scripts/get_data.bash
@@ -1,3 +1,4 @@
 #!/bin/bash

-curl --location --output data/uniprot_sprot.fasta.gz http://bit.ly/1l6SAKb
+FNAME="data/uniprot_sprot.$(date +'%Y-%m-%d').fasta.gz"
+curl --location --output $FNAME http://bit.ly/1l6SAKb
```

The command above tells us that one line has been removed, the one prefixed by a minus sign, and that two lines have been added, the ones prefixed by a plus sign. In fact we have modified one line and added one, but the effect is the same.

Let us now add and commit the changes to Git. We need to do this as working with Git is a cyclical process. You make changes by editing the files, you add the changes that you want to snapshot to the staging area, then you commit the staged changes. At this point the cycle starts all over again Fig. 6.1.

```
$ git add scripts/get_data.bash
$ git commit -m "Updated download script to include date in file name."
[master 7512894] Updated download script to include date in file name.
 1 file changed, 2 insertions(+), 1 deletion(-)
```

By adding the date of download to the file name reproducibility is improved and it means that we can download the file on different dates and ensure that no data is overwritten.

However, it is still possible to accidentally delete or modify the data file. To overcome this, and further improve reproducibility, it is good practise to give the data file read-only permissions. This means that the file cannot be modified or deleted, only read. To do this we will make use of the `chmod` command. In this instance we will make use of an absolute mode. Absolute modes encode the permissions using the numbers 1, 2 and 4 that represent execute, write and read modes respectively. These numbers can be combined to create any permission, for example 7 represents read, write and execute permissions and 5 represents read and execute permissions.

Value	Permission
1	execute
2	write
3	write & execute
4	read
5	read & execute
6	read & write
7	read & write & execute

To set the permissions for the owner, group and all other users one simply uses three such numbers. For example to give the owner read and write permissions and the group and all other users read-only permissions one would use the absolute mode 644.

In this instance we want to set the file to read-only for the owner, group and all other users so we will use the absolute mode 444.

```
#!/bin/bash

FNAME="data/uniprot_sprot.$(date +'%Y-%m-%d').fasta.gz"
curl --location --output $FNAME http://bit.ly/1l6SAKb
chmod 444 $FNAME
```

If you run the script now you will see that it changes the permissions of the downloaded file. If you run the script again, on the same day, you will notice that the it complains that it has not got permissions to write to the file. This is expected as the `curl` command is wanting to overwrite the existing read-only file.

Let's add these changes to the staging area.

```
$ git add scripts/get_data.bash
```

It is good practise to try to make the commit message no more than 50 characters long. Sometimes this is not enough. In these cases you can create a multi line commit message using a text editor (likely to be `vim` by default) by omitting the `-m` flag.

Let's try this now.

```
$ git commit
```

This should open a text editor with the text below.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   scripts/get_data.bash
```

Use your text editor to edit this message to the below.

```
Set permissions of data file to read only

The intention of this change is to prevent accidental deletion or
modification of the raw data file.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   scripts/get_data.bash
```

When you save the file you should see the output below in the terminal.

```
$ git commit
[master ad2a4c5] Set permissions of data file to read only
 1 file changed, 1 insertion(+)
```

Help! I'm stuck in Vim.

If you tried out the `git commit` command you may depeding on the configuration of your computer get thrown into Vim. If you don't know how to use Vim, it can be tricky just to get out of it.

1. Press the Esc key. This ensure that you are in "normal" mode.
2. Press the colon (`:`) key. This puts you in "command-line" mode.
3. Enter the text `q!`. This is the command to quit (`q`) without saving (`!`).
4. Press the Enter key. This executes the command from step 3.

What if I want to edit or delete a file that is read only?

In this case you first need to change the mode of the file so that you have write permissions to it. This is achieved using the `chmod` command, for example:

```
$ chmod u+w file_i_want_to_edit.txt
```

6.6 Create script for counting the number of proteins in a genome

Now that we have a script for downloading the SwissProt FASTA file let us convert what we learnt in First steps towards automation (page 13) into a script for counting the number of proteins for a particular species.

Add the lines below to the file `scripts/protein_count.bash`.

```
#!/bin/bash

gunzip -c data/uniprot_sprot.fasta.gz | grep 'OS=Homo sapiens' \
| cut -d '|' -f 2 | wc -l
```

Make the file executable and test the script.

```
$ chmod +x scripts/protein_count.bash
$ ./scripts/protein_count.bash
   20194
```

At the moment the path to the data file and the species are hard coded into the script. It would be nice if we could turn these two parameters into command line arguments. We can do this using the special variables `$1` and `$2` that represent the first and second command line arguments, respectively.

```
#!/bin/bash

DATA_FILE_PATH=$1
SPECIES=$2
echo "Input file: $DATA_FILE_PATH"
echo "Species: $SPECIES"

gunzip -c $DATA_FILE_PATH | grep "OS=$SPECIES" \
| cut -d '|' -f 2 | wc -l
```

Warning: Bash makes a distinction between single and double quotes. To expand variables one needs to use double quotes. If not one will get the literal value of the string within the single quotes. For example, the command `echo 'Species: $SPECIES'` would print the literal string `Species: $SPECIES`.

```
$ SPECIES=H.sapiens
$ echo "Species: $SPECIES"
Species: H.sapiens
$ echo 'Species: $SPECIES'
Species: $SPECIES
```

This is a good point to test if things are working as expected.

```
$ ./scripts/protein_count.bash data/uniprot_sprot.2015-11-26.fasta.gz "Homo sapiens"
Input file: data/uniprot_sprot.2015-11-26.fasta.gz
Species: Homo sapiens
   20194
```

Success! Let us add and commit the script to Git.

```
$ git add scripts/protein_count.bash
$ git commit -m "Added script for counting the numbers of proteins."
[master b9de9bc] Added script for counting the numbers of proteins.
 1 file changed, 9 insertions(+)
 create mode 100755 scripts/protein_count.bash
```

6.7 More useful git commands

We've covered a lot of ground in this chapter. Can you remember everything that we did and the motivation behind each individual step? If not, that is okay, we can use Git to remind us using the `git log` command.

```
$ git log --oneline
b9de9bc Added script for counting the numbers of proteins.
a672257 Added command to set permissions of data file to read only.
7512894 Updated download script to include date in file name.
6c6f65b Added gitignore file.
f80731e Added script for downloading SwissProt FASTA file.
e1dc880 Added readme file.
```

Note that the comments above give a decent description of what was done. However, it would have been useful to include more information about the motive behind some changes. If one does not make use of the `-m` argument when using `git commit` one can use the default text editor to write a more comprehensive commit message. For example, a more informative commit message for commit `a672257` could have looked something along the lines of:

```
Added command to set permissions of data file to read only.

The intention of this change is to prevent accidental deletion or
modification of the raw data file.
```

Another useful feature of Git is that it allows us to inspect the changes between commits using the `git diff` command. For example to understand what changed in commit `a672257` we can compare it to the previous commit `7512894`.

```
$ git diff 7512894 a672257
diff --git a/scripts/get_data.bash b/scripts/get_data.bash
```

```
index 338d82c..0bbc17b 100755
--- a/scripts/get_data.bash
+++ b/scripts/get_data.bash
@@ -2,3 +2,4 @@

 FNAME="data/uniprot_sprot.$(date +'%Y-%m-%d').fasta.gz"
 curl --location --output $FNAME http://bit.ly/1l6SAKb
+chmod 444 $FNAME
```

In the above we can see that we added the line `chmod 444 $FNAME` to the `scripts/get_data.bash` file.

6.8 Key concepts

- When working with files it is often desirable to be able to track changes
- When programming it is particularly useful to be able to save working states of the code
- This gives one the opportunity to roll back to a previously working state if things go wrong
- Git is a powerful version control system
- To get started with Git one only needs to get familiar with a handful of commands
- Use `git init` to initialise a Git repository
- Use `git add file-to-add` to start tracking a file in Git
- Use `git commit -m "your summary message here"` to record a snapshot in Git
- The overhead of using Git whilst programming is minimal
- The benefits of using Git are great
- Start using Git in your day-to-day work right now!
- On Unix-like systems files have write, read and execute permissions that can be turned on and off
- By making a file executable it can be run as an independent program
- By giving raw data files read only permissions one can ensure that they are not accidentally modified or deleted

CHAPTER 7

Data analysis

Streptomyces coelicolor is a soil-dwelling bacterium with a complex life-cycle involving mycelial growth and sporulation. It is of particular interest in that it can produce a range of natural products of pharmaceutical relevance. In fact it is the major source of natural antibiotics.

In 2002 the genome of *Streptomyces coelicolor* A3(2), the model actinomycete organism, was sequenced. In this chapter we will do some basic bioinformatics on this genome to illustrate some fundamental concepts of computer programming.

One feature of interest when examining DNA is the guanine-cytosine (GC) content. DNA with high GC-content is more stable than DNA with low GC-content. The GC-content is defined as:

$$100 * \frac{G + C}{A + T + G + C}$$

To solve this problem we need to write code to be able to read in sequence data from a file, process the input to calculate the local GC-content and write out the results to another file. In the process the computational concepts of variables, functions and loops will be introduced.

In this chapter we will use the Python scripting language to perform our data analysis.

7.1 What is Python?

Python is a high-level scripting language that is growing in popularity in the scientific community. It uses a syntax that is relatively easy to get to grips with and which encourages code readability.

7.2 Using Python in interactive mode

To start off with we will make use of Python using its interactive mode[35], which means that we can type Python commands straight into the terminal. In fact when working with Python in interactive mode one can think of it as switching the terminal's shell from Bash to Python.

To start Python in its interactive mode simply type `python` into the terminal.

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

35 https://docs.python.org/2/tutorial/interpreter.html#interactive-mode

Note that this prints out information about the version of Python that is being used and how it was compiled before leaving you at the interactive prompt. In this instance I am using Python version 2.7.10.

The three greater than signs (>>>) represent the primary prompt into which commands can be entered.

```
>>> 1 + 2
3
```

The secondary prompt is represented by three dots (...) and denotes a continuation line.

```
>>> line = ">sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R"
>>> if line.startswith(">"):
...     print(line)
...
>sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R
```

7.3 Variables

A variable is a means of storing a piece of information using using a descriptive name. The use of variables is encouraged as it allows us to avoid having to repeat ourselves, the *DRY* principle.

In Python variables are assigned using the equals sign.

```
>>> pi = 3.14
```

When naming variables being explicit is more important than being succinct. One reason for this is that you will spend more time reading your code than you will writing it. Avoiding the mental overhead of trying to understand what all the acronyms mean is a good thing. For example, suppose that we wanted to create a variable for storing the radius of a circle. Please avoid the temptation of naming the variable `r`, and go for the longer but more explicit name `radius`.

```
>>> radius = 1.5
```

Note: Many modern text editors have auto complete functionality so longer variable names does not necessarily need to mean that there is more typing required. Remember, spend time finding a text editor that works well for you!

7.4 Determining the GC count of a sequence

Suppose that we had a *string* representing a DNA sequence.

```
>>> dna_string = "attagcgcaatctaactacactactgccgcgcggcatatatttaaatata"
>>> print(dna_string)
attagcgcaatctaactacactactgccgcgcggcatatatttaaatata
```

A string is a data type for representing text. As such it is not ideal for data processing purposes. In this case the DNA sequence would be better represented using a *list*, with each item in the list representing a DNA letter.

In Python we can convert a string into a list using the built-in `list()` function.

```
>>> dna_list = list(dna_string)
>>> print(dna_list)
['a', 't', 't', 'a', 'g', 'c', 'g', 'c', 'a', 'a', 't', 'c', 't', 'a', 'a',
 'c', 't', 'a', 'c', 'a', 'c', 't', 'a', 'c', 't', 'g', 'c', 'c', 'g', 'c',
 'g', 'c', 'g', 'g', 'c', 'a', 't', 'a', 't', 'a', 't', 't', 't', 'a', 'a',
 'a', 't', 'a', 't', 'a']
```

Python's list *class* has got a method called `count()` that we can use to find out the counts of particular elements in the list.

```
>>> dna_list.count("a")
17
```

To find out the total number of items in a list one can use Python's built-in `len()` function, which returns the *length* of the list.

```
>>> len(dna_list)
50
```

When using Python you need to be careful when dividing integers, because in Python 2 the default is to use integer division, i.e. to discard the remainder.

```
>>> 3 / 2
1
```

One can work around this by ensuring that at least one of the numbers is represented using floating point.

```
>>> 3 / 2.0
1.5
```

Warning: In Python 3, the behaviour of the division operator has been changed, and dividing two integers will result in normal division.

One can convert an integer to a floating point number using Python's built-in `float()` function.

```
>>> float(2)
2.0
```

We now have all the information required to calculate the GC-content of the DNA sequence.

```
>>> gc_count = dna_list.count("g") + dna_list.count("c")
>>> gc_fraction = float(gc_count) / len(dna_list)
>>> 100 * gc_fraction
38.0
```

7.5 Creating reusable functions

Suppose that we wanted to calculate the GC-content for several sequences. In this case it would be very annoying, and error prone, to have to enter the commands above into the Python shell manually for each sequence. Rather, it would be advantageous to be able to create a piece of code that could be called repeatedly to calculate the GC-content. We can achieve this using the concept of functions. In other words functions are a means for programmers to avoid repeating themselves, thus adhering to the *DRY* principle.

Let us create a simple function that adds two items together.

```
>>> def add(a, b):
...     return a + b
...
>>> add(2, 3)
5
```

In Python functions are defined using the def keyword. Note that the def keyword is followed by the name of the function. The name of the function is followed by a parenthesized set of arguments, in this case the function takes two arguments a and b. The end of the function definition is marked using a colon.

The body of the function, in this example the return statement, needs to be indented. The standard in Python is to use four white spaces to indent code blocks. In this case the function body only contains one line of code. However, a function can include several indented lines of code.

Warning: Whitespace really matters in Python! If your code is not correctly aligned you will see IndentationError messages telling you that everything is not as it should be. You will also run into IndentationError messages if you mix white spaces and tabs.

Now we can create a function for calculating the GC-content of a sequence. As with variables explicit trumps succinct in terms of naming.

```
>>> def gc_content(sequence):
...     gc_count = sequence.count("g") + sequence.count("c")
...     gc_fraction = float(gc_count) / len(sequence)
...     return 100 * gc_fraction
...
>>> gc_content(dna_list)
38.0
```

7.6 List slicing

Suppose that we wanted to look at local variability in GC-content. To achieve this we would like to be able to select segments of our initial list. This is known as "slicing", as in slicing up a salami.

In Python slicing uses a [start:end] syntax that is inclusive for the start index and exclusive for the end index. To illustrate slicing let us first create a list to work with.

```
>>> zero_to_five = ["zero", "one", "two", "three", "four", "five"]
```

To get the first two elements we therefore use 0 for the start index, as Python uses a zero-based indexing system, and 2 for the end index as the element from the end index is excluded.

```
>>> zero_to_five[0:2]
['zero', 'one']
```

Note that the start position for the slicing is 0 by default so we could just as well have written.

```
>>> zero_to_five[:2]
['zero', 'one']
```

To get the last three elements.

```
>>> zero_to_five[3:]
['three', 'four', 'five']
```

It is worth noting that we can use negative indices, where -1 represents the last element. So to get all elements except the first and the last, one could slice the list using the indices 1 and -1.

```
>>> zero_to_five[1:-1]
['one', 'two', 'three', 'four']
```

We can use list slicing to calculate the local GC-content measurements of our DNA.

```
>>> gc_content(dna_list[:10])
40.0
>>> gc_content(dna_list[10:20])
30.0
>>> gc_content(dna_list[20:30])
70.0
>>> gc_content(dna_list[30:40])
50.0
>>> gc_content(dna_list[40:50])
0.0
```

7.7 Loops

It can get a bit repetitive, tedious, and error prone specifying all the ranges manually. A better way to do this is to make use of a loop construct. A loop allows a program to cycle through the same set of operations a number of times.

In lower level languages *while* loops are common because they operate in a way that closely mimic how the hardware works. The code below illustrates a typical setup of a while loop.

```
>>> cycle = 0
>>> while cycle < 5:
...     print(cycle)
...     cycle = cycle + 1
...
0
1
2
3
4
```

In the code above Python moves through the commands in the while loop executing them in order, i.e. printing the value of the `cycle` variable and then incrementing it. The logic then moves back to the `while` statement and the conditional (`cycle < 5`) is re-evaluated. If true the commands in the while statment are executed in order again, and so forth until the conditional is false. In this example the `print(cycle)` command was called five times, i.e. until the `cycle` variable incremented to 5 and the `cycle < 5` conditional evaluated to false.

However, when working in Python it is much more common to make use of *for* loops. For loops are used to iterate over elements in data structures such as lists.

```
>>> for item in [0, 1, 2, 3, 4]:
...     print(item)
...
0
1
2
3
4
```

In the above we had to manually write out all the numbers that we wanted. However, because iterating over a range of integers is such a common task Python has a built-in function for generating such lists.

```
>>> range(5)
[0, 1, 2, 3, 4]
```

So a typical for loop might look like the below.

```
>>> for item in range(5):
...     print(item)
...
0
1
2
3
4
```

The `range()` function can also be told to start at a larger number. Say for example that we wanted a list including the numbers 5, 6 and 7.

```
>>> range(5, 8)
[5, 6, 7]
```

As with slicing the start value is included whereas the end value is excluded.

It is also possible to alter the step size. To do this we must specify the start and end values explicitly before adding the step size.

```
>>> range(0, 50, 10)
[0, 10, 20, 30, 40]
```

We are now in a position where we can create a naive loop for for calculating the local GC-content of our DNA.

```
>>> for start in range(0, 50, 10):
...     end = start + 10
...     print(gc_content(dna_list[start:end]))
...
40.0
30.0
70.0
50.0
0.0
```

Loops are really powerful. They provide a means to iterate over lots of items and can be used to automate repetitive tasks.

7.8 Creating a sliding window GC-content function

So far we have been working with Python in interactive mode. This is a great way to explore what can be achieved with Python. It is also handy to simply use Python's interactive mode as a command line calculator. However, it can get a little bit clunky when trying to write constructs that span several lines, such as functions.

Now we will examine how one can write a Python script as a text file and how to run that text file through the Python interpreter, i.e. how to run a Python script from the command line.

Start off by creating a new directory to work in.

```
$ mkdir S.coelicolor-local-GC-content
$ cd S.coelicolor-local-GC-content
```

Use your favourite text editor to enter the code below into a file named `gc_content.py`.

```
sequence = list("attagcgcaatctaactacactactgccgcgcggcatatatttaaatata")
print(sequence)
```

Note: If your text editor is not giving you syntax highlighting find out how it can be enabled. If your text editor does not support syntax highlighting find a better text editor!

Open up a terminal and go to the directory where you saved the `gc_content.py` script. Run the script using the command below.

```
$ python gc_content.py
```

You should see the output below printed to your terminal.

```
['a', 't', 't', 'a', 'g', 'c', 'g', 'c', 'a', 'a', 't', 'c', 't', 'a', 'a',
'c', 't', 'a', 'c', 'a', 'c', 't', 'a', 'c', 't', 'g', 'c', 'c', 'g', 'c',
'g', 'c', 'g', 'g', 'c', 'a', 't', 'a', 't', 'a', 't', 't', 't', 'a', 'a',
'a', 't', 'a', 't', 'a']
```

In the script we used Python's built-in `list()` function to convert the DNA string into a list. We then printed out the `sequence` list.

Now let us add the `gc_content()` function to the script.

```
1  def gc_content(sequence):
2      "Return GC-content as a percentage from a list of DNA letters."
3      gc_count = sequence.count("g") + sequence.count("c")
4      gc_fraction = float(gc_count) / len(sequence)
5      return 100 * gc_fraction
6
7  sequence = list("attagcgcaatctaactacactactgccgcgcggcatatatttaaatata")
8  print(gc_content(sequence))
```

In the above the `gc_content()` function is implemented as per our exploration in our interactive session. The only difference is the addition of a, so called, "docstring" (documentation string) to the body of the function (line 2). The docstring is meant to document the purpose and usage of the function. Documenting the code in this way makes it easier for you, and others, to understand it.

Note that the script now prints out the GC-content rather than the sequence (line 8). Let us run the updated script from the command line.

```
$ python gc_content.py
38.0
```

The next piece of code will be a bit more complicated. However, note that that it represents the most complicated aspect of this chapter. So if you find it difficult, don't give up, it gets easier again later on.

Now let us implement a new function for performing a sliding window analysis. Add the code below to the start of the `gc_content.py` file.

```
def sliding_window_analysis(sequence, function, window_size=10):
    """Return an iterator that yields (start, end, property) tuples.

    Where start and end are the indices used to slice the input list
    and property is the return value of the function given the sliced
    list.
    """
    for start in range(0, len(sequence), window_size):
        end = start + window_size
        if end > len(sequence):
            break
        yield start, end, function(sequence[start:end])
```

There is quite a lot going on in the code above so let us walk through it slowly. One of the first things to note is that the `sliding_window_analysis()` function takes another `function` as its second argument. Functions can be passed around just like variables and on line 12 the `function` is repeatedly called with slices of the input sequence.

The `sliding_window_analysis()` function also takes a `window_size` argument. This defines the step size of the `range()` function used to generate the `start` indices for the slicing. Note that in this case we provide the `window_size` argument with a default value of 10. This means that the `window_size` argument does not need to be explicitly set when calling the function (if one is happy with the default).

On line 9, inside the for loop, we generate the `end` index by adding the `window_size` to the `start` index. This is followed by a check that the generated `end` index would not result in a list slice that spanned beyond the end of the sequence.

On line 11, inside the `if` statment there is a `break` clause. The `break` statement is used to break out of the loop immediately. In other words if the `end` variable is greater than the length of the sequence we break out of the loop immediately.

At the end of the for loop we make use of the `yield` keyword to pass on the `start` and `end` indices as well as the value resulting from calling the input `function` with the sequence slice. This means that rather than returning a value the `sliding_window_analysis()` function returns an iterator. As the name suggests an "iterator" is an object that one can iterate over, for example using a *for* loop. Let us add some code to the script to illustrate how one would use the `sliding_window_analysis()` function in practise.

```
sequence = list("attagcgcaatctaactacactactgccgcgcggcatatatttaaatata")
for start, end, gc in sliding_window_analysis(sequence, gc_content):
    print(start, end, gc)
```

Let us test the code again.

```
$ python gc_content.py
(0, 10, 40.0)
(10, 20, 30.0)
(20, 30, 70.0)
(30, 40, 50.0)
(40, 50, 0.0)
```

The current implementation of the `sliding_window_analysis()` is very dependent on the frame of reference as the window slides along. For example if the `window_size` argument was set to 3 one would obtain the analysis of the first codon reading frame, but one would have no information about the second and third codon reading frames. To overcome this one can perform sliding window analysis with overlapping windows. Let us illustrate this visually by extracting codons from a DNA sequence.

```
# Original sequence.
atcgctaaa

# Non overlapping windows.
atc
   gct
      aaa

# Overlapping windows.
atc
 tcg
  cgc
   gct
    cta
     taa
      aaa
```

To enable overlapping windows in our `sliding_window_analysis()` function we need to add a `step_size` argument to it and make use of this in the call to the `range()` function.

```python
def sliding_window_analysis(sequence, function, window_size=10, step_size=5):
    """Return an iterator that yields (start, end, property) tuples.

    Where start and end are the indices used to slice the input list
    and property is the return value of the function given the sliced
    list.
    """
    for start in range(0, len(sequence), step_size):
        end = start + window_size
        if end > len(sequence):
            break
        yield start, end, function(sequence[start:end])
```

Let us run the script again to see what the output of this overlapping sliding window analysis is.

```
$ python gc_content.py
(0, 10, 40.0)
(5, 15, 40.0)
(10, 20, 30.0)
(15, 25, 40.0)
(20, 30, 70.0)
(25, 35, 100.0)
(30, 40, 50.0)
(35, 45, 0.0)
(40, 50, 0.0)
```

Note that the `gc_content()` function is now applied to overlapping segments of DNA. This allows us, for example, to note that the 25 to 35 region has a GC-content of 100%, which is something that we did not manage to pick out before.

7.9 Downloading the genome

It is time to start working on some real data. Let us download the genome of *Streptomyces coelicolor* from the Sanger Centre ftp site[36]. The URL shortened using bitly[37] point to the `Sco.dna` file.

```
$ curl --location --output Sco.dna http://bit.ly/1Q8eKWT
$ head Sco.dna
SQ   Sequence 8667507 BP; 1203558 A; 3121252 C; 3129638 G; 1213059 T; 0 other;
     cccgcggagc gggtaccaca tcgctgcgcg atgtgcgagc gaacacccgg gctgcgcccg        60
     ggtgttgcgc tcccgctccg cgggagcgct ggcgggacgc tgcgcgtccc gctcaccaag       120
     cccgcttcgc gggcttggtg acgctccgtc cgctgcgctt ccggagttgc ggggcttcgc       180
     cccgctaacc ctgggcctcg cttcgctccg ccttgggcct gcggcgggtc cgctgcgctc       240
     ccccgcctca agggcccttc cggctgcgcc tccaggaccc aaccgcttgc gcgggcctgg       300
     ctggctacga ggatcggggg tcgctcgttc gtgtcgggtt ctagtgtagt ggctgcctca       360
     gatagatgca gcatgtatcg ttggcagaaa tatgggacac ccgccagtca ctcgggaatc       420
     tcccaagttt cgagaggatg gccagatgac cggtcaccac gaatctaccg gaccaggtac       480
     cgcgctgagc agcgattcga cgtgccgggt gacgcagtat cagacggcgg gtgtgaacgc       540
```

[36] ftp://ftp.sanger.ac.uk/pub/project/pathogens/S_coelicolor/whole_genome/
[37] https://bitly.com/

7.10 Reading and writing files

In order to be able to process the genome of *Streptomyces coelicolor* we need to be able to read in the `Sco.dna` file. In Python reading and writing of files is achieved using the built-in `open()` function, which returns a file handle.

What is a file handle?

A file handle is a data structure that handles the book keeping of the position within the file as well as the mode in which the file was opened. The mode of the file determines what one can do with it. For example one cannot write to a file that has been opened for reading. The position of the file handle determines where the next operation will take place. For example, if one is about to write something to a file existing content will be overwritten unless the position is pointing at the end of the file, in which case the new content will be appended to the old.

Before we start adding code to our script let us examine reading and writing of files using in Python's interactive mode. Let us open up the `Sco.dna` file for reading.

```
>>> file_handle = open("Sco.dna", mode="r")
```

We can access the current position within the file using the `tell()` method of the file handle.

```
>>> file_handle.tell()
0
```

The integer zero indicates that we are at the beginning of the file.

To read in the entire content of the file as a single string of text one can use the `read()` method of the file handle.

```
>>> text = file_handle.read()
```

After having read in the content of the file the position of the file handle will point at the end of the file.

```
>>> file_handle.tell()
11701261
```

When one has finished working with a file handle it is important to remember to close it.

```
>>> file_handle.close()
```

Let us examine the text that we read in.

```
>>> type(text)
<type 'str'>
>>> len(text)
11701261
>>> text[:60]
'SQ   Sequence 8667507 BP; 1203558 A; 3121252 C; 3129638 G; 1'
```

However, rather than reading in files as continuous strings one often want to process files line by line. One can read in a file as a list of lines using the `readlines()` method.

```
>>> file_handle = open("Sco.dna", "r")
>>> lines = file_handle.readlines()
>>> file_handle.close()
```

Let us examine the lines that we read in.

```
>>> type(lines)
<type 'list'>
>>> len(lines)
144461
>>> lines[0]
'SQ   Sequence 8667507 BP; 1203558 A; 3121252 C; 3129638 G; 1213059 T; 0 other;\n'
```

A third way of accessing the content of a file handle is to simply treat it as an iterator. This is possible as the Python file handles implement a method called `next()` that returns the next line in the file. When it reaches the end of the file the `next()` function raises a `StopIteration` exception, which tells the iterator to stop iterating.

