Unix and Perl Primer for Biologists

Keith Bradnam & Ian Korf

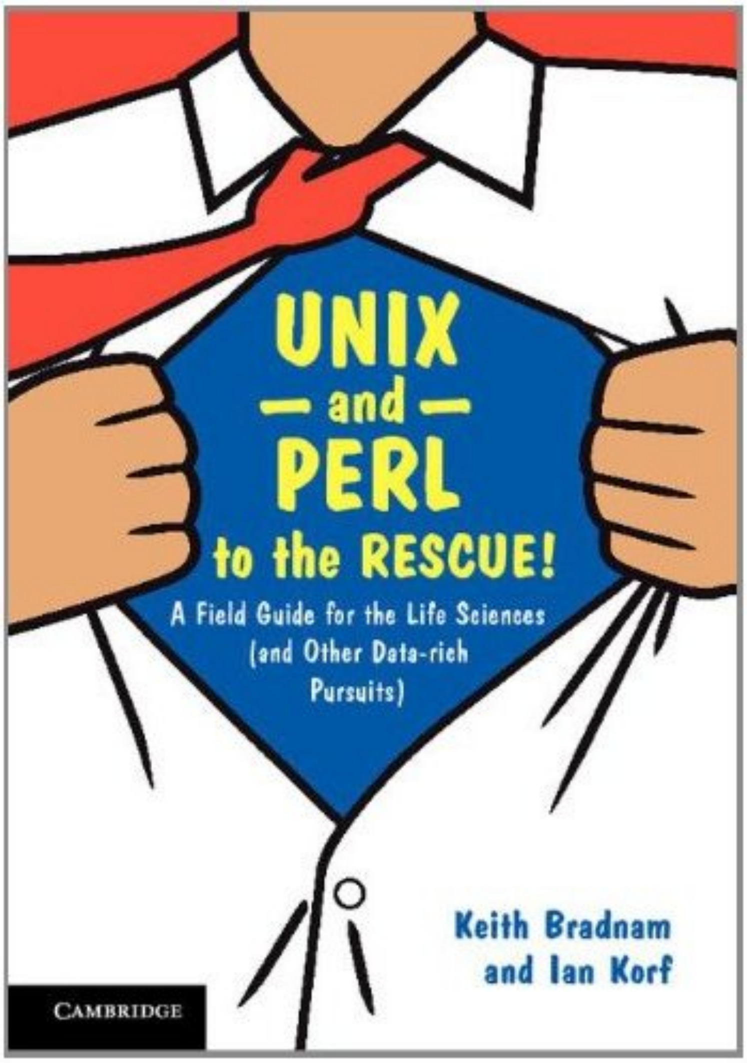
Version 3.1.2 — October 2016

Contents

Shameless Plug 1

This course has been greatly extended and reworked into a book that has been published by Cambridge University Press. It is available to order on Amazon.com and at many other online stores. It is also available in various ebook formats.

Unix and Perl to the Rescue! A field guide for the life sciences (and other data-rich pursuits)


UNIX
— and —
PERL
to the RESCUE!
A Field Guide for the Life Sciences (and Other Data-rich Pursuits)
Keith Bradnam and Ian Korf
CAMBRIDGE

Unix and Perl to the Rescue!

This primer will remain freely available, though we of course hope that if you find the primer useful, you will consider taking a look at our book. In the book we greatly expand on every subject that is in the primer, as well as covering many more topics. Some of these extra topics include more coverage of Unix and Perl, but we also devote sections to areas such as 'Data Management', 'Revision Control', and 'Code Beautification'. There are also many more jokes and geeky cultural references.

We have also created a website at http://rescuedbycode.com/ to support both the primer and the book, and should there ever be a movie adaptation of the book (starring Tom Cruise as 'grep'?) I expect that you'll be able to find out about that on the website as well.

Enjoy!

Keith Bradnam & Ian Korf May 2012

Shameless Plug 2

We are slowly — make that *very slowly* — in the process of writing a new book which will hopefully explain all the fun that can be had when using *Python* rather than Perl as your scriping language of choice for bioscience work.

We've been posting some blog posts at http://rescuedbycode.com to chronicle some of our thoughts as we write this book; most notably on the differences between Perl and Python but also from issues arising in the differences in Python 2 and Python 3. You can find all such blog posts listed here in the post Lessons learned while writing a book about programming.

Introduction

Advances in high-throughput biology have transformed modern biology into an incredibly data-rich science. Biologists who never thought they needed computer programming skills are now finding that using an Excel spreadsheet is simply not enough. Learning to program a computer can be a daunting task, but it is also incredibly worthwhile. You will not only improve your research, you will also open your mind to new ways of thinking and have a lot of fun.

This course is designed for Biologists who want to learn how to program but never got around to it. Programming, like language or math, comes more naturally to some than others. But we all learn to read, write, add, subtract, etc., and we can all learn to program. Programming, more than just about any other skill, comes in waves of understanding. You will get stuck for a while and a little frustrated, but then suddenly you will see how a new concept aggregates a lot of seemingly disconnected information. And then you will embrace the new way, and never imagine going back to the old way.

As you are learning, if you are getting confused and discouraged, slow down and ask questions. You can contact us either in person, by email, or (preferably) on the associated [Unix and Perl for Biologists Google Group][Google group] The lessons build on each other, so do not skip ahead thinking you will return to the confusing concept at a later date.

Why Unix?

The Unix operating system has been around since 1969. Back then there was no such thing as a graphical user interface. You typed everything. It may seem archaic to use a keyboard to issue commands today, but it's much easier to automate keyboard tasks than mouse tasks. There are several variants of Unix (including Linux), though the differences do not matter much. Though you may not have noticed it, Apple has been using Unix as the underlying operating system on all of their computers since 2001.

Increasingly, the raw output of biological research exists as *in silico* data, usually in the form of large text files. Unix is particularly suited to working with such files and has several powerful (and flexible) commands that can process your data for you. The real strength of learning Unix is that most of these commands can be combined in an almost unlimited fashion. So if you can learn just five Unix commands, you will be able to do a lot more than just five things.

Why Perl?

Perl is one of the most popular Unix programming languages. It doesn't matter much which language you learn first because once you know how one works, it is much easier to learn others. Among languages, there is often a distinction between interpreted (e.g. Perl, Python, Ruby) and compiled (e.g. C, C++, Java) languages. People often call interpreted programs scripts. It is generally easier to learn programming in a scripting language because you don't have to worry as much about variable types and memory allocation. The downside is the interpreted programs often run much slower than compiled ones (100-fold is common). But let's not get lost in petty details. Scripts are programs, scripting is programming, and computers can solve problems quickly regardless of the language.

Typeset Conventions

All of the Unix and Perl code in these guides is written in constant-width font with line numbering. Here is an example with 3 lines:

```
1. for ($i = 0; $i < 10; $i++) {
2.     print $i, "\n";
3. }
```

Text you are meant to type into a terminal is indented in constant-width font without line numbering. Here is an example:

```
ls -lrh
```

Sometimes a paragraph will include a reference to a Unix command, Perl function, or a file that you should be working with, Any such text will be in a constant-width, boxed font. E.g.

Type the `pwd` command again.

From time to time this documentation will contain web links to pages that will help you find out more about certain Unix commands and Perl functions. Usually, the *first* mention of a command or function will be a hyperlink to Wikipedia (for Unix commands) or to http://perldoc.perl.org (for Perl functions). Important or critical points will be styled like so:

> ***This is an important point!***

About the authors

Keith Bradnam started out his academic career studying ecology. This involved lots of field trips and and throwing quadrats around on windy hillsides. He was then lucky to be in the right place at the right time to do a Masters degree in Bioinformatics (at a time when nobody was very sure what bioinformatics was). From that point onwards he has spent most of his waking life sat a keyboard (often staring into a Unix terminal). A PhD studying eukaryotic genome evolution followed; this was made easier by the fact that only one genome had been completed at the time he started (this soon changed). After a brief stint working on an Arabidopsis genome database, he moved to working on the excellent model organism database WormBase at the Wellcome Trust Sanger Institute. It was here that he first met Ian Korf and they bonded over a shared love of Macs, neatly written code, and English puddings. Ian then tried to run away and hide in California at the UC Davis Genome Center but Keith tracked him down and joined his lab. Apart from doing research, he also gets to look after all the computers in the lab and teach the occasional class or two. However, he would give it all up for the chance to be able to consistently beat Ian at foosball, but that seems unlikely to happen anytime soon. Keith still likes Macs and neatly written code, but now has a much harder job finding English puddings.

Ian Korf believes that you can tell what a person will do with their life by examining their passions as a teen. Although he had no idea what a ‘sequence analysis algorithm’ was at 16, a deep curiosity about biological mechanisms and an obsession with writing/playing computer games is only a few bits away. Ian’s first experience with bioinformatics came as a post-doc at Washington University (St. Louis) where he was a member of the Human Genome Project. He then went across the pond to the Sanger Centre for another post-doc. There he met Keith

Bradnam, and found someone who truly understood the role of communication and presentation in science. Ian was somehow able to persuade Keith to join his new lab in Davis California, and this primer on Unix and Perl is but one of their hopefully useful contributions.

Preamble

What computers can run Perl?

One of the main goals of this course is to learn Perl. As a programming language, Perl is platform agnostic. You can write (and run) Perl scripts on just about any computer. We will assume that >99% of the people who are reading this use either a Microsoft Windows PC, an Apple Mac, or one of the many Linux distributions that are available (Linux can be considered as a type of Unix, though this claim might offend the Linux purists reading this). A small proportion of you may be using some other type of dedicated Unix platform, such as Sun or SGI. For the Perl examples, none of this matters. All of the Perl scripts in this course should work on any machine that you can install Perl on (if an example doesn't work then please let us know!).

What computers can run Unix?

Unlike our Perl documentation, the Unix part of this course is not quite so portable to other types of computer. We decided that this course should include an introduction to Unix because most bioinformatics happens on Unix/Linux platforms; so it makes sense to learn how to run your Perl scripts in the context of a Unix operating system. If you read the Introduction, then you will know that all modern Mac computers are in fact Unix machines. This makes teaching Perl & Unix on a Mac a relatively straightforward proposition, though we are aware that this does not help those of you who use Windows. This is something that we will try to specifically address in later updates to this course. For now, we would like to point out that you can achieve a Unix-like environment on your Windows PC in one of two ways:

1. Install Cygwin — this provides a Linux-like environment on your PC, it is also free to download. There are some differences between Cygwin and other types of Unix which may mean that not every Unix example in this course works exactly as described, but overall it should be sufficient for you to learn the basics of Unix.
2. Install Linux by using virtualization software — there are many pieces of software that will now allow you effectively install one operating system within another operating system. Microsoft has it's own (free) Virtual PC software, and here are some guidelines for using Ubuntu Linux with various virtualization software tools.

You should also be aware that there is a lot of variation within the world of Unix/Linux. Most commands will be the same, but the layout of the file system may look a little different. Hopefully our documentation should work for most types of Unix, but bear in mind it was written (and tested) with Apple's version of Unix.

Do I need to run this course from a USB drive?

We originally developed this course to be taught in a computer classroom environment. Because of this we decided to put the entire course (documentation & data) on to a USB flash drive. One reason for doing this was so that people could take the flash drive home with them and continue working on their own computers.

If you have your own computer which is capable of running a Unix/Linux environment then you might prefer to use that, rather than using a flash drive. If you have downloaded the course material, then after unpacking it you should have a directory called 'Unix_and_Perl_course'. You can either copy this directory (about 100 MB in size at the time of writing) to a flash drive or to any other directory within your Unix environment. Instructions in this document will assume that you are working on a flash drive on a Mac computer, so many of the Unix examples will not work exactly as written on other systems. In most cases you will just need to change the name of any directories the are used in the examples.

In our examples, we assume that the course material is located on a flash drive that is named ‘USB’. If you run the course from your own flash-drive, you might find it easier to rename it to ‘USB’ as well, though you don’t have to do this.

Part 1: Unix - Learning the essentials

Introduction to Unix

These exercises will (hopefully) teach you to become comfortable when working in the environment of the Unix terminal. Unix contains many hundred of commands but you will probably use just 10 or so to achieve most of what you want to do.

You are probably used to working with programs like the Apple Finder or the Windows File Explorer to navigate around the hard drive of your computer. Some people are so used to using the mouse to move files, drag files to trash etc. that it can seem strange switching from this behavior to typing commands instead. Be patient, and try — as much as possible — to stay within world of the Unix terminal. Please make sure you complete and understand each task before moving on to the next one.

First steps

The lessons from this point onwards will assume the following:

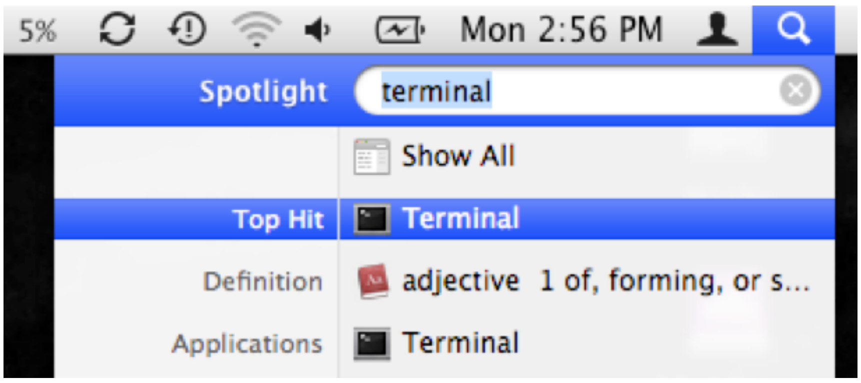
1. You have downloaded the Unix and Perl course material and copied it to a USB flash drive .
2. The flash drive has been renamed to ‘USB’.
3. You have removed the downloaded files from your Desktop/Downloads folder (this is often the source of confusion when you have one copy on your USB drive and a separate copy on your Desktop) .

U1. The Terminal

A ‘terminal’ is the common name for the program that does two main things. It allows you to type input to the computer (i.e. run programs, move/view files etc.) and it allows you to see output from those programs. All Unix machines will have a terminal program and on Apple computers, the terminal application is unsurprisingly named ‘Terminal’.

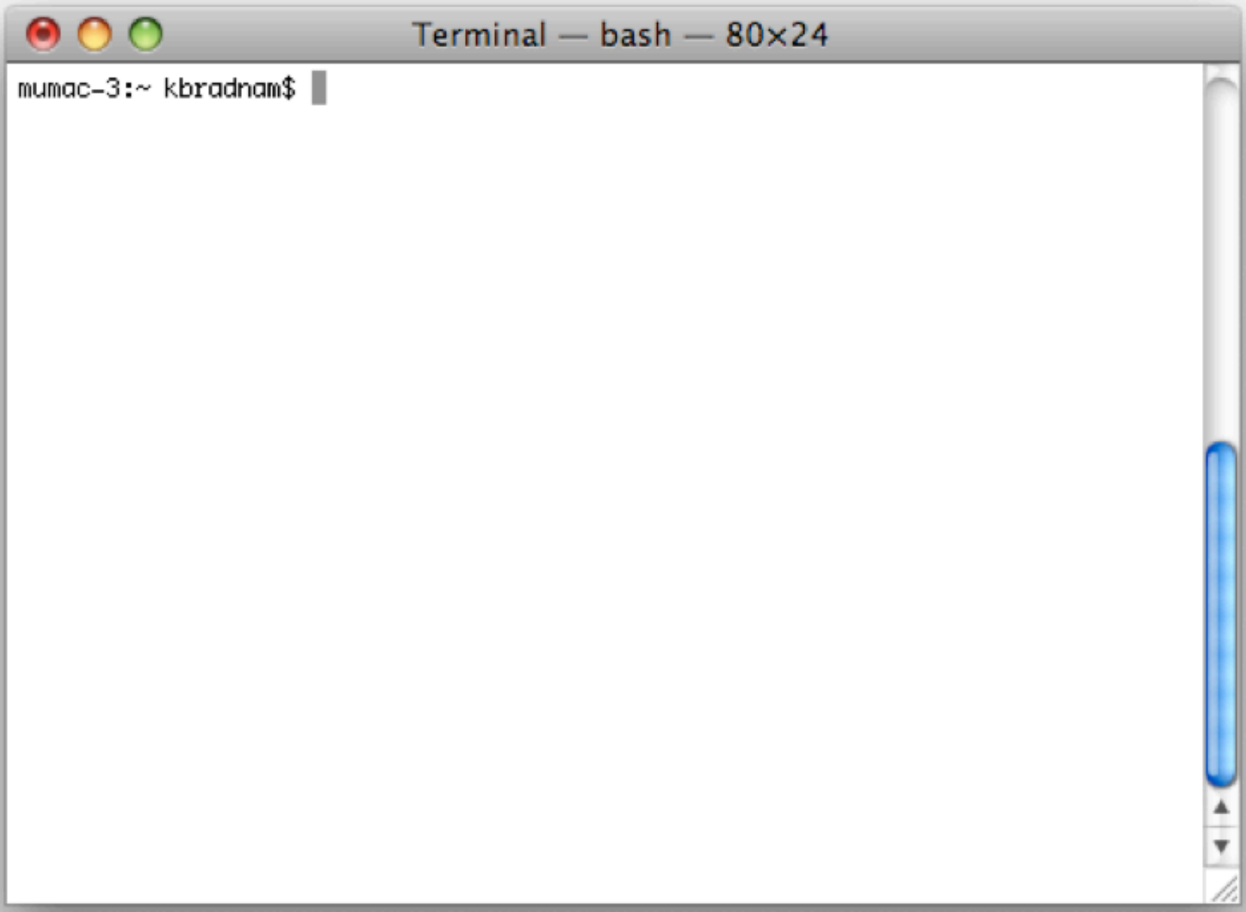
Task U1.1

Use the ‘Spotlight’ search tool (the little magnifying glass in the top right of the menu bar) to find, and then launch, Apple’s Terminal application:



Spotlight

You should now see something that looks like the following (any text that appears inside your terminal window will look different):



```
mumac-3:~ kbradnam$
```

(http://korflab.ucdavis.edu/Unix_and_Perl/terminal.png)

Before we go any further, you should note that you can:

- make the text larger/smaller (hold down ‘command’ and either ‘+’ or ‘–’)
- resize the window (this will often be necessary)
- have multiple terminal windows on screen (see the ‘Shell’ menu)
- have multiple tabs open within each window (again see the ‘Shell’ menu)

There will be many situations where it will be useful to have multiple terminals open and it will be a matter of preference as to whether you want to have multiple windows, or one window with multiple tabs (there are keyboard shortcuts for switching between windows, or moving between tabs).