Let's see the workings of the `next()` method in action.

```
>>> file_handle = open("Sco.dna", "r")
>>> file_handle.next()
'SQ   Sequence 8667507 BP; 1203558 A; 3121252 C; 3129638 G; 1213059 T; 0 other;\n'
>>> file_handle.next()
'     cccgcggagc gggtaccaca tcgctgcgcg atgtgcgagc gaacacccgg gctgcgcccg        60\n'
```

We can go to the end of the file using the `seek()` method of the file handle.

```
>>> file_handle.seek(11701261)
```

Let's see what happens when we call the `next()` method now.

```
>>> file_handle.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

As explained above this raises a `StopIteration` exception. Now that we are done with our experiment we must remember to close the file handle.

```
>>> file_handle.close()
```

In practise one tends to use file handles directly within for loops.

```
>>> num_lines = 0
>>> file_handle = open("Sco.dna", "r")
>>> for line in file_handle:
...     num_lines = num_lines + 1
...
>>> print(num_lines)
144461
```

In the for loop above the file handle acts as an iterator, yielding the lines within the opened file.

Again we must not forget to close the file handle.

```
>>> file_handle.close()
```

Having to constantly remember to close file handles when one is done with them can become tedious. Furthermore, forgetting to close file handles can have dire consequences. To make life easier one can make use of Python's built-in `with` keyword.

The `with` keywords works with context managers. A context manager implements the so called "context manager protocol". In the case of a file handle this means that the file is opened when one enters into the context of the `with` statement and that the file is automatically closed when one exits out of the context. All that is a fancy way of saying that we do not need to worry about remembering to close files if we access file handles using the syntax below.

```
>>> with open("Sco.dna", mode="r") as file_handle:
...     text = file_handle.read()
...
```

Let us shift the focus to the writing of files. There are two modes for writing files `w` and `a`. The former will overwrite any existing files with the same name whereas the latter would append to them. Let us illustrate this with an example.

```
>>> with open("tmp.txt", "w") as file_handle:
...     file_handle.write("Original message")
...
>>> with open("tmp.txt", "r") as file_handle:
...     print(file_handle.read())
...
Original message
>>> with open("tmp.txt", "a") as file_handle:
...     file_handle.write(", with more info")
...
>>> with open("tmp.txt", "r") as file_handle:
...     print(file_handle.read())
...
Original message, with more info
>>> with open("tmp.txt", "w") as file_handle:
...     file_handle.write("scrap that...")
...
>>> with open("tmp.txt", "r") as file_handle:
...     print(file_handle.read())
...
scrap that...
```

Armed with our new found knowledge of how to read and write files we will now create a function for reading in the DNA sequence from the `Sco.dna` file.

7.11 Creating a function for reading in the *Streptomyces* sequence

Let us create a function that reads in a genome as a string and returns the DNA sequence as a list. At this point we have a choice of what the input parameter should be. We could give the function the name of the file containing the genome or we could give the function a file handle of the genome. Personally, I prefer to create functions that accept file handles, because they are more generic. Sometimes the data to be read comes from sources other than a file on disk. However, as long as these behave as a file object one can still pass them to the function.

Let us have a look at the file containing the *Streptomyces coelicolor* genome.

```
$ head Sco.dna
SQ   Sequence 8667507 BP; 1203558 A; 3121252 C; 3129638 G; 1213059 T; 0 other;
     cccgcggagc gggtaccaca tcgctgcgcg atgtgcgagc gaacacccgg gctgcgcccg        60
     ggtgttgcgc tcccgctccg cgggagcgct ggcgggacgc tgcgcgtccc gctcaccaag       120
     cccgcttcgc gggcttggtg acgctccgtc cgctgcgctt ccggagttgc ggggcttcgc       180
     cccgctaacc ctgggcctcg cttcgctccg ccttgggcct gcggcgggtc cgctgcgctc       240
     ccccgcctca agggcccttc cggctgcgcc tccaggaccc aaccgcttgc gcgggcctgg       300
     ctggctacga ggatcggggg tcgctcgttc gtgtcgggtt ctagtgtagt ggctgcctca       360
     gatagatgca gcatgtatcg ttggcagaaa tatgggacac ccgccagtca ctcgggaatc       420
     tcccaagttt cgagaggatg gccagatgac cggtcaccac gaatctaccg gaccaggtac       480
     cgcgctgagc agcgattcga cgtgccgggt gacgcagtat cagacggcgg gtgtgaacgc       540
```

From this we want a function that:

1. Discards the first line, as it does not contain any sequence
2. Iterates over all subsequent lines extracting the relevant sequence from them

Extracting the relevant sequence can be achieved by noting that each sequence line consists of seven “words”, where a word is defined as a set of characters separated by one or more white spaces. The first six words correspond to sequence, whereas the last word is an index listing the number of nucleotide bases.

Let us implement such a function. Add the lines below to the top of the `gc_content.py` file.

```
1  def parse_dna(file_handle):
2      """Return DNA sequence as a list."""
3      first_line = file_handle.next()  # Discard the first line.
4      sequence = []
5      for line in file_handle:
6          words = line.split()
7          seq_string = "".join(words[:-1])
8          seq_list = list(seq_string)
9          sequence.extend(seq_list)
10     return sequence
```

There are a couple of new string methods introduced in the above, let’s explain them now.

Let’s look at line six first.

```
6          words = line.split()
```

Here we use the `split()` method to split the string into a list of words, by default the `split()` method splits text based on one or more white space characters.

On line seven we use the `join()` method to join the words together.

```
7          seq_string = "".join(words[:-1])
```

In this instance there are no characters separating the words to be joined. It is worth clarifying this with an example, if we wanted to join the words using a comma character one would use the syntax `",".join(words[:-1])`.

On line seven it is also worth noting that we exclude the last word (the numerical index) by making use of list slicing `words[:-1]`.

Finally, on line nine we make use of the list method `extend()`, this extends the existing `sequence` list with all the elements from the `seq_list` list. Because `words` `seq_string` and `seq_list` will be overwritten when the loop moves on to the next line in the input file.

```
9          sequence.extend(seq_list)
```

Adding elements to lists `append()` vs `extend()`

There are two main ways of adding elements to a list. The first is the append method, which adds a single element.

```
>>> colors = ["red", "green"]
>>> colors.append("blue")
```

The `extend()` method is different in that it extends the list with the content of another list.

```
>>> colors.extend(["yellow", "purple"])
>>> print(colors)
['red', 'green', 'blue', 'yellow', 'purple']
```

Now let us update the `gc_content.py` script to initalise the sequence by parsing the `Sco.dna` file.

```
with open("Sco.dna", "r") as file_handle:
    sequence = parse_dna(file_handle)

for start, end, gc in sliding_window_analysis(sequence, gc_content):
    print(start, end, gc)
```

Finally, let us change the default `window_size` and `step_size` values. In the below I have split the function definition over two lines so as not to make the line exceed 78 characters. Exceeding 78 characters is considered poor "style" because it makes it difficult to read the code.

```
def sliding_window_analysis(sequence, function,
                            window_size=100000, step_size=50000):
    """Return an iterator that yields (start, end, property) tuples.

    Where start and end are the indices used to slice the input list
    and property is the return value of the function given the sliced
    list.
    """
```

Let us run the script again.

```
$ python gc_content.py
```

Note that this will produce a lot of output. To find out the number of lines that are generated we can make use of piping and the `wc -l` command (mnemonic `wc` word count, `-l` lines) .

```
$ python gc_content.py | wc -l
    173
```

The result makes sense as there are 8667507 base pairs in the sequence and we are stepping through it using a step size of 50000.

```
>>> 8667507 / 50000
173
```

7.12 Writing out the sliding window analysis

Finally we will write out the analysis to a text file. Since this data is tabular we will use the CSV file format. Furthermore, since we will want to plot the data using ggplot2 in Data visualisation (page 61) we will use a "tidy" data structure, see *Tidy data* (page 28) for details.

Edit the end of the `gc_content.py` script to make it look like the below.

```python
32  with open("Sco.dna", "r") as file_handle:
33      sequence = parse_dna(file_handle)
34
35  with open("local_gc_content.csv", "w") as file_handle:
36      header = "start,middle,end,gc_content\n"
37      file_handle.write(header)
38      for start, end, gc in sliding_window_analysis(sequence, gc_content):
39          middle = (start + end) / 2
40          row = "{},{},{},{}\n".format(start, middle, end, gc)
41          file_handle.write(row)
```

On line 35 we open a file handle to write to. On lines 36 and 37 we write a header to the CSV file. Lines 38 to 41 then performs the sliding window analysis and writes the results as rows, or lines if you prefer, to the CSV file. Line 39 calculates the middle of the local sequence by calculating the mean of the start and end positions.

The main new feature introduced in the code snippet above is on line 40 where we use Python's built in string formatting functionality. The matching curly braces in the string are replaced with the content of the `format()` string method. Let us illustrate this with an example in interactive mode.

```
>>> print("{},{},{},{}")
{},{},{},{}
>>> print("{},{},{},{}".format(1, 5, 10, 38.5))
1,5,10,38.5
```

Okay, let us see what happens when we run the script.

```
$ python gc_content.py
```

This should have created a file named `local_gc_content.csv` in the working directory.

```
$ ls
Sco.dna
gc_content.py
local_gc_content.csv
```

We can examine the top of this newly created file using the `head` command.

```
$ head local_gc_content.csv
start,middle,end,gc_content
0,50000,100000,69.124
50000,100000,150000,70.419
100000,150000,200000,72.495
150000,200000,250000,71.707
200000,250000,300000,71.098
250000,300000,350000,72.102
300000,350000,400000,72.712
350000,400000,450000,73.15
400000,450000,500000,73.27
```

Great, everything seems to be working. Let's start tracking our code using Git, see Keeping track of your work (page 31).

```
$ git init
$ git status
On branch master

Initial commit

```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Sco.dna
        gc_content.py
        local_gc_content.csv

nothing added to commit but untracked files present (use "git add" to track)
```

We have got three untracked files in our directory, the script, the input data and the output data. We don't want to track the input and the output data so let's create a `.gitignore` file and add those files to it.

```
Sco.dna
local_gc_content.csv
```

Let's check the status of our Git repository again.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        gc_content.py

nothing added to commit but untracked files present (use "git add" to track)
```

Let's start tracking the `gc_content.py` and the `.gitignore` files and take a snapshot of them in their current form.

```
$ git add gc_content.py .gitignore
$ git commit -m "Initial file import"
[master (root-commit) 6d8e0cf] Initial file import
 2 files changed, 43 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 gc_content.py
```

Well done! We have covered a lot of ground in this chapter. I suggest digging out some good music and chilling out for a bit.

7.13 Key concepts

- Python is a powerful scripting language that is popular in the scientific community
- You can explore Python's syntax using its interactive mode
- Variables and functions help us avoid having to repeat ourselves, the *DRY* principle
- When naming variables and functions explicit trumps succinct
- Loops are really powerful, they form the basis of automating repetitive tasks
- Files are accessed using file handles
- A file handle is a data structure that handles the book keeping of the position within the file and the mode in which it was opened

- The mode of the file handle determines how you will interact with the file
- Read mode only allows reading of a file
- Append mode will keep the existing content of a file and append to it
- Write mode will delete any previous content before writing to it

CHAPTER 8

Data visualisation

So far we have been learning how to crunch data. Now we will look into how to visualise it.

There are two main purposes of representing data visually:

1. To explore the data
2. To convey a message to an audience

In this chapter we will look at how both of these can be accomplished using R[38] and its associated ggplot2[39] package. This choice is based on ggplot2 being one of the best plotting tools available. It is easy to use and has got sensible defaults that result in beautiful plots out of the box. An added advantage is that this gives us a means to get more familiar with R, which is an awesome tool for statistical computing (although this topic is beyond the scope of this book). Other tools will be discussed later in the section *Other useful tools for scripting the generation of figures* (page 82).

We will start off by exploring Fisher's Iris flower data set[40]. This will be an exercise in data exploration.

Finally, we will use R and ggplot2 to build up a sliding window plot of the GC content data from the Data analysis (page 43) chapter. This will serve to illustrate some points to consider when trying to convey a message to an audience.

8.1 Starting R and loading the Iris flower data set

R like Python can be run in an interactive shell. This is a great way to get familiar with R. To access R's interactive shell simply type `R` into your terminal and press enter.

```
$ R

R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin14.5.0 (64-bit)

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

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
```

[38] https://www.r-project.org/
[39] http://ggplot2.org/
[40] https://en.wikipedia.org/wiki/Iris_flower_data_set

```
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

>
```

R comes bundled with a number of data sets. To view these you can use the `data()` function, which lists all the available data sets.

```
> data()
Data sets in package ‘datasets’:

AirPassengers           Monthly Airline Passenger Numbers 1949-1960
BJsales                 Sales Data with Leading Indicator
BJsales.lead (BJsales)
                        Sales Data with Leading Indicator
BOD                     Biochemical Oxygen Demand
CO2                     Carbon Dioxide Uptake in Grass Plants
...
```

If you want to stop displaying the data sets without having to scroll to the bottom press the q button on your keyboard.

We are interested in the Iris flower data set, called `iris`, let us load it.

```
> data(iris)
```

This loads the `iris` data set into the workspace. You can list the content of the workspace using the `ls()` function.

```
> ls()
[1] "iris"
```

What is a workspace?

R has the concept of a "workspace". The workspace is the current working environment and includes any user defined objects. At the end of a R session the user can save the workspace. If saved the workspace will be automatically loaded the next time R is started.

8.2 Understanding the structure of the `iris` data set

First of all let us find out about the internal structure of the `iris` data set using the `str()` funciton.

```
> str(iris)
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

This reveals that the `iris` data set is a data frame with 150 observations and five variables. It is also worth noting that `Species` is recorded as a Factor data structure. This means that it has categorical data. In this case three different species.

In R a data frame is a data structure for storing two-dimensional data. In a data frame each column contains the same type of data and each row has values for each column.

What is the difference between vectors, lists and data frames?

As well as data frames R also has concepts of vectors and lists. A vector is a list where each item is of the same type. A list is more flexible in that the items can be of different types. The data frame is essentially a list of equal length lists.

You can find the names of the columns in a data frame using the `names()` function.

```
> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

To find the number of columns and rows one can use the `ncol()` and `nrow()` functions, respectively.

```
> ncol(iris)
[1] 5
> nrow(iris)
[1] 150
```

To view the first six rows of a data frame one can use the `head()` function.

```
    > head(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
```

To view the last six rows of a data frame one can use the `tail()` function.

```
> tail(iris)
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

Warning: Often the most difficult aspect of data visualisation using R and ggplot2 is getting the data into the right structure. The `iris` data set is well structured in the sense that each observation is recorded as a separate row and each row has the same number of columns. Furthermore, each column in a row has a value, note for example how each row has a value indicating the `Species`. Once your data is well structured plotting it becomes relatively easy. However, if you are used to adding heterogeneous data to Excel the biggest issue that you will face is formatting the data so that it becomes well structured and can be loaded as a data frame in R. For more detail on how to structure your data see *Tidy data* (page 28).

8.3 A note on statistics in R

Although this book is not about statistics it is worth mentioning that R is a superb tool for doing statistics. It has many built in functions for statistical computing. For example to calculate the median `Sepal.Length` for the `iris` data one can use the built in `median()` function.

```
> median(iris$Sepal.Length)
[1] 5.8
```

In the above the `$` symbol is used to specify the column of interest in the data frame.

Another useful tool for getting an overview of a data set is the `summary()` function.

```
> 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  
```

8.4 Default plotting in R

Before using ggplot2 let us have a look at how to generate default plots in R.

First of all let us plot a histogram of the `Sepal.Length` (Fig. 8.1).

```
> hist(iris$Sepal.Length)
```

Scatter plots can be produced using the `plot()` function. The command below produces a scatter plot of `Sepal.Width` versus `Sepal.Length` (Fig. 8.2).

```
> plot(iris$Sepal.Length, iris$Sepal.Width)
```

Finally, a decent overview of the all data can be obtained by passing the entire data frame to the `plot()` function (Fig. 8.3).

```
> plot(iris)
```

R's built in plotting functions are useful for getting quick exploratory views of the data. However, they are a bit dull. In the next section we will make use of the ggplot2 package to make more visually pleasing and informative figures.

8.5 Installing the ggplot2 package

If you haven't used it before you will need to install the ggplot2 package. Detailed instructions can be found in *Installing R packages* (page 144). In summary you need to run the command below.

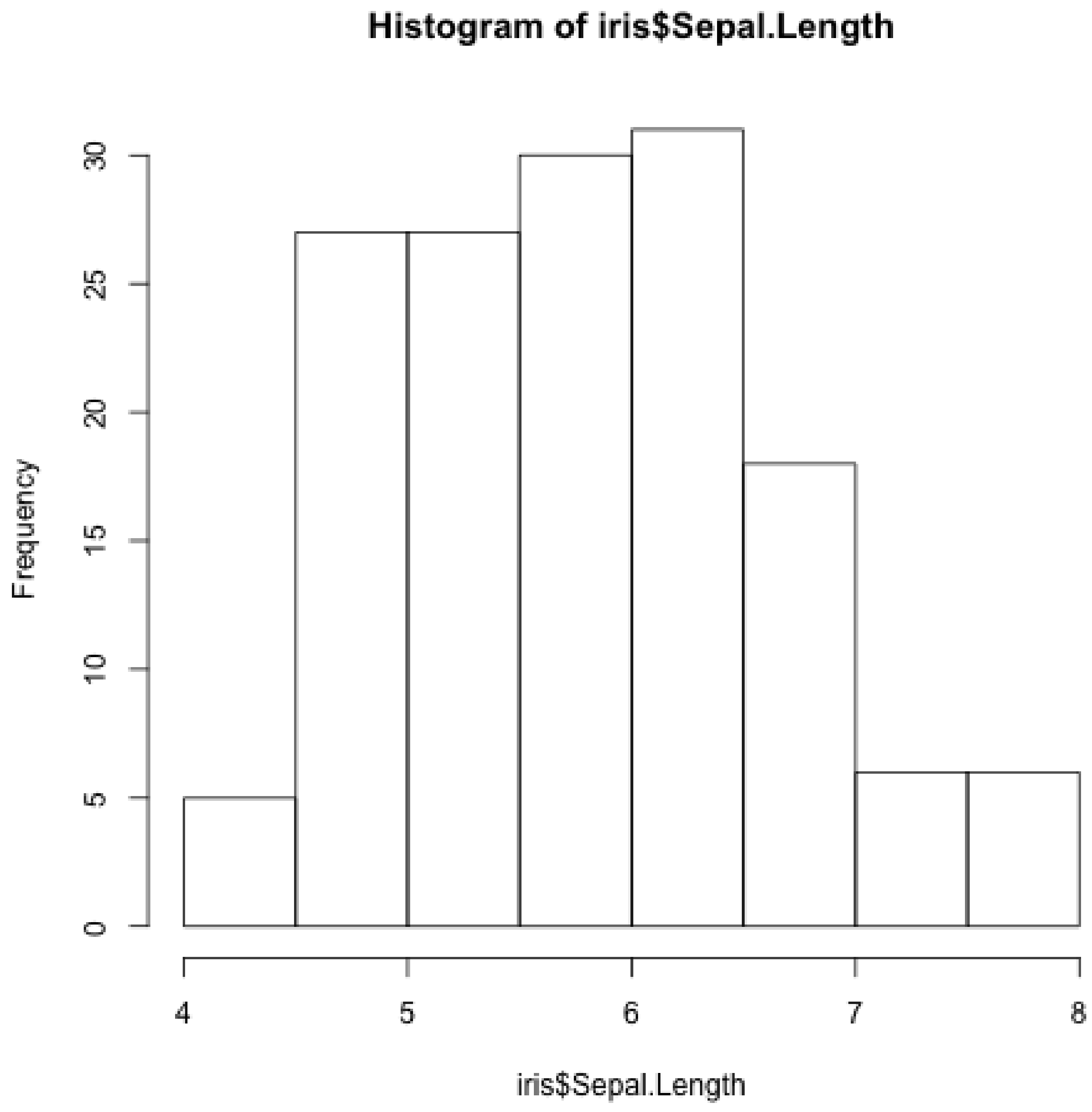


Fig. 8.1: Histogram of Iris sepal length data generated using R's built in `hist()` function.

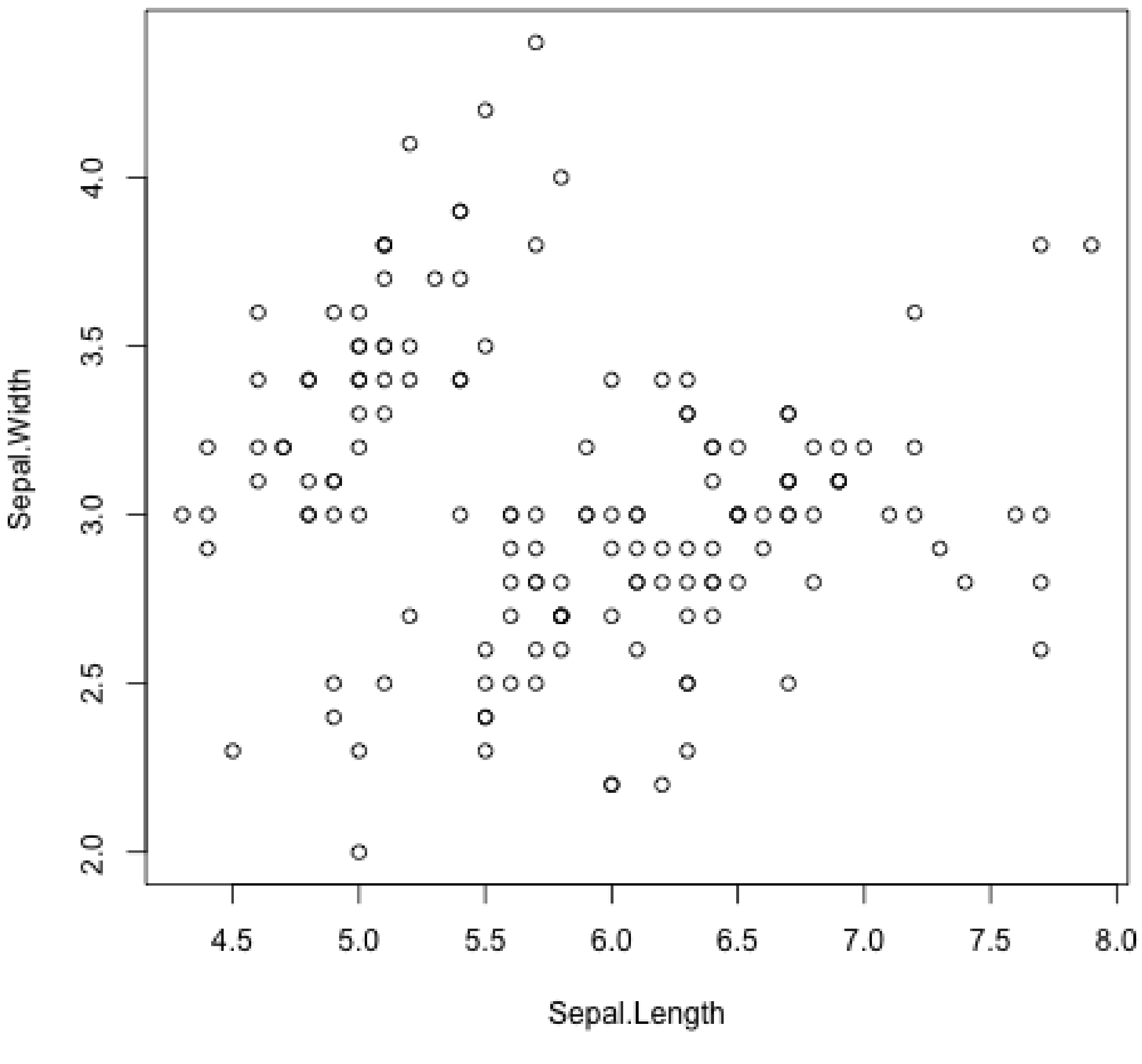


Fig. 8.2: Scatter plot of Iris sepal length vs width generated using R's built in `plot()` function.

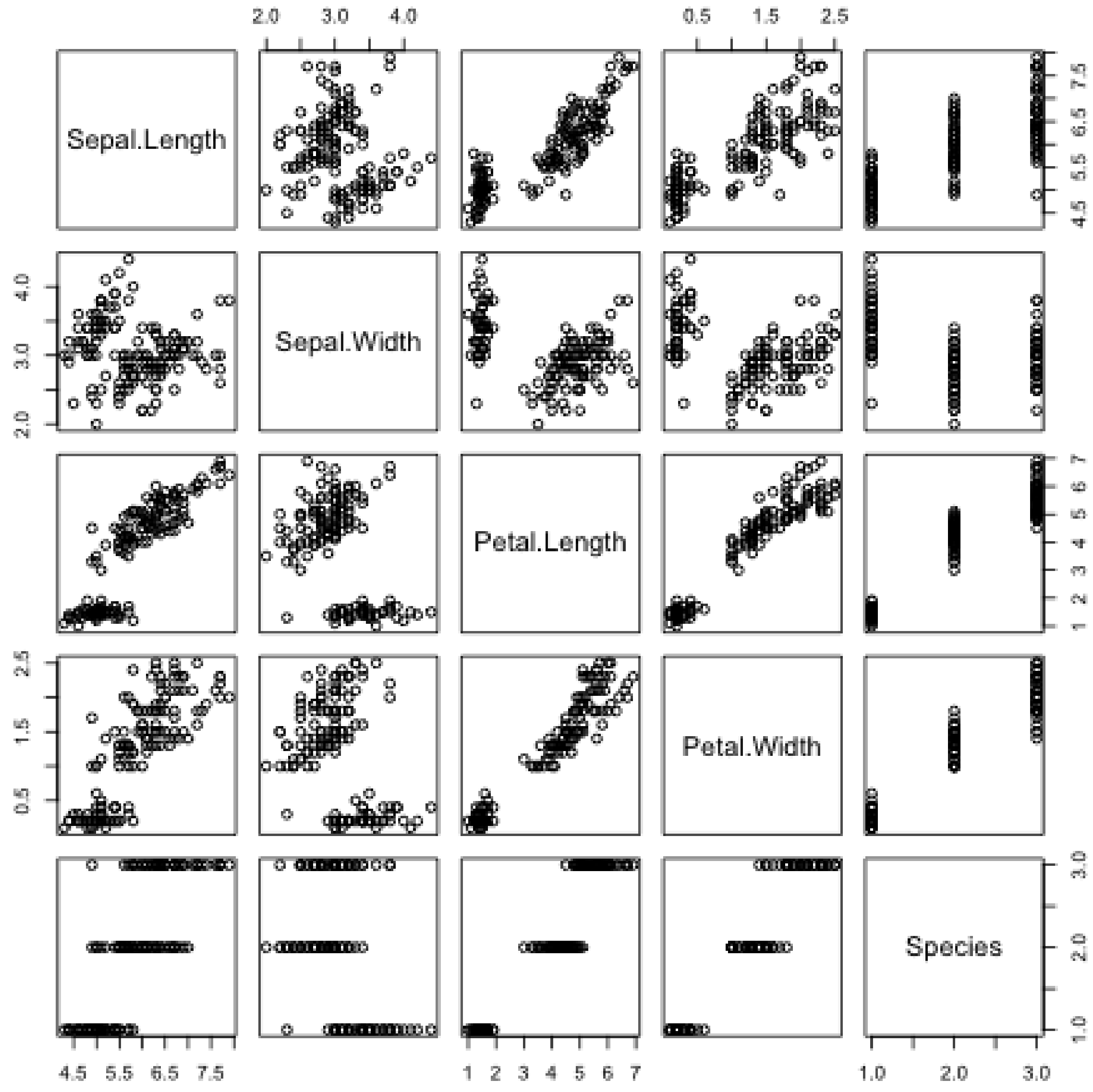


Fig. 8.3: Overview plot of Iris data using R's built in `plot()` function.

```
> install.packages("ggplot2")
```

8.6 Loading the ggplot2 package

In order to make use of the ggplot2 package we need to load it. This is achieved using the `library()` function.

```
> library("ggplot2")
```

8.7 Plotting using ggplot2

To get an idea of what it feels like to work with ggplot2 let us re-create the previous histogram and scatter plot with it.

Let us start with the histogram (Fig. 8.4).

```
> ggplot(data=iris, mapping=aes(Sepal.Length)) + geom_histogram()
```

The syntax used may look a little bit strange at first. However, before going into more details about what it all means let's create the scatter plot (Fig. 8.5) to get a better feeling of how to work with ggplot2.

```
> ggplot(data=iris, mapping=aes(x=Sepal.Length, y=Sepal.Width)) + geom_point()
```

In the examples above we provide the `ggplot()` function with two arguments `data` and `mapping`. The former contains the data frame of interest and the latter specifies the columns(s) to be plotted.

The `ggplot` function returns a ggplot object that can be plotted. However, in order to view an actual plot one needs to add a layer to the ggplot object defining how the data should be presented. In the examples above this is achieved using the `+ geom_histogram()` and `+ geom_point()` syntax.

A ggplot object consists of separate layers. The reason for separating out the generation of a figure into separate layers is to allow the user to better be able to reason about the best way to represent the data.

The three layers that we have come across so far are:

- Data: the data to be plotted
- Aesthetic: how the data should be mapped to the aesthetics of the plot
- Geom: what type of plot should be used

Of the above the "aesthetic" layer is the trickiest to get one's head around. However, take for example the scatter plot, one aesthetic choice that we have made for that plot is that the `Sepal.Length` should be on the x-axis and the `Sepal.Width` should be on the y-axis.

To reinforce this let us augment the scatter plot by sizing the points in the scatter plot by the `Petal.Width` and coloring them by the `Species` (Fig. 8.6), all of which could be considered to be aesthetic aspects of how to plot the data.

```
> ggplot(iris, aes(x=Sepal.Length,
+                  y=Sepal.Width,
+                  size=Petal.Width,
+                  color=Species)) + geom_point()
```

In the above the secondary prompt, represented by the plus character (+), denotes a continuation line. In other words R interprets the above as one line of code.

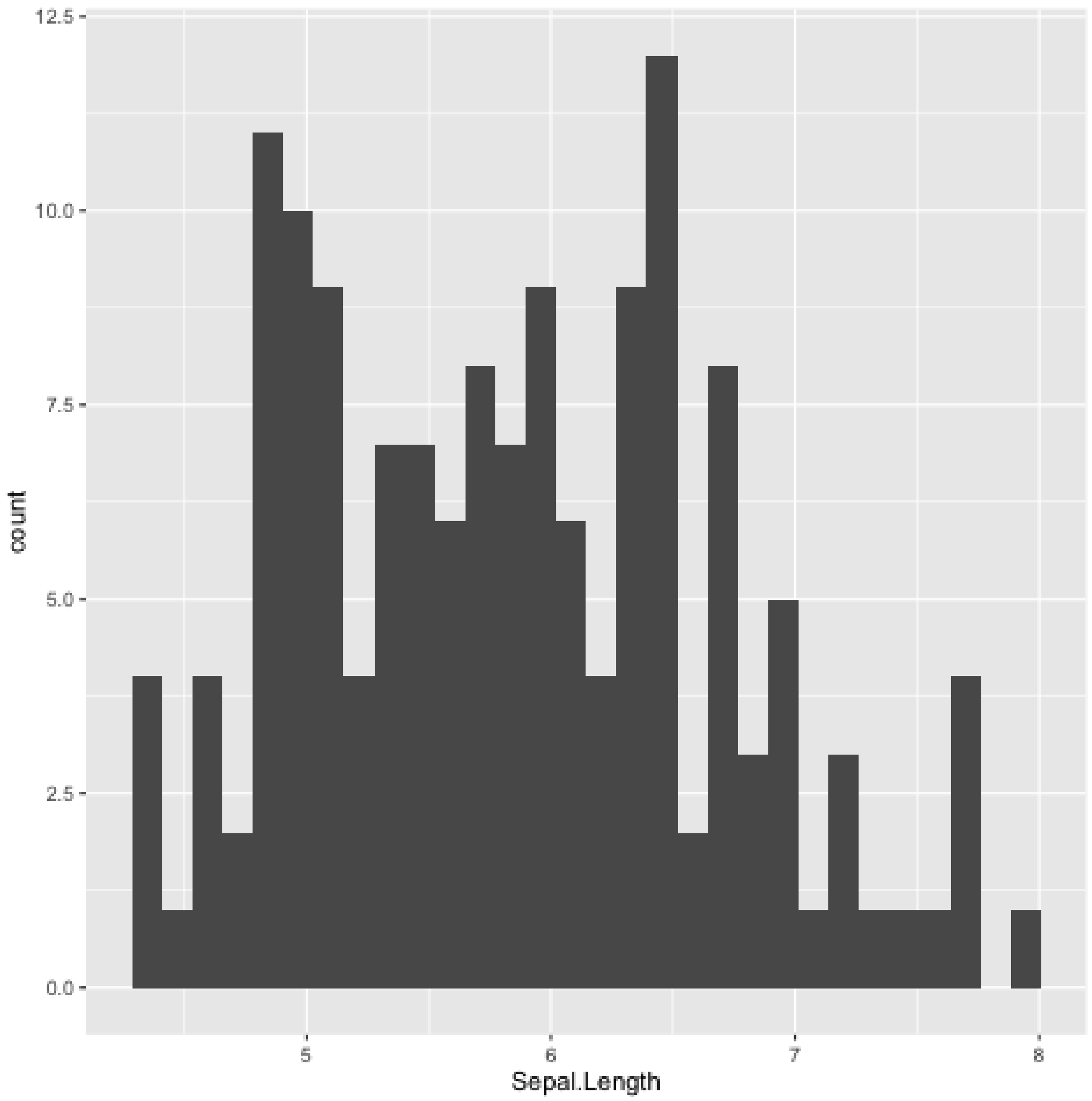


Fig. 8.4: Histogram of Iris sepal length data generated using R's ggplot2 package. The default bin width used is different from the one used by R's built in `hist()` function, hence the difference in the appearance of the distribution.

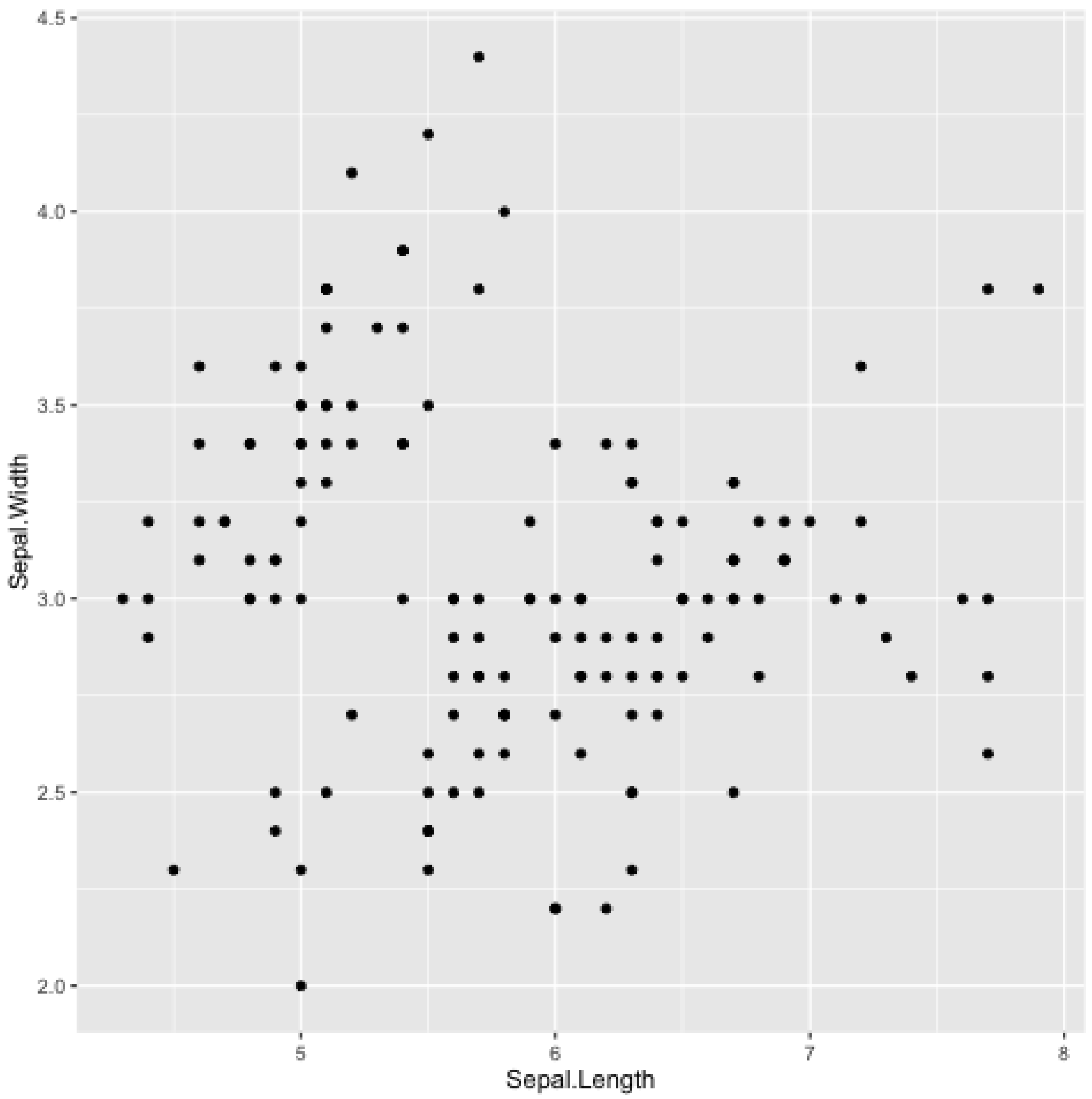


Fig. 8.5: Scatter plot of Iris sepal length vs width generated using R's ggplot2 package.

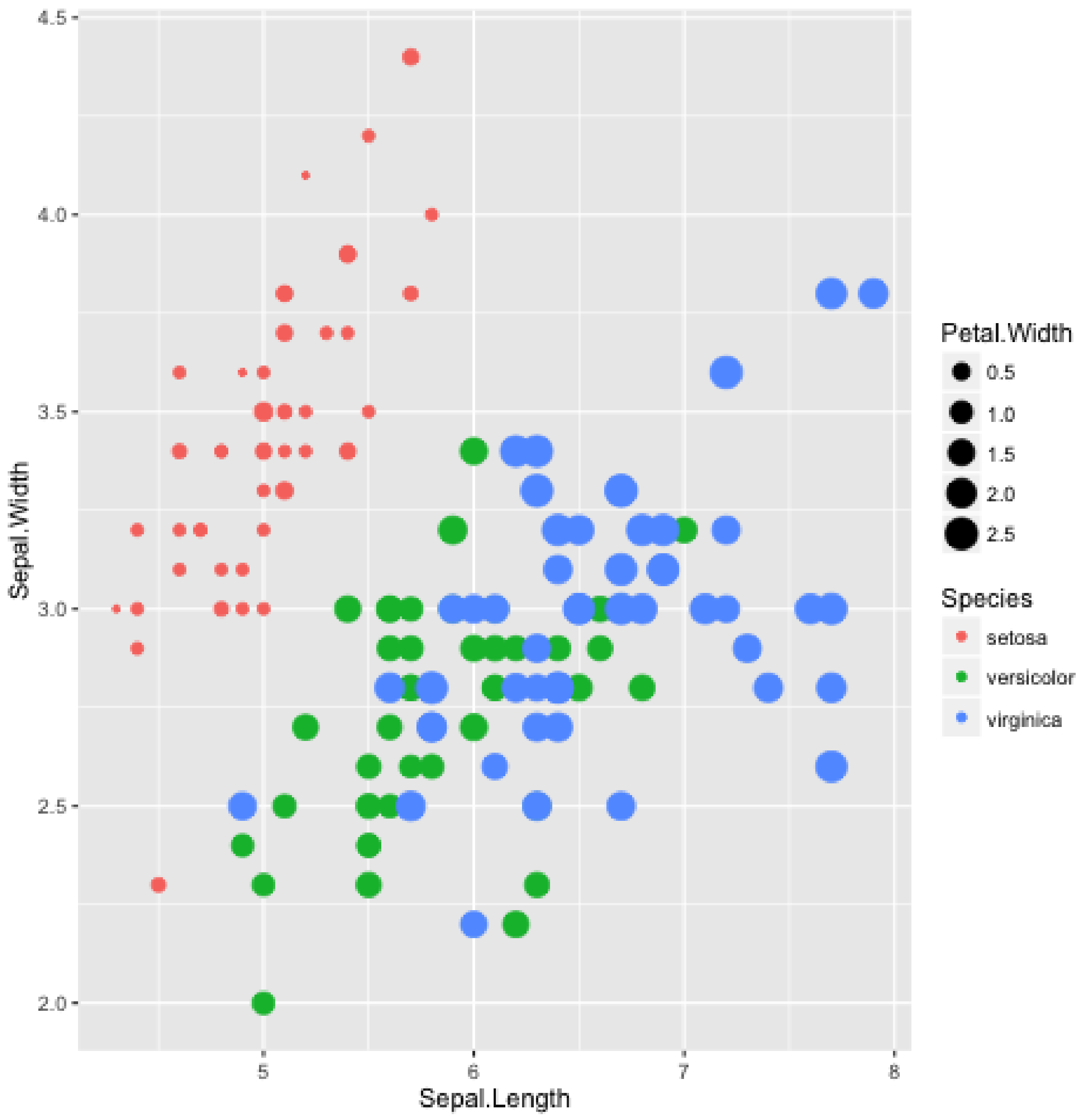


Fig. 8.6: Scatter plot of Iris sepal length vs width, where the size of each point represents the petal width and the colour is used to indicate the species.

The information in the scatterplot is now four dimensional! The x- and y-axis show `Sepal.Length` and `Sepal.Width`, the size of the point indicates the `Petal.Width` and the colour shows the `Species`. By adding `Petal.Width` and `Species` as additional aesthetic attributes to the figure we can start to discern structure that was previously hidden.

8.8 Available "Geoms"

The ggplot2 package comes bundled with a wide range of geoms ("geometries" for plotting data). So far we have seen `geom_histogram()` and `geom_point()`. To find out what other geoms are available you can start typing geom_ into a R session and hit the Tab key to list the available geoms using tab-completion.

```
> geom_
geom_abline      geom_errorbarh   geom_quantile
geom_area        geom_freqpoly    geom_raster
geom_bar         geom_hex         geom_rect
geom_bin2d       geom_histogram   geom_ribbon
geom_blank       geom_hline       geom_rug
geom_boxplot     geom_jitter      geom_segment
geom_contour     geom_label       geom_smooth
geom_count       geom_line        geom_spoke
geom_crossbar    geom_linerange   geom_step
geom_curve       geom_map         geom_text
geom_density     geom_path        geom_tile
geom_density_2d  geom_point       geom_violin
geom_density2d   geom_pointrange  geom_vline
geom_dotplot     geom_polygon
geom_errorbar    geom_qq
```

These work largely as expected, for example the `geom_boxplot()` results in a boxplot and the `geom_line()` results in a line plot. For illustrations of these different geoms in action have a look at the examples in the ggplot2 documentation[41].

8.9 Scripting data visualisation

Now that we have a basic understanding of how to interact with R and the functionality in the ggplot library, let's take a step towards automating the figure generation by recording the steps required to plot the data in a script.

Let us work on the histogram example. In the code below we store the ggplot object in the variable `g` and make use of `ggsave()` to write the plot to a file named `iris_sepal_length_histogram.png`. Save the code below to a file named `iris_sepal_length_histogram.R` using your favorite text editor.

```r
library("ggplot2")
data(iris)

g <- ggplot(data=iris, aes(Sepal.Length)) +
     geom_histogram()

ggsave('iris_sepal_length_histogram.png')
```

To run this code we make use of the program `Rscript`, which comes bundled with your R installation.

[41] http://docs.ggplot2.org/current/index.html

```
$ Rscript iris_sepal_length_histogram.R
```

This will write the file `iris_sepal_length_histogram.png` to your current working directory.

8.10 Faceting

In the extended scatterplot example (Fig. 8.6) we found that it was useful to be able to visualise which data points belonged to which species. Maybe it would be useful to do something similar with the data in our histogram.

In order to achieve this ggplot2 provides the concept of faceting, the ability to split your data by one or more variables. Let us split the data by `Species` using the `facet_grid()` function (Fig. 8.7).

```
library("ggplot2")
data(iris)

g <- ggplot(data=iris, aes(Sepal.Length)) +
     geom_histogram() +
     facet_grid(Species ~ .)

ggsave('iris_sepal_length_histogram.png')
```

In the above the `facet_grid(Species ~ .)` states that we want one `Species` per row, as opposed to one species per column (`facet_grid(. ~ Species)`). Replacing the dot (`.`) with another variable would result in faceting the data into a two dimensional grid.

Faceting is a really powerful feature of ggplot2! It allows you to easily split data based on one or more variables and can result in insights about underlying structures and multivariate trends.

8.11 Adding more colour

The faceted histogram plot (Fig. 8.7) clearly illustrates that there are differences in the distributions of the sepal length between the different Iris species.

However, suppose that the faceted histogram figure was meant to be displayed next to the extended scatter plot (Fig. 8.6). To make it easier to make the mental connection between the two data representations it may be useful to colour the histograms by species as well.

The colour of a histogram is an aesthetic characteristic. Let us add the fill colour as an aesthetic to the histogram geometric object (Fig. 8.8).

```
library("ggplot2")
data(iris)

g <- ggplot(data=iris, aes(Sepal.Length)) +
     geom_histogram(aes(fill=Species)) +
     facet_grid(Species ~ .)

ggsave('iris_sepal_length_histogram.png')
```

8.12 Purpose of data visualisation

There are two main purposes of representing data visually:

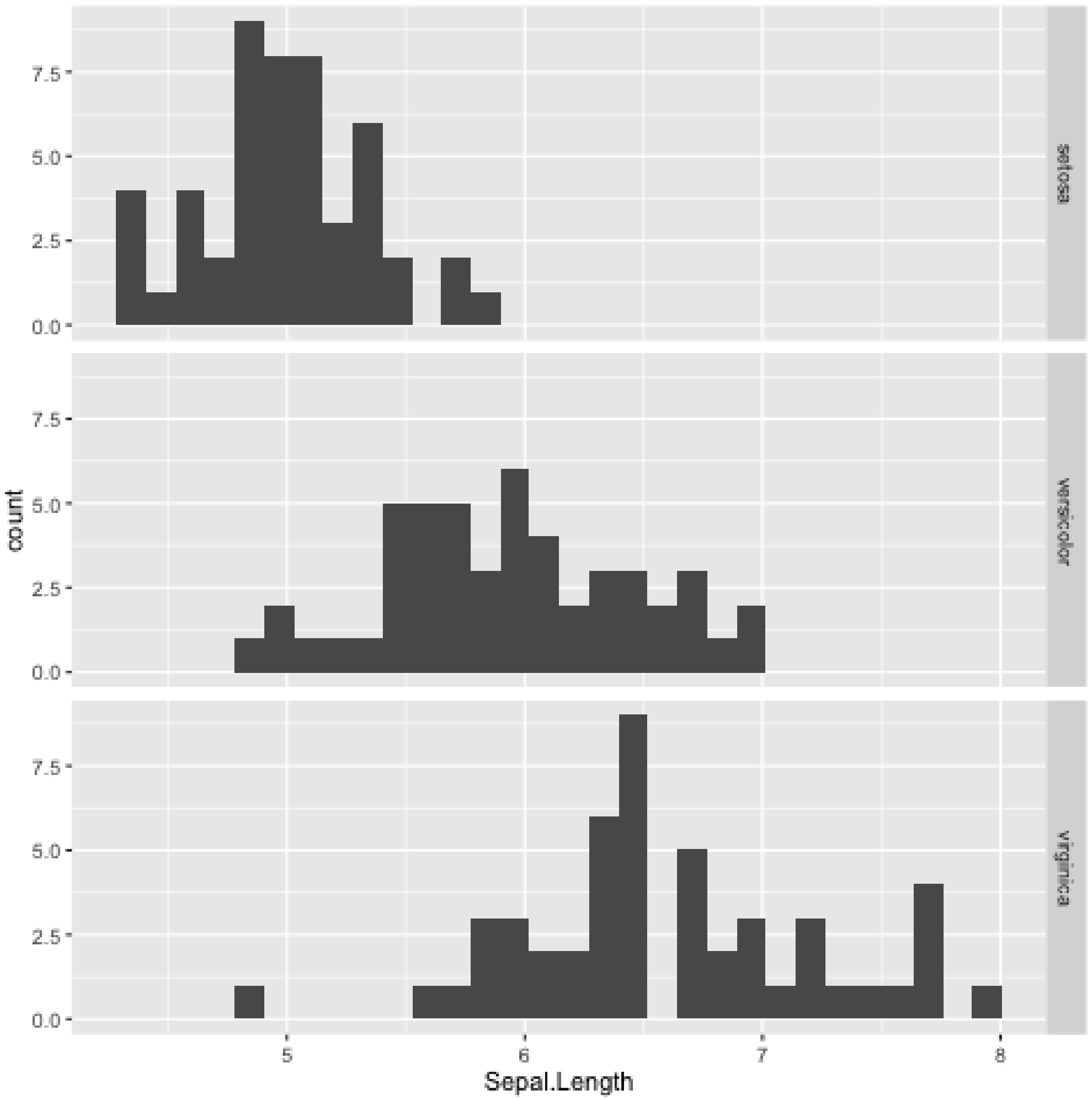


Fig. 8.7: Histogram of Iris sepal length data faceted by species.

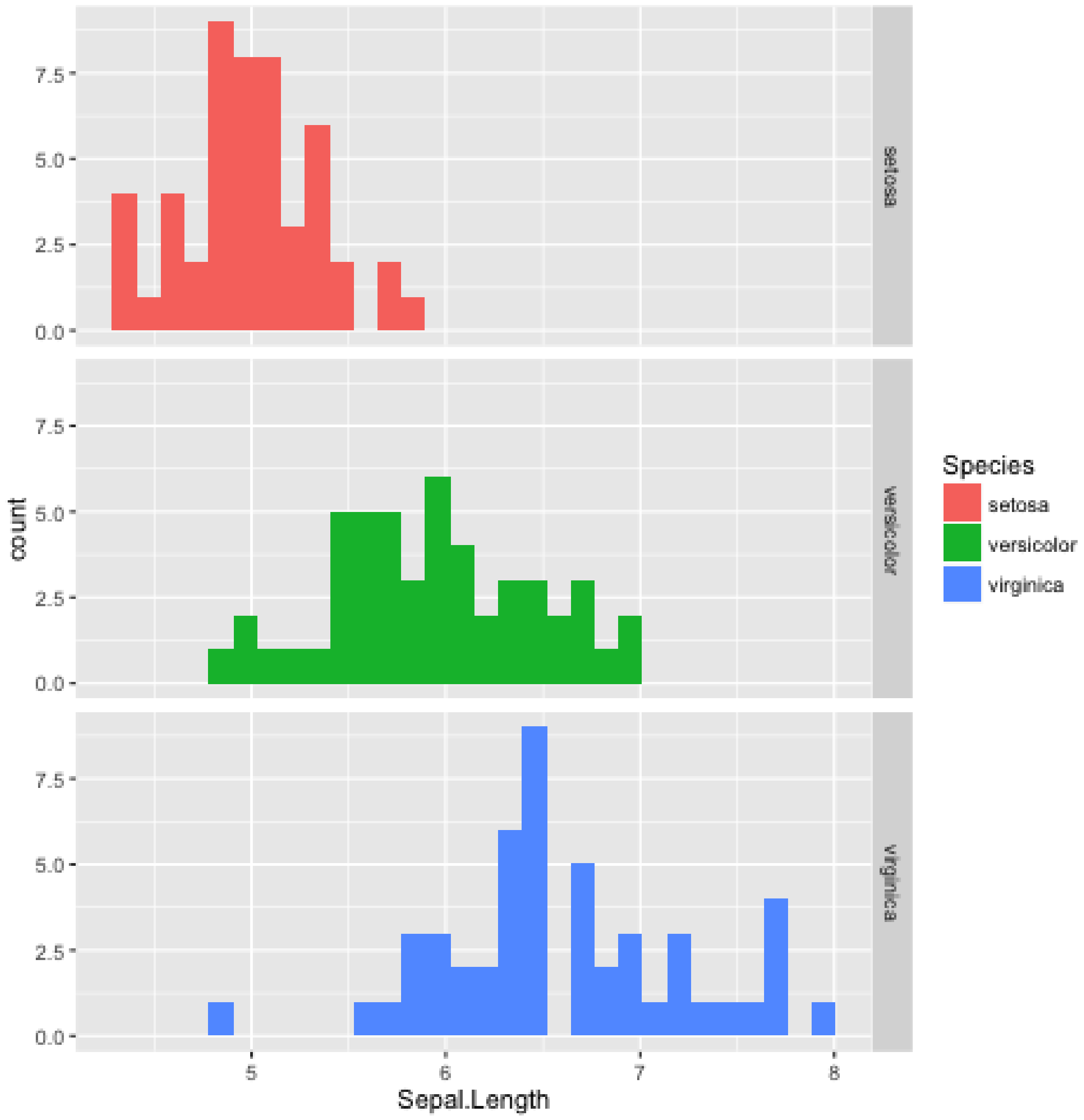


Fig. 8.8: Histogram of Iris sepal length data faceted and coloured by species.

1. To explore the data
2. To convey a message to an audience

So far we have been focussing on the former. Let us now think a little bit more about the latter, conveying a message to an audience. In other words creating figures for presentations and publications.

8.13 Conveying a message to an audience

The first step in creating an informative figure is to consider who the intended audience is. Is it the general public, school children or plant biologists? The general public may be interested in trends, whereas a plant scientist may be interested in a more detailed view.

Secondly, what is the message that you want to convey? Stating this explicitly will help you when making decisions about the content and the layout of the figure.

Thirdly, what medium will be used to display the figure? Will it be displayed for a minute on a projector or will it be printed in an article? The audience will have less time to absorb details from the former.

So suppose we wanted to create a figure intended for biologists, illustrating that there is not much variation in the local GC content of *Streptomyces coelicolor* A3(2), building on the example in Data analysis (page 43). Let us further suppose that the medium is in a printed journal where the figure can have a maximum width of 89 mm.

First of all make sure that you are in the `S.coelicolor-local-GC-content` directory created in Data analysis (page 43).

```
$ cd S.coelicolor-local-GC-content
```

Now create a file named `local_gc_content_figure.R` and add the R code below to it.

```
library("ggplot2")

df = read.csv("local_gc_content.csv", header=T)

g1 = ggplot(df, aes(x=middle, y=gc_content))
g2 = g1 + geom_line()

ggsave("local_gc_content.png", width=89, height=50, units="mm")
```

The code above loads the data and plots it as a line, resulting in a local GC content plot. Note also that we specify the width and height of the plot in millimeters in the `ggsave()` function.

Let's try it out.

```
$ Rscript local_gc_content_figure.R
```

It should create a file named `local_gc_content.png` containing the image in Fig. 8.9.