U2. Your first Unix command

Unix keeps files arranged in a hierarchical structure. From the ‘top-level’ of the computer, there will be a number of directories, each of which can contain files and subdirectories, and each of those in turn can of course contain more files and directories and so on, ad infinitum. It’s important to note that you will always be “in” a directory when using the terminal. The default behavior is that when you open a new terminal you start in your own ’home” directory (containing files and directories that only you can modify).

To see what files are in our home directory, we need to use the ls command. This command ‘lists’ the contents of a directory. So why don’t they call the command ‘list’ instead? Well, this is a good thing because typing long commands over and over again is tiring and time-consuming. There are many (frequently used) Unix commands that are just two or three letters. If we run the ls command we should see something like:

```
olson27-1:~ kbradnam$ ls
Application Shortcuts   Documents   Library
Desktop                 Downloads
olson27-1:~ kbradnam$
```

There are four things that you should note here:

1. You will probably see different output to what is shown here, it depends on your computer. Don’t worry about that for now.
2. The `olson27-1:~ kbradnam$` text that you see is the Unix command prompt. It contains a user name (kbradnam), the name of the machine that this user is working on (‘olson27–1’ and the name of the current directory (‘, more on that later). Note that the command prompt might not look the same on different Unix systems. In this case, the $ sign marks the end of the prompt.
3. The output of the `ls` command lists five things. In this case, they are all directories, but they could also be files. We’ll learn how to tell them apart later on.
4. After the `ls` command finishes it produces a new command prompt, ready for you to type your next command.

The `ls` command is used to list the contents of *any* directory, not necessarily the one that you are currently in. Plug in your USB drive, and type the following:

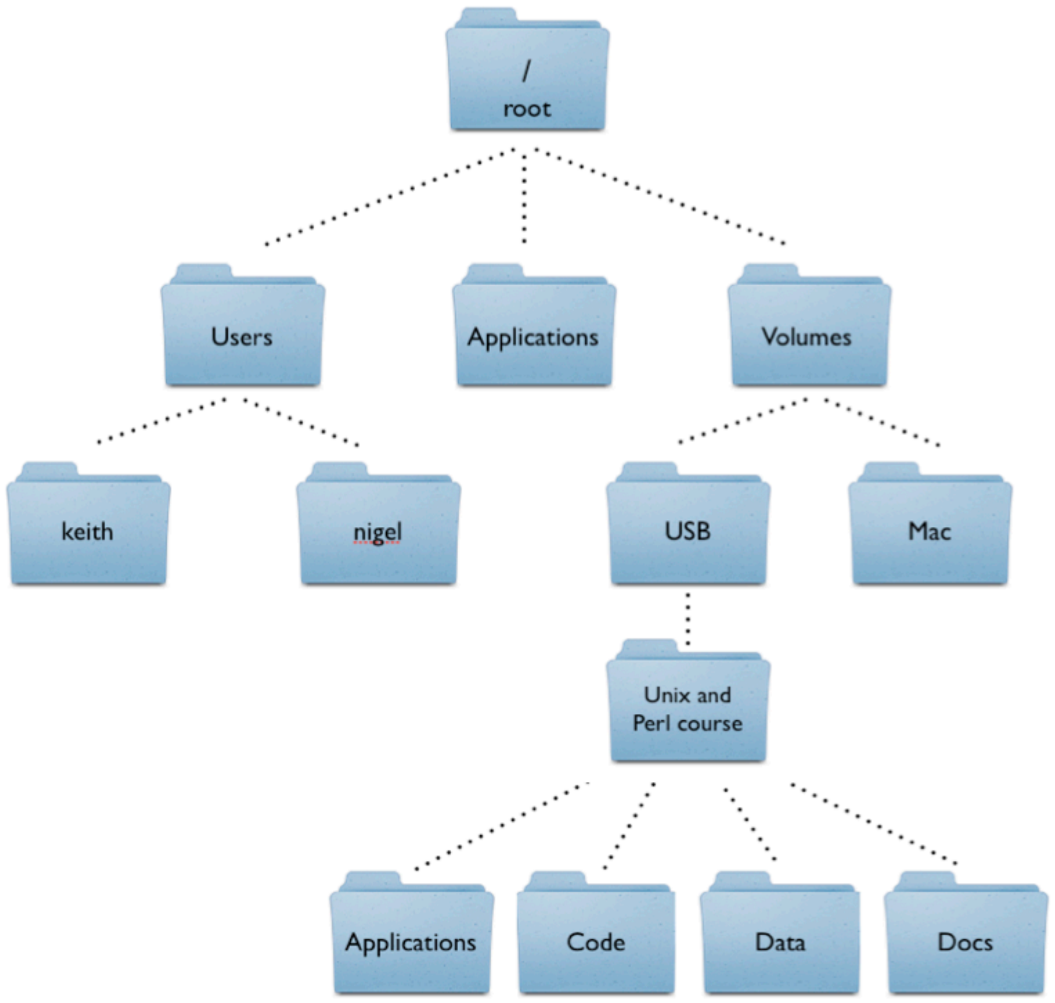
```
olson27-1:~ kbradnam$ ls /Volumes/USB/Unix_and_Perl_course
Applications    Code        Data        Documentation
```

On a Mac, plugged in drives appear as subdirectories in the special ‘Volumes’ directory. The name of the USB flash drive is ‘USB’. The above output shows a set of four directories that are all “inside” the ‘Unix_and_Perl_course’ directory). Note how the underscore character ‘_’ is used to space out words in the directory name.

U3: The Unix tree

Looking at directories from within a Unix terminal can often seem confusing. But bear in mind that these directories are exactly the same type of folders that you can see if you use Apple's graphical file-management program (known as 'The Finder'). A tree analogy is often used when describing computer filesystems. From the root level (/) there can be one or more top level directories, though most Macs will have about a dozen. In the example below, we show just three. When you log in to a computer you are working with your files in your home directory, and this will nearly always be inside a 'Users' directory. On many computers there will be multiple users.

All Macs have an applications directory where all the GUI (graphical user interface) programs are kept (e.g. iTunes, Microsoft Word, Terminal). Another directory that will be on all Macs is the Volumes directory. In addition to any attached *external* drives, the Volumes directory should also contain directories for every *internal* hard drive (of which there should be at least one, in this case it's simply called 'Mac'). It will help to think of this tree when we come to copying and moving files. E.g. if we had a file in the 'Code' directory and wanted to copy it to the 'keith' directory, we would have to go *up* four levels to the root level, and then *down* two levels.



Example directory structure

U4: Finding out where you are

There may be many hundreds of directories on any Unix machine, so how do you know which one you are in? The command
pwd will Print the Working Directory and that's pretty much all this command does:

```
olson27-1:~ kbradnam$ pwd
/users/clmuser
```

When you log in to a Unix computer, you are typically placed into your *home* directory. In this example, after we log in, we are placed in a directory called 'clmuser' which itself is a subdirectory of another directory called 'users'. Conversely, 'users' is the parent directory of 'clmuser'. The first forward slash that appears in a list of directory names always refers to the top level directory of the file system (known as the root directory). The remaining forward slash (between 'users' and 'clmuser') delimits the various parts of the directory hierarchy. If you ever get 'lost' in Unix, remember the `pwd` command.

As you learn Unix you will frequently type commands that don't seem to work. Most of the time this will be because you
are in the wrong directory, so it's a really good habit to get used to running the `pwd` command a lot.

U5: Getting from ‘A’ to ‘B’

We are in the home directory on the computer but we want to to work on the USB drive. To change directories in Unix, we
use the cd command:

```
olson27-1:~ kbradnam$ cd /Volumes/USB/Unix_and_Perl_course
olson27-1:USB kbradnam$ ls
Applications    Code        Data        Documentation
olson27-1:USB kbradnam$ pwd
/Volumes/USB/Unix_and_Perl_course
```

The first command reads as “change directory to the Unix_and_Perl_course directory that is inside a directory called ‘USB’, which itself is inside the Volumes directory that is at the root level of the computer”. Did you notice that the command prompt changed after you ran the `cd` command? The ‘~’ sign should have changed to ‘Unix_and_Perl_course’. This is a useful feature of the command prompt. By default it reminds you where you are as you move through different directories on the computer.

> ***NB. For the sake of clarity, we will now simplify the command prompt in all of the following examples***

U6: Root is the root of all evil

In the previous example, we could have achieved the same result in three separate steps:

```
$ cd /Volumes
$ cd USB
$ cd Unix_and_Perl_course
```

Note that the second and third commands do not include a forward slash. When you specify a directory that starts with a forward slash, you are referring to a directory that should exist one level below the root level of the computer. What happens if you try the following two commands? The first command should produce an error message.

```
$ cd Volumes
$ cd /Volumes
```

The error is because without including a leading slash, Unix is trying to change to a ‘Volumes’ directory below your current level in the file hierarchy (/Volumes/USB/Unix_and_Perl_course), and there is no directory called Volumes at this location.

U7: Up, up, and away

Frequently, you will find that you want to go ‘upwards’ one level in the directory hierarchy. Two dots `..` are used in Unix to refer to the *parent* directory of wherever you are. Every directory has a parent except the root level of the computer:

```
$ cd /Volumes/USB/Unix_and_Perl_course
$ pwd
/Volumes/USB/Unix_and_Perl_course
$ cd ..
$ pwd
/Volumes/USB
```

What if you wanted to navigate up *two* levels in the file system in one go? It’s very simple, just use two sets of the `..` operator, separated by a forward slash:

```
$ cd /Volumes/USB/Unix_and_Perl_course
$ pwd
/Volumes/USB/Unix_and_Perl_course
$ cd ../..
$ pwd
/Volumes
```

U8: I’m absolutely sure that this is all relative

Using `cd ..` allows us to change directory *relative* to where we are now. You can also always change to a directory based on its *absolute* location. E.g. if you are working in the `/Volumes/USB/Unix_and_Perl_course/Code` directory and you then want to change to the `/Volumes/USB/Unix_and_Perl_course/Data` directory, then you could do either of the following:

```
$ cd ../Data
```

or…

```
$ cd /Volumes/USB/Unix_and_Perl_course/Data
```

They both achieve the same thing, but the 2nd example requires that you know about the full *path* from the root level of the computer to your directory of interest (the ‘path’ is an important concept in Unix). Sometimes it is quicker to change directories using the relative path, and other times it will be quicker to use the absolute path.

U9: Time to go home

Remember that the command prompt shows you the name of the directory that you are currently in, and that when you are in your home directory it shows you a tilde character (~) instead? This is because Unix uses the tilde character as a short-hand way of specifying a home directory.

Task U9.1

See what happens when you try the following commands (use the `pwd` command after each one to confirm the results):

```
$ cd /
$ cd ~
$ cd /
$ cd
```

Hopefully, you should find that `cd` and `cd ~` do the same thing, i.e. they take you back to your home directory (from wherever you were). Also notice how you can specify the single forward slash to refer to the root directory of the computer. When working with Unix you will frequently want to jump straight back to your home directory, and typing `cd` is a very quick way to get there.

U10: Making the `ls` command more useful

The `..` operator that we saw earlier can also be used with the `ls` command. Can you see how the following command is listing the contents of the root directory? If you want to test this, try running `ls /` and see if the output is any different.

```
$ cd /Volumes/USB/Unix_and_Perl_course
$ ls ../../..
Applications    Volumes     net
CRC             bin         oldlogins
Developer       cores       private
Library         dev         sbin
Network         etc         tmp
Server          home        usr
System          mach_kernel var
Users           mach_kernel.ctfsys
```

The `ls` command (like most Unix commands) has a set of options that can be added to the command to change the results. Command-line options in Unix are specified by using a dash ('-') after the command name followed by various letters, numbers, or words. If you add the letter 'l' to the `ls` command it will give you a 'longer' output compared to the default:

```
$ ls -l /Volumes/USB/Unix_and_Perl_course
total 192
drwxrwxrwx  1 keith  staff  16384 Oct  3 09:03 Applications
drwxrwxrwx  1 keith  staff  16384 Oct  3 11:11 Code
drwxrwxrwx  1 keith  staff  16384 Oct  3 11:12 Data
drwxrwxrwx  1 keith  staff  16384 Oct  3 11:34 Documentation
```

For each file or directory we now see more information (including file ownership and modification times). The 'd' at the start of each line indicates that these are directories

Task U10.1

There are many, many different options for the ls command. Try out the following (against any directory of your choice) to see how the output changes.

```
ls -l
ls -R
ls -l -t -r
ls -lh
```

Note that the last example combine multiple options but only use one dash. This is a very common way of specifying multiple command-line options. You may be wondering what some of these options are doing. It's time to learn about Unix documentation….

U11: Man your battle stations!

If every Unix command has so many options, you might be wondering how you find out what they are and what they do. Well,
thankfully every Unix command has an associated ‘manual’ that you can access by using the `man` command. E.g.

```
$ man ls
$ man cd
$ man man # yes even the man command has a manual page
```

When you are using the man command, press `space` to scroll down a page, `b` to go back a page, or `q` to quit. You can also use the up and down arrows to scroll a line at a time. The man command is actually using another Unix program, a text viewer called `less`, which we’ll come to later on.

Some Unix commands have very long manual pages, which might seem very confusing. It is typical though to always list the command line options early on in the documentation, so you shouldn’t have to read too much in order to find out what a command-line option is doing.

U12: Make directories, not war

If we want to make a new directory (e.g. to store some work related data), we can use the mkdir command:

```
$ cd /Volumes/USB/Unix_and_Perl_course
$ mkdir Work
$ ls
Applications    Code        Data        Documentation   Work
$ mkdir Temp1
$ cd Temp1
$ mkdir Temp2
$ cd Temp2
$ pwd
/Volumes/USB/Unix_and_Perl_course/Temp1/Temp2
```

In the last example we created the two temp directories in two separate steps. If we had used the `-p` option of the `mkdir` command we could have done this in one step. E.g.

```
$ mkdir -p Temp1/Temp2
```

Task U12.1

Practice creating some directories and navigating between them using the `cd` command. Try changing directories using both the *absolute* as well as the *relative* path (see section U8).

U13: Time to tidy up

We now have a few (empty) directories that we should remove. To do this use the rmdir command, this will only remove empty directories so it is quite safe to use. If you want to know more about this command (or any Unix command), then remember that you can just look at its man page.

```
$ cd /Volumes/USB/Unix_and_Perl_course
$ rmdir Work
```

Task U13.1

Remove the remaining empty Temp directories that you have created

U14: The art of typing less to do more

Saving keystrokes may not seem important, but the longer that you spend typing in a terminal window, the happier you
will be if you can reduce the time you spend at the keyboard. Especially, as prolonged typing is not good for your body.
So the best Unix tip to learn early on is that you can tab complete the names of files and programs on most Unix systems. Type enough letters that uniquely identify the name of a file, directory or program and press tab…Unix will do the rest. E.g. if you type 'tou' and then press tab, Unix will autocomplete the word to touch (which we will learn more about in a minute). In this case, tab completion will occur because there are no other Unix commands that start
with 'tou'. If pressing tab doesn't do anything, then you have not have typed enough unique characters. In this case pressing tab *twice* will show you all possible completions. This trick can save you a LOT of typing…if you don't use tab-completion then you must be a masochist.

Task U14.1

Navigate to your home directory, and then use the `cd` command to change to the `/Volumes/USB/Unix_and_Perl_course/Code/` directory. Use tab completion for each directory name. This should only take 13 key strokes compared to 41 if you type the whole thing yourself.

Another great time-saver is that Unix stores a list of all the commands that you have typed in each login session. You can access this list by using the history command or more simply by using the up and down arrows to access anything from your history. So if you type a long command but make a mistake, press the up arrow and then you can use the left and right arrows to move the cursor in order to make a change.

U15: U *can* touch this

The following sections will deal with Unix commands that help us to work with files, i.e. copy files to/from places, move files, rename files, remove files, and most importantly, look at files. Remember, we want to be able to do all of these things without leaving the terminal. First, we need to have some files to play with. The Unix command touch will let us create a new, empty file. The touch command does other things too, but for now we just want a couple of files to work with.

```
$ cd /Volumes/USB/Unix_and_Perl_course
$ touch heaven.txt
$ touch earth.txt
$ ls
Applications Code   Data    Documentation   earth.txt     heaven.txt
```

U16: Moving heaven and earth

Now, let’s assume that we want to move these files to a new directory (‘Temp’). We will do this using the Unix mv (move) command:

```
$ mkdir Temp
$ mv heaven.txt Temp/
$ mv earth.txt Temp/
$ ls
Applications   Code     Data    Documentation   Temp
$ ls Temp/
earth.txt heaven.txt
```

For the `mv` command, we always have to specify a source file (or directory) that we want to move, and then specify a
target location. If we had wanted to we could have moved both files in one go by typing any of the following commands:

```
$ mv *.txt Temp/
$ mv *t Temp/
$ mv *ea* Temp/
```

The asterisk `*` acts as a wild-card character, essentially meaning ‘match anything’. The second example works because there are no other files or directories in the directory that end with the letters ‘t’ (if there was, then they would be copied too). Likewise, the third example works because only those two files contain the letters ‘ea’ in their names. Using wild-card characters can save you a lot of typing.