The scale of the y-axis makes the plot misleading. It looks like there is a lot of variation in the data. Let's expand the y range to span from 0 to 100 percent (Fig. 8.10).

```
library("ggplot2")

df = read.csv("local_gc_content.csv", header=T)

g1 = ggplot(df, aes(x=middle, y=gc_content))
g2 = g1 + geom_line()
g3 = g2 + ylim(0, 100)
```

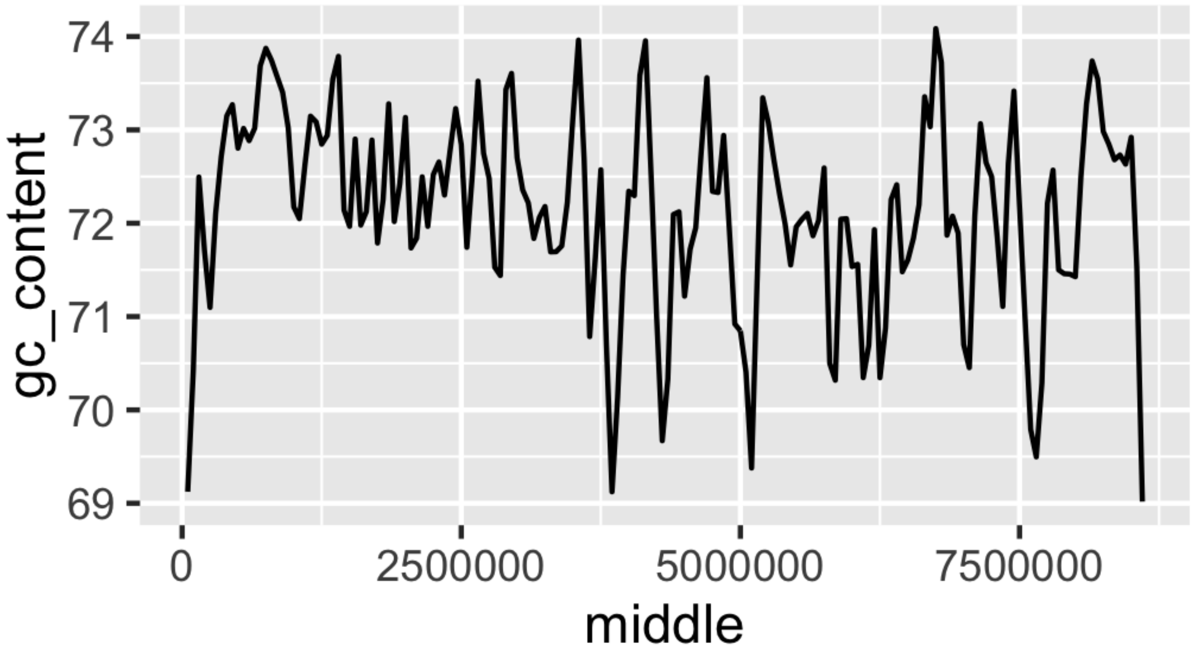


Fig. 8.9: Initial attempt at plotting local GC content.

```
ggsave("local_gc_content.png", width=89, height=50, units="mm")
```

At the moment it looks like the line is floating in mid-air. This is because ggplot adds some padding to the limits. Let's turn this off (Fig. 8.11).

```
library("ggplot2")

df = read.csv("local_gc_content.csv", header=T)

g1 = ggplot(df, aes(x=middle, y=gc_content))
g2 = g1 + geom_line()
g3 = g2 + ylim(0, 100)
g4 = g3 + coord_cartesian(expand=FALSE)

ggsave("local_gc_content.png", width=89, height=50, units="mm")
```

The labels on the x-axis are a bit difficult to read. To make it easier to understand the content of the x-axis let's scale it to use kilobases as its units (Fig. 8.12).

```
library("ggplot2")

df = read.csv("local_gc_content.csv", header=T)

g1 = ggplot(df, aes(x=middle, y=gc_content))
g2 = g1 + geom_line()
g3 = g2 + ylim(0, 100)
g4 = g3 + coord_cartesian(expand=FALSE)
g5 = g4 + scale_x_continuous(labels=function(x)x/1000)

ggsave("local_gc_content.png", width=89, height=50, units="mm")
```

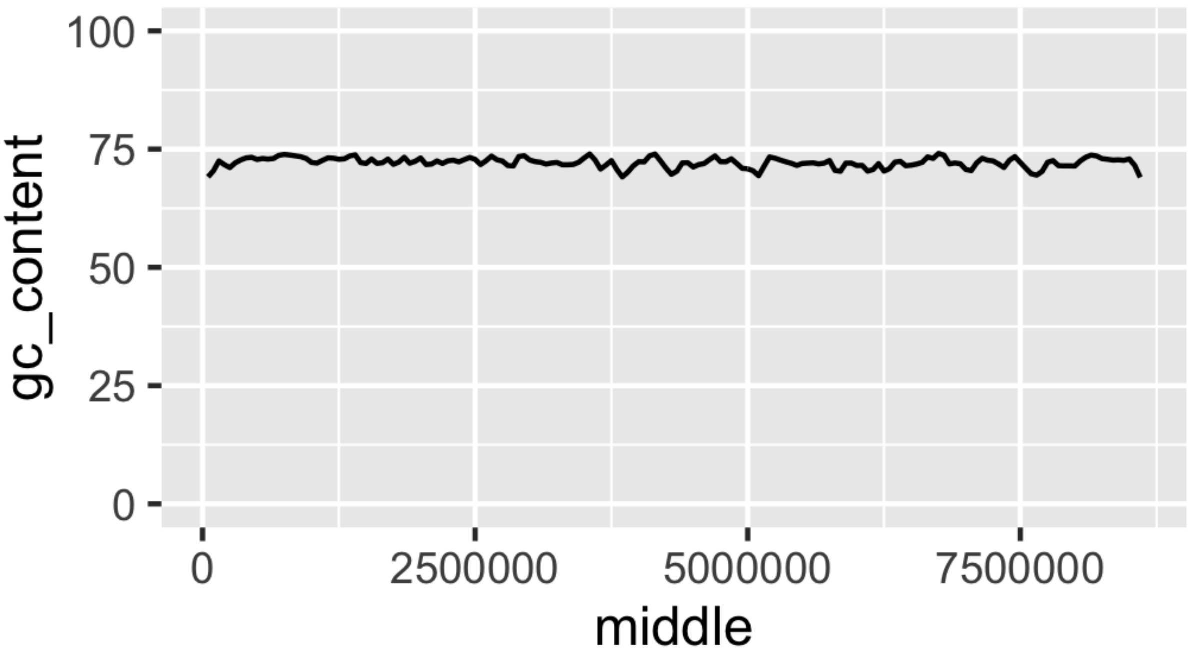


Fig. 8.10: Local GC content with y-axis scaled corretly.

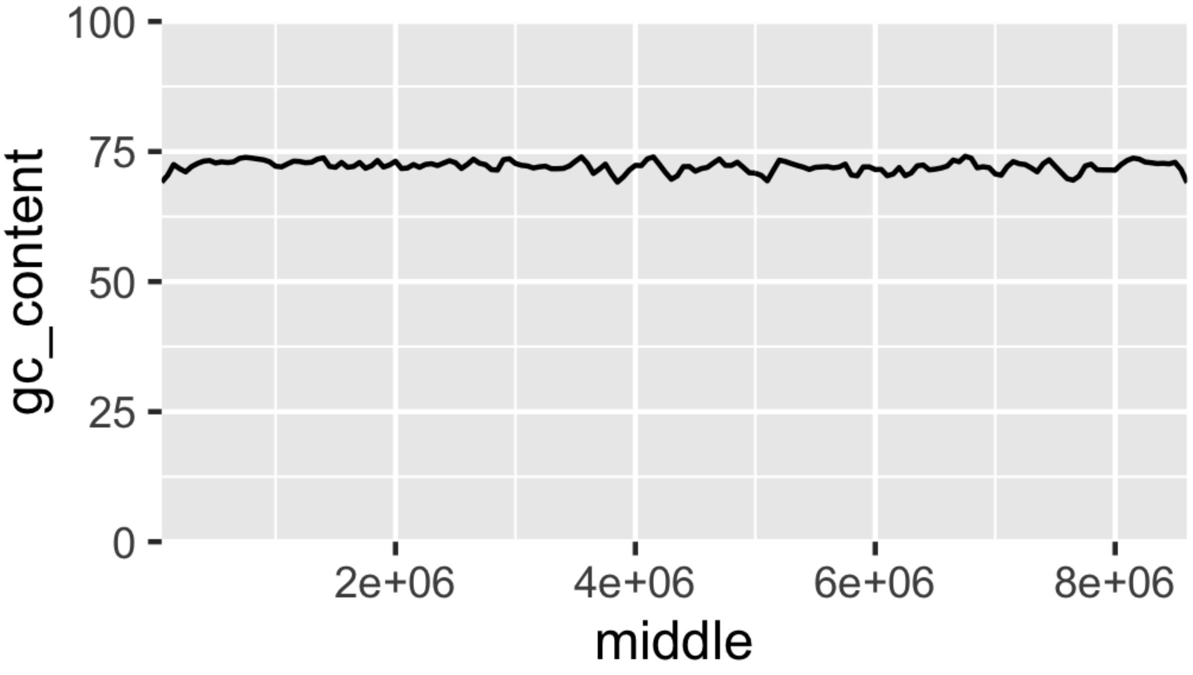


Fig. 8.11: Local GC content without padding.

In the above `function(x)x/1000` is a function definition. The function returns the input (`x`) divided by 1000. In this case the function is passed anonymously (the function is not given a name) to the `labels` argument of the `scale_x_continuous()` function.

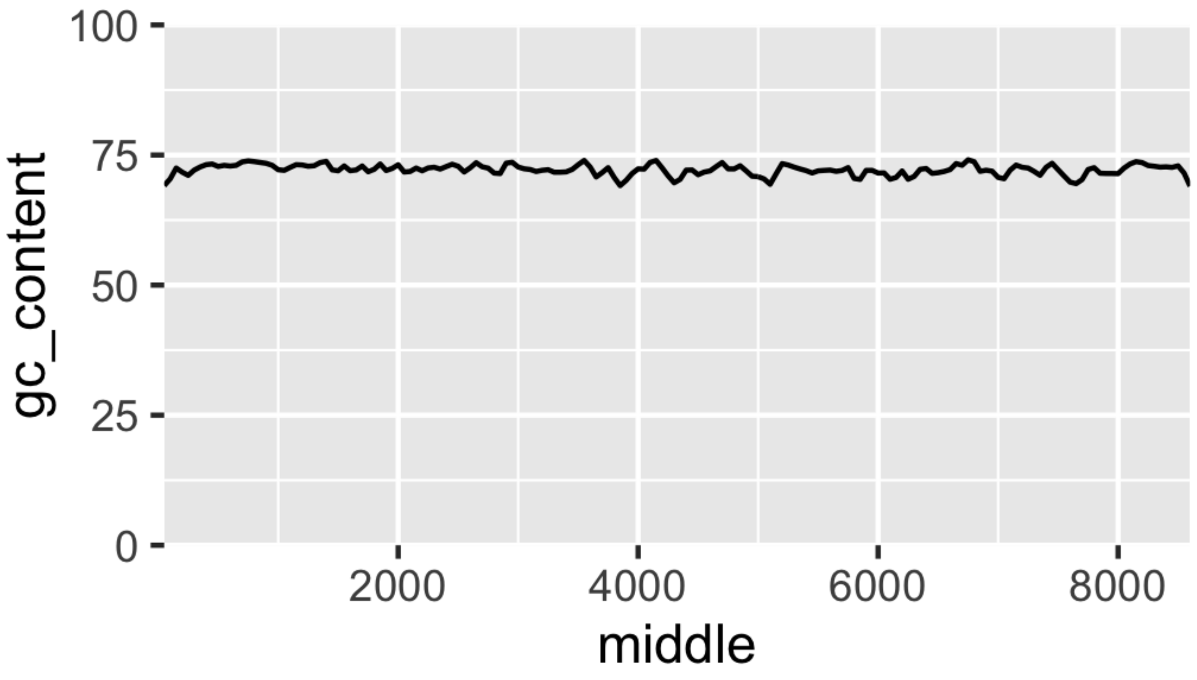


Fig. 8.12: Local GC content with kilobases as the x axis units.

Finally, let us add labels to the x-axis (Fig. 8.13).

```
library("ggplot2")

df = read.csv("local_gc_content.csv", header=T)

g1 = ggplot(df, aes(x=middle, y=gc_content))
g2 = g1 + geom_line()
g3 = g2 + ylim(0, 100)
g4 = g3 + coord_cartesian(expand=FALSE)
g5 = g4 + scale_x_continuous(labels=function(x)x/1000)
g6 = g5 + xlab("Nucleotide position (KB)") + ylab("GC content (%)")

ggsave("local_gc_content.png", width=89, height=50, units="mm")
```

This is a good point to start tracking the `local_gc_content_figure.R` file in version control, see Keeping track of your work (page 31) for more details.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        local_gc_content.png
        local_gc_content_figure.R

nothing added to commit but untracked files present (use "git add" to track)
```

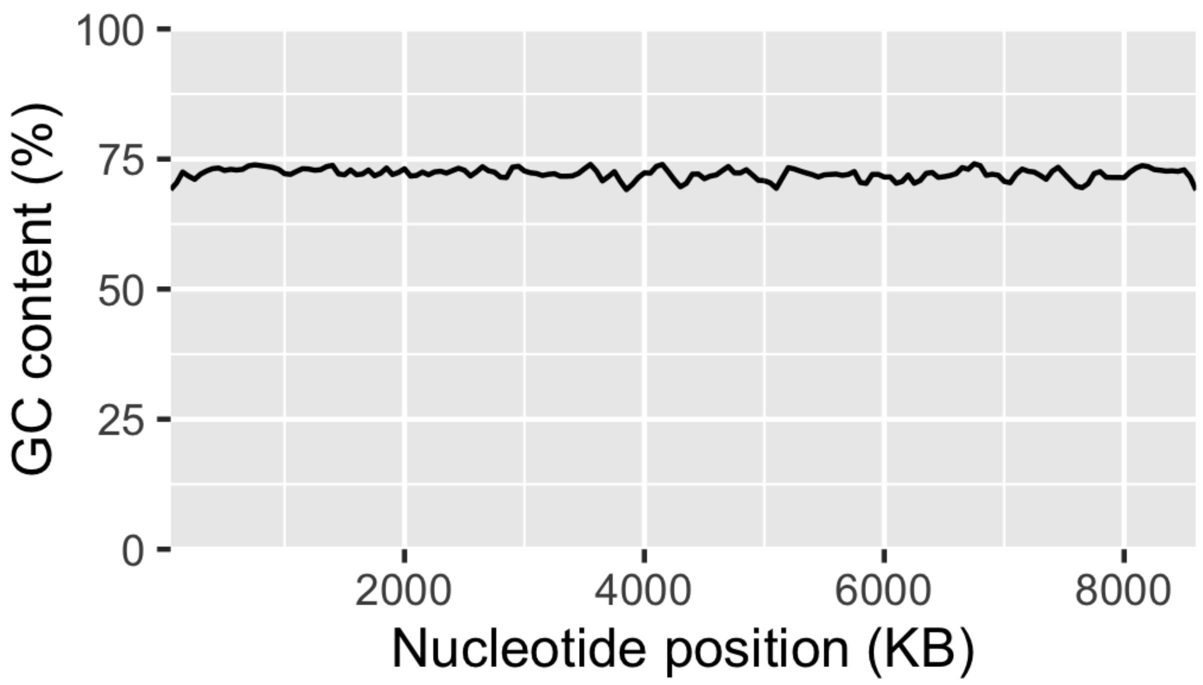


Fig. 8.13: Local GC content with labelled axis.

So we have two untracked files: the R script and the PNG file the R script generates. We do not want to track the latter in version control as it can be generated from the script. Let us therefore update our `.gitignore` file.

```
Sco.dna
local_gc_content.csv
local_gc_content.png
```

Let's check the status of the Git repository now.

```
$ 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 in working directory)

        modified:   .gitignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        local_gc_content_figure.R

no changes added to commit (use "git add" and/or "git commit -a")
```

Great, let us add the `local_gc_content_figure.R` and `.gitignore` files and commit a snapshot.

```
$ git add local_gc_content_figure.R .gitignore
$ git commit -m "Added R script for generating local GC plot"
[master cba7277] Added R script for generating local GC plot
 2 files changed, 13 insertions(+)
```

```
create mode 100644 local_gc_content_figure.R
```

8.14 Writing a caption

All figures require a caption to help the audience understand how to interpret the plot.

In this particular case two particular items stand out as missing from being able to interpret the plot. First of all what is the source of the DNA, i.e. where is the data from? Secondly, what was the window and step sizes of the sliding window analysis?

It would also be appropriate to explicitly state what the message of the plot is, i.e. that there is not much variation in the local GC content. In this case we could further substantiate this claim by citing the mean and standard deviation of the local GC content across the genome.

```
> df = read.csv("local_gc_content.csv", header=T)
> mean(df$gc_content)
[1] 72.13948
> sd(df$gc_content)
[1] 1.050728
```

Below is the final figure and its caption (Fig. 8.14). The caption has been added by the tool chain used to build this book, not by R.

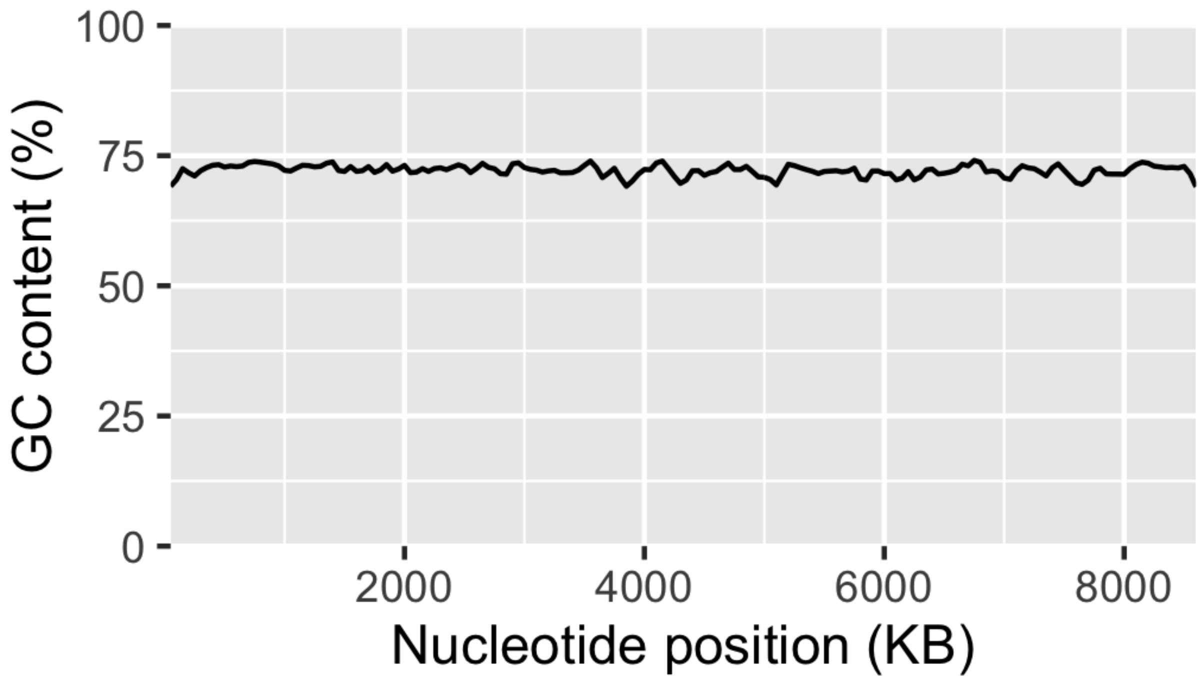


Fig. 8.14: There is little variation in the local GC content of *Streptomyces coelicolor* A3(2). Using a window size of 100 KB and a step size of 50 KB the local GC content has a mean of 72.1% and a standard deviation of 1.1%.

8.15 Other useful tools for scripting the generation of figures

There are many different ways of visualising data and generating figures. A broad distinction can be made between *ad-hoc* methods, usually using graphical user interfaces and button clicking, and methods that can be automated, i.e. methods that can reproduce the figure without human intervention.

One take home message from this chapter is that you should automate the generation of your figures. This will save you time when you realise that you need to alter the style of all the figures when submitting a manuscript for publication. It will also make your research more reproducible.

Apart from R and ggplot2 there are several tools available for automating the generation of your figures. In Python there is the matplotlib[42] package, which is very powerful and it is a great tool for plotting data generated from Python scripts. Gnuplot[43] is a scripting language designed to plot data and mathematical functions, it is particularly good at depicting three-dimensional data.

If you are dealing with graphs, as in evolutionary trees and metabolic pathways, it is worth having a look at Graphviz.

A general purpose tool for manipulating images on the command line is ImageMagick. It can be used to resize, crop and transform your images. It is a great tool for automating your image manipulation.

If you are wanting to visualise data in web pages it is worth investing some time looking at the various JavaScript libraries available for data visualisation. One poplar option is D3.js[44].

8.16 Key concepts

- R and ggplot2 are powerful tools for visualising data
- R should also be your first port of call for statistical computing (statistical computing is not covered in this book)
- In ggplot2 the visual representation of your data is built up using layers
- Separating out different aspects of plotting into different layers makes it easier to reason about the data and the best way to represent it visually
- Faceting allows you to separate out data on one or more variables, this is one of the strengths of ggplot2
- Scripting your data visualisation makes it easier to modify later on
- Scripting your data visualisation makes it reproducible
- When creating a figure ask yourself: who is the audience and what is the message

[42] http://matplotlib.org/
[43] http://www.gnuplot.info/
[44] https://d3js.org/

CHAPTER 9

Collaborating on projects

In this chapter you will learn how to work collaboratively on projects using Git.

Collaborating using Git has several benefits. Changes to the documents are kept under version control giving a transparent audit trail. Merging contributions from collaborators is simple, and automatic for all non-conflicting changes. Furthermore, as Git is a distributed version control system the project becomes stored on many different computers, on yours and your collaborators' computers as well as on remote servers, effectively backing up your code and text files.

Warning: Although you can store anything in Git it is bad practise to track large files such as genomes and microscopy files in it.
It is bad practise in that large files makes it difficult to move the repository between different computers. Furthermore, once a file has been committed to Git it is almost impossible to remove it to free up the space.
Also, if you are using a free service to host your code it is impolite to use it as a backup system for your raw data.
It is therefore good practise to add large files in your project to the `.gitignore` file.

9.1 Find a friend

The first step to collaboration is to have someone to collaborate with. If you have not already shared your experiences of this book with anyone I suggest that you engage the person sitting next to you in the office as soon as possible. Things are more fun when you work with others!

9.2 Get a GitHub or BitBucket account

Although you can host your own Git server I would strongly recommend making use of either GitHub[45] or BitBucket[46] as your remote server for backing up your Git repositories.

Registration is free for both GitHub and BitBucket. GitHub gives you unlimited number of public repositories and collaborators for free. However, you have to pay for private repositories. BitBucket allows you to have private repositories for free, but limits the number of collaborators that you can have on these. Personally, I started with BitBucket because I was ashamed of my code and did not want others to see it. Then I realised that code did not have to be perfect to be useful and then I started using public GitHub repositories more

[45] https://github.com/
[46] https://bitbucket.org/

and more. It is up to you. However the rest of this chapter will make use of GitHub as a remote server, as it is my current favorite.

9.3 Pushing the project to a remote server

Now that you have a GitHub/BitBucket account it is time to push the `S.coelicolor-local-GC-content` project from the Data analysis (page 43) and Data visualisation (page 61) chapters to it.

Go to the web interface of your account and find the button that says something along the lines of "New repostiory" or "Create repository". Give the project the name `S.coelicolor-local-GC-content` and make sure that the backend uses Git (if you are using BitBucket there is an option to use an alternative source control management system named Mercurial) and that it is initialised as an empty repository (i.e. don't tick any checkboxes to add readme, licence or `.gitignore` files).

Now on your local computer (the one in front of you where you have been doing all the exercises from this book so far) go into the directory containing the `S.coelicolor-local-GC-content` repository created in Data analysis (page 43) and built on in Data visualisation (page 61).

```
$ cd S.coelicolor-local-GC-content
```

We now need to specify the remote repository. My user name on GitHub is `tjelvar-olsson` and the repository I created on GitHub was named `S.coelicolor-local-GC-content`. To add a remote to my local repository (the one on my computer) I use the command below.

Note: In the command below `tjelvar-olsson` is my GitHub user name.

```
$ git remote add origin \
https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content.git
```

The command adds the GitHub *URL* as a remote named `origin` to the local repository. However, we have not yet *pushed* any data to this remote. Let's do this now.

```
$ git push -u origin master
```

The `git push` command pushes the changes made to the local repository to the remote repository (named `origin`). In this case the last argument (`master`) specifies the specific branch to push to.

The `-u` option is the short hand for `--set-upstream` and is used to set up the association between your local branch (`master`) and the branch on the remote. You only need to specify the `-u` option the first time you push a local repository to a remote.

If you are using GitHub or BitBucket to host your remotes you do not need to remember these commands as the web interface will display the commands you need to push an existing project to a newly created repository.

What is a branch?

When one initialises a Git repository a default branch named `master` is created. In this book all work has been done on the `master` branch.
However, it is possible to create new branches from any committed snapshot in the history. For example before submitting a paper to Nature one might create a branch named `nature`. During the lengthy review process one could then continue working on the `master` branch. After six months when the reviewers come back with their comments one could then switch back to the `nature` branch and implement all the suggested changes and send it back to the editor. At that point the `nature` branch will have diverged from the `master` branch. The editor of Nature then comes back stating that in spite of all the changes the manuscript will still be rejected due to the lack of a "wow" factor. At this point one may want to submit to Science. However, one wants to incorporate all the changes made on the `master` and the `nature` branch. That is not a problem as Git has a powerful system for *merging* changes. In this case one could merge the `nature` branch back onto `master`. At that point one could take a new branch named `science` from the master branch before submitting the manuscript to science, and so forth.
Although branching is powerful, it is beyond the scope of this book. If you are interested in learning more about it I would recommend the free Learn Git course on codecademy.

https://www.codecademy.com/learn/learn-git

9.4 Collaboration using Git

Now it is time for your collaborator to get access to the repository. Use the web interface and your friend's GitHub/BitBucket user name to give them write access to the repository.

Now your friend should be able to *clone* the repository. Alternatively, if all your friends are busy or uninterested you can clone the repository on a different computer or in a different directory to simulate the collaboration process by working on the same project in two different locations.

```
$ git clone git@github.com:tjelvar-olsson/S.coelicolor-local-GC-content.git
Cloning into 'S.coelicolor-local-GC-content'...
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 8 (delta 0), reused 8 (delta 0), pack-reused 0
Receiving objects: 100% (8/8), done.
Checking connectivity... done.
```

The command above clones my `S.coelicolor-local-GC-content.git` repository. You will need to replace `tjelvar-olsson` with your user name. Alternatively, have a look in the web interface for a string that can be used to clone the repository.

Now that your friend has cloned your repository it is time for him/her to add something to it. Create the file `README.md` and add the markdown text below to it.

```
# Local GC content variation in *S. coelicolor*

Project investigating the local GC content of the
*Streptomyces coelicolor* A3(2) genome.
```

Now let your friend add the `README.md` file to the repository and commit a snapshot of the repository.

```
$ git add README.md
$ git commit -m "Added readme file"
[master a531ea4] Added readme file
 1 file changed, 4 insertions(+)
 create mode 100644 README.md
```

Finally, your friend can push the changes to the remote repository.

```
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 384 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content.git
   cba7277..a531ea4  master -> master
```

Have a look at the repository in the using the GitHub/BitBucket web interface. You should be able to see the changes reflected there.

Now look at your local repository. You will not see your friends changes reflected there yet, because you have not yet *pulled* from the remote. It is time for you to do that now. Run the `git pull` command in your local repository.

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:tjelvar-olsson/S.coelicolor-local-GC-content
   cba7277..a531ea4  master     -> origin/master
Updating cba7277..a531ea4
Fast-forward
 README.md | 4 ++++
 1 file changed, 4 insertions(+)
 create mode 100644 README.md
```

You have now successfully pulled in the contributions from your friend into your local repository. Very cool indeed!

Let's go over what happened again.