Task U16.1

Use `touch` to create three files called ‘fat’, ‘fit’, and ‘feet’ inside the Temp directory. I.e.

```
$ cd Temp
$ touch fat fit feet
```

Then type either `ls f?t` or `ls f*t` and see what happens. The ? character is also a wild-card but with a slightly different meaning. Try typing `ls f??t` as well.

U17: Renaming files

In the earlier example, the destination for the `mv` command was a directory name (Temp). So we moved a file from its source location to a target location (‘source’ and ‘target’ are important concepts for many Unix commands). But note that the target could have also been a (different) file name, rather than a directory. E.g. let’s make a new file and move it whilst renaming it at the same time:

```
$ touch rags
$ ls
Applications    Code        Data    Documentation   Temp  rags
$ mv rags Temp/riches
$ ls Temp/
earth.txt      heaven.txt     riches
```

In this example we create a new file (‘rags’) and move it to a new location and in the process change the name (to ‘riches’). So `mv` can rename a file as well as move it. The logical extension of this is using `mv` to rename a file without moving it (you have to use `mv` to do this as Unix does not have a separate ‘rename’ command):

```
$ mv Temp/riches Temp/rags
$ ls Temp/
earth.txt      heaven.txt     rags
```

U18: Stay on target

It is important to understand that as long as you have specified a ‘source’ and a ‘target’ location when you are moving a file, then it doesn’t matter what your current directory is. You can move or copy things within the same directory or between different directories regardless of whether you are “in” any of those directories. Moving directories is just like moving files:

```
$ mkdir Temp2
$ ls
Applications    Code        Data    Documentation   Temp  Temp2
$ mv Temp2 Temp/
$ ls Temp/
Temp2       earth.txt       heaven.txt  rags
```

This step moves the Temp2 directory inside the Temp directory.

Task U18.1

Create another Temp directory (Temp3) and then change directory to your home directory (/users/clmuser). **Without** changing directory, move the Temp3 directory to inside the /Volumes/USB/Unix_and_Perl_course/Temp directory.

U19: Here, there, and everywhere

The philosophy of ‘not having to be in a directory to do something in that directory’, extends to just about any operation that you might want to do in Unix. Just because we need to do something with file X, it doesn’t necessarily mean that we have to change directory to wherever file X is located. Let’s assume that we just want to quickly check what is in the Data directory before continuing work with whatever we were previously doing in /Volumes/USB/Unix_and_Perl_course. Which of the following looks more convenient:

```
$ cd Data
$ ls
Arabidopsis     C_elegans   GenBank     Misc        Unix_test_files
$ cd ..
```

or…

```
$ ls Data/
Arabidopsis     C_elegans   GenBank     Misc        Unix_test_files
```

In the first example, we change directories just to run the ls command, and then we change directories back to where we were again. The second example shows how we could have just stayed where we were.

U20: To slash or not to slash?

Task U20.1

Run the following two commands and compare the output

```
$ ls Documentation

$ ls Documentation/
```

The two examples are not quite identical, but they produce identical output. So does the trailing slash character in the second example matter? Well not really. In both cases we have a directory named ‘Documentation’ and it is optional as to whether you include the trailing slash. When you tab complete any Unix directory name, you will find that a trailing slash character is automatically added for you. This becomes useful when that directory contains subdirectories which you also want to tab complete.

I.e. imagine if you had to type the following (to access a buried directory ‘ggg’) and tab-completion *didn’t* add the trailing slash characters. You’d have to type the seven slashes yourself.

```
$ cd aaa/bbb/ccc/ddd/eee/fff/ggg/
```

U21: The most dangerous Unix command you will ever learn!

You've seen how to remove a directory with the `rmdir` command, but `rmdir` won't remove directories if they contain any files. So how can we remove the files we have created (in /Volumes/USB/Unix_and_Perl_course/Temp)? In order to do this, we will have to use the rm (remove) command.

> ***Please read the next section VERY carefully. Misuse of the rm command can lead to needless death & destruction***

Potentially, `rm` is a very dangerous command; if you delete something with `rm`, you will not get it back! It does not go into the trash or recycle can, it is permanently removed. It is possible to delete everything in your home directory (all directories and subdirectories) with `rm`, that is why it is such a dangerous command.

Let me repeat that last part again. It is possible to delete EVERY file you have ever created with the `rm` command. Are you scared yet? You should be. Luckily there is a way of making `rm` a little bit safer. We can use it with the `-i` command-line option which will ask for confirmation before deleting anything:

```
$ pwd
/Volumes/USB/Unix_and_Perl_course/Temp
$ ls
Temp2       Temp3       earth.txt   heaven.txt  rags
$ rm -i earth.txt
remove earth.txt? y
$ rm -i heaven.txt
remove heaven.txt? y
```

We could have simplified this step by using a wild-card (e.g. `rm -i *.txt`).

Task U21.1

Remove the last file in the Temp directory ('rags') and then remove the two empty directories (Temp 2 & Temp3).

U22: Go forth and multiply

Copying files with the cp (copy) command is very similar to moving them. Remember to always specify a source and a target location. Let’s create a new file and make a copy of it.

```
$ touch file1
$ cp file1 file2
$ ls
file1   file2
```

What if we wanted to copy files from a different directory to our current directory? Let’s put a file in our home directory (specified by ‘ remember) and copy it to the USB drive:

```
$ touch ~/file3
$ ls
file1   file2
$ cp ~/file3 .
$ ls file1 file2 file3
```

This last step introduces another new concept. In Unix, the current directory can be represented by a ‘.’ (dot) character. You will mostly use this only for copying files to the current directory that you are in. But just to make a quick point, compare the following:

```
$ ls
$ ls .
$ ls ./
```

In this case, using the dot is somewhat pointless because `ls` will already list the contents of the current directory by default. Also note again how the trailing slash is optional.

Let’s try the opposite situation and copy these files back to the home directory (even though one of them is already there). The default behavior of copy is to overwrite (without warning) files that have the same name, so be careful.

```
$ cp file* ~/
```

Based on what we have already covered, do you think the trailing slash in ‘~/’ is necessary?

U23: Going deeper and deeper

The `cp` command also allows us (with the use of a command-line option) to copy entire directories (also note how the `ls` command in this example is used to specify multiple directories):

```
$ mkdir Storage
$ mv file* Storage/
$ ls
Storage
$ cp -R Storage Storage2
$ ls Storage Storage2
Storage:
file1   file2   file3

Storage2:
file1   file2   file3
```

Task U23.1

The `-R` option means 'copy recursively', many other Unix commands also have a similar option. See what happens if you don't include the `-R` option. We've finished with all of these temporary files now. Make sure you remove the Temp directory and its contents (remember to always use `rm -i`).

U24: When things go wrong

At this point in the course, you may have tried typing some of these commands and have found that things did not work as expected. Some people will then assume that the computer doesn't like them and that it is being deliberately mischievous. The more likely explanation is that you made a typing error. Maybe you have seen one the following error messages:

```
$ ls Codee
ls: Codee: No such file or directory

$ cp Data/Unix_test_files/* Docmentation
usage: cp [-R [-H | -L | -P]] [-fi | -n] [-pvX] source_file target_file
       cp [-R [-H | -L | -P]] [-fi | -n] [-pvX] source_file ... target_directory
```

In both cases, we included a deliberate typo when specifying the name of the directories. With the `ls` command, we get a fairly useful error message. With the `cp` command we get a more cryptic message that reveals the correct usage statement for this command. In general, if a command fails, check your current directory (`pwd`) and check that all the files or directories that you mention actually exist (and are in the right place). Many errors occur because people are not in the right directory!

U25: Less is more

So far we have covered listing the contents of directories and moving/copying/deleting either files and/or directories. Now we will quickly cover how you can look at files; in Unix the less command lets you view (but not edit) text files. Let's take a look at a file of *Arabidopsis thaliana* protein sequences:

```
$ less Data/Arabidopsis/At_proteins.fasta
```

When you are using less, you can bring up a page of help commands by pressing `h`, scroll forward a page by pressing `space`, or go forward or backwards one line at a time by pressing `j` or `k`. To exit less, press `q` (for quit). The `less` program also does about a million other useful things (including text searching).

U26: Directory enquiries

When you have a directory containing a mixture of files and directories, it is not often clear which is which. One solution is to use `ls -l` which will put a ‘d’ at the start of each line of output for items which are directories. A better solution is to use `ls -p`. This command simply adds a trailing slash character to those items which are directories. Compare the following:

```
$ ls
Applications    Data    file1    Code    Documentation    file2

$ ls -p
Applications/   Data/   file1    Code/   Documentation/   file2
```

Hopefully, you’ll agree that the second example makes things a little clearer. You can also do things like always capitalizing directory names (like I have done) but ideally we would suggest that you always use `ls -p`. If this sounds a bit of a pain, then it is. Ideally you want to be able to make `ls -p` the default behavior for `ls`. Luckily, there is a way of doing this by using Unix aliases. It’s very easy to create an alias:

```
$ alias ls='ls -p'
$ ls
Applications/   Data/   file1    Code/   Documentation/   file2
```

If you have trouble remembering what some of these very short Unix commands do, then aliases allow you to use human-readable alternatives. I.e. you could make a ‘copy’ alias for the cp command’ or even make ‘list_files_sorted_by_date’ perform the `ls -lt` command. Note that aliases do not replace the original command. It can be dangerous to use the name of an existing command as an alias for a different command. I.e. you could make an `rm` alias that put files to a ‘trash’ directory by using the `mv` command. This might work for you, but what if you start working on someone else’s machine who doesn’t have that alias? Or what if someone else starts working on your machine?

Task U26.1

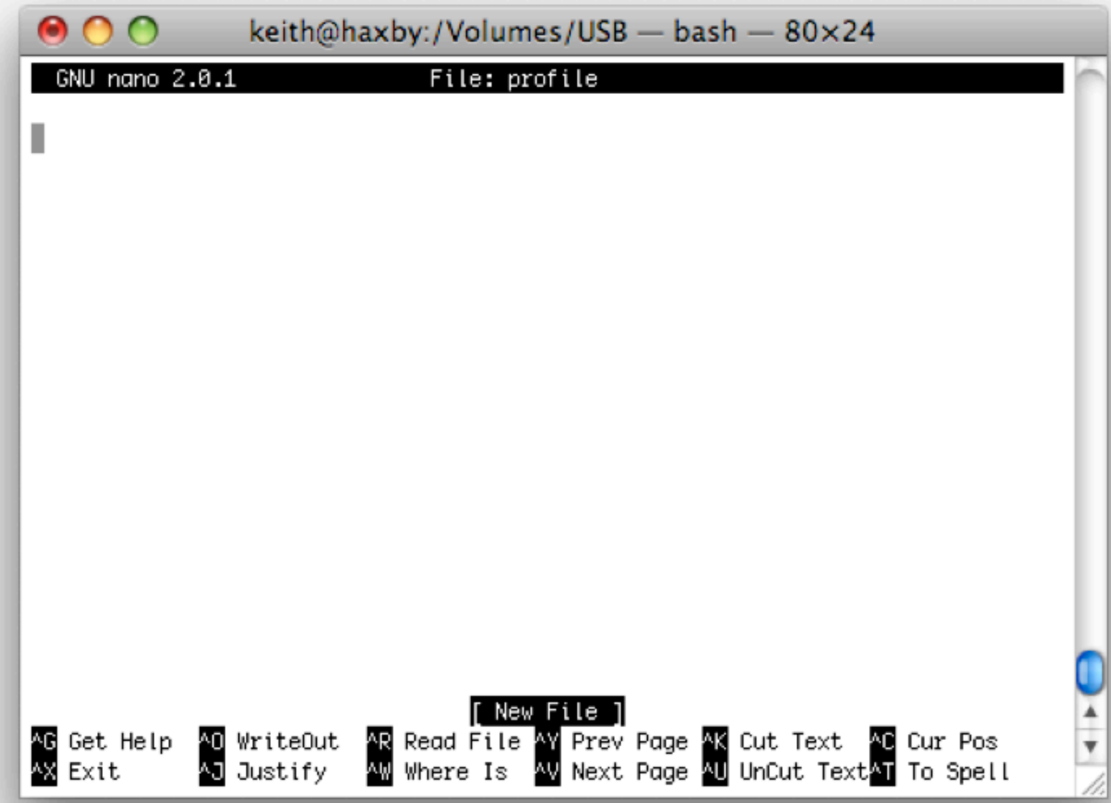
Create an alias such that typing `rm` will always invoke `rm -i`. Try running the alias command on its own to see what happens. Now open a new terminal window (or a new tab) and try running your `ls` alias. What happens?

U27: Fire the editor

The problem with aliases is that they only exist in the current terminal session. Once you log out, or use a new terminal window, then you'll have to retype the alias. Fortunately though, there is a way of storing settings like these. To do this, we need to be able to create a configuration file and this requires using a text editor. We could use a program like TextEdit to do this (or even Microsoft Word), but as this is a Unix course, we will use a simple Unix editor called [`][]. Let's create a file called profile:

```
$ cd /Volumes/USB/Unix_and_Perl_course
$ nano profile
```

You should see the following appear in your terminal:



```
keith@haxby:/Volumes/USB — bash — 80×24
  GNU nano 2.0.1                File: profile





                                   [ New File ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

the nano editor

The bottom of the nano window shows you a list of simple commands which are all accessible by typing 'Control' plus a letter. E.g. Control + X exits the program.

Task U27.1

Type the following text in the editor and then save it (Control + O). Nano will ask if you want to 'save the modified buffer' and then ask if you want to keep the same name. Then exit nano (Control + X) and use `less` to confirm that the profile file contains the text you added.

```
# some useful command line short-cuts
alias ls='ls -p'
alias rm='rm -i'
```

Now you have successfully created a configuration file (called ‘profile’) which contains two aliases. The first line that starts with a hash (#) is a comment, these are just notes that you can add to explain what the other lines are doing. But how do you get Unix to recognize the contents of this file? The source command tells Unix to read the contents of a file and treat it as a series of Unix commands (but it will ignore any comments).

Task U27.2

Open a new terminal window or tab (to ensure that any aliases will not work) and then type the following (make sure you first change to the correct directory):

```
$ source profile
```

Now try the `ls` command to see if the output looks different. Next, use `touch` to make a new file and then try deleting it with the `rm` command. Are the aliases working?

U28: Hidden treasure

In addition to adding aliases, profile files in Unix are very useful for many other reasons. We have actually already created a profile for you. It's in /Volumes/USB/Unix_and_Perl_course but you probably won't have seen it yet. That's because it is a hidden file named '.profile' (dot profile). If a filename starts with a dot, Unix will treat it as a hidden file. To see it, you can use `ls -a` which lists all hidden files (there may be several more files that appear).

Task U28.1

Use `less` to look at the profile file that we have created. See if you can understand what all the lines mean (any lines that start with a # are just comments). Use `source` to read this file. See how this changes the behavior of typing `cd` on its own. You can now delete the profile file that you made earlier, from now on we will use the .profile file.

If you have a .profile file in your *home* directory then it will be automatically read every time you open a new terminal. A problem for this class is your home directories are wiped each day, so we can't store files on the computer (which is why we are using the USB drive). So for this course we have to do a bit of extra work.

> ***Remember to type:***
> ***source /Volumes/USB/Unix_and_Perl_course/.profile***
> ***every time you use a new terminal window***

U29: Sticking to the script

Unix can also be used as a programming language just like Perl. Depending on what you want to do, a Unix script might solve all your problems and mean that you don't really need to learn Perl at all.

So how do you make a Unix script (which are commonly called 'shell scripts')? At the simplest level, we just write one or more Unix commands to a file and then treat that file as if it was any other Unix command or program.

Task U29.1

Copy the following two lines to a file (using `nano`). Name that file hello.sh (shell scripts are typically given a .sh extension) and **make sure that you save this file in /Volumes/USB/Unix_and_Perl_course/Code**.

```
# my first Unix shell script
echo "Hello World"
```

When you have done that, simply type 'hello.sh' and see what happens. If you have previously run `source .profile` then you should be able to run 'hello.sh' from any directory that you navigate to. If it worked, then it should have printed 'Hello world'. This very simple script uses the Unix command echo which just prints output to the screen. Also note the comment that precedes the `echo` command, it is a good habit to add explanatory comments.

Task U29.2

Try moving the script outside of the Code directory (maybe move it 'up' one level) and then `cd` to that directory. Now try running the script again. You should find that it doesn't work anymore. Now try running `./hello.sh` (that's a dot + slash at the beginning). It should work again.

U30: Keep to the $PATH

The reason why the script worked when it was in the Code directory and then stopped working when you moved it is because we did something to make the Code directory a bit special. Remember this line that is in your .profile file?

```
PATH=$PATH":$HOME/Code"
```

When you try running *any* program in Unix, your computer will look in a set of predetermined places to see if a program by that name lives there. All Unix commands are just files that live in directories somewhere on your computer. Unix uses something called $PATH (which is an *environment variable*) to store a list of places to look for programs to run. In our .profile file we have just told Unix to also look in your Code directory. If we didn't add the Code directory to the $PATH, then we have to run the program by first typing ./ (dot slash). Remember that the dot means the current directory. Think of it as a way of forcing Unix to run a program (including Perl scripts).

U31: Ask for permission

Programs in Unix need permission to be run. We will normally always have to type the following for any script that we create:

```
$ chmod u+x hello.sh
```

This would use the chmod to add *executable* permissions (+x) to the file called ‘hello.sh’ (the ‘u’ means add this permission to just you, the user). Without it, your script won’t run. Except that it did. One of the oddities of using the USB drive for this course, is that files copied to a USB drive have all permissions turned on by default. Just remember that you will normally need to run `chmod` on any script that you create. It’s probably a good habit to get into now.

The chmod command can also modify read and write permissions for files, and change any of the three sets of permissions (read, write, execute) at the level of ‘user’, ‘group’, and ‘other’. You probably won’t need to know any more about the chmod command other than you need to use it to make scripts executable.

U32: The power of shell scripts

Time to make some Unix shell scripts that might actually be useful.

Task U32.1

Look in the Data/Unix_test_files directory. You should see several files (all are empty) and four directories. Now put the following information into a shell script (using `nano`) and save it as cleanup.sh.