1. You created a new repository on GitHub
2. You added this repository as a remote to your local repository
3. You pushed the content of your local repository to the remote on GitHub
4. Your friend cloned your GitHub repository, creating a local repository on their machine
5. Your friend committed changes to their local repository
6. Your friend pushed the changes from their local repository to the remote on GitHub
7. You pulled in your friend's changes from the remote on GitHub to your local repository

9.5 Working in parallel

The workflow described above was linear. You did some work, you friend cloned your work, your friend did some work, you pulled your friends work. What if you both worked on the project in parallel in your local repositories, how would that work?

Let's try it out. In your local repository add some information on how to download the genome to the `README.md` file.

~~~
# Local GC content variation in *S. coelicolor*

Project investigating the local GC content of the
*Streptomyces coelicolor* A3(2) genome.

Download the genome using ``curl``.

```
$ curl --location --output Sco.dna http://bit.ly/1Q8eKWT
```
~~~

Now commit these changes.

```
$ git add README.md
$ git commit -m "Added info on how to download genome"
[master 442c433] Added info on how to download genome
 1 file changed, 6 insertions(+)
```

Now your friend tries to work out what the `gc_content.py` and the `local_gc_content_figure.R` scripts do. The names are not particularly descriptive so he/she looks at the code and works out that the `gc_content.py` script produces a local GC content CSV file from the genome and that the `local_gc_content_figure.R` script produces a local GC plot from the CSV file. Your friend therefore decides to rename these scripts to `dna2csv.py` and `csv2png.R`.

```
$ git mv gc_content.py dna2csv.py
$ git mv local_gc_content_figure.R csv2png.R
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    local_gc_content_figure.R -> csv2png.R
        renamed:    gc_content.py -> dna2csv.py

$ git commit -m "Improved script file names"
[master 1f868ad] Improved script file names
 2 files changed, 0 insertions(+), 0 deletions(-)
 rename local_gc_content_figure.R => csv2png.R (100%)
 rename gc_content.py => dna2csv.py (100%)
```

At this point your friend pushes their changes to the GitHub remote.

```
$ git push
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 351 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To git@github.com:tjelvar-olsson/S.coelicolor-local-GC-content.git
   a531ea4..1f868ad  master -> master
```

Now you realise that you have not pushed the changes that you made to the `README.md` file to the GitHub remote. You therefore try to do so.

```
$ git push
To https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/tjelvar-olsson/...'
```

```
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Rejected? What is going on? Well, it turns out that the "hints" give us some useful information. They inform us that the update was rejected because the remote contained work that we did not have in our local repository. It also suggests that we can overcome this by using `git pull`, which would pull in your friends updates from the GitHub remote and merge them with your local updates.

```
$ git pull
```

Now unless you have defined an alternative editor as your default (using the `$EDITOR` environment variable) you will be dumped into a `vim` session with the text below in the editor.

```
Merge branch 'master' of https://github.com/tjelvar-olsson/...

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

To accept these changes you need to save the file and exit the editor (in Vim `Esc` followed by `:wq`). Once this is done you will be dumped back into your shell.

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 2 (delta 0), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content
   a531ea4..1f868ad  master     -> origin/master
Merge made by the 'recursive' strategy.
 local_gc_content_figure.R => csv2png.R | 0
 gc_content.py => dna2csv.py            | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 rename local_gc_content_figure.R => csv2png.R (100%)
 rename gc_content.py => dna2csv.py (100%)
```

Note that Git managed to work out how to merge these changes together automatically. Let's have a look at the history.

```
$ git log --oneline
a5779d6 Merge branch 'master' of https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content
1f868ad Improved script file names
442c433 Added info on how to download genome
a531ea4 Added readme file
cba7277 Added R script for generating local GC plot
6d8e0cf Initial file import
```

Now we should now be able to push to the GitHub remote.

```
$ git push
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 765 bytes | 0 bytes/s, done.
```

```
Total 5 (delta 2), reused 0 (delta 0)
To https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content.git
   1f868ad..a5779d6  master -> master
```

Collaboratively working on projects in parallel, really cool stuff!

Let's go over what happened.

1. You committed some changes to your local repository
2. Your friend committed some changes to their local repository
3. Your friend pushed their changes to the GitHub remote
4. You tried, but failed, to push your changes to the GitHub remote
5. You pulled in your friend's changes from the GitHub remote
6. Git automatically worked out how to merge these changes with yours
7. You pushed the merged changes to the remote

9.6 Resolving conflicts

So far so good, but what happens if both you and your friend edit the same part of the same file in your local repositories? How does Git deal with this? Let's try it out.

First of all your friend pulls in your changes from the GitHub remote. By pulling your friend ensures that he/she is working on the latest version of the code.

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 2), reused 5 (delta 2), pack-reused 0
Unpacking objects: 100% (5/5), done.
From github.com:tjelvar-olsson/S.coelicolor-local-GC-content
   1f868ad..a5779d6  master     -> origin/master
Updating 1f868ad..a5779d6
Fast-forward
 README.md | 6 ++++++
 1 file changed, 6 insertions(+)
```

Your friend then edits the `README.md` file.

```
# Local GC content variation in *S. coelicolor*

Project investigating the local GC content of the
*Streptomyces coelicolor* A3(2) genome.

Download the genome using ``curl``.

```
$ curl --location --output Sco.dna http://bit.ly/1Q8eKWT
```

Generate local GC content csv file from the genome.

```
$ python dna2csv.py
```
```

Your friend then commits and pushes these changes.

```
$ git add README.md
$ git commit -m "Added info on how to generate local GC content CSV"
[master 9f41c21] Added info on how to generate local GC content CSV
 1 file changed, 6 insertions(+)
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 399 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:tjelvar-olsson/S.coelicolor-local-GC-content.git
   a5779d6..9f41c21  master -> master
```

Now you work on your local repository. You too are concerned with giving more detail about how to make use of the scripts so you edit your local copy of the `README.md` file.

~~~
# Local GC content variation in *S. coelicolor*

Project investigating the local GC content of the
*Streptomyces coelicolor* A3(2) genome.

Download the genome using ``curl``.

```
$ curl --location --output Sco.dna http://bit.ly/1Q8eKWT
```

Data processing.

```
$ python dna2csv.py
$ Rscript csv2png.R
```
~~~

And you commit the changes to your local repository.

```
$ git add README.md
[-- olssont@ exit=0 S.coelicolor-local-GC-content --]
$ git commit -m "Added more info on how to process data"
[master 2559b5d] Added more info on how to process data
 1 file changed, 7 insertions(+)
```

However, when you try to push you realise that your friend has pushed changes to the remote.

```
$ git push
Username for 'https://github.com': tjelvar-olsson
Password for 'https://tjelvar-olsson@github.com':
To https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/tjelvar-olsson/...'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

So you pull.

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content
   a5779d6..9f41c21  master     -> origin/master
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

The automatic merge failed! The horror!

Let's find out what the status of the repository is.

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Okay, so both you and your friend have been editing the `README.md` file. Let's have a look at it.

~~~
# Local GC content variation in *S. coelicolor*

Project investigating the local GC content of the
*Streptomyces coelicolor* A3(2) genome.

Download the genome using ``curl``.

```
$ curl --location --output Sco.dna http://bit.ly/1Q8eKWT
```

<<<<<<< HEAD
Data processing.

```
$ python dna2csv.py
$ Rscript csv2png.R
=======
Generate local GC content csv file from the genome.

```
$ python dna2csv.py
>>>>>>> 9f41c215cce3500e80c747426d1897f93389200c
```
~~~

So Git has created a "merged" file for us and highlighted the section that is conflicting. In the above the first highlighted region contains the changes from the `HEAD` in your local repository and the second highlighted region shows the changes from your friend's commit `9f41c21`.

Now you need to edit the file so that you are happy with it. Your friend's idea of documenting what the command does is a good one so you could edit the `README.md` file to look like the below.

~~~
# Local GC content variation in *S. coelicolor*

Project investigating the local GC content of the
*Streptomyces coelicolor* A3(2) genome.

Download the genome using ``curl``.

```
$ curl --location --output Sco.dna http://bit.ly/1Q8eKWT
```

Generate ``local_gc_content.csv`` file from ``Sco.dna`` file.

```
$ python dna2csv.py
```

Generate ``local_gc_content.png`` file from ``local_gc_content.csv`` file.

```
$ Rscript csv2png.R
```
~~~

Now you need to add and commit these changes.

```
$ git add README.md
[-- olssont@ exit=0 S.coelicolor-local-GC-content --]
$ git commit -m "Manual merge of readme file"
[master 857b470] Manual merge of readme file
```

Now that you have merged the changes you can push to the remote.

```
$ git push
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 708 bytes | 0 bytes/s, done.
Total 6 (delta 4), reused 0 (delta 0)
To https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content.git
   9f41c21..857b470  master -> master
```

Let's go over what just happened.

1. Your friend pulled in your changes from the GitHub remote
2. Your friend edited the `README.md` file
3. Your friend committed and pushed their changes to the remote
4. You edited the `README.md` file in your local repository
5. You committed the changes to your local repository
6. You tried but failed to push to the GitHub remote repository
7. You pulled the changes from the GitHub remote repository; but the automatic merging failed
8. You looked at the status of your local repository to find out what state it was in

9. You resolved the conflicts manually by editing the `README.md` file
10. You added the updated `README.md` file to the staging area and committed it
11. You pushed your manual merge to the GitHub remote repository

That's it, you now have all the tools you need to start collaborating with your colleagues using Git! May the force be with you.

9.7 Key concepts

- Git is a powerful tool for collaborating on projects
- Every person working on a Git project have all the files on their local computer
- By adding a remote to a local repository one can push updates to another repository (the remote)
- It is also possible to pull changes from a remote
- One way to collaborate using Git is to have multiple people pulling from and pushing to the same remote repository
- It is only possible to push to a remote if you have all the updates that are on the remote in your local repository
- When pulling Git will do its best to resolve any conflicts between your local updates and updates from the remote
- If there are conflicts that Git cannot resolve you have to fix them manually
- By pushing your updates to a remote server such as GitHub you are effectively backing up your project

CHAPTER 10

Creating scientific documents

Communication forms an important part of the scientific process. In Collaborating on projects (page 83) we added a `README.md` file describing the purpose of the project and how to perform the analysis. This is a good thing. However it may not be clear how this process can be extended to produce scientific documents with figures, equations and reference management. In this chapter we will learn how to do this.

10.1 Starting the manuscript

Here we will build on the work from Data analysis (page 43), Data visualisation (page 61) and Collaborating on projects (page 83) chapters. Let's start by going into the `S.coelicolor-local-GC-content` directory.

```
$ cd S.coelicolor-local-GC-content
```

Now use your favorite text editor to add the text below to a file named `manuscipt.md`.

```
## Introduction

*Streptomyces coelicolor* is a bacteria that can produce a range of natural
products of pharmaceutical relevance.

In 2002 the genome of *S. coelicolor*  A3(2), the model actinomycete
organism, was sequenced.  In this study we investigate the local GC content
of this organism.


## Methodology

The genome of *S. coelicolor*  A3(2) was downloaded from the Sanger Centre
ftp site.

The local GC content was then calculated using a sliding window of 100 KB
and a 50 KB step size.


## Results

The mean of the local GC content is 72.1% with a standard deviation of 1.1.


## Conclusion

There is little variation in the local GC content of *S. coelicolor* A3(2).
```

The manuscript above won't make it into Nature, but it will do for us to illustrate how to build up a manuscript using plain text files.

In terms of the markdown syntax the two hashes `##` are used to mark level 2 headings and the words surrounded by asterisk symbols (`*`) will be emphasized using italic font.

10.2 Converting the document to HTML using pandoc

Markdown is a lightweight syntax that can easily be converted to HTML. Here we will use of the tool Pandoc[48] to convert the markdown document to HTML.

First of all you will need to install Pandoc. Generic information on how to install software can be found in Managing your system (page 137). On a Mac you can use Homebrew and on Linux based systems it should be available for install using the distributions package manager. For more detail have a look at the Pandoc installation notes[49].

Now that we have installed Pandoc, let's use it to convert our `manuscript.md` file to a standalone HTML file.

```
$ pandoc -f markdown -t html -s manuscript.md > manuscript.html
```

In the above the `-f markdown` option means *from markdown* and the `-t html` option means *to html*. The `-s` option means *standalone*, i.e. encapsulate the content with appropriate appropriate headers and footers.

Pandoc writes to the standard output stream so we redirect it (>) to a file named `manuscript.html`. Have a look at the `manuscript.html` file using a web browser.

Alternatively, we could have used the `-o` option to specify the name of an output file. The command below produces the same outcome as the previous command.

```
$ pandoc -f markdown -t html -s manuscript.md -o manuscript.html
```

Now use a web browser to view the generated `manuscript.html` file.

10.3 Adding a figure

At this point it would be good to add the figure produced in Data visualisation (page 61) to the "Results" section of the manuscript.

In markdown images can be added using the syntax below.

```
![Alternative text](path/to/image.png)
```

In HTML the intention of the alternative text (the "alt" attribute) is to provide a descriptive text in case the image cannot be displayed for some reason. Pandoc makes use of the alternative text attribute to create a caption for the image.

```
## Results

The mean of the local GC content is 72.1% with a standard deviation of 1.1.

![**Variation in the local GC content of *S. coelicolor* A3(2).** Using a
window size of 100 KB and a step size of 50 KB the local GC content has a
mean of 72.1% and a standard deviation of 1.1.](local_gc_content.png)
```

[48] http://pandoc.org/
[49] http://pandoc.org/installing.html

In the above the double asterix (`**`) is used as markup for bold text. This will serve as a title for the figure caption.

Now we can build the document again.

```
$ pandoc -f markdown -t html -s manuscript.md -o manuscript.html
```

10.4 Converting the document to PDF

HTML is great for websites. However, scientific documents tend to be read as PDF. Let us use Pandoc to convert our document to PDF.

However, before we can do this we need to install LaTeX[50]. On Mac install MacTeX[51]. On Linux use you package manager to install LaTeX, possibly known as "TeX Live". See the section on Obtaining LaTeX[52] on the LaTeX project website for more information.

Now that you have installed LaTeX you can convert the `manuscript.md` markdown file to PDF using the command below.

```
$ pandoc -f markdown -t latex -s manuscript.md -o manuscript.pdf
```

In the above we use the `-t latex` option to specify that the `manuscript.pdf` output file should be built using LaTeX.

10.5 Reference management

Reference management is a particularly prominent feature of scientific writing. Let us therefore look at how we can include references to websites and papers in our document.

Let's start by creating a bibliography file. Copy and paste the content below into a file named `references.bib`.

```
@online{S.coelicolor-genome,
title={{S. coelicolor genome}},
url={ftp://ftp.sanger.ac.uk/pub/project/pathogens/S_coelicolor/whole_genome/},
urldate={2016-07-10}
}
```

This is a so called BibTex record. In this particular case it is a BibTex record for an online resource, as indicated by the `@online` type. You would also use the `@online` type to reference web pages.

The text `S.coelicolor-genome` is the "key" assigned to this record. The key could have been called anything as long as it is unique. This key will be used within our document when citing this record.

Now append the text below to the bottom of the `references.bib` file.

```
@article{Bentley2002,
title={Complete genome sequence of the model actinomycete
       Streptomyces coelicolor A3 (2)},
author={Bentley, Stephen D and Chater, Keith F and Cerdeno-Tarraga, A-M and
        Challis, Greg L and Thomson, NR and James, Keith D and
        Harris, David E and Quail, Michael A and Kieser, H and
        Harper, David and others},
```

[50] https://www.latex-project.org
[51] http://www.tug.org/mactex
[52] https://latex-project.org/ftp.html

```
journal={Nature},
volume={417},
number={6885},
pages={141--147},
year={2002},
publisher={Nature Publishing Group}
}
```

Do not type in BibTex records by hand. The entire `Bentley2002` record was copied and pasted from Google Scholar[53]. Note that in the record above the identifier was changed from `bentley2002complete` (key used by Google Scholar) to `Bentley2002`.

References managers such as Mendeley[54] and Zotero[55] can also be used to export BibTex records. More suggestions on how to access BitTex records can be found on the Tex StackExchange[56] site.

Now let's add some references to our `manuscript.md` file.

```
## Introduction

*Streptomyces coelicolor* is a bacteria that can produce a range of natural
products of pharmaceutical relevance.

In 2002 the genome of *S. coelicolor*  A3(2), the model actinomycete
organism, was sequenced [@Bentley2002].

In this study we investigate the local GC content of this organism.


## Methodology

The genome of *S. coelicolor*  A3(2) was downloaded from the Sanger Centre
ftp site [@S.coelicolor-genome].
```

Now we can add referenes using Pandoc's built in `pandoc-citeproc` filter.

```
$ pandoc -f markdown -t latex -s manuscript.md -o manuscript.pdf   \
  --filter pandoc-citeproc --bibliography=references.bib
```

The `--filter pandoc-citeproc` argument results in automatically adding citations and a bibliography to the document. However, this requires some knowledge of where the bibliographic information is, this is specified using the `--bibliography=references.bib` argument.

"CiteProc" is in fact a generic name for a program that can be used to produce citations and bibliographies based on formatting rules using the Citation Style Langauge (CSL) syntax. Zotero provides CSL styles for lots of journals in the Zotero Style Repository[57].

Let's download Zotero's CSL file for Nature, copy and paste this text into a file named `nature.csl`.

```
$ curl https://www.zotero.org/styles/nature > nature.csl
```

We can now produce our document using Nature's citation style.

```
$ pandoc -f markdown -t latex -s manuscript.md -o manuscript.pdf   \
  --filter pandoc-citeproc --bibliography=references.bib  \
  --csl=nature.csl
```

[53] https://scholar.google.co.uk/
[54] https://www.mendeley.com
[55] https://www.zotero.org/
[56] http://tex.stackexchange.com/a/207
[57] https://www.zotero.org/styles

Have a look at the generated PDF file. Pretty neat right?! One thing that is missing is a title for the reference section. Let's add that to the `manuscript.md` file.

```
## Conclusion

There is little variation in the local GC content of *S. coelicolor* A3(2).


## References
```

10.6 Adding meta data

To turn this into a research article we need to add a title, authors, an abstract and a date. In Pandoc this can be achieved by adding meta data to the top of the file, using a YAML syntax (see *Useful plain text file formats* (page 25) for information on YAML).

Add the header below to the top of the `manuscript.md` file.

```
---
title: "*S. coelicolor* local GC content analysis"
author: Tjelvar S. G. Olsson and My Friend
abstract: |
  In 2002 the genome of *S. coelicolor*  A3(2), the model actinomycete
  organism, was sequenced.

  The local GC content was calculated using a sliding window of
  100 KB and a 50 KB step size.

  The mean of the local GC content was found to be 72.1% with a standard
  deviation of 1.1. We therefore conclude that there is little variation
  in the local GC content of *S. coelicolor* A3(2).
date: 25 July 2016
---
## Introduction
```

Let's give some explanation of the meta data above. The YAML meta data is encapsulated using `---`. The title string is quoted to avoid the `*` symbols confusing Pandoc. The pipe symbol at the beginning of the abstract allows for multi-line input with newlines, note that the multi-lines must be indented.

Let's generate the document again.

```
$ pandoc -f markdown -t latex -s manuscript.md -o manuscript.pdf   \
  --filter pandoc-citeproc --bibliography=references.bib  \
  --csl=nature.csl
```

The `manuscript.pdf` document is now looking pretty good!

Anther useful feature of Pandoc's meta data section is that we can add information for some of the data that we previously had to specify on the command line. Let's add items for the `--bibliograpy` and `--csl` options (these options are in fact short hand for `--metadata bibliograpy=FILE` and `--metadata csl=FILE`).

```
date: 25 July 2016
bibliography: references.bib
csl: nature.csl
---
## Introduction
```

Now we can generate the documentation using the command below.

```
$ pandoc -f markdown -t latex -s manuscript.md -o manuscript.pdf   \
  --filter pandoc-citeproc
```

This is a good point to commit a snapshot to version control. Let's look at the status of our repository first.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        bioinformatics.csl
        manuscript.html
        manuscript.md
        manuscript.pdf
        nature.csl
        references.bib

nothing added to commit but untracked files present (use "git add" to track)
```

We have created many new files. We want to track all of them except `manuscript.pdf` and `manuscript.html` as they can be generated by Pandoc. Let us therefore update the `.gitignore` file to look like the below.

```
manuscript.*
!manuscript.md

Sco.dna
local_gc_content.csv
local_gc_content.png
```

In the above the first two lines are new. Let's explain what they do. The first line states that all files starting with `manuscript.` should be ignored. This includes the file we want to track `manuscript.md`. On the second line we therefore add an exception for this file, the exclamation mark (`!`) is used to indicate that the `manuscript.md` should be excluded from the previous rule to ignore it.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .gitignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        bioinformatics.csl
        manuscript.md
        nature.csl
        references.bib

no changes added to commit (use "git add" and/or "git commit -a")
```

Now we can add the remaining files and commit the snapshot.

```
$ git add bioinformatics.csl manuscript.md nature.csl references.bib
$ git commit -m "Added draft manuscript"
[master 7b06d9d] Added draft manuscript
```

```
 4 files changed, 332 insertions(+)
 create mode 100644 bioinformatics.csl
 create mode 100644 manuscript.md
 create mode 100644 nature.csl
 create mode 100644 references.bib
```

Finally, let us also add and commit the updated `.gitignore` file.

```
$ git add .gitignore
$ git commit -m "Updated gitignore to ignore generated manuscript files"
[master bea89f4] Updated gitignore to ignore generated manuscript files
 1 file changed, 3 insertions(+)
```

10.7 Benefits of using Pandoc and plain text files

Why go through all this trouble to produce a PDF document? Would it not be easier to simply write it using a word processor and export it as PDF?

There are three main advantages to the methodology outlined in this chapter.

1. There are lots of awesome tools for working with plain text files. If you decide to create your manuscript using plain text files you can take advantage of them. Worthy of special mention is Git which is one of the most powerful collaboration tools in the world, with the added advantage of giving you unlimited undo functionality and a transparent audit trial.
2. Automation! We will go into this more in the next chapter, Automation is your friend (page 103). However, for now imagine that someone discovered that there was something wrong with the raw data that you had been using. How long would it take you to update your manuscript? Using plain text files it is possible to create automated work flows to build entire documents from raw data.
3. Ability to convert to any file format. We have already seen how you can covert the document to HTML. What if your collaborator really needs Word? No problem.

```
     $ pandoc -f markdown -t docx -s manuscript.md -o manuscript.docx  \
       --filter pandoc-citeproc
```

Incidentally, another advantage of learning to use Pandoc is that it is not limited in going from markdown to other formats. It can take almost any file format and convert it to any other file format. For example, one could convert the Word document we just created to TeX.

```
$ pandoc -f docx -t latex manuscript.docx -o manuscript.tex
```

10.8 Alternative plain text approaches

There are other methods of creating scientific documents using plain text files.

One option is to write them using the LaTeX syntax. This is a less intuitive than using markdown. However, it gives you a bit more control and flexibility.

A good alternative for people that miss the benefits of a graphical user interface (GUI) is LyX[58]. LyX allows you to write and format text using an intuitive GUI. It is different from many other word processing tools in that it places more focus on the structure of the document and less focus on the appearance. The outputs of LyX are also plain text files, a derivative of LaTeX files. However, LyX also has built-in functionality for exporting files in a variety of file formats including PDF.

[58] https://www.lyx.org/

10.9 Using non-plain text file formats

This chapter has outlined how you can work with plain text file formats to produce scientific documents. However, many people really like using Microsoft Word. You might be one of these people. In this case I hope that this chapter has given you some food for thought. The purpose of this chapter was not to try to force you to change your habits, but to outline alternative methods.

There are some disadvantages to using Microsoft Word. The biggest one probably being collaborative editing. You tend to end up with files named along the lines of `manuscript_revision3_TO_edits.docx` being emailed around. This is not ideal. However, on the other hand if it is the tool that you and/or your collaborators are familiar with it is easy to work with.

If you and/or your collaborators do not want to use plain text files, but want to avoid emailing Word documents around, you may want to consider Google Docs[59]. It is similar to Word and LyX in that it has a GUI. Furthermore, it has built in support for collaborative working and version control.

For an extended discussion on the pros and cons of various ways of creating scientific documents I would recommend reading Good Enough Practices in Scientific Computing[60]. One of the key take home messages from this paper is for everyone involved to agree on the tools before starting the process of writing.

10.10 Key concepts

- Pandoc is a powerful tool that can be used to convert (almost) any document format to any other
- Markdown can be used to add document structure to plain text files
- The `pandoc-citeproc` filter can be used to add citations and a bibliography
- The Citation Style Language (CSL) can be used to format the citations and the bibliography
- You can access BibTex records from Google Scholar
- Mendeley and Zotero are a free reference managers that can be used to export BibTex records (try to avoid having to type out BibTex records by hand)
- Zotero provides CSL citation style files for lots and lots of journals
- It is possible to add YAML meta data to markdown files specifying attributes such as title, author and abstract
- Using plain text files for scientific documents allows you to make use of awesome tools such as Git
- Speak to your collaborators and agree on the tools that best suit your needs

[59] https://docs.google.com
[60] https://arxiv.org/abs/1609.00037

CHAPTER 11

Automation is your friend

Have you ever felt your heart sink upon reading the reviewer comments along the lines of: *"very nice analysis, but it would be useful if it was performed on species x, y and z as well"*? Or perhaps you have spent weeks analysing data and generating figures only to find out that a newer and more complete data set has become available?

In these circumstances would it not be nice if all you had to do was point your project at a new URL from where the latest version of the data could be downloaded and all the analyses, figures and the manuscript were magically updated for you?

This is possible! In this chapter we will make use of a tool called `make` to automate all the steps from Data analysis (page 43), Data visualisation (page 61) and Creating scientific documents (page 95).

11.1 Introduction to `make`

The `make` tool is commonly used to build software. As such `make` is a tool for managing dependencies in a build system, where dependencies are organised into a tree (as in a graph). People in the software industry often talk about dependency graphs.

The `make` program makes use of a file named `Makefile` in the working directory. A "Makefile" contains rules. Each rule consists of a target (the stuff to be created), prerequisites (the stuff needed to build the target) and a recipe (the instructions for how to create the target from the prerequisites).

Creating rules in a file named `Makefile` is therefore a means to create an automated build process.

11.2 Creating a `Makefile`

First of all ensure that you are in the `S.coelicolor-local-GC-content` directory created in Data analysis (page 43).

```
$ cd S.coelicolor-local-GC-content
```

One of the first steps in Data analysis (page 43) was to download the *S. coelicolor* genome. Let's start by creating a rule for this step. Using your favorite text editor create a file named `Makefile` and add the content below to it.

```
Sco.dna:
	curl --location --output Sco.dna http://bit.ly/1Q8eKWT
```

Warning: The make program expects the lines to be indented using the Tab character so make sure that the curl command is preceded by a Tab character and not a bunch of white spaces.

In this first rule Sco.dna is the target. The target has no prerequisites and the recipe has one instruction to download the file using curl.

Now we can run the make command.

```
$ make
make: `Sco.dna' is up to date.
```

Let's add another rule for cleaning up the working directory.

```
Sco.dna:
	curl --location --output Sco.dna http://bit.ly/1Q8eKWT

clean:
	rm Sco.dna
```

When invoking make it is possible to specify which rule to run. Let's clean up.

```
$ make clean
rm Sco.dna
```

If we run make now it will notice that the Sco.dna file does not exist and will download it.

```
$ make
curl --location --output Sco.dna http://bit.ly/1Q8eKWT
```

Now we need a rule to create the local_gc_content.csv (the target) from the Sco.dna file (the prerequisite). Add the lines below to the top of the Makefile.

```
local_gc_content.csv: Sco.dna
	python dna2csv.py
```

Now update the clean rule.

```
clean:
	rm Sco.dna
	rm local_gc_content.csv
```

Let's clean up again.

```
$ make clean
rm Sco.dna
rm local_gc_content.csv
```

Now we have removed Sco.dna and local_gc_content.csv. This is a good opportunity to show that make resolves dependencies. We can do this by calling the local_gc_content.csv rule, this will in turn call the Sco.dna rule.

```
$ make local_gc_content.csv
curl --location --output Sco.dna http://bit.ly/1Q8eKWT
python dna2csv.py
```

That's cool, make uses the information about requirements to build any missing pieces.