```
#!/bin/bash
mv *.txt Text
mv *.jpg Pictures
mv *.mp3 Music
mv *.fa Sequences
```

Make sure that this script is saved in your `Unix_and_Perl_course/Code directory`. Now return to the `Unix_and_Perl_course/Data/Unix_test_files` directory and run this script. It should place the relevant files in the correct directories. This is a relatively simple use of shell scripting. As you can see the script just contains regular Unix commands that you might type at the command prompt. But if you had to do this type of file sorting every day, and had many different types of file, then it would save you a lot of time.

Did you notice the #!/bin/bash line in this script? There are several different types of shell script in Unix, and this line makes it clearer that a) that this is actually a file that can be treated as a program and b) that it will be a bash script (bash is a type of Unix). As a general rule, all type of scriptable programming languages should have a similar line as the first line in the program.

Task U32.2

Here is another script. Copy this information into a file called change_file_extension.sh and again place that file in the Code directory.

```
#!/bin/bash

for filename in *.$1
do
    mv $filename ${filename%$1}$2
done
```

Now go to the `Data/Unix_test_files/Text` directory. If you have run the exercise from Task U32.1 then your text directory should now contain three files. Run the following command:

```
$ change_file_extension.sh txt text
```

Now run the `ls` command to see what has happened to the files in the directory. You should see that all the files that ended with 'txt' now end with 'text'. Try using this script to change the file extensions of other files.

It’s not essential that you understand exactly how this script works at the moment (things will become clearer as you learn Perl), but you should at least see how a relatively simple Unix shell script can be potentially very useful.

End of part 1.

You can now continue to learn a series of much more powerful Unix commands, or you can switch to Part 3 in order to start learning Perl. The choice is yours!

Part 2: Advanced Unix

How to Become a Unix power user

The commands that you have learnt so far are essential for doing any work in Unix but they don’t really let you do anything that is very useful. The following sections will introduce a few new commands that will start to show you how powerful Unix is.

U33: Match making

You will often want to search files to find lines that match a certain pattern. The Unix command grep does this (and much more). You might already know that FASTA files (used frequently in bioinformatics) have a simple format: one header line which must start with a '>' character, followed by a DNA or protein sequence on subsequent lines. To find only those header lines in a FASTA file, we can use grep, which just requires you specify a pattern to search for, and one or more files to search:

```
$ cd Data/Arabidopsis/
$ grep ">" intron_IME_data.fasta

>AT1G68260.1_i1_204_CDS
>AT1G68260.1_i2_457_CDS
>AT1G68260.1_i3_1286_CDS
>AT1G68260.1_i4_1464_CDS
.
.
.
```

This will produce lots of output which will flood past your screen. If you ever want to stop a program running in Unix, you can type Control+C (this sends an interrupt signal which should stop most Unix programs). The grep command has many different command-line options (type `man grep` to see them all), and one common option is to get `grep` to show lines that don't match your input pattern. You can do this with the -v option and in this example we are seeing just the sequence part of the FASTA file.

```
$ grep -v ">" intron_IME_data.fasta

GTATACACATCTCTCTACTTTCATATTTTGCATCTCTAACGAAATCGGATTCCGTCGTTG
TGAAATTGAGTTTTCGGATTCAGTGTTGTCGAGATTCTATATCTGATTCAGTGATCTAAT
GATTCTGATTGAAAATCTTCGCTATTGTACAG
GTTAGTTTTCAATGTTGCTGCTTCTGATTGTTGAAAGTGTTCATACATTTGTGAATTTAG
TTGATAAAATCTGAACTCTGCATGATCAAAGTTACTTCTTTACTTAGTTTGACAGGGACT
TTTTTTGTGAATGTGGTTGAGTAGAATTTAGGGCTTTGGATTAAATGTGACAAGATTTTG
.
.
.
```

U34: Your first ever Unix pipe

By now, you might be getting a bit fed up of waiting for the `grep` command to finish, or you might want a cleaner way of controlling things without having to reach for Ctrl-C. Ideally, you might want to look at the output from any command in a controlled manner, i.e. you might want to use a Unix program like less to view the output.

This is very easy to do in Unix, you can send the output from any command to any other Unix program (as long as the second program accepts input of some sort). We do this by using what is known as a pipe. This is implemented using the ‘|’ character (which is a character which always seems to be on different keys depending on the keyboard that you are using). Think of the pipe as simply connecting two Unix programs. In this next example we send the output from `grep` down a pipe to the less program. Let’s imagine that we just want to see lines in the input file which contain the pattern “ATGTGA” (a potential start and stop codon combined):

```
$ grep "ATGTGA" intron_IME_data.fasta | less

TTTTTTGTGAATGTGGTTGAGTAGAATTTAGGGCTTTGGATTAAATGTGACAAGATTTTG
CTGAATGTGACTGGAAGAATGAAATGTGTTAAGATCTTGTTCGTTAAGTTTAGAGTCTTG
GGTGGAATGAATTTATGTATCATGTGATAGCTGTTGCATTACAAGATGTAATTTTGCAAA
GTCTATGTGATGGCCATAGCCCATAGTGACTGATAGCTCCTTACTTTGTTTTTTTTTTCT
TTACTTGCAAAATTCCATGTGATTTTTTATATTACTTTGAAGAATTTTATAATATATTTT
TTGCATCAAGATATGTGACATCTTCAAAAAGATAACTTGTGAGAAGACAATTATAATATG
GTAACTTATTTATTGATTGAATCAGTAACTGTATTGTTATCATGATTTGTGAATATGTGA
AATCTTTGTGGTGGGTCTACGATATGAGCTGTCAATATATTTTTGTTTATACATGTGATC
GTATGTGAGCAAACGATGTCTCGTTTTCTCTCTCTCAATGATCAAGCACCTAACTTAAAT\
.
.
.
```

Notice that you still have control of your output as you are now in the `less` program. If you press the forward slash (/) key in `less`, you can then specify a search pattern. Type ATGTGA after the slash and press enter. The `less` program will highlight the location of these matches on each line. Note that `grep` matches patterns on a per line basis. So if one line ended ATG and the next line started TGA, then `grep` would not find it.

> ***Any time you run a Unix program or command that outputs a lot of text to the screen, you can instead pipe that output into the `less` program.***

U35: Heads and tails

Sometimes we do not want to use `less` to see *all* of the output from a command like grep. We might just want to see a few lines to get a feeling for what the output looks like, or just check that our program (or Unix command) is working properly. There are two useful Unix commands for doing this: head and tail. These commands show (by default) the first or last 10 lines of a file (though it is easy to specify more or fewer lines of output). So now, let's look for another pattern which might be in all the sequence files in the directory. If we didn't know whether the DNA/protein sequence in a FASTA files was in upper-case or lower-case letters, then we could use the `-i` option of `grep` which 'ignores' case when searching:

```
$ grep -i ACGTC * | head
At_proteins.fasta:TYRSPRCNSAVCSRAGSIACGTCFSPPRPGCSNNTCGAFPDNSITGWATSGEFALDVVSIQSTNGSNPGRFVKIPNLIFS
At_proteins.fasta:FRRYGHYISSDVFRRFKGSNGNFKESLTGYAKGMLSLYEAAHLGTTKDYILQEALSFTSSHLESLAACGTCPPHLSVHIQ
At_proteins.fasta:MAISKALIASLLISLLVLQLVQADVENSQKKNGYAKKIDCGSACVARCRLSRRPRLCHRACGTCCYRCNCVPPGTYGNYD
At_proteins.fasta:MAVFRVLLASLLISLLVLDFVHADMVTSNDAPKIDCNSRCQERCSLSSRPNLCHRACGTCCARCNCVAPGTSGNYDKCPC
chr1.fasta:TGTCTACTGATTTGATGTTTTCCTAAACTGTTGATTCGTTTCAGGTCAACCAATCACGTCAACGAAATTCAGGATCTTA
chr1.fasta:TATGCTGCAAGTACCAGTCAATTTTAGTATGGGAAACTATAAACATGTATAATCAACCAATGAACACGTCAATAACCTA
chr1.fasta:TTGAACAGCTTAGGGTGAAAATTATGATCCGTAGAGACAGCATTTAAAAGTTCCTTACGTCCACGTAAAATAATATATC
chr1.fasta:GGGATCACGAGTCTGTTGAGTTTTCCGACGTCGCTTGGTGTTACCACTTTGTCGAACATGTGTTCTTTCTCCGGAGGTG
chr1.fasta:CTGCAAAGGCCTACCTGTTTGTCCCTGTTACTGACAATACGTCTATGGAACCCATAAAAGGGATCAACTGGGAATTGGT
chr1.fasta:ACGTCGAAGGGGGTAAGATTGCAGCTAATCATTTGATGAAATGGATTGGGATTCACGTGGAGGATGATCCTGATGAAGT
```

The `*` character acts as a wildcard meaning 'search all files in the current directory' and the `head` command restricts the total amount of output to 10 lines. Notice that the output also includes the name of the file containing the matching pattern. In this case, the `grep` command finds the ACGTC pattern in four protein sequences and several lines of the the chromosome 1 DNA sequence (we don't know how many exactly because the head command is only giving us ten lines of output).

U36: Getting fancy with regular expressions

A concept that is supported by many Unix programs and also by most programming languages (including Perl) is that of using regular expressions. These allow you to specify search patterns which are quite complex and really help restrict the huge amount of data that you might be searching for to some very specific lines of output. E.g. you might want to find lines that start with an 'ATG' and finish with 'TGA' but which have at least three AC dinucleotides in the middle:

```
$ grep "^ATG.*ACACAC.*TGA$" chr1.fasta

ATGAACCTTGTACTTCACCGGGTGCCCTCAAAGACGTTCTGCTCGGAAGGTTTGTCTTACACACTTTGATGTCAAATGA
ATGATAGCTCAACCACGAAATGTCATTACCTGAAACCCTTAAACACACTCTACCTCAAACTTACTGGTAAAAACATTGA
ATGCATACCTCAGTTGCATCCCGGCGCAGGGCAAGCATACCCGCTTCAACACACACTGCTTTGAGTTGAGCTCCATTGA
```

You'll learn more about regular expressions when you learn Perl. The `^` character is a special character that tells `grep` to only match a pattern if it occurs at the start of a line. Similarly, the `$` tells `grep` to match patterns that occur at the end of the line.

Task U36.1

The `.` and `*` characters are also special characters that form part of the regular expression. Try to understand how the following patterns all differ. Try using each of these these patterns with `grep` against any one of the sequence files. Can you predict which of the five patterns will generate the most matches?

```
ACGT
AC.GT
AC*GT
AC.*GT
```

> ***The asterisk in a regular expression is similar to, but NOT the same, as the other asterisks that we have seen so far. An asterisk in a regular expression means: 'match zero or more of the preceding character or pattern'.***

Try searching for the following patterns to ensure you understand what `.` and `*` are doing:

```
A...T
AG*T
A*C*G*T*
```

U37: Counting with `grep`

Rather than showing you the lines that match a certain pattern, `grep` can also just give you a count of how many lines match. This is one of the frequently used `grep` options. Running `grep -c` simply counts how many lines match the specified pattern. It doesn't show you the lines themselves, just a number:

```
$ grep -c i2 intron_IME_data.fasta
9785
```

Task U37.1

Count how many times each pattern from Task U36.1 occurs in all of the sequence files (specifying `*.fasta` will allow you to specify all sequence files).

U38: Regular expressions in `less`

You have seen already how you can use `less` to view files, and also to search for patterns. If you are viewing a file with `less`, you can type a forward-slash `/` character, and this allows you to then specify a pattern and it will then search for (and highlight) all matches to that pattern. Technically it is searching forward from whatever point you are at in the file. You can also type a question-mark `?` and `less` will allow you to search backwards. The real bonus is that the patterns you specify can be regular expressions.

Task U38.1

Try viewing a sequence file with `less` and then searching for a pattern such as `ATCG.*TAG$`. This should make it easier to see exactly where your regular expression pattern matches. After typing a forward-slash (or a question-mark), you can press the up and down arrows to select previous searches.

U39: Let me transl(iter)ate that for you

We have seen that these sequence files contain upper-case characters. What if we wanted to turn them into lower-case
characters (because maybe another bioinformatics program will only work if they are lower-case)? The Unix command tr
(short for transliterate) does just this, it takes one range of characters that you specify and changes them into another range of characters:

```
$ head -n 2 chr1.fasta

>Chr1 dumped from ADB: Mar/14/08 12:28; last updated: 2007-12-20
CCCTAAACCCTAAACCCTAAACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAATCTTTAAATCCTACATCCAT

$ head -n 2 chr1.fasta | tr 'A-Z' 'a-z'

>chr1 dumped from adb: mar/14/08 12:28; last updated: 2007-12-20
ccctaaaccctaaaccctaaaccctaaacctctgaatccttaatccctaaatccctaaatctttaaatcctacatccat
```

U40: That's what she sed

The `tr` command let's you change a range of characters into another range. But what if you wanted to change a particular pattern into something completely different? Unix has a very powerful command called sed that is capable of performing a variety of text manipulations. Let's assume that you want to change the way the FASTA header looks:

```
$ head -n 1 chr1.fasta >Chr1 dumped from ADB: Mar/14/08 12:28; last updated: 2007-12-20

$ head -n 1 chr1.fasta | sed 's/Chr1/Chromosome 1/' >Chromosome 1 dumped from ADB: Mar/14/08 12:28
```

The 's' part of the `sed` command puts `sed` in 'substitute' mode, where you specify one pattern (between the first two forward slashes) to be replaced by another pattern (specified between the second set of forward slashes). Note that this doesn't actually change the contents of the file, it just changes the screen output from the previous command in the pipe. We will learn later on how to send the output from a command into a new file.

U41: Word up

For this section we want to work with a different type of file. It is sometimes good to get a feeling for how large a file is before you start running lots of commands against it. The `ls -l` command will tell you how big a file is, but for many purposes it is often more desirable to know how many ‘lines’ it has. That is because many Unix commands like `grep` and `sed` work on a line by line basis. Fortunately, there is a simple Unix command called wc (word count) that does this:

```
$ cd Data/Arabidopsis/ $ wc At_genes.gff 531497 4783473 39322356 At_genes.gff
```

The three numbers in the output above count the number of lines, words and bytes in the specified file(s). If we had run `wc -l`, the `-l` option would have shown us just the line count.

U42: GFF and the art of redirection

The Arabidopsis directory also contains a GFF file. This is a common file format in bioinformatics and GFF files are used to describe the location of various features on a DNA sequence. Features can be exons, genes, binding sites etc, and the sequence can be a single gene or (more commonly) an entire chromosome.

This GFF file describes of all of the gene-related features from chromosome I of *A. thaliana*. We want to play around with some of this data, but don't need all of the file…just 10,000 lines will do (rather than the ~500,000 lines in the original). We will create a new (smaller) file that contains a subset of the original:

```
$ head -n 10000 At_genes.gff > At_genes_subset.gff
$ ls -l
total 195360
-rwxrwxrwx  1 keith  staff  39322356 Jul  9 15:02 At_genes.gff
-rwxrwxrwx  1 keith  staff    705370 Jul 10 13:33 At_genes_subset.gff
-rwxrwxrwx  1 keith  staf f 17836225 Oct  9  2008 At_proteins.fasta
-rwxrwxrwx  1 keith  staff  30817851 May  7  2008 chr1.fasta
-rwxrwxrwx  1 keith  staff  11330285 Jul 10 11:11 intron_IME_data.fasta
```

This step introduces a new concept. Up till now we have sent the output of any command to the screen (this is the default behavior of Unix commands), or through a pipe to another program. Sometimes you just want to redirect the output into an actual file, and that is what the `>` symbol is doing, it acts as one of three redirection operators in Unix.

As already mentioned, the GFF file that we are working with is a standard file format in bioinformatics. For now, all you really need to know is that every GFF file has 9 fields, each separated with a tab character. There should always be some text at every position (even if it is just a '.' character). The last field often is used to store a lot of text.

U43: Not just a pipe dream

The 2nd and/or 3rd fields of a GFF file are usually used to describe some sort of biological feature. We might be interested in seeing how many different features are in our file:

```
$ cut -f 3 At_genes_subset.gff | sort | uniq

CDS
chromosome
exon
five_prime_UTR
gene
mRNA
miRNA
ncRNA
protein
pseudogene
pseudogenic_exon
pseudogenic_transcript
snoRNA
tRNA
three_prime_UTR
transposable_element_gene
```

In this example, we combine three separate Unix commands together in one go. Let's break it down (it can be useful to just run each command one at at time to see how each additional command is modifying the preceding output):

1. The cut command first takes the At_genes_subset.gff file and 'cuts' out just the 3rd column (as specified by the `-f` option). Luckily, the default behavior for the `cut` command is to split text files into columns based on tab characters (if the columns were separated by another character such as a comma then we would need to use another command line option to specify the comma).
2. The sort command takes the output of the cut command and sorts it alphanumerically.
3. The uniq command (in its default format) only keeps lines which are unique to the output (otherwise you would see thousands of fields which said 'curated', 'Coding_transcript' etc.)

Now let's imagine that you might want to find which features start earliest in the chromosome sequence. The start coordinate of features is always specified by column 4 of the GFF file, so:

```
$ cut -f 3,4 At_genes_subset.gff | sort -n -k 2 | head

chromosome  1
exon    3631
five_prime_UTR  3631
gene    3631
mRNA    3631
CDS 3760
protein 3760
CDS 3996
exon    3996
CDS 4486
```

Here we first cut out just two columns of interest (3 & 4) from the GFF file. The `-f` option of the `cut` command lets us specify which columns we want to remove. The output is then sorted with the `sort` command. By default, `sort` will sort alphanumerically, rather than numerically, so we use the `-n` option to specify that we want to sort numerically. We have two columns of output at this point and we could sort based on either column. The `-k 2` specifies that we use the second column. Finally, we use the `head` command to get just the 10 rows of output. These should be lines from the GFF file that have the lowest starting coordinate.

U44: The end of the line

When you press the return/enter key on your keyboard you may think that this causes the same effect no matter what computer you are using. The visible effects of hitting this key are indeed the same…if you are in a word processor or text editor, then your cursor will move down one line. However, behind the scenes pressing enter will generate one of two different events (depending on what computer you are using). Technically speaking, pressing enter generates a newline character which is represented internally by either a *line feed or carriage return* character (actually, Windows uses a combination of both to represent a newline). If this is all sounding confusing, well it is, and it is even more complex than we are revealing here.

The relevance of this to Unix is that you will sometimes receive a text file from someone else which looks fine on their computer, but looks unreadable in the Unix text viewer that you are using. In Unix (and in Perl and other programming languages) the patterns `\n` and `\r` can both be used to denote newlines. A common fix for this requires substituting `\r` for `\n`.

Use `less` to look at the Data/Misc/excel_data.csv file. This is a simple 4-line file that was exported from a Mac version of Microsoft Excel. You should see that if you use `less`, then this appears as one line with the newlines replaced with ^M characters. You can convert these carriage returns into Unix-friendly line-feed characters by using the tr command like so:

```
$ cd Data/Misc
$ tr '\r' '\n' < excel_data.csv
sequence 1,acacagagag
sequence 2,acacaggggaaa
sequence 3,ttcacagaga
sequence 4,cacaccaaacac
```

This will convert the characters but not save the resulting output, if you wanted to send this output to a new file you will have to use a second redirect operator:

```
$ tr '\r' '\n' < excel_data.csv > excel_data_formatted.csv
```

U45: This one goes to 11

Finally, let’s parse the Arabidopsis `intron_IME_data.fasta` file to see if we can extract a subset of sequences that match criteria based on something in the FASTA header line. Every intron sequence in this file has a header line that contains the following pieces of information:

- gene name
- intron position in gene
- distance of intron from transcription start site (TSS)
- type of sequence that intron is located in (either CDS or UTR)

Let’s say that we want to extract five sequences from this file that are: a) from first introns, b) in the 5’ UTR, and c) closest to the TSS. Therefore we will need to look for FASTA headers that contain the text ‘i1’ (first intron) and also the text ‘5UTR’.

We can use `grep` to find header lines that match these terms, but this will not let us extract the associated sequences. The distance to the TSS is the number in the FASTA header which comes after the intron position. So we want to find the five introns which have the lowest values.

Before I show you one way of doing this in Unix, think for a moment how you would go about this if you didn’t know any Unix or Perl…would it even be something you could do without manually going through a text file and selecting each sequence by eye? Note that this Unix command is so long that — depending on how you are viewing this document — it may appear to wrap across two lines. When you type this, it should all be on a single line:

```
$ tr '\n' '@' < intron_IME_data.fasta | sed 's/>/#>/g' | tr '#' '\n' | grep "i1_.*5UTR" | sort -nk

>AT4G39070.1_i1_7_5UTR
GTGTGAAACCAAAACCAAAACAAGTCAATTTGGGGGCATTGAAAGCAAAGGAGAGAGTAG
CTATCAAATCAAGAAAATGAGAGGAAGGAGTTAAAAAAGACAAAGGAAACCTAAGCTGCT
TATCTATAAAGCCAACACATTATTCTTACCCTTTTGCCCACACTTATACCCCATCAACCT
CTACATACACTCACCCACATGAGTGTCTCTACATAAACACTACTATATAGTACTGGTCCA
AAGGTACAAGTTGAGGGAG

>AT5G38430.1_i1_7_5UTR
GCTTTTTGCCTCTTACGGTTCTCACTATATAAAGATGACAAAACCAATAGAAAAACAATT
AAG

>AT1G31820.1_i1_14_5UTR
GTTTGTACTTCTTTACCTCTCGTAAATGTTTAGACTTTCGTATAAGGATCCAAGAATTTA
TCTGATTGTTTTTTTTTCTTTGTTTCTTTGTGTTGATTCAG

>AT3G12670.1_i1_18_5UTR
GTAGAATTCGTAAATTTCTTCTGCTCACTTTATTGTTTCGACTCATACCCGATAATCTCT
TCTATGTTTGGTAGAGATATCTTCTCAAAGTCTTATCTTTCCTTACCGTGTTCTGTGTTT
TTTGATGATTTAG

>AT1G26930.1_i1_19_5UTR
GTATAATATGAGAGATAGACAAATGTAAAGAAAAACACAGAGAGAAAATTAGTTTAATTA
ATCTCTCAAATATATACAAATATTAAAACTTCTTCTTCTTCAATTACAATTCTCATTCTT
TTTTTCTTGTTCTTATATTGTAGTTGCAAGAAAGTTAAAAGATTTTGACTTTTCTTGTTT
CAG
```

That’s a long command, but it does a lot. Try to break down each step and work out what it is doing (you will need to consult the man page for some commands maybe). Notice that I use one of the other redirect operators `<` to read from a file. It took seven Unix commands to do this, but these are all relatively simple Unix commands; it is the combination of them together which makes them so powerful. One might argue that when things get this complex with Unix that it might be easier to do it in Perl!

Summary

Congratulations are due if you have reached this far. If you have learnt (and understood) all of the Unix commands so far then you probably will never need to learn anything more in order to do a lot of productive Unix work. But keep on dipping into the man page for all of these commands to explore them in even further detail.

The following table provides a reminder of most of the commands that we have covered so far. If you include the three, as-yet-unmentioned, commands in the last column, then you will probably be able to achieve >95% of everything that you will ever want to do in Unix (remember, you can use the `man` command to find out more about `top`, `ps`, and `kill`). The power comes from how you can use combinations of these commands.

The absolute basics	***File control***	***Viewing, creating, or editing files***	***Misc. useful commands***	***Power commands***	***Process-related commands***
ls	mv	less	man	uniq	top
cd	cp	head	chmod	sort	ps
pwd	mkdir	tail	source	cut	kill
	rmdir	touch	wc	tr	
	rm	nano		grep	
	| (pipe)			sed	
	> (write to file)				
	< (read from file)				

Part 3: Perl

Your programming environment

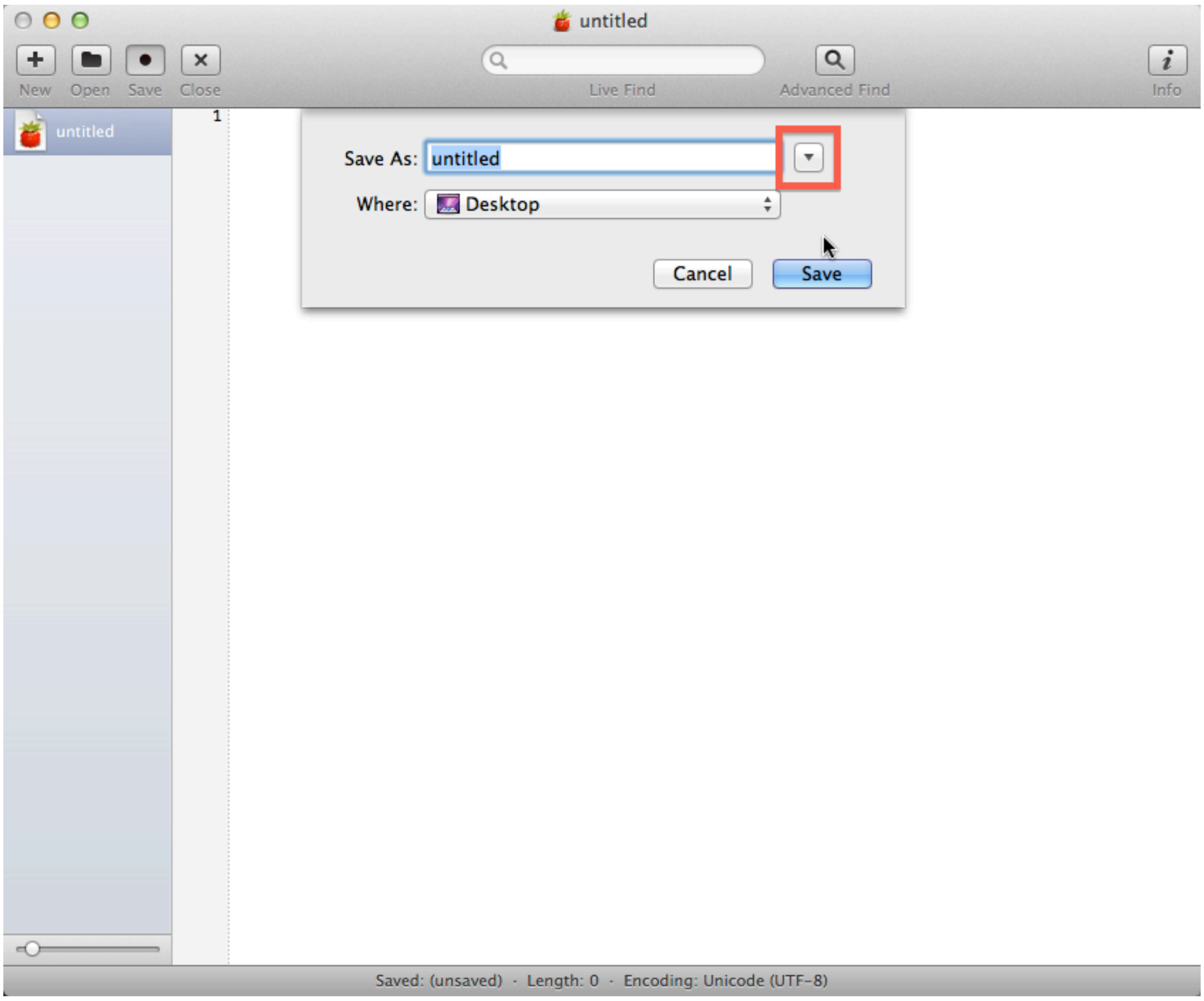
For this course, you will be using two applications, a code editor and a terminal. You should already be familiar with the Terminal application from the Unix lessons. If you are using a Mac then we recommend using a free code editor such as [Text Wrangler][]. Windows users might want to consider using [Notepad++][]. But please remember that there are many more editors out there and Wikipedia has a useful page comparing many of them. All of these editors will share several useful features such as syntax highlighting, automatic indentation, line numbering, and advanced search & replace.

> ***Remember to type:*** `source /Volumes/USB/Unix_and_Perl_course/.profile` ***at the beginning of every session***

Where to save your Perl scripts

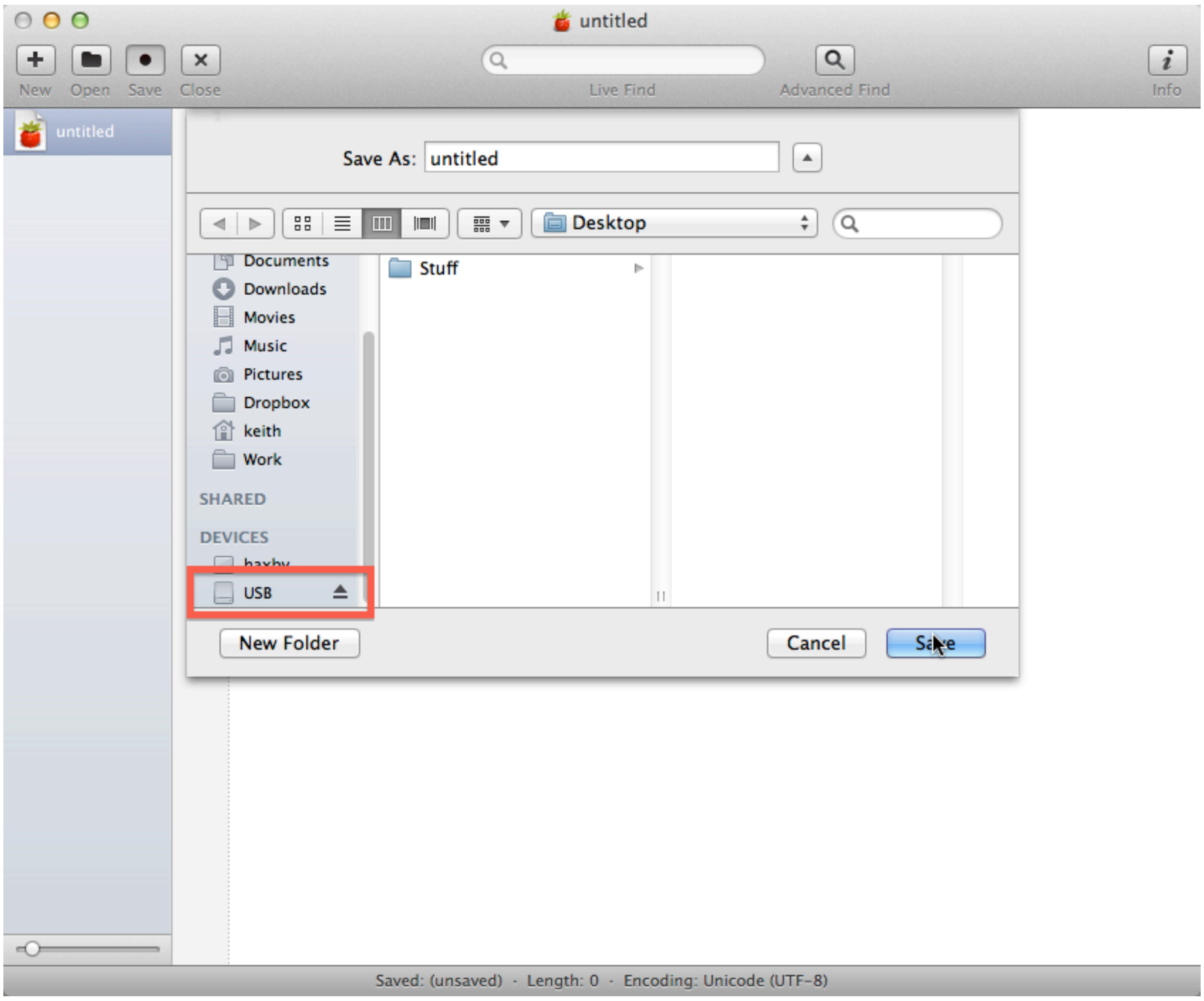
Every time you write a script you should save it in the `Unix_and_Perl_course/Code` directory. This is because we have specified this directory to be part of your Unix PATH (see section U30). If you keep your Perl scripts here then you can call them from any directory.

If you are new to Macs then it can be confusing to find out how to save a file to specific directory. When you click on the Save button in your code editor the default is to offer to save the file on the Desktop. Click on the *disclosure triangle* and this will expand the save dialog sheet and let you other folders and drives as the save destination:



Default (unexpanded) save dialog box

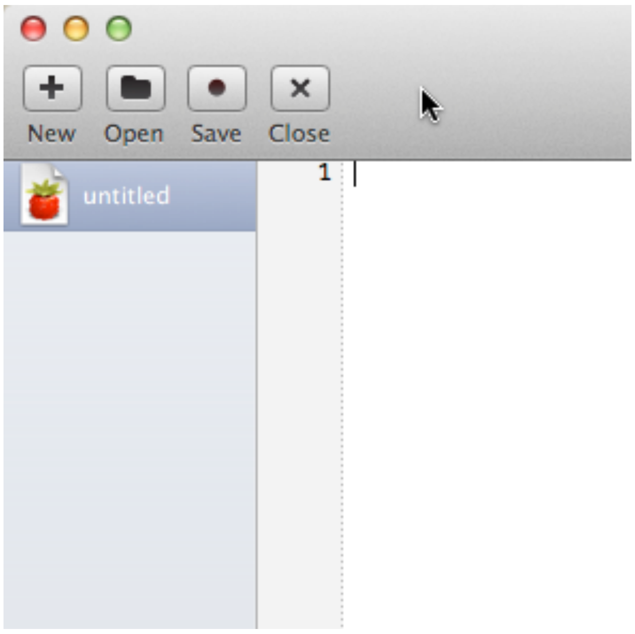
You should now be able to select your USB drive from the list of devices on the left hand side of the save sheet (you might need to scroll to make this available):



Expanded save dialog box

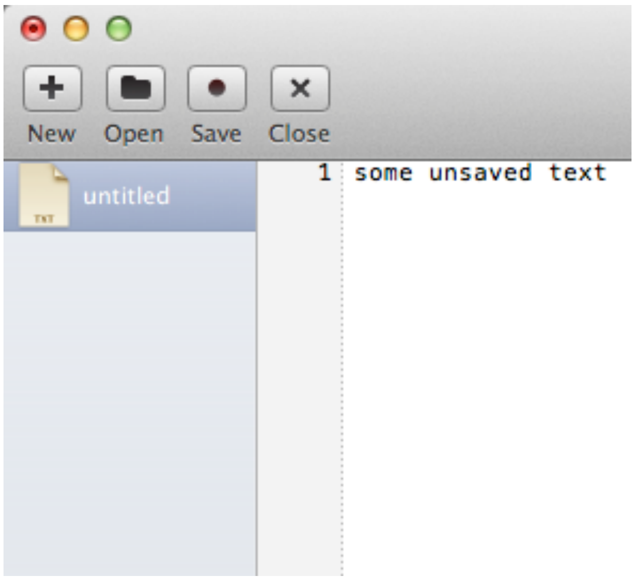
When to save your Perl scripts

Here is a handy Mac tip that will apply to Fraise and also to any other Mac graphical application that allows you to edit and save text. When you first open a new empty document, the program is — as yet — unsaved. If you haven't written anything then this is not a problem, and the top left corner of your application should look like this:



Unsaved document with no text

Now notice what happens when you start entering text into the main Fraise window. The window’s ‘close’ button (the red circle in the top left of the window), now has a small black dot inside it:



```
some unsaved text
```

Unsaved document with text

This is meant to serve as a visual reminder that your file is still unsaved. As soon as you click the ‘Save’ button, this black dot will disappear. From time to time you will have problems with your Perl scripts, and this might simply be because you have not saved any changes that you have made.

P1. Hello World

The first program you write in any language is always called [Hello World][]. The purpose of this program is to demonstrate that the programming environment is working, so the program is as simple as possible.

Task P1.1

Enter the text below into your text editor, but do not include the numbers. The numbers are there only so that we can reference specific lines.