Let's add another rule for generating the local_gc_content.png file from the local_gc_content.csv file. Add the lines below to the top of the Makefile.

```
local_gc_content.png: local_gc_content.csv
        Rscript csv2png.R
```

Let's also remember to update the rule for cleaning up.

```
clean:
        rm Sco.dna
        rm local_gc_content.csv
        rm local_gc_content.png
```

Finally, let's add a rule for building the manuscript as a PDF file and update the `clean` rule to remove it.

```
manuscript.pdf: local_gc_content.png
        pandoc -f markdown -t latex -s manuscript.md -o manuscript.pdf   \
        --filter pandoc-citeproc

local_gc_content.png: local_gc_content.csv
        Rscript csv2png.R

local_gc_content.csv: Sco.dna
        python dna2csv.py

Sco.dna:
        curl --location --output Sco.dna http://bit.ly/1Q8eKWT

clean:
        rm Sco.dna
        rm local_gc_content.csv
        rm local_gc_content.png
        rm manuscript.pdf
```

Let's try it.

```
$ make clean
rm Sco.dna
rm local_gc_content.csv
rm local_gc_content.png
rm manuscript.pdf
```

Double check that the files have actually been removed.

```
$ ls
Makefile           csv2png.R          nature.csl
README.md          dna2csv.py         references.bib
bioinformatics.csl manuscript.md
```

Now let's build the `manuscript.pdf` file from scratch.

```
$ make
curl --location --output Sco.dna http://bit.ly/1Q8eKWT
python dna2csv.py
Rscript csv2png.R
pandoc -f markdown -t latex -s manuscript.md -o manuscript.pdf   \
        --filter pandoc-citeproc
```

Very cool!

Now suppose that for some reason we needed to use a different genome file. In this case we would only need to update the URL used in the Makefile's `Sco.dna` rule and run `make` again! Amazing!

This is a good point to commit a snapshot of the Git repository. First of all let's clean up.

```
$ make clean
rm Sco.dna
rm local_gc_content.csv
rm local_gc_content.png
rm manuscript.pdf
```

Now let's check the status of the project.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Makefile

nothing added to commit but untracked files present (use "git add" to track)
```

Great let's add and commit the `Makefile`.

```
$ git add Makefile
$ git commit -m "Added Makefile to build manuscript.pdf"
[master 5d74b6a] Added Makefile to build manuscript.pdf
 1 file changed, 18 insertions(+)
 create mode 100644 Makefile
```

Finally, we push the changes to GitHub.

```
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 498 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tjelvar-olsson/S.coelicolor-local-GC-content.git
   bea89f4..5d74b6a  master -> master
```

This was quite a short chapter which illustrates that automation does not need to be painful!

Note that as well as automating the building of the manuscript the rules in the Makefile serve as authoritative documentation that can be used to understand how the different components of the project fit together.

11.3 Key concepts

- Automation is your friend
- The `make` command builds projects using rules specified in the `Makefile`
- By specifying requirements for each target in the `Makefile` the `make` command can work out what the dependency graph is
- It is good practice to specify a `clean` rule for removing files that are built automatically
- Automation saves you time in the long run
- Good automation scripts also serve a purpose as technical documentation of a project

CHAPTER 12

Practical problem solving

So far we have been learning about computing and coding.

The purpose of this chapter is to introduce some practical techniques for tackling data analysis problems.

At this point we go back to looking at the Swiss-Prot FASTA file used in First steps towards automation (page 13). You can download this file using the `curl` command below.

```
$ curl --location --output uniprot_sprot.fasta.gz http://bit.ly/1l6SAKb
```

The downloaded `uniprot_sprot.fasta.gz` file is gzipped. You can view the content of this file without permanently unzipping it using the command below.

```
$ gunzip -c uniprot_sprot.fasta.gz
```

If the `curl` and `gunzip` commands above look unfamiliar you may want to review the First steps towards automation (page 13) chapter.

By briefly inspecting some random FASTA description lines it becomes clear that some species include variants. Below are some sample *E. coli* extracts from the FASTA file.

```
...IraP OS=Escherichia coli (strain SE11) GN=iraP PE=3 SV=1
...IraP OS=Escherichia coli (strain SMS-3-5 / SECEC) GN=iraP PE=3 SV=1
...IraP OS=Escherichia coli (strain UTI89 / UPEC) GN=iraP PE=3 SV=1
```

How many variants are there per species? Do all variants contain the same number of proteins?

To answer these questions we will make use of the Python scripting language and some practical problem solving techniques.

12.1 Stop and think

One of the most common mistakes when programming is to start coding before the problem is clearly defined.

What do we want to achieve? Pause for a second and try to answer this question to yourself. It may help trying to explain it out loud.

Rubber duck debugging

When faced with a tricky *bug* many programmers find that one of the best methods to *debug* their code is to explain it to a colleague. The act of verbalising ones assumptions often identifies the problem at hand without the colleague having to do anything except listen. This technique is so powerful that some programmers use a rubber duck, as a substitute for a colleague, when the latter is unavailable. This method is not limited to debugging, hence the suggestion to verbalise the problem statement out loud.

Now describe all the steps required to achieve this goal. Again it may help speaking out loud to yourself or imagining that you are describing the required steps to a colleague.

Did you find that easy or difficult? Thinking about what one wants to achieve and all the steps required to get there can often be quite overwhelming. When this happens it is easy to think to oneself that the detail will become clear later on and that one should stop wasting time and get on with the coding. Try to avoid doing this. Rather, try to think of a simpler goal.

For example consider the goal of working out how many different species there are in the FASTA file. This goal would require us to:

1. Identify all the description lines in the FASTA file
2. Extract the organism name from the line
3. Ensure that there are no duplicate entries
4. Count the number of entries

If you know the tools and techniques required to do all of the steps above this may not seem overwhelming. However, if you don't it will. In either case it is worthwhile to treat each step as a goal in itself and work out the sub-steps required to achieve these. By iterating over this process you will either get down to steps sizes that do not feel overwhelming or you will identify the key aspect that you feel unsure about.

For example, the first step to identify all the description lines in the FASTA file can be decomposed into the sub-steps below.

1. Create a list for storing the description lines
2. Iterate over all the lines in the input file
3. For each line check if it is a description line
4. If it is add the line to the list

Now suppose one felt unsure about how to check if a line was a description line. This would represent a good place to start experimenting with some actual code. Can we come up with a code snippet that can check if a string is a FASTA description line?

12.2 Measuring success

Up until this point we have been testing our code snippets by running the code and verifying the results manually. This is terribly inefficient and soon breaks down once you have more than one thing to test.

It is much better to write tests that can be run automatically. Let's examine this concept. Create a file named `fasta_utils.py` and add the Python code below to it.

```python
"""Module containing utility functions for working with FASTA files."""

def test_is_description_line():
    """Test the is_description_line() function."""
```

```
5      print("Testing the is_description_line() function...")
6
7  test_is_description_line()
```

The first line is a module level "docstring" and it is used to explain the intent of the module as a whole. A "module" is basically a file with a `.py` extension, i.e. a Python file. Modules are used to group related functionality together.

Lines three to five represent a function named `test_is_description_line()`. Python functions consist of two parts: the function definition (line 3) and the function body (lines 4 and 5).

In Python functions are defined using the `def` keyword. Note that the `def` keyword is followed by the name of the function. The name of the function is followed by a parenthesized set of arguments, in this case the function takes no arguments. The end of the function definition is marked using a colon.

The body of the function, lines four and five, need to be indented. The standard in Python is to use four white spaces to indent code blocks.

Warning: Whitespace really matters in Python! If your code is not correctly aligned you will see `IndentationError` messages telling you that everything is not as it should be. You will also run into `IndentationError` messages if you mix white spaces and tabs.

The first line of the function body, line four, is a docstring explaining the intent of the function. Line five makes use of the built-in `print()` function to write a string to the *standard output stream*. Python's built-in `print()` function is similar to the `echo` command we used earlier in Keeping track of your work (page 31).

Finally, on line seven the `test_is_description_line()` function is called, i.e. the logic within the function's body is executed. In this instance this means that the `"Testing the is_description_line() function..."` string is written to the standard output stream.

Let us run this script in a terminal. To run a Python script we use the `python` command followed by the name of the script.

```
$ python fasta_utils.py
Testing the is_description_line() function...
```

So far so good? At the moment our `test_is_description_line()` function does not actually test anything. Let us rectify that now.

```
1  """Module containing utility functions for working with FASTA files."""
2
3  def test_is_description_line():
4      """Test the is_description_line() function."""
5      print("Testing the is_description_line() function...")
6      assert is_description_line(">This is a description line") is True
7
8  test_is_description_line()
```

There are quite a few things going on in the newly added line. First of all it makes use of three built-in features of Python: the `assert` and `is` keywords, as well as the `True` constant. Let's work through these in reverse order.

Python has some built-in constants[61], most notably `True`, `False` and `None`. The `True` and `False` constants are the only instances of the `bool` (boolean) type and `None` is often used to represent the absence of a value.

In Python `is` is an operator that checks for object identity, i.e. if the object returned by the `is_description_line()` function and `True` are the same object. If they are the same object the comparison evaluates to `True` if not it evaluates to `False`.

[61] https://docs.python.org/2/library/constants.html

The assert keyword is used to insert debugging statements into a program. It provides a means to ensure that the state of a program is as expected. If the statement evaluates to False an AssertionError is raised.

So, what will happen if we run the code in its current form? Well, we have not yet defined the is_description_line() function, so Python will raise a NameError. Let us run the code.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 8, in <module>
    test_is_description_line()
  File "fasta_utils.py", line 6, in test_is_description_line
    assert is_description_line(">This is a description line") is True
NameError: global name 'is_description_line' is not defined
```

Great now we are getting somewhere! What? Well, we have impemented some code to test the functionality of the is_description_line() and it tells us that the function does not exist. This is useful information. Let us add a placeholder is_description_line() function to the Python module.

```
1   """Module containing utility functions for working with FASTA files."""
2
3   def is_description_line(line):
4       """Return True if the line is a FASTA description line."""
5
6   def test_is_description_line():
7       """Test the is_description_line() function."""
8       print("Testing the is_description_line() function...")
9       assert is_description_line(">This is a description line") is True
10
11  test_is_description_line()
```

Note that the function we have added on lines three and four currently does nothing. By default the function will return None. However, when we run the script we should no longer get a NameError. Let's find out what happens when we run the code.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 11, in <module>
    test_is_description_line()
  File "fasta_utils.py", line 9, in test_is_description_line
    assert is_description_line(">This is a description line") is True
AssertionError
```

More progress! Now we see the expected AssertionError, becuase None is not True. Let us add some code to try to get rid of this error message. To achieve this we simply need to make the function return True.

```
1   """Module containing utility functions for working with FASTA files."""
2
3   def is_description_line(line):
4       """Return True if the line is a FASTA description line."""
5       return True
6
7   def test_is_description_line():
8       """Test the is_description_line() function."""
9       print("Testing the is_description_line() function...")
10      assert is_description_line(">This is a description line") is True
11
12  test_is_description_line()
```

Now, we can run the code again.

```
$ python fasta_utils.py
Testing the is_description_line() function...
```

No error message, the code is now working to the specification described in the test. However, the test does not specify what the behaviour should be for a biological sequence line. Let us add another assert statement to specify this.

```python
"""Module containing utility functions for working with FASTA files."""

def is_description_line(line):
    """Return True if the line is a FASTA description line."""
    return True

def test_is_description_line():
    """Test the is_description_line() function."""
    print("Testing the is_description_line() function...")
    assert is_description_line(">This is a description line") is True
    assert is_description_line("ATCG") is False

test_is_description_line()
```

Now we can run the code again.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 13, in <module>
    test_is_description_line()
  File "fasta_utils.py", line 11, in test_is_description_line
    assert is_description_line("ATCG") is False
AssertionError
```

More progress, we now have a test to ensure that the `is_description_line()` function returns `False` when the input line is a sequence. Let us try to implement the desired functionality to make the test pass. For this we will use the startswith()[62] method, that is built into strings, to check if the string starts with a greater than (>) character.

```python
"""Module containing utility functions for working with FASTA files."""

def is_description_line(line):
    """Return True if the line is a FASTA description line."""
    if line.startswith(">"):
        return True
    else:
        return False

def test_is_description_line():
    """Test the is_description_line() function."""
    print("Testing the is_description_line() function...")
    assert is_description_line(">This is a description line") is True
    assert is_description_line("ATCG") is False

test_is_description_line()
```

In the code above we make use of conditional logic, i.e. `if` something is `True` the code does something otherwise it does something `else`. As mentioned previously whitespace is important in Python and four

[62] https://docs.python.org/2/library/stdtypes.html#str.startswith

spaces are used to indent the lines after the if and else statements to tell Python which statement(s) belong in the conditional code blocks. In this case we only have one statement per conditional, but it is possible to group several statements together based on their indentation.

Let us test the code again.

```
$ python fasta_utils.py
Testing the is_description_line() function...
```

Fantastic the code behaves in the way that we want it to behave!

However, the current implementation of the is_description_line() function is a little bit verbose. Do we really need the else conditional? What would happen if it was not there?

The beauty of tests now become more apparent. We can start experimenting with the implementation of a function and feel confident that we are not breaking it. As long as the tests do not fail that is!

Let us test out our hypothesis that the else conditional is redundant by removing it and de-denting the return False statement.

```python
"""Module containing utility functions for working with FASTA files."""

def is_description_line(line):
    """Return True if the line is a FASTA description line."""
    if line.startswith(">"):
        return True
    return False
```

Now we can simply run the tests to see what happens.

```
$ python fasta_utils.py
Testing the is_description_line() function...
```

Amazing, we just made a change to our code and we can feel pretty sure that it is still working as intended. This is very powerful. Incidentally, the code works by virtue of the fact that if the return True statement is reached the function returns True and exits, as such the second return statement is not called.

The methodology used in this section is known as Test-Driven Development, often referred to as TDD. It involves three steps:

1. Write a test
2. Write minimal code to make the test pass
3. Refactor the code if necessary

In this instance we started off by writing a test checking that the is_description_line() function returned True when the input was a description line. We then added *minimal* code to make the test pass, i.e. we simply made the function return True. At this point no refactoring was needed so we added another test to check that the function returned False when the input was a sequence line. We then added some naive code to make the tests pass. At this point, we believed that there was scope to improve the implementation of the function, so we refactored it to remove the redundant else statement.

Well done! That was a lot of information. Go make yourself a cup of tea.

12.3 More string processing

Because information about DNA and proteins are often stored in plain text files many aspects of biological data processing involves manipulating text. In computing text is often referred to as strings of characters.

String manipulation is is therefore a common task both for processing biological sequences and for interpreting sequence identifiers.

This section will provide a brief summary of how Python can be used for such string processing.

Start Python in its interactive mode by typing `python` into the terminal.

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

12.3.1 The Python string object

When parsing in strings from a text file one often has to deal with lines that have leading and/or trailing white spaces. Commonly one wants to get rid of them. This can be achieved using the `strip()` method built into the string object.

```
>>> "  text with leading/trailing spaces ".strip()
'text with leading/trailing spaces'
```

Another common use case is to replace a word in a line. For example, when we strip out the leading and trailing white spaces one might want to update the word "with" to "without" to make the resulting string reflect its current state. This can be achieved using the `replace()` method.

```
>>> "  text with leading/trailing spaces ".strip().replace("with", "without")
'text without leading/trailing spaces'
```

Note: In the example above we chain the `strip()` and `replace()` methods together. In practise this means that the `replace()` methods acts on the return value of the `strip()` method.

What is the difference between a function and a method?

Often the two terms are used interchangeably. However, a method refers to a function that is part of a class and the term function refers to a function which stands on its own.

Earlier we saw how the `startswith()` method can be used to identify FASTA description lines.

```
>>> ">MySeq1|description line".startswith(">")
True
```

The `endswith()` method complements the `startswith()` method and is often used to examine file extensions.

```
>>> "/home/olsson/images/profile.png".endswith("png")
True
```

This example above only works if the file extension is in lower case.

```
>>> "/home/olsson/images/profile.PNG".endswith("png")
False
```

However, we can overcome this issue by adding a call to the `lower()` method, which converts the string to lower case.

```
>>> "/home/olsson/images/profile.PNG".lower().endswith("png")
True
```

Another common use case is to search for a particular string within another string. For example one might want to find out if the UniProt identifier "Q6GZX4" is present in a FASTA description line. To achieve this one can use the `find()` method, which returns the index position (zero-based) where the search term was first identified.

```
>>> ">sp|Q6GZX4|001R_FRG3G".find("Q6GZX4")
4
```

If the search term is not identified `find()` returns -1.

```
>>> ">sp|P31946|1433B_HUMAN".find("Q6GZX4")
-1
```

When iterating over lines in a file one often wants to split the line based on a delimiter. This can be achieved using the `split()` method. By default this splits on white space characters and returns a list of strings.

```
>>> "text without leading/trailing spaces".split()
['text', 'without', 'leading/trailing', 'spaces']
```

A different delimiter can be used by providing it as an argument to the `split()` method.

```
>>> ">sp|Q6GZX4|001R_FRG3G".split("|")
['>sp', 'Q6GZX4', '001R_FRG3G']
```

There are many variations on the string operators described above. It is useful to familiarise yourself with the Python documentation on strings[63].

12.3.2 Regular expressions

Regular expressions can be defined as a series of characters that define a search pattern.

Regular expressions can be very powerful. However, they can be difficult to build up. Often it is a process of trial and error. This means that once they have been created, and the trial and error process has been forgotten, it can be extremely difficult to understand what the regular expression does and why it is constructed the way it is.

Warning: Use regular expression as a last resort. A good rule of thumb is to always try to use string operations to implement the desired functionality and only switch to regular expressions when the code implemented using these operations becomes more difficult to understand than the equivalent regular expression.

To use regular expressions in Python we need to import the `re` module. The `re` module is part of Python's standard library. Importing modules in Python is achieved using the `import` keyword.

What is a standard library?

In computing a standard library refers to a set of functionality that comes built-in with the core programming language.

```
>>> import re
```

[63] https://docs.python.org/2/library/string.html

Let us store a FASTA description line in a variable.

```
>>> fasta_desc = ">sp|Q6GZX4|001R_FRG3G"
```

Now, let us search for the UniProt identifer `Q6GZX4` within the line.

```
>>> re.search(r"Q6GZX4", fasta_desc)
<_sre.SRE_Match object at 0x...>
```

There are two things to note here:

1. We use a raw string to represent our regular expression, i.e. the string prefixed with an `r`
2. The regular expression `search()` method returns a match object (or None if no match is found)

What is a "raw string"?

In Python "raw" strings differ from regular strings in that the bashslash `\` character is interpreted literally. For example the regular string equivalent of `r"\n"` would be `"\\n"` where the first backslash is used to escape the effect of the second (remember that `\n` represents a newline).
Raw strings were introduced in Python to make it easier to create regular expressions that rely heavily on the use of literal backslashes.

The index of the first matched character can be accessed using the match object's `start()` method. The match object also has an `end()` method that returns the index of the last character + 1.

```
>>> match = re.search(r"Q6GZX4", fasta_desc)
>>> if match:
...     print(fasta_desc[match.start():match.end()])
...
Q6GZX4
```

In the above we make use of the fact that Python strings support slicing. Slicing is a means to access a subsection of a sequence. The `[start:end]` syntax is inclusive for the start index and exclusive for the end index.

```
>>> "012345"[2:4]
'23'
```

To see the merit of regular expressions we need to create one that matches more than one thing. For example a regular expression that could match all the patterns `id0`, `id1`, ..., `id9`.

Now suppose that we had a list containing FASTA description lines with these types of identifiers.

```
>>> fasta_desc_list = [">id0 match this",
...                    ">id9 and this",
...                    ">id100 but not this (initially)",
...                    "AATCG"]
...
```

Note that the list above also contains a sequence line that we never want to match.

Let us loop over the items in this list and print out the lines that match our identifier regular expression.

```
>>> for line in fasta_desc_list:
...     if re.search(r">id[0-9]\s", line):
...         print(line)
...
>id0 match this
>id9 and this
```

There are two noteworthy aspects of the regular expression. Firstly, the `[0-9]` syntax means match any digit. Secondly, the `\s` regular expression meta character means match any white space character.

The `[0-9]` syntax works in Bash too!

For example to list the files `photo_0.png`, `photo_1.png`, ..., `photo_9.png` you could use the command.

```
$ ls photo_[0-9].png
```

If one wanted to create a regular expression to match an identifier with an arbitrary number of digits one can make use of the `*` meta character, which causes the regular expression to match the preceding expression 0 or more times.

```
>>> for line in fasta_desc_list:
...     if re.search(r">id[0-9]*\s", line):
...         print(line)
...
>id0 match this
>id9 and this
>id100 but not this (initially)
```

It is possible to extract specific pieces of information from a line using regular expressions. This uses a concept known as "groups", which are indicated using parenthesis. Let us try to extract the UniProt identifier from a FASTA description line.

```
>>> print(fasta_desc)
>sp|Q6GZX4|001R_FRG3G
>>> match = re.search(r">sp\|([A-Z,0-9]*)\|", fasta_desc)
```

Warning: Note how horrible and incomprehensible the regular expression is.

It took me a couple of attempts to get this regular expression right as I forgot that `|` is a regular expression meta character that needs to be escaped using a backslash `\`.

The regular expression representing the UniProt idendifier `[A-Z,0-9]*` means match capital letters (`A-Z`) and digits (`0-9`) zero or more times (`*`). The UniProt regular expression is enclosed in parenthesis. The parenthesis denote that the UniProt identifier is a group that we would like access to. In other words, the purpose of a group is to give the user access to a section of interest within the regular expression.

```
>>> match.groups()
('Q6GZX4',)
>>> match.group(0)  # Everything matched by the regular expression.
'>sp|Q6GZX4|'
>>> match.group(1)
'Q6GZX4'
```

Note: There is a difference between the `groups()` and the `group()` methods. The former returns a tuple containing all the groups defined in the regular expression. The latter takes an integer as input and returns a specific group. However, confusingly `group(0)` returns everything matched by the regular expression and `group(1)` returns the first group making the `group()` method appear as if it used a one-based indexing scheme.

Finally Let us have a look at a common pitfall when using regular expressions in Python: the difference between the methods search() and match().

```
>>> print(re.search(r"cat", "my cat has a hat"))
<_sre.SRE_Match object at 0x...>
>>> print(re.match(r"cat", "my cat has a hat"))
None
```

Basically `match()` only looks for a match at the beginning of the string to be searched. For more information see the search() vs match()[64] section in the Python documentation.

There is a lot more to regular expressions in particular all the meta characters. For more information have a look at the regular expressions operations[65] section in the Python documentation.

12.4 Extracting the organism name

Armed with our new found knowledge of string processing let's create a function for extracting the organism name from a SwissProt FASTA description line. In other words given the lines:

```
>sp|P01090|2SS2_BRANA Napin-2 OS=Brassica napus PE=2 SV=2
>sp|Q15942|ZYX_HUMAN Zyxin OS=Homo sapiens GN=ZYX PE=1 SV=1
>sp|Q6QGT3|A1_BPT5 A1 protein OS=Escherichia phage T5 GN=A1 PE=2 SV=1
```

We would like to extract the strings:

```
Brassica napus
Homo sapiens
Escherichia phage T5
```

There are three things which are worth noting:

1. The organism name string is always preceeded by the key `OS` (Organism Name)
2. The organism name string can contain more than two words
3. The two letter key after the organism name string can vary, in the case above we see both `PS` (Protein Existence) and `GE` (Gene Name)

For more information about the UniProt FASTA description line go to UniProt's FASTA header[66] page.

The three FASTA description lines examined above provide an excellent basis for creating a test for the function that we want. Add the lines below to your `fasta_utils.py` file.

```
def test_extract_organism_name():
    """Test the extract_organism_name() function."""
    print("Testing the extract_organism_name() function...")
    lines = [">sp|P01090|2SS2_BRANA Napin-2 OS=Brassica napus PE=2 SV=2",
        ">sp|Q15942|ZYX_HUMAN Zyxin OS=Homo sapiens GN=ZYX PE=1 SV=1",
        ">sp|Q6QGT3|A1_BPT5 A1 protein OS=Escherichia phage T5 GN=A1 PE=2 SV=1"]
    organism_names = ["Brassica napus", "Homo sapiens", "Escherichia phage T5"]
    for line, organism_name in zip(lines, organism_names):
        assert extract_organism_name(line) == organism_name

test_is_description_line()
```

In the above we make use of pythons built-in `zip()` function. This function takes two lists as inputs and returns a list with paired values from the input lists.

[64] https://docs.python.org/2/library/re.html#search-vs-match
[65] https://docs.python.org/2/library/re.html
[66] http://www.uniprot.org/help/fasta-headers

```
>>> zip(["a", "b", "c"], [1, 2, 3])
[('a', 1), ('b', 2), ('c', 3)]
```

Let's make sure that the tests fail.

```
$ python fasta_utils.py
Testing the is_description_line() function...
```

What, no error message, what is going on? Ah, we added the test, but forgot to add a line to call it. Let's rectify that.

```
25 test_is_description_line()
26 test_extract_organism_name()
```

Let's try again.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 26, in <module>
    test_extract_organism_name()
  File "fasta_utils.py", line 23, in test_extract_organism_name
    assert extract_organism_name(line) == organism_name
NameError: global name 'extract_organism_name' is not defined
```

Success! We now have a failing test informing us that we need to create the `extract_organism_name()` function. Let's do that.

```
15 def extract_organism_name(line):
16     """Return the organism name from a FASTA description line."""
```

Let's find out where this minimal implementation gets us.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 29, in <module>
    test_extract_organism_name()
  File "fasta_utils.py", line 26, in test_extract_organism_name
    assert extract_organism_name(line) == organism_name
AssertionError
```

The test fails as expected. However, since we are looping over many input lines it would be good to get an idea of which test failed. We can achieve this by making use of the fact that we can provide a custom message to be passed to the `AssertionError`. Let us pass it the input line. Note the addition of the trailing `, line` in line 26.

```
25     for line, organism_name in zip(lines, organism_names):
26         assert extract_organism_name(line) == organism_name, line
```

Let's see what we get now.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 29, in <module>
    test_extract_organism_name()
  File "fasta_utils.py", line 26, in test_extract_organism_name
    assert extract_organism_name(line) == organism_name, line
AssertionError: >sp|P01090|2SS2_BRANA Napin-2 OS=Brassica napus PE=2 SV=2
```

Much better! Let us try to implement a basic regular expression to make this pass. First of all we need to make sure we import the re module.

```python
"""Module containing utility functions for working with FASTA files."""

import re
```

Then we can implement a regular expression to try to extract the organism name.

```python
def extract_organism_name(line):
    """Return the organism name from a FASTA description line."""
    match = re.search(r"OS=(.*) PE=", line)
    return match.group(1)
```

Remember the .* means match any character zero or more times and the surrounding parenthesis creates a group.

Let us see what happens now.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 34, in <module>
    test_extract_organism_name()
  File "fasta_utils.py", line 31, in test_extract_organism_name
    assert extract_organism_name(line) == organism_name, line
AssertionError: >sp|Q15942|ZYX_HUMAN Zyxin OS=Homo sapiens GN=ZYX PE=1 SV=1
```

Progress! We are now seeing a different error message. The issue is that the key after the regular expression is GN rather than PE. Let us try to rectify that.

```python
def extract_organism_name(line):
    """Return the organsim name from a FASTA description line."""
    match = re.search(r"OS=(.*) [A-Z]{2}=", line)
    return match.group(1)
```

The regular expression now states that instead of PE it wants any capital letter [A-Z] repeated twice {2}. Let's find out if this fixes the issue.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 33, in <module>
    test_extract_organism_name()
  File "fasta_utils.py", line 30, in test_extract_organism_name
    assert extract_organism_name(line) == organism_name, line
AssertionError: >sp|P01090|2SS2_BRANA Napin-2 OS=Brassica napus PE=2 SV=2
```

What, back at square one again? As mentioned previously, regular expressions can be painful and should only be used as a last resort. This also exemplifies why it is important to have tests. Sometimes you think you make an innocuous change, but instead things just fall apart.

At this stage the error message is not very useful, let us change it to print out the value returned by the function instead.