```
1.  # helloworld.pl by _insert_your_name_here_
2.  print("Hello World!\n");
```

Line 1 has a `#` sign on it. When Perl sees a `#` sign, everything that follows on that line is considered a comment. Programmers use comments to describe what a program does, who wrote the program, what needs to be fixed, etc. It's a good idea to put comments in your code, especially as they grow larger.

Line 2 is the only line of this program that does anything. The `print()` function outputs its arguments to your terminal. In this case, there is only one argument, the text `"Hello World\n"`. The funny `\n` at the end is a newline character, which is like a carriage return. Most of the time, Perl statements end with a semicolon. This is like a period at the end of a sentence. The last statement in a block does not require a semicolon. We will revisit this in a later lesson.

Save the program as `helloworld.pl` (in your Code directory). To run the program, type the following in the terminal and hit return (making sure you have first changed directory to your 'Code' directory).

```
$ perl helloworld.pl
```

This will run the perl program and tell it to execute the instructions of the helloworld.pl file. If it worked, great. If it doesn't work, then you may see an error message like the one below:

```
Can't open perl script "helloworld.pl": No such file or directory
```

If you see this, you may have forgotten to save the file, misspelled the file name, or saved the file to someplace unintended. Always use tab-completion to prevent spelling mistakes. Always save your programs to the `Unix_and_Perl_course/Code` directory (for now anyway).

Task P1.2

Modify the program to output some other text, for example the date. Add a few more print statements and experiment with what happens if you omit or add extra newlines.

Task P1.3

Make a few deleterious mutations to your program. For example, leave off the semicolon or one of the parentheses. Observe the error messages. One of the most important aspects of programming is debugging. Probably more time is spent debugging than programming, so it’s a good idea to start recognizing errors now.

P2. Scalar variables

Variables hold data. In Perl, the main variable type is called a scalar variable. A scalar holds one thing. This thing could be a number, some text, or an entire genome. We will see other data types later. You can always tell a scalar variable because it has a $ on the front (the dollar sign is a mnemonic for scalar). For example, a variable might be named $x. When speaking aloud, we do not say “dollar x”. We just call it “x”.

Task P2.1

Create a new blank text document. Enter the text below and save this program as `scalar.pl` in your Code directory.

```
1.  #!/usr/bin/perl
2.  # scalar.pl by _insert_your_name_here_
3.  use warnings;
4.
5.  $x = 3;
6.  print($x, "\n");
```

Line 1 will appear at the top of every Perl script that we write from now on. This line of code is very similar to the line that appeared at the top of our Unix shell script. It lets Unix know that the Perl program (located at `/usr/bin/perl`) can read this file and run the remaining code inside it.

Line 2 is simply a comment. You should always include a few comments in your programs.

Line 3 is another line that we will add to every script from now on. This line effectively tells Perl that we would like to be warned if we start writing certain types of ‘bad’ code. This is a good thing! We will return to this later on.

Line 4 is deliberately blank. You should use spaces and blank lines to improve the readability of your code. In this case we are separating the first three lines of the script (which don’t actually calculate anything) from the rest.

Lines 5 is a variable assignment. The variable `$x` gets the value of 3

Line 6 prints the value of `$x` and then print a newline. As you can see, the `print()` function can take multiple arguments separated by commas.

Run the program by typing the line below in your terminal. Observe the output and go back through the code and line descriptions to make sure you understand everything.

```
$ scalar.pl
```

The addition of `#!/usr/bin/perl` to the script means that we no longer have to type:

```
$ perl scalar.pl
```

What is actually happening here is that we are making it clear that these text files contain instructions written in Perl. The line that we added tells Unix that it should expect to find a program called perl in the `/usr/bin` directory and that program should be capable of making sense of your Perl commands. Now try adding the following lines to your program.

```
7.  $s = "something";
8.  print($s, "\n");
9.  print("$s\n");
```

Line 7 is another variable assignment, but unlike `$x`, our new variable `$s` gets a character string, which is just another term for text.

Lines 8–9 print our new variable `$s` and then print a newline character.

Save the script and run it again. You should see that although lines 8–9 are different they produce exactly the same output. The print function can print a list of items (all separated by commas), but it often makes more sense to print just one thing instead. It would have been possible to rewrite our very first Perl script with the following:

```
print("H","e","l","l","o"," ","W","o","r","l","d","!","\n");
```

Hopefully you will agree that printing this phrase as one string and not thirteen separate strings is a lot easier on the eye. Now add the following line to your program, and run it again.

```
10.  print "$s\n";
```

Line 10 calls the print function without parentheses. You do not have to use parentheses for Perl functions, but they are often useful to keep a line organized. In most cases, you will see the print function without parentheses. Now add the final two lines to the program:

```
11.  print '$x $s\n';
12.  print "$x $s\n";
```

Line 11 puts the two variables between single quotes. Any text between single quotes will print exactly as shown. This also means that `\n` loses its special meaning as a newline character. In contrast, strings between double quotes will undergo a process known as *variable interpolation*. This means that variables are always expanded inside double quotes, and print will always show what those variables contain.

Task P2.2

Mutate your program. Delete a `$` and see what error message you get.

Task P2.3

Modify the program by changing the contents of the variables. Observe the output. Try experimenting by creating more variables.

Variables summary

You can use (almost) anything for your variable names, though you should try to use names which are descriptive and not too long. You should also use lower case names for your variables. This is not essential though. Which of the following is the best variable name for a variable that will store a DNA sequence?

```
$x = "ATGCAGTGA";         # $x is not a good choice
$dna_sequence_variable;  # also not a good choice, too long
$sequence = "ATGCAGTGA"; # $sequence is better
$dna = "ATGCAGTAGA";     # $dna is even better
```

It is perfectly fine to give a variable the same name as an existing function in Perl though this might be confusing. I.e. a variable named `$print` might look a bit too similar to the `print()` function. Sometimes though the choice of variable name is obvious: `$length` is often a good name for variables that contain the length of something, even though there is also a `length()` function in Perl (which we will learn about later on).

As shown in the example above, variable names can contain underscore characters to separate 'words'. This is often useful and helps make things easier to understand. E.g.

```
$first_name  = "Keith";
$second_name = "Bradnam";
```

Finally, you should be aware that (with a few exceptions) you can use spaces to make things clearer (or less clear if you so desire). The following lines are all treated by Perl in exactly the same way:

```
$dna = "ATGCAGTGA";       # one space either side of the '=' sign
$dna="ATGCAGTGA";         # no spaces either side of the '=' sign
$dna    =    "ATGCAGTGA"; # lots of spaces!
```

P3. Safer programming: `use strict`

Task 3.1

Create the following program, but don't run it yet. Instead try to to predict what it will do. Knowing how a program should work (before you run it) is a good programming skill to develop. If you don't understand what a program should be doing, then you will probably not realize if it is doing something wrong.

```
1.  #!/usr/bin/perl
2.  # strict.pl by _insert_your_name_here_
3.  use strict; use warnings;
4.
5.  $pi = 3.14;
6.  print "pi = $pi\n";
7.
8.  $pi = 3.141593;
9.  print "pi = $pi\n";
```

You hopefully noticed that this program introduces another new concept; line 3 includes another usage statement: `use strict;` (in addition to `use warnings;`). Up till now we have ended each line of Perl code with a semi-colon, but there are times when it is simpler to put two lines of Perl code into one line in an editor. Perl will still treat these as two separate lines of code.

Telling Perl to `use strict` means that Perl will insist your script is written in a certain way which is widely considered to be a 'better' way of writing code. At this point it is not important to go into the details of what exactly `use strict` is doing. Just accept our word that including a `use strict; use warnings;` line in every script that you write is a good thing to do (we will return to these issues later).

Task 3.2

Now try running the script. You should hopefully see the following errors:

```
Global symbol "$pi" requires explicit package name at strict.pl line 5.
Global symbol "$pi" requires explicit package name at strict.pl line 6.
Global symbol "$pi" requires explicit package name at strict.pl line 8.
Global symbol "$pi" requires explicit package name at strict.pl line 9.
Execution of strict.pl aborted due to compilation errors.
```

We see one error message for each use of the `$pi` variable in the script. Now see what happens if you remove the `use strict;` statement and re-run the script. It should now work. What is happening here? When we tell Perl that we want to `use strict;` Perl will first check the code and one of the things it will do is to look at how variables are declared. In Perl, when we first introduce any variable we can *optionally* describe whether they are available to all parts of a program or not. However, if we turn on `use strict;` it becomes *mandatory* to say whether the variable is a *local* or *global* variable. At this time it is not important to understand the details of this (we will return to it later on), other than that we want our programs to include `use strict` and so we will be making our variables local variables.

Task 3.3

Make sure that `use strict;` is back in your program. Now change line 5 of the program to the following and run your script again (it should now work and should not produce any errors):

```
5.  my $pi = 3.14;
```

We are now declaring the `$pi` variable using the word `my`. This makes the variable a local variable and we will now be doing this most of the time that we introduce any new variable. It might help to think of the `my` word as reading as ‘let’. At this point you are probably thinking that including `use strict;` in your programs is making things more complex. That is true but the benefits of including `use strict;` outweigh the costs associated with it.

Task 3.4

The other point of this programming exercise is to introduce you to the simple fact you can reassign variables to different values or strings. Try declaring a new variable and and assign it a value. Add two more lines to change that value and print it out again.

P4. Math

Perl, like most programming languages supports a variety of mathematical operators and functions. Let's experiment with some of these.

Task P4.1

Write the program below, save it as `math.pl`, and then run it. But wait, this time we are going to take a slightly different strategy. The program is getting longer. If you type the whole thing and have a lot of errors, it will become difficult to debug. So instead, write only a few lines, and then save, run, and observe the output. Debug if necessary. Try to check that your program is working every few lines. As you get more experience, you will gain skill and confidence and not need to check as frequently.

```
1.  #!/usr/bin/perl
2.  # math.pl
3.  use strict; use warnings;
4.
5.  my $x = 3;
6.  my $y = 2;
7.  print "$x plus $y is ", $x + $y, "\n";
8.  print "$x minus $y is ", $x - $y, "\n";
9.  print "$x times $y is ", $x * $y, "\n";
10. print "$x divided by $y is ", $x / $y, "\n";
11. print "$x modulo $y is ", $x % $y, "\n";
12. print "$x to the power of $y is ", $x ** $y, "\n";
```

Task P4.2

In addition to the mathematical operators we've just seen, there are a number of built-in numeric functions: e.g. `abs()`, `int()`, `log()`, `rand()`, `sin()`. Add the following lines to the program, run it, and observe the output.

```
13. print "the absolute value of -$x is ", abs(-$x), "\n";
14. print "the natural log of $x is ", log($x), "\n";
15. print "the square root of $x is ", sqrt($x), "\n";
16. print "the sin of $x is ", sin($x), "\n";
17. print "a random number up to $y is ", rand($y), "\n";
18. print "a random integer up to $x x $y is ", int(rand($x * $y)), "\n";
```

Line 18 could have been written as `int rand $x * $y`. This is another example where you can omit parentheses if you like. But just because you can doesn't mean you should.

Task P4.3

In the examples above, the `print()` function outputs text as well as the actual mathematical operations. This is fairly uncommon in real programming. Generally, we want to make some computation, store that value, and do more computations. To store values, we need to create a new variable that will hold the contents.

```
my $z = ($x + $y) / 2;
print "$z\n";
```

Task P4.4

In this next exercise, you will build a simple calculator that calculates X to the power of Y. Instead of assigning the variables inside the code, we will let the user input the values without editing the file. In general, this is how programs should work. Once written, they can be used without editing the source code.

```
1.  #!/usr/bin/perl
2.  # pow.pl
3.  use strict; use warnings;
4.
5.  my ($x, $y) = @ARGV;
6.  print $x ** $y, "\n";
```

Line 5 has an unfamiliar construct. `@ARGV` is list of values from the command line. We will discuss lists and arrays in greater detail later. For now, just accept that the values from the command line will end up being contained in `$x` and `$y`. For example, if you type the line below in the terminal, when the program runs, `$x` will contain 3.14 and `$y` will contain 2.718.

```
$ pow.pl 3.14 2.718
```

Task 4.5

Let's make one more calculator for fun (yes, coding is fun!). This one will compute the factorial of a number. Factorials are usually computed with some kind of a loop (we will talk a lot about loops later). Here is an alternate method that provides a reasonable approximation. Unlike the true factorial, this method can use non-integers. Note that we have written the 6th line of code such that it is split across five lines in our coding editor (lines 7—10 are also indented with tabs). This is a common practice to make code easier to read, though it would still be valid — albeit a little untidy — to write this as a single long line of code (in your editor).

```
1.  #!/usr/bin/perl
2.  # stirling.pl (Stirling's approximation to the factorial)
3.  use strict; use warnings;
4.
5.  my ($n) = (@ARGV);
6.  my $ln_factorial =
7.      (0.5 * log(2 * 3.14159265358979))
8.      + ($n + 0.5) * log($n)
9.      - $n + 1 / (12 * $n)
10.     - 1 / (360 * ($n ** 3));
11. print 2.71828 ** $ln_factorial, "\n";
```

Try it out:

```
$ stirling.pl 5
$ stirling.pl 7.1
```

Operator Precedence

Let’s quickly discuss operator precedence. Some operators have higher precedence than others. We’re used to seeing this in math where multiplication and division come before addition and subtraction: 3 + 2 * 5 = 13. If you want to force addition before multiplication, you can do this as (3 + 2) * 5 = 25. Perl has a lot of operators in addition to the mathematical operators and there are a lot of precedence rules. Don’t bother memorizing them. The universal precedence rule is this: multiplication comes before addition, use parentheses for everything else.

P5. Conditional statements

One of the most basic foundations of programming is the conditional statement. This is simply: if *condition* then do something, otherwise do something else. The *condition* is some kind of true-false statement.

Task P5.1

In the following program, note that equality is tested with two equals signs! One of the most common errors of novice programmers is using a single equals sign.

```
1.  #!/usr/bin/perl
2.  # conditional.pl
3.  use strict; use warnings;
4.
5.  my ($x, $y) = @ARGV;
6.  if ($x == $y) {
7.      print "equal\n";
8.  }
8.  else {
9.      print "not equal\n";
10. }
```

Did you notice how the print statements on lines 7 and 9 are indented? This is no accident! It shows the logical hierarchy of the code. The spacing is achieved by using a tab character. Many code editors will be smart enough to put tabs in for you automatically.

Numerical comparison operators in Perl

We have just seen the `==` operator which tests for numerical equality, here are all the ways of comparing two numbers:

Operator	Meaning	Example
`==`	equal to	`if ($x == $y)`
`!=`	not equal to	`if ($x != $y)`
`>`	greater than	`if ($x > $y)`
`<`	less than	`if ($x < $y)`
`>=`	greater than or equal to	`if ($x >= $y)`
`<=`	less than or equal to	`if ($x <= $y)`
`<=>`	comparison	`if ($x <=> $y)`

Indentation and block structure

In general, all the statements that are conditional on some other statement are indented with a tab character. You can have conditional statements inside other conditional statements, in which case you will have multiple levels of indentation. Is this necessary? Yes and no. It is necessary to aid readability, but it is not necessary to get your program to run. Pay attention to the indentation in the example programs and follow them closely. Wikipedia has a good page describing different indentation styles. Feel free to choose one of those, but do not make up your own style! Your #1 job as a programmer is to write programs that can be easily understood by others, and inventing new programming paradigms defeats that goal.

Task P5.2

Modify the program by changing the variables and relational operators. The numeric relational operators are in the accompanying table. Experiment to see if you can figure out what the `<=>` operator does (it is called the spaceship operator).

Task P5.3

Hierarchy is one of the most important concepts in programming. We are used to seeing hierarchical file systems where files are inside of folders which might be inside other folders. Programming uses the same concept. In Perl, hierarchy is shown with tabs and curly brackets. Statements (files) are inside curly brackets (folders) which might be inside other curly brackets (more folders). The following code contains an `if-else` statement, with the `if` part containing a second `if` statement.

```perl
#!/usr/bin/perl
# nested_conditional.pl
use strict; use warnings;

my ($x, $y) = @ARGV;
if ($x > $y) {
    print "$x is greater than $y\n";
    if ($x < 5) {
        print "$x is greater than $y and less than 5\n";
    }
} else {
    print "$x is not greater than $y\n";
}
```

Whitespace

Indentation and white space improve readability. Consider the following legal but confusing code which omits tabs and spaces (and even some semicolons):

```perl
if($x>$y){print"1\n";if($x<5){print"2\n"}}else{print"3\n"}
```

A program must be readable above all else. A program that works but is unreadable is difficult to improve or maintain.

P5.4

Sometimes you want to test a series of conditions. This next example shows you how to do this by using the `elsif` statement. Note that this example also ends with an `else` statement. This ensures that that something will always be printed no matter what value is held in `$x`.

```perl
#!/usr/bin/perl
# elsif.pl
use strict; use warnings;

my ($x) = @ARGV;

if ($x >= 3) {
    print "x is at least as big as 3\n";
}
elsif ($x >= 2) {
    print "x is at least as big as 2, but less than 3\n";
}
elsif ($x >= 1) {
    print "x is at least as big as 1, but less than 2\n";
}
else { print "x is less than 1\n"; }
```

Task P5.5

For simple switches such as the above example, it is sometimes useful to break the usual indentation rule. In the example below, note that the obligatory semicolons have been dropped. It turns out that the last line of a block does not need to be terminated with a semicolon precisely for this kind of code beautification. Also note that spaces are added so that the braces line up in columns.