```
    for line, organism_name in zip(lines, organism_names):
        assert extract_organism_name(line) == organism_name, extract_organism_name(line)
```

Now, let's see what is going on.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 33, in <module>
    test_extract_organism_name()
  File "fasta_utils.py", line 30, in test_extract_organism_name
    assert extract_organism_name(line) == organism_name, extract_organism_name(line)
AssertionError: Brassica napus PE=2
```

Our regular expression is basically matching too much. The reason for this is that the `*` meta character acts in a "greedy" fashion matching as much as possible, see Fig. 12.1. In this case the `PE=2` is included in the match group as `[A-Z]{2}` is matched by the `SV=` key at the end of the line. The fix is to make the `*` meta character act in a "non-greedy" fashion. This is achieved by adding a `?` suffix to it.

```
def extract_organism_name(line):
    """Return the organism name from a FASTA description line."""
    match = re.search(r"OS=(.*?) [A-Z]{2}=", line)
    return match.group(1)
```

Let's find out what happens now.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
```

All the tests pass! Well done, time for another cup of tea.

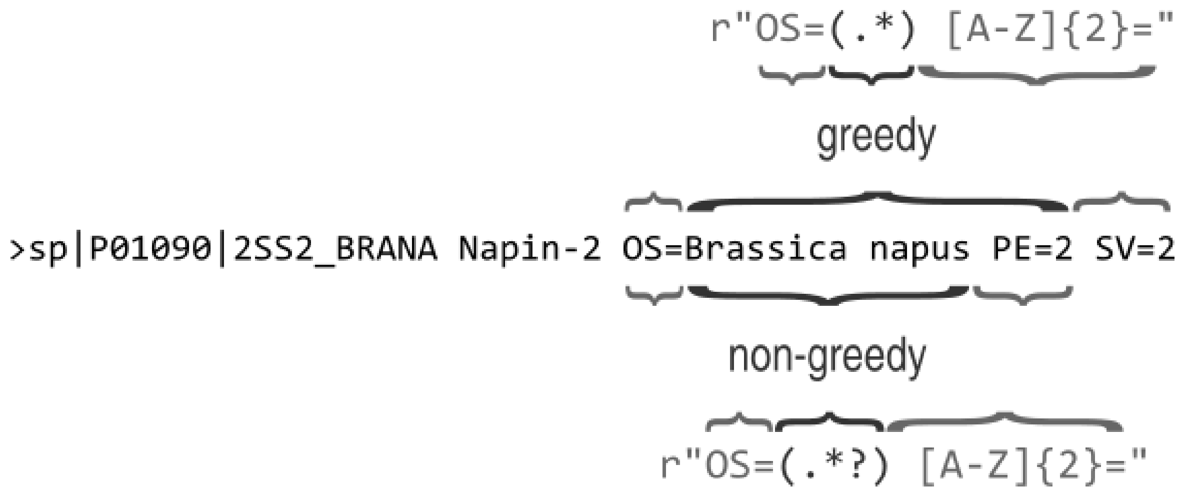


Fig. 12.1: Figure illustrating the difference between a greedy and a non-greedy regular expression. The `*` meta-character causes the previous regular expression, in this case the any character (`.`), to be matched zero or more times. This is done in a greedy fashion, i.e. trying to match as much text as possible. To make that part of the regular expression match as little text as possible, i.e. to work in a non-greedy fashion, we use the meta-character `*?`.

12.5 Only running tests when the module is called directly

In Python, modules provide a means to group related functionality together. For example we have already looked at and made use of the re module, which groups functionality for working with regular expressions.

In Python any file with a .py extension is a module. This means that the file that we have been creating, fasta_utils.py, is a module.

To make use of the functionality within a module one needs to import it. Let's try this out in an interactive session

```
$ python
```

Now we can import the module.

```
>>> import fasta_utils
Testing the is_description_line() function...
Testing the extract_organism_name() function...
```

Note that the tests run just like when we call the fasta_utils.py script directly. This is an undesired side effect of the current implementation. It would be better if the tests were not run when the module is imported.

To improve the behaviour of the fasta_utils module we will make use of a special Python attribute called __name__, which provides a string representation of *scope*. When commands are run from a script or the interactive prompt the name attribute is set to __main__. When a module is imported the __name__ attribute is set to the name of the module.

```
>>> print(__name__)
__main__
>>> print(fasta_utils.__name__)
fasta_utils
```

Using this information we can update the fasta_utils.py file with the changes highlighted below.

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

Let us make sure that the tests still run if we run the script directly.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
```

Now we can reload the module in the interactive prompt we were working in earlier to make sure that the tests no longer get executed.

```
>>> reload(fasta_utils)
<module 'fasta_utils' from 'fasta_utils.py'>
```

The test suite is no longer being called upon loading the module and no lines logging the progress of the testing suite is being printed to the terminal.

Note that simply calling the import fasta_utils command again will not actually detect the changes that we made to the fasta_utils.py file, which is why we make use of Python's built-in reload() function. Alternatively, one could have exited the Python shell, using Ctrl-D or the exit() function, and then started a new interactive Python session and imported the fasta_utils module again.

12.6 Counting the number of unique organisms

We can now use the fasta_utils module to start answering some of the biological questions that we posed at the beginning of this chapter. For now let us do this using an interactive Python shell.

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now we will start by importing the modules that we want to use. In this case fasta_utils for processing the data and gzip for opening and reading in the data from the uniprot_sprot.2015-11-26.fasta.gz file.

```
>>> import fasta_utils
>>> import gzip
```

Now we create a list for storing all the FASTA description lines.

```
>>> fasta_desc_lines = []
```

We can then open the file using the gzip.open() function. Note that this returns a file handle.

```
>>> file_handle = gzip.open("../data/uniprot_sprot.2015-11-26.fasta.gz")
```

Using a for loop we can iterate over all the lines in the input file.

```
>>> for line in file_handle:
...     if fasta_utils.is_description_line(line):
...         fasta_desc_lines.append(line)
...
```

When we are finished with the input file we must remember to close it.

```
>>> file_handle.close()
```

Let's check that the number of FASTA description lines using the built-in len() function. This function returns the number of items in the list, i.e. the length of the list.

```
>>> len(fasta_desc_lines)
549832
```

Okay now it is time to find the number of unique organisms. For this we will make use of a data structure called set. In Python sets are used to compare collections of unique elements. This means that sets are ideally suited for operations that you may associate with Venn diagrams.

However, in this instance we simply use the set data structure to ensure that we only get one unique representative of each organism. In other words even if one calls the set.add() function several times with the same item the item will only occur once in the set.

```
>>> organisms = set()
>>> for line in fasta_desc_lines:
...     s = fasta_utils.extract_organism_name(line)
...     organisms.add(s)
...
>>> len(organisms)
13251
```

Great, now we know that there are 13,251 unique organisms represented in the FASTA file.

12.7 Finding the number of variants per species and the number of proteins per variant

Suppose we had a FASTA file containing only four entries with the description lines below.

```
>sp|P12334|AZUR1_METJ Azurin iso-1 OS=Methylomonas sp. (strain J) PE=1 SV=2
>sp|P12335|AZUR2_METJ Azurin iso-2 OS=Methylomonas sp. (strain J) PE=1 SV=1
>sp|P23827|ECOT_ECOLI Ecotin OS=Escherichia coli (strain K12) GN=eco PE=1 SV=1
>sp|B6I1A7|ECOT_ECOSE Ecotin OS=Escherichia coli (strain SE11) GN=eco PE=3 SV=1
```

The analysis in the previous section would have identified these as three separate entities.

```
Methylomonas sp. (strain J)
Escherichia coli (strain K12)
Escherichia coli (strain SE11)
```

Now, suppose that we wanted to find out how many variants there were of each species. In the example above there would be be one variant of `Methylomonas sp.` and two variants of `Escherichia coli`. Furthermore, suppose that we also wanted to find out how many proteins were associated with each variant.

We could achieve this by creating a nested data structure using Python's built in dictionary type. At the top level we should have a dictionary whose keys were the species, e.g. `Escherichia coli`. The values in the top level dictionary should themselves be dictionaries. The keys of the nested dictionaries should be the full organism name, e.g. `Escherichia coli (strain K12)`. The values in the nested dictionary should be an integer representing the number of proteins found for that organism. Below is a YAML representation of the data structure that should be created from the four entries above.

```
---
Methylomonas sp.:
  Methylomonas sp. (strain J): 2
Escherichia coli:
  Escherichia coli (strain K12): 1
  Escherichia coli (strain SE11): 1
```

So what type of functionality would we need to achieve this? First of all we need a function that given an organism name returns the associated species. In other words something that converts `Escherichia coli (strain K12)` to `Escherichia coli`. Secondly, we need a function that given a list of organism names returns the data structure described above.

Let us start by creating a test for converting the organism name into a species. Add the test below to `fasta_utils.py`.

```
def test_organism_name2species():
    print("Testing the organism_name2species() function...")
    assert organism_name2species("Methylomonas sp. (strain J)") == "Methylomonas sp."
    assert organism_name2species("Homo sapiens") == "Homo sapiens"

if __name__ == "__main__":
    test_is_description_line()
    test_extract_organism_name()
    test_organism_name2species()
```

Let's find out what error this gives us.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Testing the organism_name2species() function...
```

```
Traceback (most recent call last):
  File "fasta_utils.py", line 40, in <module>
    test_organism_name2species()
  File "fasta_utils.py", line 34, in test_organism_name2species
    assert organism_name2species("Methylomonas sp. (strain J)") == "Methylomonas sp."
NameError: global name 'organism_name2species' is not defined
```

The error message is telling us that we need to define the `organism_name2species()` function. Add the lines below to define it.

```python
def organism_name2species(organism_name):
    """Return the species from the FASTA organism name."""
```

Now we get a new error message when we run the tests.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Testing the organism_name2species() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 43, in <module>
    test_organism_name2species()
  File "fasta_utils.py", line 37, in test_organism_name2species
    assert organism_name2species("Methylomonas sp. (strain J)") == "Methylomonas sp."
AssertionError
```

Great, let us add some logic to the function.

```python
def organism_name2species(organism_name):
    """Return the species from the FASTA organism name."""
    words = organism_name.split()
    return words[0] + " " + words[1]
```

Above, we split the organism name based on whitespace separators and return the first two words joined by a space character.

Note: In Python, and many other scripting languages, strings can be concatenated using the + operator.

```
>>> "hello" + " " + "world"
'hello world'
```

Time to test the code again.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Testing the organism_name2species() function...
```

Great, the function is working! Let us define a new test for the function that will generate the nested data structure we described earlier.

```python
def test_summarise_species_protein_data():
    print("Testing summarise_species_protein_data() function...")
    fasta_desc_lines = [
">sp|P12334|AZUR1_METJ Azurin iso-1 OS=Methylomonas sp. (strain J) PE=1 SV=2",
">sp|P12335|AZUR2_METJ Azurin iso-2 OS=Methylomonas sp. (strain J) PE=1 SV=1",
">sp|P23827|ECOT_ECOLI Ecotin OS=Escherichia coli (strain K12) GN=eco PE=1 SV=1",
">sp|B6I1A7|ECOT_ECOSE Ecotin OS=Escherichia coli (strain SE11) GN=eco PE=3 SV=1"
```

```python
    ]
    summary = summarise_species_protein_data(fasta_desc_lines)

    # The top level dictionary will contain two entries.
    assert len(summary) == 2
    assert "Methylomonas sp." in summary
    assert "Escherichia coli" in summary

    # The value of the Methylomonas sp. entry is a dictionary with one
    # entry in it.
    assert len(summary["Methylomonas sp."]) == 1
    assert summary["Methylomonas sp."]["Methylomonas sp. (strain J)"] == 2

    # The value of the Escherichia coli entry is a dictionary with two
    # entries in it.
    assert len(summary["Escherichia coli"]) == 2
    assert summary["Escherichia coli"]["Escherichia coli (strain K12)"] == 1
    assert summary["Escherichia coli"]["Escherichia coli (strain SE11)"] == 1

if __name__ == "__main__":
    test_is_description_line()
    test_extract_organism_name()
    test_organism_name2species()
    test_summarise_species_protein_data()
```

This should all be getting familiar now. Time to run the tests.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Testing the organism_name2species() function...
Testing summarise_species_protein_data() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 70, in <module>
    test_summarise_species_protein_data()
  File "fasta_utils.py", line 50, in test_summarise_species_protein_data
    summary = summarise_species_protein_data(fasta_desc_lines)
NameError: global name 'summarise_species_protein_data' is not defined
```

Again we start by defining the function.

```python
def summarise_species_protein_data(fasta_desc_lines):
    """Return data structure summarising the organism and protein data."""
```

And then we run the tests again.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Testing the organism_name2species() function...
Testing summarise_species_protein_data() function...
Traceback (most recent call last):
  File "fasta_utils.py", line 74, in <module>
    test_summarise_species_protein_data()
  File "fasta_utils.py", line 57, in test_summarise_species_protein_data
    assert len(summary) == 2
TypeError: object of type 'NoneType' has no len()
```

Time to add an implementation.

```
42 def summarise_species_protein_data(fasta_desc_lines):
43     """Return data structure summarising the organism and protein data."""
44     summary = dict()
45     for line in fasta_desc_lines:
46         variant_name = extract_organism_name(line)
47         species_name = organism_name2species(variant_name)
48         variant_dict = summary.get(species_name, dict())
49         variant_dict[variant_name] = variant_dict.get(variant_name, 0) + 1
50         summary[species_name] = variant_dict
51     return summary
```

In the above we make use of the dictionary's built-in `get()` method, which returns the value associated with the key provided as the first argument. If the key does not yet exist in the dictionary it returns the second argument, the default value.

On line 48 we try to access a `variant_dict` dictionary from within the `summary` dictionary, we are in other words working with a nested data structure. If there is no entry associated with the `species_name` the `get()` method will return an empty dictionary (`dict()`).

On line 49 we keep count of the number of times a variant of a species, as defined by the full organism name, has been observed. In this instance we use the `get()` method to get the number of times the variant has been observed previously and then we add 1 to it. If the variant has never been observed previously it will not be in the dictionary and the `variant_dict.get(variant_name, 0)` method call will return 0 (the default value specified).

Line 50 updates the top level dictionary by adding the `variant_dict` dictionary to the `summary` dictionary.

Let's see if it the implementation works as expected.

```
$ python fasta_utils.py
Testing the is_description_line() function...
Testing the extract_organism_name() function...
Testing the organism_name2species() function...
Testing summarise_species_protein_data() function...
```

Hurray!

Finally, let us write a separate script to convert an input FASTA file into a YAML summary file. Create the file `fasta2yaml_summary.py` and add the code below to it.

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

import sys
import yaml

import fasta_utils

fasta_desc_lines = list()

with sys.stdin as fh:
    for line in fh:
        if fasta_utils.is_description_line(line):
            fasta_desc_lines.append(line)

summary = fasta_utils.summarise_species_protein_data(fasta_desc_lines)
yaml_text = yaml.dump(summary, explicit_start=True, default_flow_style=False)

with sys.stdout as fh:
    fh.write(yaml_text)
```

In the code above we make use of the yaml module to convert our data structure to the YAML file format. The PyYAML package is not part of the Python's standard library, but it is easily installed using `pip` (for more detailed instructions see *Installing Python packages* (page 142)).

```
$ sudo pip install pyyaml
```

The script also makes use of `sys.stdin` and `sys.stdout` to read from the *standard input stream* and write to the *standard output stream*, respectively. This means that we can *pipe* in the content to our script and pipe output from our script. For example to examine the YAML output using the `less` pager one could use the command below.

```
$ gunzip -c data/uniprot_sprot.2015-11-26.fasta.gz | python fasta2yaml_summary.py | less
```

This immediately reveals that there are organisms in the SwissProt FASTA file that have few protein associated with them.

Note: Remember the benefits of using pipes was described in *Creating a work flow using pipes* (page 17).

```
---
AKT8 murine:
  AKT8 murine leukemia virus: 1
AKV murine:
  AKV murine leukemia virus: 3
Abelson murine:
  Abelson murine leukemia virus: 3
Abies alba:
  Abies alba: 1
Abies balsamea:
  Abies balsamea: 3
```

Note: The formatting in the YAML file above was created for us by the call to the `yaml.dump()` method in the `fasta2yaml_summary.py` script.

Great work!

12.8 Key concepts

- Think about the problem at hand before you start coding
- Break down large tasks into smaller more manageable tasks, repeat until the tasks seem trivial
- Test-driven development is a software development practise that can help break tasks into manageable chunks that can be tested
- Many aspects of biological data processing boil down to string manipulations
- Regular expressions are a powerful tool for performing string manipulations, but use with caution as they can result in confusion
- Python has many built-in packages for performing complex tasks, in this chapter we used the `re` package for working with regular expressions
- There are also many third party Python packages that can be installed, in this chapter we made use of the `yaml` package for writing out a data structure as a YAML file

- If in doubt, turn to the abundance of resources for Python online including manuals, tutorials and help forums

CHAPTER 13

Working remotely

Sometimes you need to work on a computer different to the one that is in front of you. This may be because you are working from home and need to log into your work computer. Alternatively it may be because you need to do some work on a high-performance computing cluster.

In this chapter we will cover tools for logging into remote machines using the SSH (Secure SHell) protocol. SSH is an open standard that is available on all UNIX based systems. In fact it has gained so much traction that Microsoft have decided to add support for it in Windows.

13.1 Jargon busting

Although, it is relatively simple to log into a remote machine using SSH explaining it requires the use of some networking terminology (as we are logging into the remote machine over a network). Let us therefore start off with some jargon busting before we get into the practical details.

A computer can be connected to a network using physical Ethernet cables or in a wireless fashion using WiFI. Computers connected to the same network can communicate with each other. This communication takes place over so called “ports”, which are identified by numbers. A famous port is port 80, which is used for HTTP (the HyperText Transfer Protocol) a.k.a. web-browsing. There are many other ports. For example port 443 is used for HTTPS and port 22 is used for SSH.

Machines on a network can be identified and accessed via their IP (Internet Protocol) address. An IP address that you may have come across is `127.0.0.1`, it identifies the local machine that you are working on.

Although very useful, IP addresses can be difficult to remember. In order to overcome this problem, internet service providers as well as the people responsible for looking after your organisations network make use of domain name servers (DNS). The purpose of a DNS server is to translate unique resource locations (URLs) to IP addresses.

A DNS server can also be used to lookup the IP address of a machine given a simpler more memorable name. To illustrate this we can find the IP address(es) of Google’s web servers using the command `host`.

```
$ host www.google.com
www.google.com has address 62.164.169.148
www.google.com has address 62.164.169.185
www.google.com has address 62.164.169.163
www.google.com has address 62.164.169.177
www.google.com has address 62.164.169.166
www.google.com has address 62.164.169.152
www.google.com has address 62.164.169.155
www.google.com has address 62.164.169.154
www.google.com has address 62.164.169.176
www.google.com has address 62.164.169.159
www.google.com has address 62.164.169.165
www.google.com has address 62.164.169.181
www.google.com has address 62.164.169.187
www.google.com has address 62.164.169.170
www.google.com has address 62.164.169.144
www.google.com has address 62.164.169.174
www.google.com has IPv6 address 2a00:1450:4009:810::2004
```

If you copy and paste one of the IP addresses above into a web browser you should see the Google home page.

Throughout the remainder of this chapter we will connect to remote machines using their hostname. What happens in practise is that your institutes DNS server translates this hostname to an IP address and you send your communications over the network to the machine identified by that IP address. Furthermore when you communicate with the remote machine you will be using a specific port. The default port for the SSH protocol is port 22.

13.2 Logging in using the Secure Shell protocol

In its most basic form the `ssh` command takes on the form `ssh hostname`. Where `hostname` is the name of the remote machine that you want to log in to. This assumes that you want to login to the remote machine using the same user name as that on the local machine. This is often not the case and it is common to see the `ssh` command in the form `ssh user@hostname`, where `user` is the Unix user name you want to log in as on the remote machine.

Let us illustrate this with an example. Suppose that one wanted to login to a remote computer named `hpc`, this could for example be the head node on your institutes high-performance computing cluster. Assuming that your user name on the head node was `olssont` then you could log in using the command below.

```
$ ssh olssont@hpc
```

Warning: All of the remote machines in this chapter are fictional. This means that if you try to run the commands verbatim you will see errors along the lines of the below.

```
ssh: Could not resolve hostname ...
```

When trying these examples make sure that you are trying to connect to a machine that exists on your network.

If the machine that you are trying to log in to has a port 22 open (the default SSH port) you will be prompted for your password.

What is a head node?

A computing cluster is basically a collection of computers. Computing clusters tend to make use of a scheduler to distribute jobs on the cluster. A scheduler is basically a piece of software that has an understanding of the computing resources available on the cluster. A user submits a job to the scheduler and the scheduler puts the job in a queue and when appropriate resources become available it starts the job on the cluster.

Most schedulers require a so called head node, also known as a master node, which acts as the control centre. A user of a cluster would therefore login to the head node and from there the user would submit a job to the scheduler. The scheduler would then dispatches the job on one or more cluster nodes when the appropriate resources became available.

The above assumes that your DNS server can resolve the name `hpc` into an IP address. Depending on how things are setup in your organisation you may need to use a more explicit name, for example `hpc.awesomeuni.ac.uk`.

```
$ ssh olssont@hpc.awesomeuni.ac.uk
```

If your user name and password authenticates successfully the shell in your terminal will now be from the remote machine. To find out the name of the machine that you are logged into you can run the command `hostname`.

```
$ hostname
hpc
```

To disconnect from the remote host you can use `Ctrl-D` or the `exit` command.

```
$ exit
$ hostname
olssont-laptop
```

In the above the `hostname` command prints out the hostname of the local machine.

By default port 22 is used for the SSH protocol. However, sometimes a machine may expose its SSH server on a different port. For example if we had a machine called `bishop` that exposed its SSH server on port 2222 one could login to it using the command below.

```
$ ssh -p 2222 olssont@bishop
```

In the above the `-p` flag is used to specify the port to connect to.

Sometimes you want to be able to run software that makes use of windowing systems (i.e. all software with a graphical user interface). For example the statistical software package `R` has built in functionality for displaying plots in a graphical window, which means it requires a windowing system. Most Unix-based systems make use of the X11 as their windowing system. We therefore need to enable X11-forwarding in SSH to be able to run programs that require graphics. This is achieved using the `-X` flag.

```
$ ssh -X olssont@pawn
```

In the above we are connecting to a machine named `pawn` with X11-forwarding enabled.

13.3 Copying files using Secure Copy

Now that we know how to login to a remote machine we need to work out how to copy data to and from it. This is achieved using the `scp`, *secure copy*, command.

Suppose that we wanted to copy the file `mydata.csv` over to `olssont`'s home directory on the `hpc` head node, we could achieve this using the command below.

```
$ scp mydata.csv olssont@hpc:
```

Note the colon (`:`) after the host name. It demarcates the end of the host name and the beginning of the location to copy the file to on the remote machine. In this instance the latter is left empty and as such the original file name is used and the location for the file defaults to `olssont`'s home directory on the remote machine. The command above is equivalent to that below which specifies the home directory using a relative path (`~/`).

```
$ scp mydata.csv olssont@hpc:~/
```

It is also possible to specify the location using an absolute path. For example if we wanted to save the file in the `/tmp` directory this could be achieved using the command below.

```
$ scp mydata.csv olssont@hpc:/tmp/
```

Just like with the `cp` command it is possible to give the copied file a different name. For example to name it `data.csv` (and place it in the `/tmp` directory) one could use the command below.

```
$ scp mydata.csv olssont@hpc:/tmp/data.csv
```

If the SSH server is listening on a port other than 22 one needs to specify the port explicitly. Confusingly the argument for this is not the same as for the `ssh` command. The `scp` command uses the argument `-P`, i.e. it uses upper rather than lower case. So if we wanted to copy the data to `bishop`, where the SSH server is listening on port 2222 one could use the command below.

```
$ scp -P 2222 mydata.csv olssont@bishop:
```

Sometimes one wants to copy the entire content of a directory. In this case one can use the `-r` option to *recursively* copy all the content of the specified directory. For example if we had a directory named `data` and we wanted to copy it to `pawn` one could use the command below.

```
$ scp -r data/ olssont@pawn:
```

All of the commands above will prompt you for your password. This can get tedious. In the next section we will look at a more secure and less annoying way of managing the authentication step when working with remote machines.

13.4 Password-less authentication using SSH keys

An alternative and more secure method to using password based authentication is to use public-key cryptography. Public-key cryptography, also known as asymmetric cryptography, uses a pair of so called "keys". One of these keys is public and one is private. The public key is one that you can distribute freely, in this case to all the remote machines that you want to be able to login to. However, the private key must never be compromised as it is what allows you access to all the remote machines. One way to think about this system is to view the public key as a lock that only the private key can open. You can fit the all the machines that you want secure access to with copies of the same public key as long as you keep the private key safe.

Enough theory let's try it out.

The first step is to generate a public/private key pair. This is achieved using the command `ssh-keygen`. This will prompt you for the file to save the key as, the default `~/.ssh/id_rsa` file is a good option if you have not yet setup any key pairs. You will then be prompted, to optionally, enter a passphrase. This provides another layer of protection in case someone gets hold of your private key. However, it does mean that you will be prompted for the passphrase the first time you make use of the key in a newly booted system. Personally, I am paranoid so I make use of the passphrase and I suggest that you do too.

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/olssont/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

If you used the default naming scheme for your key pair you should now have two files in your `.ssh` directory: `id_rsa` (your private key) and `id_rsa.pub` (your public key).

```
$ ls -l .ssh/
-rw-------  1 olssont  NR4\Domain Users   1679 23 Feb  2015 id_rsa
-rw-r--r--  1 olssont  NR4\Domain Users    407 23 Feb  2015 id_rsa.pub
```

Note that only the user has read/write permissions on the private key, whereas the pubic key is readable by everyone.

Now let us setup password-less login to the cluster head node. First of all let us copy the public key to the remote machine using `scp`.

```
$ scp ~/.ssh/id_rsa.pub olssont@hpc:
```

Now we need to login to the head node to configure it. At this point we will still need to use our password. Once logged into the head node we need to create a `.ssh` directory in the user's home directory (if it does not already exist). We then need to append the public key to a file named `authorized_keys` in the `.ssh` directory. Finally we logout of the head node.

```
$ ssh olssont@hpc
$ hostname
hpc
$ mkdir .ssh
$ cat id_rsa.pub >> .ssh/authorized_keys
$ exit
```

What does the >> expression do?

The >> symbol is similar to the > redirection symbol. However, redirection using > will replace an existing file whereas >> will append to it. In this case we do not want to destroy any previously added public keys so we append to the `authorized_keys` file.

Now we should be able to `ssh` and `scp` to the head node in a password-less fashion. If you setup your key pair using a passphrase you will be prompted for it the first time you use the key pair.

Great that's really cool! However, it was quite a lot of work to get the public key onto the remote machine. There is a better way to do this using the program `ssh-copy-id`. Depending on the operating system that you are using may need to install this program, see Managing your system (page 137) for details on how to install software.

Once you have `ssh-copy-id` on your system you can provision a remote machine with your public key using a single command. Below we use it to add our pubic key to `bishop`.

```
$ ssh-copy-id -i ~/.ssh/id_rsa.pub olssont@bishop
```

The optional `-i` flag is used to specify which public key should be copied to the remote machine.