```perl
if    ($x >= 3) {print "x is at least as big as 3\n"}
elsif ($x >= 2) {print "x is at least as big as 2, but less than 3\n"}
elsif ($x >= 1) {print "x is at least as big as 1, but less than 2\n"}
else            {print "x is less than 1\n"}
```

Other Conditional Constructs

An alternative to `if ($x != $y)` is `unless ($x == y)`. There are times when 'unless' is more expressive than 'if not'. You cannot use `elsif` or `else` statements with an `unless` statement however.

Perl also lets you do something called *post-fix* notation. This allows you to put the `if` or `unless` at the end of the statement rather than at the beginning. You can't use `elsif` or else in this case, but sometimes the post-fix notation just reads much better. Here are two examples:

```perl
print "x is less than y\n" if $x < $y;
print "x is less than y\n" unless $x >= $y;
```

Finally, Perl includes something called the *trinary* operator that lets you do very simple if-then-else statements with just a few symbols. Consider the following statement:

```perl
if ($x == $y) {print "yes"}
else          {print "no"}
```

This can be written more succinctly as:

```perl
print $x == $y ? "yes\n" : "no\n";
```

The trinary operator is not that commonly used, but you will see it from time to time.

Numeric Precision and Conditionals

Although Perl hides the details, numbers in a computer are generally stored either as *integer* or *floating point* (decimal) numbers. Both *ints* and *floats* have minimum and maximum values, and floats have limited precision. You have probably run into these concepts with your calculator. If you keep squaring a number greater than 1.0 you will eventually run into an *overflow error*. In Perl, this will happen at approximately 1e+308. Similarly, if you repeatedly square a number less than 1.0, you will eventually reach an underflow error. In Perl, the lowest non-zero value you can get to zero is approximately 1e–308. Try some extreme values in pow.pl or stirling.pl to reach underflow and overflow.

Floating point numbers do not have the exact value you may expect. For example, 0.1 is not exactly one-tenth. Perl sometimes hides these details. Try the following code. When you run this, you expect to see 0.3 0.3 0.0, but that’s not what happens because adding the imprecise 0.1 three times is not the same as the imprecise 0.3.

```perl
#!/usr/bin/perl
# float.pl
use strict; use warnings;

my $x = 0.1 + 0.1 + 0.1;
my $y = 0.3;
print $x, "\t", $y, "\t", $x - $y, "\n"; # \t is a tab character
```

Since floating point numbers are approximations, you should not compare them in conditional statements. Never ask `if ($x == $y)` if the values are floats because as we have seen, 0.3 is not necessarily equal to 0.3. Instead, ask if their difference is smaller than some acceptable threshold value:

```perl
my $threshold = 0.001;
if (abs($x - $y) < $threshold) {print "close enough\n"}
```

Project 0: Poisson

The Poisson distribution is
commonly (mis)used as a null model in biology. For example, suppose you are
aligning genomic sequence reads back to a genome. Your average depth of coverage
is expected to be 10x. In one particular region you observe 5x. In another
region you observe 20x. One interpretation is that the 5x region is haploid
(assuming the organism is diploid). Similarly, the 20x region could be a
duplication. But there is some chance that these fluctuations are entirely
random. In this project, we will calculate how likely 5x or 20x coverage is
given an expectation of 10x.

The Poisson distribution has two parameters: lambda and k. Lambda is an integer
greater than zero. It is both the mean (expected value) and the variance. In our
example above, lambda is 10. The second parameter, k, is the number of
observations seen. In our example above, this is 5 or 20. This parameter can be
any non-negative integer, and zero is a very useful value because we can ask how
likely is it if we did not observe something we expected. For example, some
areas of a genome may have no reads aligned to them. There may be biological reasons for this, but
under a Poisson model, there are also stochastic reasons to expect this from
time to time.

One of the reasons to use a Poisson distribution is because it gives you a feel
for how important it is to have a large number of counts. Let's say you have
observed some phenomenon 9 times. Under a Poisson model, the standard deviation
is 3, as this is the square root of the variance. So 9 counts of something can
also be thought of as 9 +/- 3. Zero is only 3 standard deviations away from
expected. So the difference between 9 counts and 0 counts is not that large. In
contrast, the difference between 400 counts and 300 counts is much larger, since
300 is 5 standard deviations away from 400 (20 being the square root of 400). In
a gene expression context, suppose you see 9 counts of something in a cancer
cell and 0 counts in a normal cell. Is this significant? Is 400 vs. 300 more
significant? Statistically speaking, 400 vs. 300 is much more robust than 9 vs.
0. But biologically speaking, 9 vs. 0 may be more important if it turns out to
be an accurate representation of the data. However, it could be completely
random. When looking at thousands of data points, things that happen only once
in a thousand times are supposed to occur! This is why it is important to
perform replicate experiments. Enough science, let's get coding.

Goals of your program

- Give your program a reasonable name
- Include informative comments
- Use descriptive variable names
- Use whitespace to show hierarchy and to separate thoughts

- Use `@ARGV` (see lesson P4.4) so that you can specify lambda and k on the command line
- Use conditional statements to make sure that lambda and k are not out of bounds
- Use Stirling's approximation to compute factorials (see lesson P4.5)
- You could check to make sure that lambda and k are integers but it is not strictly necessary when using Stirling's approximation
- Report error messages where applicable
- Compute and report the answer (the probability of k, given lambda)

Note that the final item in the list is to compute the answer. Proper programming is more than just getting the right answer. A beautiful program that is not quite correct is better than a correct indecipherable program. The former can be maintained and improved in the future, the latter cannot.

Play around with the program and try a bunch of different values for lambda and
k. As a thought experiment, if you set k = 0, what value of lambda gives you
confidence that k is significantly different from lambda?

P6. String operators

From algebra, we are used to the idea of using variables in math. But what about strings? Can you add, subtract, multiply, divide, and compare strings? Not exactly, but there are analogous operations. We'll see more of this later. For now, let's just look at some simple operators.

Task P6.1

Create the following program and run it.

```
1.  #!/usr/bin/perl
2.  # strings.pl
3.  use strict; use warnings;
4.
5.  my $s1 = "Hello";
6.  my $s2 = "World\n";
7.  my $s3 = $s1 . " " . $s2;
8.  print $s3;
```

Line 7 introduces the concatenate operator which in Perl is represented by the dot `.` character. This operator allows you to join two or more strings together and (optionally) store the result in a new variable. In this case we create a new variable `$s3` which stores the result of joining three things together (`$s1`, a space character " ", and `$s2`). Now add the following lines to the script.

```
9.  if    ($s1 eq $s2) {print "same string\n"}
10. elsif ($s1 gt $s2) {print "$s1 is greater than $s2\n"}
11. elsif ($s1 lt $s2) {print "$s1 is less than $s2\n"}
```

How are these strings compared? It might make sense to compare them by length, but that's not what is happening. They are compared by their ASCII values. So 'A' is less than 'B' which is less than 'Z'. Similarly 'AB' is less than 'AC' and 'ABCDE' is also less than 'AC'. Oddly, 'a' is greater than 'A'. See the wikipedia page on ASCII to see the various values. To get the length of a string, you use the length() function.

String comparison operators in Perl

Operator	Meaning	Example
`eq`	equal to	`if ($x eq $y)`
`ne`	not equal to	`if ($x ne $y)`
`gt`	greater than	`if ($x gt $y)`
`lt`	less than	`if ($x lt $y)`
`.`	concatenation	`$z = $x . $y`
`cmp`	comparison	`if ($x cmp $y)`

Task P6.2

Modify the program in P6.1 to experiment with different string comparison operators. Then try comparing a number and a string using both numeric and string comparison operators. Try using the `length()` function.

Task P6.3

If you are interested in ASCII values, try using the `ord()` and `chr()` functions, which convert letters to numbers and vice-versa.

```
print ord("A"), "\n";
print chr(66), "\n";
```

Matching Operators

One of the most common tasks you may have as a programmer is to find a string within another string. In a biological context, you might want to find a restriction site in some DNA sequence. These kinds of operations are really easy in Perl. We are only going to touch on a few examples here. In a few lessons we will get much more detailed.

Task P6.4

Enter the program below and observe the output. The binding operator `=~` signifies that we are going to do some string manipulation next. The exact form of that manipulation depends on the next few characters. The most common is the match operator `m//`. This is used so commonly that the m can be omitted. There are also substitution and transliteration operators. If your script is working then try changing line 6 to make the matching operator match other patterns.

```
1.  #!/usr/bin/perl
2.  # matching.pl
3.  use strict; use warnings;
4.
5.  my $sequence = "AACTAGCGGAATTCCGACCGT";
6.  if ($sequence =~ m/GAATTC/) {print "EcoRI site found\n"}
7.  else {print "no EcoRI site found\n"}
```

Matching operators in Perl

Operator	Meaning	Example
`=~ m//`	match	`if ($s =~ m/GAATTC/)`
`=~ //`	match	`if ($s =~ /GAATTC/)`
`!~ //`	not match	`if ($s !~ m/GAATTC/)`
`=~ s///`	substitution	`$s =~ s/thing/other/;`
`=~ tr///`	transliteration	`$count = $s =~ tr/A/A/;`

Task P6.5

Add the following lines and observe what happens when you use the substitution operator. This behaves in a similar way to the `sed` command in Unix.

```
8.  $sequence =~ s/GAATTC/gaattc/;
9.  print "$sequence\n";
```

Now add the following lines and find out what happens to $sequence.

```
10. $sequence =~ s/A/adenine/;
11. print "$sequence\n";
12. $sequence =~ s/C//;
13. print "$sequence\n";
```

Line 12 replaces the occurrence of a C character with nothing (//), i.e. it deletes a C character. You should have noticed though that lines 10 and 12 only replaced the *first* occurrence of the matching pattern. What if you wanted to replace all occurrences? To specify a 'global' option (i.e. replace all occurrences), we add a letter 'g' to the end of the substitution operator:

```
14. $sequence =~ s/C//g; # adding 'g' on the end of substitution operator
```

This is similar to how we use command-line options in Unix, the 'global' option modifies the default behavior of the operator.

Task P6.6

Add the following lines to the script and try to work out what happens when you add an 'i' to the to matching operator:

```
15. my $protein = "MVGGKKKTKICDKVSHEEDRISQLPEPLISEILFHLSTKDLWQSVPGLD";
16. print "Protein contains proline\n" if ($protein =~ m/p/i);
```

Task P6.7

In bioinformatics, you will sometimes be given incorrectly formatted data files which might break your script. Therefore we often want to stop a script early on if we detect that the input data is not what we were expecting. Add the following lines to your script and see if you can work out what the die() function is doing.

```
my $input = "ACNGTARGCCTCACACQ"; # do you know your IUPAC characters?
die "non-DNA character in input\n" if ($input =~ m/[efijlopqxz]/i);
print "We never get here\n";
```

It is very common to stop scripts by using the 'die … if' syntax. There is no point letting a script continue processing data if the data contains errors. Perl does not know about rules of biology so you will need to remember to add suitable checks to your scripts.

The transliteration operator

The transliteration operator gets its own section as it is a little bit different to the other matching operators. If you worked through Part 2 of the Unix lessons you may remember that there is a `tr` command in Unix. The transliteration operator behaves in the same way as this command. It takes a list of characters and changes each item in the list to a character in a second list, though we often use it with just one thing in each list. It automatically performs this operation on all characters in a string (so no need for a 'global' option).

Task P6.8

Make a new script to test the full range of abilities of the transliteration operator. Notice how there are comments at the end of many of the lines (the hash character '#' denotes the start of a comment). You don't have to type these comments, but adding comments to your scripts is a good habit to get into. You will need to add suitable print statements to this script in order for it to do anything.

```
1.  #!/usr/bin/perl
2.  # transliterate.pl
3.  use strict; use warnings;
4.
5.  my $text = "these are letters: abcdef, and these are numbers, 123456";
6.
7.  $text =~ tr/a/b/;     # changes any occurrence of 'a' to 'b'
8.  $text =~ tr/bs/at/;   # the letter 'b' becomes 'a', and 's' becomes 't'
9.  $text =~ tr/123/321/; # 1 becomes 3, 2 stays as 2, 3 becomes 1
10. $text =~ tr/abc/ABC/; # capitalize the letters a, b, and c
11. $text =~ tr/ABC/X/;   # any 'A', 'B', or 'C' will become an X
12. $text =~ tr/d/DE/;    # incorrect use, only 'd' will be changed to 'D'
```

On Line 5 in this script we define a string and save that to a variable `$text`. Lines 7—12 then perform a series of transliterations on the text.

Task P6.9

If you have many characters to transliterate, you can use the `tr` operator in a slightly different way, which you may (or may not) find easier to understand:

```
13. $text =~ tr [abcdefgh]
14.              [hgfedcba]; # semicolon is here and not on line 13
```

In this case we use two pairs of square brackets to denote the two range of characters, rather than just three slashes. There are two lines of your code in the text editor, but Perl sees this as just one line. Perl scripts can contain any amount of *whitespace*, and it often helps to split one line of code into two separate lines in your editor. The following line would be treated by Perl as exactly the same as Lines 13–14:

```
15. $text =~ tr[abcdefgh][hgfedcba]; # whitespace removed
```

Task P6.10

The transliteration operator can also be used to count how many changes are made. This can be extremely useful when working with DNA sequences. Add the following lines to your script.

```
16. my $sequence = "AACTAGCGGAATTCCGACCGT";
17. my $g_count = ($sequence =~ tr/G/G/);
18. print "The letter G occurs $g_count times in $sequence\n";
```

Line 17 may appear confusing. The transliteration operator is changing the letter G to itself, and it then assigns the result of this operation to a new variable (`$g_count`). So what is happening? Perl performs the code inside the parentheses first and this performs the transliteration. The result of the transliteration is that lots of G->G substitutions are made which leaves $sequence unchanged. The transliteration operator counts how many changes are made. Normally it does nothing with this count, but if you ask Perl to assign the output of the transliteration to a variable (as in this example), then it will store the count in that variable.

Task P6.11

Remove the parentheses from line 17. Does the script still work? This is a case where the parentheses are not needed by Perl, but their inclusion might make your code more understandable. If you have any code where you use the assignment operator `=` , Perl always evaluates the right-hand side of the equals sign first.

Task P6.12

Add the following line to your script and see if you can understand how to specify a ‘range’ of characters with the `tr` operator.

```
19. $sequence =~ tr/A-Z/a-z/;
```

If you have added this line but nothing seems to be happening, then maybe you need to add another line of code in order to see the result of what line 19 is doing.

Project 1: DNA composition

At this point, we know enough Perl to write our first useful program. The program will read a sequence and determine its length and composition. Unlike the various tasks that appear in each chapter, we will not provide you the code for this project. You must write it yourself.

Program Name

A descriptive program name helps people understand what it does. But people often choose the name based on some other criteria. Unix program names are almost always short and lower case to minimize typing. Bioinformatics programs tend towards acronyms and abbreviations. Once you come up with a concept for a great program, choosing an appropriate name can sometimes be the hardest part (only half-joking). Feel free to choose whatever name you want for this project, but a name such as project1.pl or dnastats.pl is better than 1337.pl.

Executable

Programs should have executable permission. This will happen automatically if they are on your USB flash drive but you should still know how to use the Unix command `chmod` to add execute permission. Your script should also have a `#!` statement as the first line.

Usage Statement

Programs should have some kind of documentation that tell other people how to use the program. Users should not have to figure it out from the source code, especially if they are not programmers. A simple, but useful form of documentation is the usage statement. This is generally between 1 and 20 lines of text that informs people what the program does and what the arguments are. Usage statements are often displayed if the program is given no arguments. In that case, you want the program to report a little documentation and quit (using the `die` function again). Here are the first few lines of your program.

```perl
#!/usr/bin/perl
# dnastats.pl by ___
use strict; use warnings;

die "usage: dnastats.pl <dna sequence>\n" unless @ARGV == 1;
my ($seq) = @ARGV;
```

While it is possible to use the `die()` function without printing any output, you should always try to include a helpful statement as to why the program has stopped. It is common to see several `die` statements near the start of a script as this is the point when you should ideally check that all of the script's parameters make sense and that any input files are present (and valid).

On line 5 the `die()` function will be run *unless* the `@ARGV` array contains exactly one item (remember that the `@ARGV` array contains a list of anything you specify on the command-line after the script name). Note that we don't need to calculate the length of the array and store that in a variable, we can just test the length of the array implicitly. You will often use the `die` function in conjunction with the `if` operator, i.e. *if* something is missing, stop the script. Note that line 5 could be replaced with the following if we wanted to make things even more explicit:

```
5.  my $number_of_arguments = @ARGV;
6.  if($number_of_arguments != 1){
7.      die "usage: dnastats.pl <dna sequence>\n";
8.  }
```

Goals of your program

Your program should read a sequence that is specified on the command line and report the following:

- The length of the sequence
- The total number of A, C, G, and T nucleotides
- The fraction of A, C, G, and T nucleotides (i.e. %A, %C etc.)
- The GC fraction

P7. List context

Up till now we have only worked with single variables or values, but we often want to work with lists of things. In Perl, if you have multiple scalar values in parentheses separated by commas, this is known as *list context* (actually it's still list context even if you have one or even zero scalar values in parentheses).

Task P7.1: Create the following short program and run it.

```
1.  #!/usr/bin/perl
2.  # list.pl
3.  use strict; use warnings;
4.
5.  my ($x, $y, $z) = (1, 2, 3);
6.  print "x=$x y=$y z=$z\n";
```

The code in line 5 takes a list of three values `(1, 2, 3)` and assigns them to a list of three variables `($x, $y, $z)`. Without using lists, we would have to have three separate lines of code in order to declare and initialize each variable with a value.

Assignments in lists occur simultaneously. Because of this, line 7 below exchanges the values for `$x` and `$y`.