13.5 Managing your login details using SSH config

Suppose that access to your institutes cluster was setup in a way that required you to use the full `hpc.awesomeuni.ac.uk` host name, but that you wanted to be able to login using the shorter name `hpc`. You can configure your machine to setup access in this fashion by creating the file `.ssh/config` file and adding the lines below to it.

```
Host hpc
    HostName hpc.awsomeuni.ac.uk
    User olssont
```

The SSH configuration above also specifies the user name. This means that you can login to the head node using the command below (note the lack of a user name).

```
$ ssh hpc
```

As you start using SSH keys to manage access to various machines you are likely to find yourself using multiple key pairs. In this case you will want to be able to specify the name of the private key, also known as an identity file, in the `.ssh/config` file.

```
Host hpc
    HostName hpc.awsomeuni.ac.uk
    User olssont
    IdentityFile ~/.ssh/id_rsa
```

Finally in the examples described earlier access to `bishop` had been configured to use port 2222. To configure access to this remote machine we could use the specification below.

```
Host bishop
    HostName bishop
    User olssont
    Port 2222
    IdentityFile ~/.ssh/id_rsa
```

Again, using the `.ssh/config` file in this way means that we do not need to remember port numbers and what options to invoke the `scp` and `ssh` commands with. Copying a file can then be achieved using the concise syntax below.

```
$ scp mydata.csv bishop:
```

Logging in to the machine becomes similarly trivial.

```
$ ssh bishop
```

13.6 Executing long running commands on remote hosts

One problem that one can encounter when working on a remote machine is that if the connection is broken whilst a program is running it may fail.

Luckily, it is quite easy to work around this. All one needs to do is to prefix the command to run the program of interest with `nohup`. The `nohup` command makes the program of interest immune to hangups.

To see this in action open up two terminals on your computer. In one of them we will monitor the running processes using the command `top`.

```
$ top
```

This should display a lot of information about the current running processes. To make things a little easier to digest we can limit the output to the processes owned by you. Press `U`, which will prompt you for a user name. Enter your user name and press enter. You should now only see the processes owned by you.

In the second terminal we will simulate a long running program using the command `sleep`, which simply pauses execution for a specified number of seconds.

```
$ sleep 3600
```

In the first terminal, running `top`, you should now see the `sleep` program running.

Now close the second terminal, the one in which you are running the `sleep` command. Note that the `sleep` program disappears from the `top` display. This is because the program was interrupted by the closing of the terminal.

Open a new terminal. This time we will prefix the `sleep` command with `nohup`.

```
$ nohup sleep 3600
```

Now close the terminal running the `sleep` command again. Note that the `sleep` command is still present in the `top` display. It will keep running until it is finished in an hours time.

13.7 Key concepts

- You can use the `ssh` command to login to remote machines
- You can copy data to and from remote machines using the `scp` command
- You can use SSH keys to avoid having to type in your password every time you want to interact with a remote machine
- Using SSH keys is also more secure than using passwords
- If you need to interact with many remote machines it may make sense to create a `.ssh/config` file
- You can use `nohup` to ensure that long running processes are not killed by losing connection to the remote machine

CHAPTER 14

Managing your system

This chapter will give a brief overview of how to install software on Unix-like systems. Different systems operate in different ways and this can lead to confusion. This chapter aims to help you get an understanding of the basic principles as well as an overview of the most common systems giving you a solid foundation to build upon.

14.1 The file system

The file system on Unix-like systems is built up like a tree starting from, the so called, root. You can view the content of your root system by typing in `ls /`.

On a linux box this may look like the below.

```
$ ls /
bin  dev   etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root
run  sbin  srv  sys   tmp  usr    var
```

The files that you have been working with so far have been located in your home directory, e.g. `/home/olssont/`.

However, the programs that you have been running are also files. Programs fundamental to the operating system is located in `/bin`. Other programs installed by the systems package manager tend to be located in `/usr/bin`.

To find out the location of a program you can use the `which` command. For example, let us find out the location of the `ls` program.

```
$ which ls
/bin/ls
```

We can run the `ls` program using this absolute path, for example to view the content of the root directory again.

```
$ /bin/ls /
bin  dev   etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root
run  sbin  srv  sys   tmp  usr    var
```

14.2 The PATH environment variable

If running `ls` is equivalent to running `/bin/ls` it is worth asking how the shell knows how to translate `ls` to `/bin/ls`. Or more correctly how does the shell know to look for the `ls` program in the `/bin` directory?

The answer lies in the environment variable `PATH`. This environment variable tells the shell the locations where to go looking for programs. We can inspect the content of the `PATH` environment variable using the `echo` command.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

Note: Remember that in bash scripting we need to prefix variable names with a dollar sign (`$`) to access the value the variable is holding.

The `PATH` variable basically contains a list of colon (`:`) separated directories. When you try to run a program the shell looks in these directories, in order, and uses the first match it can find. If the program of interest is not located in any of these directories you will see a `command not found` error issued from your shell.

```
$ this-program-does-not-exist
bash: this-program-does-not-exist: command not found
```

14.3 The root user

Now let's have a look at the permissions of the `/usr/bin` directory. In the below we use the `-l` flag to list the result in "long" format, i.e. to view the permissions etc, and the `-d` flag to state that we want to list the directory itself rather than its content.

```
$ ls -ld /usr/bin/
drwxr-xr-x  1056 root  wheel  35904 22 Mar 11:15 /usr/bin/
```

In the above the directory is owned by the `root` user and belongs to the `wheel` group. The permissions on the directory states that only `root`, the owner, is allowed to write to the directory.

The `root` user is a special user that is all powerful, sometimes referred to as a superuser or "su" for short. These special "powers" of the superuser are often needed to perform systems administration tasks, like installing software and creating/deleting users.

On some systems you become the superuser, to perform systems administration tasks, by using the *switch user* (`su`) command. This defaults to switching to the superuser.

```
$ su
Password:
#
```

Note that this prompts you for the root password. However, depending on who provisioned your machine you may or may not have access to the root password. Note also that when you are logged in as the superuser the prompt tends to change to a hash symbol (`#`). This is to warn you that things that you do can have dire consequences.

A more modern approach to running commands with root privileges is to prefix the command of interest with `sudo`. This allows you to run a command as another user, the `root` user by default.

The details of who can run commands using `sudo` are stored in the `/etc/sudoers` file.

```
$ ls -l /etc/sudoers
-r--r-----  1 root  wheel  1275 10 Sep  2014 /etc/sudoers
```

Note that you need root privileges to be able to read this file. We can therefore illustrate the use of the `sudo` command by trying to read the file using the `less` pager.

```
$ sudo less /etc/sudoers
```

The only problem with the command above is that you won't be able to run it unless you are on the sudoer's list in the first place.

A consequence of the fact that only the `root` user can write files to the `/bin` and `/usr/bin` directories is that you need to have root privileges to install software (write files) to these default locations.

14.4 Using package managers

All modern Linux distribution come with a so called package manager, which should be your first port of call when trying to install software. Package managers make it easier to install software for two main reasons they resolve dependencies and they (usually) provide pre-compiled versions of software that are known to play nicely with the other software available through the package manager.

There are countless numbers of Linux distributions. However, most main stream distributions are derived from either Debian or RedHat. Debian based Linux distributions include amongst others Debian itself, Ubuntu, and Linux Mint. RedHat based distributions include RedHat, CentOS and Fedora.

Although Mac OSX comes with the AppStore this is not the place to look for scientific software. Instead two other options based on the idea of the Linux package managers have evolved the first one is Mac Ports[67] and the second one is Homebrew[68]. I would recommend using the latter as it has thriving scientific user community.

14.4.1 Installing software on Debian based systems

Debian-based systems come with a huge range of pre-package software available for installation using the Advance Package Tool (APT). To search for a piece of software package you would typically start off by updating the list of packages available for download using the `apt-get update` command.

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

One can then search for the specific software of interest, for example the multiple sequence alignment tools T-Coffee[69], using the `apt-cache search` command.

```
$ sudo apt-cache search t-coffee
t-coffee - Multiple Sequence Alignment
t-coffee-examples - annotated examples for the use of T-Coffee
```

To install the software package one uses the `apt-get install` command.

```
$ sudo apt-get install t-coffee
```

To uninstall a package one can use the `apt-get remove` command.

[67] https://www.macports.org/
[68] http://brew.sh/
[69] http://www.tcoffee.org/Projects/tcoffee/

```
$ sudo apt-get remove t-coffee
```

The command above leaves package configuration files intact in case you would want to re-use them in the future. To completely remove a package from the system one would use the `apt-get purge` command.

```
$ sudo apt-get purge t-coffee
```

14.4.2 Installing software on RedHat based systems

RedHat and its free clone CentOS come, with fewer software packages than Debian. The T-Coffee software, is for example not available. However, on the other hand RedHat is a super solid Linux distribution created by Red Hat Inc, the first billion dollar open source company.

RedHat based systems use the YUM package manager. To search for software one can use the `yum search` command. For example one could use the command below to search for the Git version control package.

```
$ yum search git
```

What doe YUM stand for?

YUM is an acronym for "Yellowdog Updater, Modified". This name symbolises that YUM is a rewritten and modified version of the "Yellowdog Updater" (YUP) tool, which was the package manager for the Yellowdog Linux distribution.

To install a package using YUM one uses the `yum install` command.

```
$ sudo yum install git
```

To uninstall a package one can use the `yum remove` command.

```
$ sudo yum remove git
```

RedHat based system also provide groups of software. One group that you will want to install is the "Development Tools" group. This includes the Gnu C Compiler (gcc) and the "make" tools that are often required to install other software from source code.

```
$ sudo yum groupinstall "Development Tools"
```

There are far fewer packages available for Redhat based distributions compared to Debian based distributions. To make available more software packages for the former it is worth adding the Extra Packages for Enterprise Linux (EPEL) repository. This can be achieved by running the command below.

```
$ sudo yum install epel-release
```

Warning: YUM has also got an "update" command. However, unlike APT where `apt-get update` updates the list of available software packages YUM's `yum update` will update all the installed software packages to the latest version.

14.4.3 Installing software on Mac OSX

This section will illustrate how to install software using the Homebrew[70] package manager.

[70] http://brew.sh/

First of all we need to install Homebrew itself. This can be achieved using the command below, taken from the Homebrew home page.

```
$ /usr/bin/ruby -e "$(curl \
-fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Homebrew refers to packages as "formulae". That is because each package/formulae is a ruby script describing how to install/brew a particular piece of software.

Homebrew, just like APT, contains a local list of formulae that can be syncronised with the online sources using the `brew update` command.

```
$ brew update
```

To search for a formulae one can use the `brew search` command. Let us for example search for the Git version control package.

```
$ brew search git
```

To install a formulae using Homebrew one uses the `brew install` command.

```
$ brew install git
```

To uninstall a formulae one uses the `brew uninstall` command.

```
$ brew uninstall git
```

One of the things that you will want to do is to add another "tap" to Homebrew. Namely, the `science` tap. In Homebrew a "tap" is an additional resource of formulae.

```
$ brew tap homebrew/science
```

We can now search for scientific software such as T-Coffee.

```
$ brew search t-coffee
```

And install it.

```
$ brew install t-coffee
```

14.5 Compiling software from source

Many scientific software packages are only available as source code. This may mean that you need to compile the software yourself in order to run it.

There are lots of different ways of compiling software from source. In all likelihood you will need to read and follow instructions provided with the software sources. The instructions are typically included in `README` or `INSTALL` text files.

The most common scenario is that you need to run three commands in the top level directory of the downloaded software.

The first command is to run a script named `configure` provided with the software.

```
$ ./configure
```

The `configure` script makes sure that all the dependencies are present on your system. For example if the software was written in `C` one of the tasks of the `configure` script would be to check that it could find a `C` compiler on your system.

Another task that is commonly performed by the `configure` script is to create a `Makefile`. We already encountered the `Makefile` in Automation is your friend (page 103). It is essentially a file describing how to build the software.

Building the software, using the instructions in the `Makefile`, is also the next step of the process. This is typically achieved by running the `make` command.

```
$ make
```

The `make` command typically creates a number of executable files, often in a subdirectory named `build`.

The final step is to install the software. This is achieved by copying the built executable files into a relevant directory present in your `PATH`. Since these directories are typically owned by root the final step typically requires superuser privileges.

```
$ sudo make install
```

14.6 Installing Python packages

Python is a high-level scripting language that is relatively easy to read and get to grips with. We have already made use of Python in the previous chapters.

It is possible to create re-usable software packages in Python. In fact there are many such Python packages aimed at the scientific community. Examples include numpy[71] and scipy[72] for numerical computing, sympy[73] for symbolic mathematics, matplotlib[74] for figure generation, pandas[75] for data structures and analysis and scikit-learn[76] for machine learning. There is also a package aimed directly at the biological community, namely biopython[77].

Most packages are hosted on PyPI[78] and can be installed using `pip`. The `pip` command comes prepackaged with Python since versions 2.7.9 and 3.4. If you have an older version of Python you may need to install `pip` manually, see the pip installation notes[79] for more details.

Another really useful package is `virtualenv`. I suggest installing it straight away.

```
$ sudo pip install virtualenv
```

Virtualenv is a tool that allows you to create virtual Python environments.

What is a Python virtual environment?

A Python virtual environment is essentially a local copy of your system's Python. The copy can live in your home directory. This means that you can install Python packages into your virtual environment without having root privileges. It also means that any packages that you install do not interfere with your system Python and vice versa. Furthermore, if you have multiple virtual environment they are all separate from each other. So if you have one virtual environment for each project that you are working on their Python packages can be of different versions without causing any problems.

[71] http://www.numpy.org/
[72] https://www.scipy.org/
[73] http://www.sympy.org/en/index.html
[74] http://matplotlib.org/
[75] http://pandas.pydata.org/
[76] http://scikit-learn.org/stable/
[77] http://biopython.org/
[78] https://pypi.python.org
[79] https://pip.pypa.io/en/latest/installing/#install-pip

Let's use `virtualenv` to create a virtual environment.

```
$ virtualenv env
```

Note that `env` is a directory containing all the required pieces for a working Python system. To make use of our virtual environment we need to activate it by sourcing the `env/bin/activate` script.

```
$ source ./env/bin/activate
```

This script basically mangles your `PATH` environment variable to ensure that virtualenv's Python is found first. We can find out which version of Python and `pip` is will be used by using the `which` command.

```
(env)$ which python
/home/olssont/env/bin/python
(env)$ which pip
/home/olssont/env/bin/pip
```

Note: The `./env/bin/activate` script also changed the look of our prompt prefixing it with the name of the virtual environment.

Now let us install `numpy` into our virtual environment.

```
(env)$ pip install numpy
```

To list installed packages you can use the `pip list` command.

```
(env)$ pip list
numpy (1.9.2)
pip (6.0.8)
setuptools (12.0.5)
```

When working on a Python project it can be useful to record the exact versions of the installed packages to make it easy to reproduce the setup at a later date. This is achieved using the `pip freeze` command.

```
(env)$ pip freeze
numpy==1.9.2
```

Let us save this information into a file named `requirements.txt`.

```
(env)$ pip freeze > requirements.txt
```

To show why this is useful let us deactivate the virtual environment.

```
(env)$ deactivate
$ which python
/usr/bin/python
```

Note: The `deactivate` command is created when you run the `./env/bin/activate` script.

Now let us create a new clean virtual environment, activate it and list its packages.

```
$ virtualenv env2
$ source ./env2/bin/activate
(env2)$ pip list
pip (6.0.8)
setuptools (12.0.5)
```

Now we can replicate the exact same setup found in our initial virtual environment, by running `pip install -r requirements.text`.

```
(env2)$ pip install -r requirements.txt
(env)$ pip list
numpy (1.9.2)
pip (6.0.8)
setuptools (12.0.5)
```

This feature allows you make your data analysis more reproducible!

14.7 Installing R packages

R is a scripting language with a strong focus on statistics and data visualisation.

There are many packages available for R. These are hosted on CRAN[80] (The Comprehensive R Archive Network).

To install an R package, for example `ggplot2`, we need to start an R session.

```
$ R

R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin14.5.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

Then one can use the built-in `install.packages` function. For exampel to install the `ggplot2` package one would use the command below.

```
> install.packages("ggplot2")
```

This will prompt you for the selection of a mirror to download the package from. Pick one close to you.

That's it, the `ggplot2` package is now available for you to use. However, you need to load it using the `library` function to use it.

```
> library(ggplot2)
```

[80] https://cran.r-project.org/

14.8 Installing Perl modules

Perl is a scripting language popular in the bioinformatics community. You may therefore have to work with it.

There are a vast number of Perl modules available. These are hosted on CPAN[81] (Comprehensive Perl Archive Network).

Traditionally, CPAN hosted packages are installed using the cpan command. However, this can be quite cumbersome as it asks the user a lot of questions with regards to how things should be configured. This resulted in people developing a simpler tool to install Perl modules: cpanm (CPAN Minus). You may be able to install cpanm using your distributions package manager, if not you can install it using cpan.

```
$ cpan App::cpanminus
```

When you run the command above you will notice that cpan prompts you for a lot of information, accepting the defaults is fine. When it prompts you to select an approach:

```
What approach do you want?  (Choose 'local::lib', 'sudo' or 'manual')
```

choose sudo. This will install cpanm into a location that is immediately available in your PATH.

Now that you have installed cpanm you can use it to install Perl modules more easily. For example to install the Bio::Tools::GFF module you can simply use the command below.

```
$ cpanm Bio::Tools::GFF
```

14.9 Installing Latex packages

TeX is a collection of programs and packages that allow you to typeset documents. LaTeX is a number of macros built on top of TeX. In Collaborating on projects (page 83) we used Latex for producing a PDF version of the document.

Confusingly there are many different distributions of TeX, for example the dominant distribution of TeX on Windows' is MiKTeX[82]. On Unix based systems the most commonly used TeX distribution is TeX Live[83]. And on Mac OSX it is MacTeX[84].

In terms of package management Tex Live has got three different concepts: packages, collections and schemes. A collection is a set of packages and a scheme is a group of collections and packages. Scheme's can only be selected during the initial install of TeX Live, whereas packages can be installed at any point.

One option is to use the scheme-full, which includes everything meaning that you are unlikely to need to install anything else. However, this can take a long time and take up quite a lot of space on your system.

Another option is to start with a smaller scheme, for example scheme-basic, scheme-minimal and scheme-small. Other packages and collections can then be installed as required.

Once you have install TeX Live you can manage it using the TeX Live Package Manager (tlmgr).

To search for a package you can use the tlmgr search command.

```
$ tlmgr search fontsrecommended
collection-fontsrecommended - Recommended fonts
```

[81] http://www.cpan.org/index.html
[82] http://miktex.org/
[83] https://www.tug.org/texlive/
[84] https://www.tug.org/mactex/

To install a package/collection.

```
$ sudo tlmgr install collection-fontsrecommended
```

14.10 Automated provisioning

As this chapter highlights managing software installations can be onerous and tedious. What makes matters worse is that after you have installed a piece of software it can be very easy to forget how one did it. So when you get a new computer you may find yourself spending quite sometime configuring it so that all your analysis pipelines work as expected.

Spending time configuring your system may be acceptable if you are the only person depending on it. However, if other people depend on the machine it is not. For example, you may end up responsible for a scientific web-service. In these instances you should look into automating the configuration of your system.

Describing how to do this is beyond the scope of this book. However, if you are interested I highly recommend using Ansible[85]. To get an idea of how Ansible works I suggest having a look at some of the blog posts on my website, for example How to create automated and reproducible work flows for installing scientific software[86].

14.11 Key concepts

- The file system is structured like a tree staring from the root `/`
- Programs are commonly located in `/bin`, `/usr/bin`, `/usr/local/bin`
- The `PATH` environment variable defines where your shell looks for programs
- You need superuser privileges to write to many of default program locations
- The `sudo` command allows you to run another command with superuser privileges if you are in the sudoers list
- Package managers allow you to easily search for and install software packages
- Some software such as Python, R and Perl have their own built-in package managers
- It can be easy to forget how you how you configured your machine, do make notes
- Once you start finding it tedious making notes you should start thinking about automating the configuration of your system using a tool such as Ansible

[85] https://www.ansible.com/
[86] http://tjelvarolsson.com/blog/how-to-create-automated-and-reproducible-work-flows-for-installing-scientific-software/

CHAPTER 15

Next steps

You've got to the end! Well done! Have a cup of tea and a biscuit.

This book has covered a lot of ground to give you a broad understanding of computing and practises to make your life easier. I hope it will stand you in good stead in the future.

This chapters gives some suggestions on how to continue exploring.

15.1 Apply what you have learnt

The only way to learn is to keep practising. Try to think about how to apply what you have been learning. You will run into problems, working your way through these problems is a great way to solidify your knowledge.

Think about your day-to-day work and your research. How could they be improved by scripting and computing? Try to solve problems that are outside of your comfort zone. This way you will extend your expertise.

15.2 Be social

Find friends and colleges that are in the same boat as you and talk to them about your computing experiences and the problems that you are trying to solve. It is much more fun to learn together and you will keep each other motivated.

Find someone more experienced than you and learn from them. Having a mentor to point you in the right direction can be a great time saver.

Connect with me on Twitter, my personal account is @tjelvar_olsson[87]. I've also set up a more specific account relating to this book @bioguide2comp[88]. Alternatively you can send me an email[89], or message me via the Facebook page[90].

Let me know what you think of this book. I would really value your feedback.

Also, if you liked the book please spread the word!

If you want to receive updates about this book do sign up to the mailing list. You can find the sign up form on the website: biologistsguide2computing.com[91].

[87] https://twitter.com/tjelvar_olsson
[88] https://twitter.com/bioguide2comp
[89] tjelvar@biologistsguide2computing.com
[90] https://www.facebook.com/biologistsguide2computing
[91] http://biologistsguide2computing.com/

15.3 Recommended reading

Peter Norvig's essay Teach Yourself Programming in Ten Years[92] is really interesting. Norvig is the director of research at Google and really knows what he is talking about. The essay should help you put your learning into perspective and motivate you when things get tough.

Scott Granneman's Linux Phrasebook[93] provides a concise reference for learning the foundations of the most important Unix command line tools.

Hadley Wickaham's article Tidy Data[94] describes the concept of "Tidy Data" in detail along with background, motivation and case studies. It is an accessible read and will give you the tools required to structure your data so that it can be visualised directly by ggplot2.

Scott Chacon and Ben Straub's Pro Git[95] book provides a comprehensive guide to Git. I would recommend that you read chapter two of this book to solidify your understanding of Git. Many of the other chapters in this book are also really valuable, in particular chapters three and five.

John D. Blischak , Emily R. Davenport, Greg Wilson's article "A Quick Introduction to Version Control with Git and GitHub" is aimed at scientists wanting to start using version control. It has some great figures for explaining how Git works visually. DOI: 10.1371/journal.pcbi.1004668[96]

Mark Pilgrim's Dive Into Python 3[97] is a great book for learning the ins and outs of the Python programming language. Note that this is a new and improved version of Dive Into Python[98]. I recommend that you read Dive Into Python 3 as it is more recent and will teach you about the latest and greatest version of Python.

Zed A. Shaw's Learning Python the Hard Way[99] is another great book for learning Python. The book uses what Shaw refers to as Educational Mithridatism[100], where the use of repetition is applied to help you learn quicker.

Nicolas P. Rougier, Michael Droettboom, Philip E. Bourne's article "Ten Simple Rules for Better Figures" is a great resource of practical advice to help you create better scientific illustrations. DOI: 10.1371/journal.pcbi.1003833[101]

Geir Kjetil Sandve , Anton Nekrutenko, James Taylor, Eivind Hovig's article "Ten Simple Rules for Reproducible Computational Research" provides advice on how you can make your research more easily reproducible. DOI: 10.1371/journal.pcbi.1003285[102]

Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, Tracy K. Teal's article Good Enough Practices in Scientific Computing[103] provides a well informed and opinionated view of tools and practises that people performing scientific computing should adopt.

Andrew Hunt and David Thomas' The Pragmatic Programmer[104] is a modern classic on software development. This book will teach you programming best practices.

Finally, if you enjoyed this book you may also want to have a look at my blog, tjelvarolsson.com[105], where I post about programming and systems administration from a scientific computing point of view.

[92] http://norvig.com/21-days.html
[93] http://www.granneman.com/writing/books/linux-phrasebook/
[94] http://vita.had.co.nz/papers/tidy-data.pdf
[95] https://git-scm.com/book/en/v2
[96] https://dx.doi.org/10.1371/journal.pcbi.1004668
[97] http://www.diveintopython3.net/
[98] http://www.diveintopython.net/
[99] https://learnpythonthehardway.org/
[100] https://zedshaw.com/2015/09/14/educational-mithridatism/
[101] https://dx.doi.org/10.1371/journal.pcbi.1003833
[102] https://dx.doi.org/10.1371/journal.pcbi.1003285
[103] https://arxiv.org/abs/1609.00037
[104] https://pragprog.com/book/tpp/the-pragmatic-programmer
[105] http://tjelvarolsson.com/

CHAPTER 16

Glossary

Bit The smallest unit of information in computing, abbreviation of the term Binary digIT.

Blob Binary Large OBject, typically a large binary file stored in a database.

Bug Defect in a computer program which stops it from working as intended, see also *debug*.

Class In *object-oriented programming* a class is a template that can be used to create an *object*.

CPU Central Processing Unit, part of the computer that executes machine instructions.

Debug The activity of identifying and removing a *bug* (defect) from a computer program.

DNS Domain Name Server a machine that translates URLs to IP addresses.

DRY Don't repeat yourself. Make use of a variables for frequently used values. Make use of functions for frequently used processing logic.

Function A set of instructions for performing a procedure. Most commonly a function takes a set of arguments and returns a value.

HTTP HyperText Transfer Protocol, commonly used for web browsing.

Hypervisor A piece of software that allows you to run a *virtual machine*.

List A common data structure representing an ordered collection of elements. Also known as an array.

Method A method refers to a *function* that is part of a *class*.

Init Abbreviation of the word "initialise". Usually used to represent a one time event that that results in the creation of a new entity.

Integer A whole number, i.e. a number that can be expressed without a fractional component.

I/O Input/Output; often used in the context of reading and writing data to disk.

Object In *object-oriented programming* an object is an instance of a *class*.

Object-oriented programming Programming methodology centered around the concept of objects rather than actions and procedures.

IP address Internet Protocol address.

Pipe A pipe is a means to redirect the output from one command into another. The character used to represent a pipe is the vertical bar: `|`.

RAM Random Access Memory, fast access volatile memory.

Recursion A procedure whose implementation calls itself. Of practical use for problems where the solution depends on smaller instances of the same problem.

Scope Determines which parts of a program has access to what. For example any part of a program can have access to a global variable. However, if a variable is contained within the scope of a function only code within that function can access the variable.

Shell Program that allows the user to interact with the operating system's services and programs, see also *terminal*.

Standard input stream Stream of characters ingested by a program, often abbreviated as `stdin`. A common way to send data to a program is to *pipe* them from the *standard output stream* of another program. In the below we use the `echo` command to send the string `foo bar baz` to the standard input stream of the `wc` command.

```
$ echo "foo bar baz" | wc
       1       3      12
```

Standard error stream Stream of characters, representing error output, emitted by a program. Commonly viewed in the shell when running a command. Often abbreviated as `stderr`.

Standard Library A set of functionality that comes built-in with the core programming language.

Standard output stream Stream of characters emitted by a program. Commonly viewed in the shell when running a command. The standard output stream can be redirected using a *pipe*. Often abbreviated as `stdout`.

State All the information, to which a program has access, at a particular point in time.

String A list of characters used to represent text.

TDD See *test-driven development*

Terminal Application for accessing a shell, see also *shell*.

Test-driven development Methodology used in software development that makes use of rapid iterations of development cycles. The development cycle includes three steps:

1. Write a test
2. Write minimal code to make the test pass
3. Refactor the code if necessary

URL Unique Resource Location

Virtual machine An operating system running within a *hypervisor* rather than on the physical computer.

VM See *virtual machine*.