```
7.  ($x, $y) = ($y, $x);
8.  print "x=$x y=$y\n";
```

Task P7.2

Exchange the value of `$x` and `$y` without using list context. This is one of those problems that appears difficult at first, but once you see the solution, it will seem so obvious that you can't imagine how you didn't think of it immediately.

P8. Safer programming: use warnings

We've been telling you to always include a use warnings; line in your Perl scripts, but we haven't really explained why. Let's see what can happen when we *don't* include it.

Task P8.1

In the last lesson, we discussed assignments in lists. What if the lists are not the same length? Let's find out. Try this program, but this time make sure that you *don't* include the `use warnings` statement.

```
1.  #!/usr/bin/perl
2.  # undefined.pl
3.  use strict;
4.
5.  my ($x, $y, $z) = (1, 2, 3, 4, 5);
6.  print "x=$x y=$y z=$z\n";
7.
8.  my ($a, $b, $c) = (1, 2);
9.  print "c=$c\n";
10. print length($c), "\n";
11. print $a + $c, "\n";
```

Line 5 assigns 3 variables with 5 values. The two extra values on the right are simply thrown away.

Line 8 assigns 3 variables from only 2 values. So what happens to `$c`? The output from line 9 suggests that `$c` is some kind of a blank, and the output from line 10 suggests it has no length. But the output from line 11 suggests that `$c` has a value of zero. What is happening in this script is that `$c` has an *undefined value*. It is simultaneously zero and an empty string. Do you find this confusing? It is.

Undefined values are bad. You should never assume the contents of a variable. Variables should always be assigned before they are used. Similarly, lists should be the same length on each side of an assignment, but Perl has no way of checking this. To find undefined values, always include `use warnings` in your program. This will alert you when undefined variables are being used. If you have undefined values, stop immediately and debug. A program that runs with undefined values can be very dangerous.

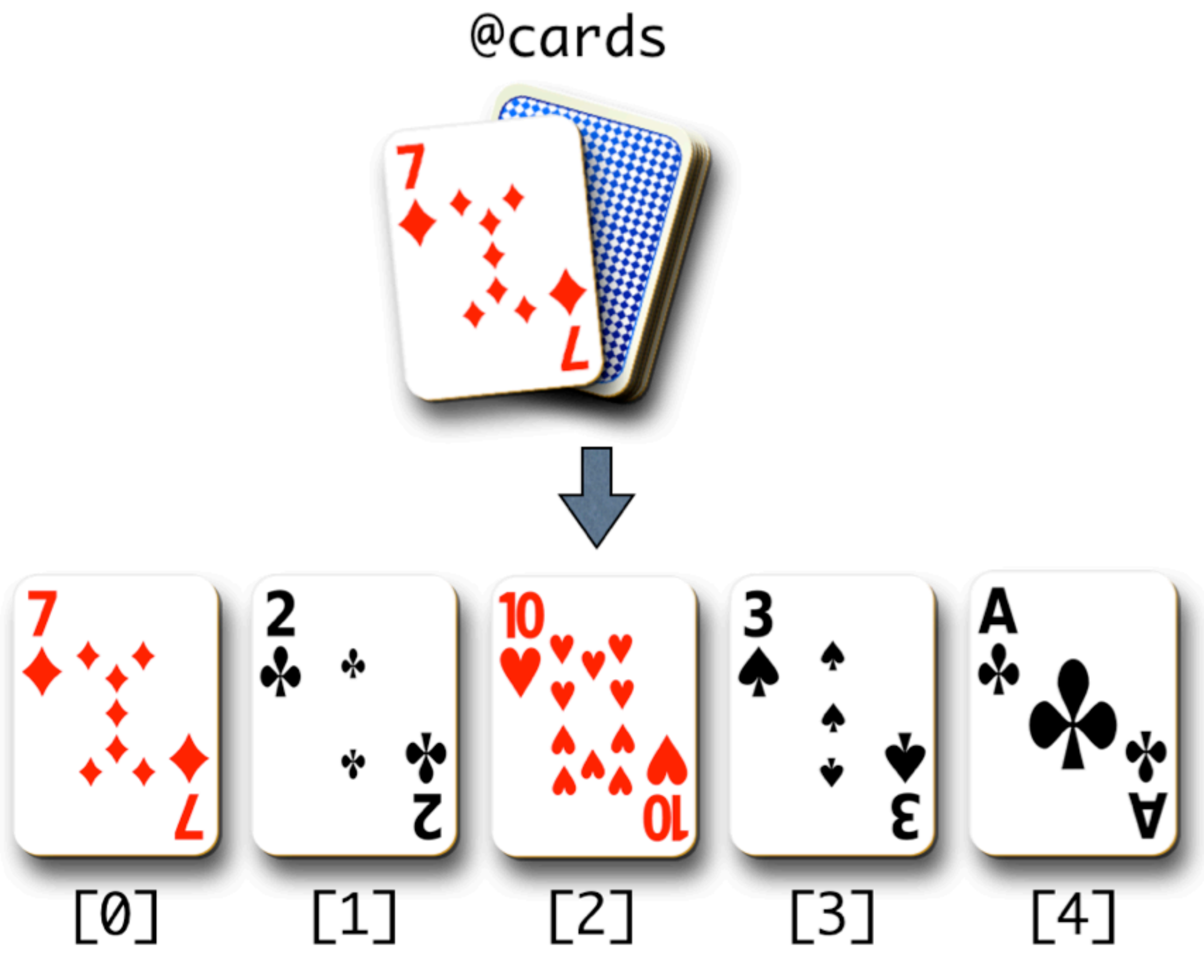
Task P8.2

Modify the original program by adding a `use warnings;` line. Run the program and observe what happens. The errors that you should see are a good thing!

P9. Arrays

Lists are useful for declaring and assigning multiple variables at once, but they are transient and if we want to store the details of a list then we have to capture all the values into separate variables. Ideally, there should be a way of referring to all of a list in one go, and there should be a way to access individual items in a list. In Perl (and in most other programming languages) we do this using *arrays*.

An array is a named list. Each array element can be any scalar variable that we have seen so far, e.g. a number, letter, word, sentence etc. In Perl, as in most programming languages, an array is indexed by integers *beginning with zero*. The first element of an array is therefore the zero-th element. This might confuse you but that's just the way it is. Arrays in Perl are named using the `@` character. Let's imagine that we have an array called `@cards` that contains five playing cards (we can imagine that each card in the array would be stored as a text string such as '7D' for 'seven of diamonds').



Array example

If we wanted to see what the individual elements of the `@cards` array were, we could access them at array positions 0 through to 4. It's important to note that arrays always have a *start* (the zero-th position), an *end* (in this case, position 4), and a *length* (in this case 5). Arrays can contain just one element in which case the start and the end would be the same. Arrays can also contain no elements whatsoever (more of that later). In biology you might frequently see arrays used to store DNA or protein sequence information. This could either be where each element is a separate DNA/protein sequence, *or* where each element is one nucleotide/amino acid and the whole array is the sequence.

Task P9.1

Create and run the following program:

```perl
1.  #!/usr/bin/perl
2.  # array.pl
3.  use strict; use warnings;
4.
5.  my @animals = ('cat', 'dog', 'pig');
6.  print "1st animal in array is: $animals[0]\n";
7.  print "2nd animal in array is: $animals[1]\n";
8.  print "Entire animals array contains: @animals\n";
```

Line 5 assigns the `@animals` array a list of 3 values. Note how we also have to declare arrays with `my` (because we are including the `use strict;` statement).

Lines 6 and 7 show how to access individual elements of an array. You specify a position in the array by putting an integer value between square brackets. The integer value is known as the ‘array index’.

Lines 6 and 7 also shows that you can interpolate individual scalars inside double quotes, i.e. Perl prints out the value stored at the specified array position rather than just printing the text `$animals[0]`.

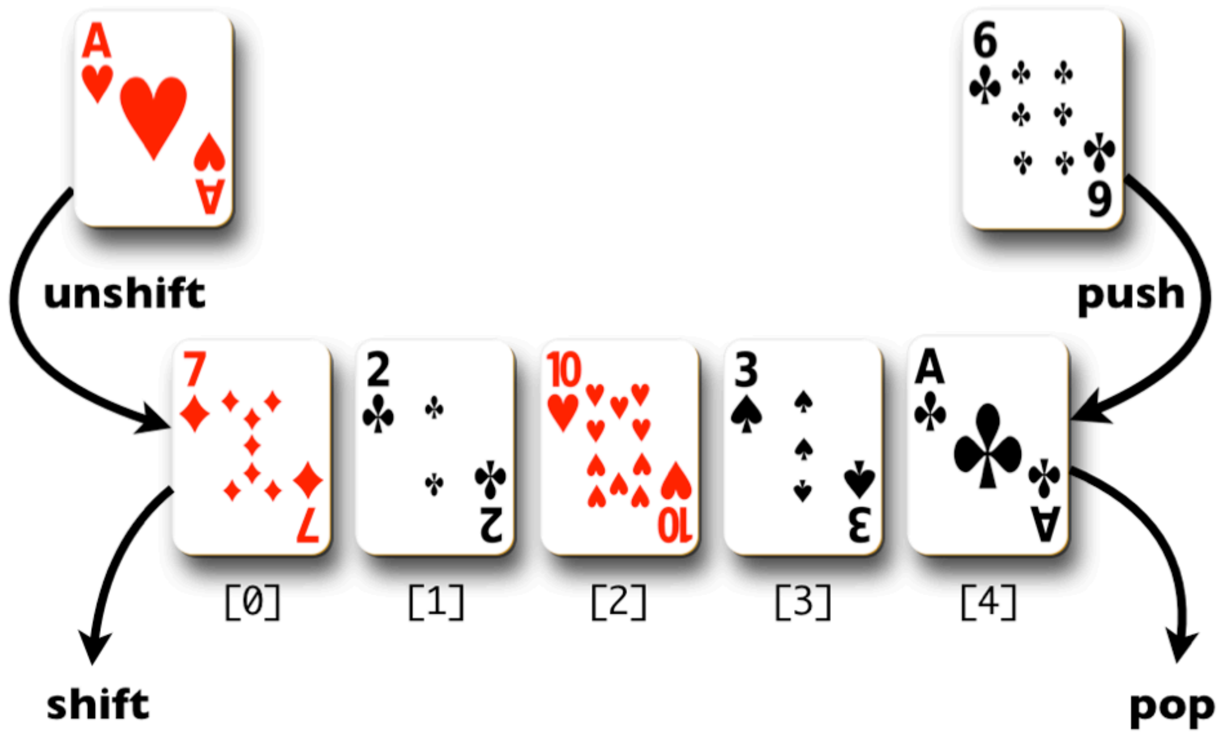
Line 8 shows that if you include an array name between double quotes, then the entire array interpolates and Perl will add spaces between each element in the printed output.

Note that each element of the list is a scalar variable. We write `$animals[0]` never `@animals[0]`. There is no such thing as `@animals[0]` in Perl. The membership of `$animals[0]` in `@animals` is shown by the square brackets. Writing `@animals[0]` is one of the most common errors of new programmers (it’s so common that it will actually be legal in the next version of Perl…). Try modifying the code to include this erroneous syntax and observe the warning message:

```perl
9.  print "@animals[0]\n"; # bad
```

Making arrays bigger and smaller

Perl arrays are *dynamic*. That is, they grow and shrink automatically as you add/remove data from them. It is very common to modify the contents of arrays, and it is also very common to start off with an array full of things, and then remove one thing at a time. Most of the time we add or remove things to either end of an array and Perl has four dedicated functions to do this:



Functions which modify arrays

Task P9.2

To examine this dynamic behavior, we will first learn to use the push() function to add some new data onto the array. The push function is used to add one thing to the end of an array. The end of an array is the element with the highest array index position. Add the following lines to your program.

```
10. push @animals, "fox"; # the array is now longer
11. my $length = @animals;
12. print "The array now contains $length elements\n";
```

Line 10 introduces a very useful concept in Perl. If you assign a list to a scalar variable, then the scalar variable becomes the *length of the list*. This is so useful that you will use it a lot in your Perl code. You can think of this in another way. Anywhere in a Perl script where it is possible to specify a numerical value, you can instead specify the name of an array. If that array contains any elements then Perl will calculate the length of the array and use the number of elements.

It is a common mistake to confuse the following two lines of code, can you work out what the difference is?

```
$length = @animals;
($length) = @animals;
```

The first line of code takes the *length* of the animals array and assigns that to the `$length` variable. But when we add parentheses around `$length` we are now making a list, and the second line of code is therefore a *list assignment*. It doesn't look much like a list because there is only one thing in it, but it is still a list. So the second line of code could be read as 'take the `@animals` array and assign all of the elements to a new list called `$length`'. Of course in this case the new list is shorter than the array so it can only receive one item. Have a look again at section P8.1 to see if that helps you understand things.

Task P9.3

Just to make sure you fully understand arrays, let's add a few more lines.

```
my ($first, $second) = @animals;
print "First two animals: $first $second\n";
my @animals2 = @animals; # make a copy of @animals
@animals = (); # assign @animals an empty list -> destroys contents
print "Animals array now contains: @animals\n";
print "Animals2 array still contains @animals2\n";
```

Common Array Functions

We already saw `push()` as a way of adding an element to the end (tail) of a list. Naturally, you can add an element to the front (head) of a list, or remove elements instead of adding them. Try modifying your program to use the following set of functions: pop(), shift(), unshift(), and if you're really brave splice(). The last function is the hardest one to understand but also the most powerful because it allows you add, remove, or substitute array elements at *any* position in the array, not just at the ends.

Function	Meaning
`push(@array, "some value")`	add a value to the end of the list
`$popped_value = pop(@array)`	remove a value from the end of the list
`$shifted_value = shift (@array)`	remove a value from the front of the list
`unshift(@array, "some value")`	add a value to the front of the list
`splice(...)`	everything above and more!

Task P9.4

Experiment with the array functions by adding some new lines to array.pl. Rather than just adding a text string to an array, try to see if you can use the `push()` or `unshift()` functions to add *variables* or even other arrays to existing arrays. For the `shift()` and `pop()` functions, try to see what happens if you don't assign the popped or shifted value to a variable. E.g. try to determine the difference between the following two lines of code:

```
my $value = pop(@array);
pop(@array);
```

More About Array Indexes

Let’s consider a couple more indexing issues. Add the following lines but before running it, try to guess what will happen.

```perl
@animals = ('cat', 'dog', 'pig'); # needed because @animals was emptied
print "Animal at array position 1.2 is $animals[1.2]\n";
print "Animal at array position 1.7 is $animals[1.7]\n";
print "Animal at array position -1 is $animals[-1]\n";
print "array length = ", scalar(@animals), "\n";
```

Floating point value such as 1.2 or 1.7 are rounded down. Using negative numbers for the array index positions have the effect of counting from the tail of the array. The `scalar()` function forces scalar context on its argument. As we know, an array gives its length in scalar context. Recall `$length = @animals`. The `scalar()` function does the same thing without the need to create an extra variable. Something else you can try is to look up an array element using a text string rather than a number. E.g. what happens if you try the following?

```perl
print "Animal at array position 'foobar' is ", $animals["foobar"], "\n";
```

You could substitute “foobar” for any text at all. The first thing that you should notice is that the Perl program should give you a useful warning message:

Argument “foobar” isn’t numeric in array element at…

Strings such as “foobar” have a numeric value of zero and so if you use any text instead of a number when trying to lookup a specific position in an array, you will always get the first (zero-th) element. Hopefully you will never try doing this.

P10. From strings to arrays and back

We saw that if we try printing an array between double quotes, then Perl interpolates the array and prints each element separated by a space. What if we want something other than spaces? In Perl, there's always more than one way to do things, but the best way is with the join() function, which allows you to create a string from an array and put whatever you want between the elements of the array.

Task P10.1

Let's say we want to create a CSV (comma separated values) format from an array of gene names from the nematode *Caenorhabditis elegans*. Here's how we could do that.

```
1.  #!/usr/bin/perl
2.  # stringarray.pl
3.  use strict; use warnings;
4.
5.  my @gene_names = qw(unc-10 cyc-1 act-1 let-7 dyf-2);
6.  my $joined_string = join(", ", @gene_names);
7.  print "$joined_string\n";
```

Line 5 uses the qw() function to make an array. `qw()` is short for 'quote words'. It's a little shorthand so that we don't have to keep typing quotation marks.

Line 6 creates a string from an array with `join()`, and specifies that each element of the array should be joined with a comma followed by a space.

The opposite function of `join()` is the split() function. This divides a string into an array. But we have to tell it where to split. This works sort of like a restriction digest but the restriction site is consumed in the process.

Task P10.2

Add the following lines to your program and run it:

```
8.  my $dna = "aaaaGAATTCttttttGAATTCggggggg";
9.  my $EcoRI = "GAATTC";
10. my @digest = split($EcoRI, $dna);
11. print "@digest\n";
```

If we want to convert a string into an array and split the string at every possible position, we need to use an empty string ("") in the `split()` function. This is often used to convert DNA/protein sequences stored in variables into arrays:

```
12. my @dna = split("", $dna);
13. print "@dna\n";
```

P11. Sorting

As in real life, lists are great, but sorted lists are even better. Imagine looking through a telephone book if it wasn't sorted… tedious. Perl has an incredibly flexible sorting function. But it's a little complicated, so you may want to come back and read this part again later.

Task P11.1

Create the following program and run it. How does Perl sort items in a list?

```
1.  #!/usr/bin/perl
2.  # sorting.pl
3.  use strict; use warnings;
4.
5.  my @list = qw( c b a C B A a b c 3 2 1); # an unsorted list
6.  my @sorted_list = sort @list;
7.  print "default: @sorted_list\n";
```

Line 6 calls the sort() function. This could have been written with parentheses (e.g. `= sort(@list)`), but this is one of those cases where parentheses are usually omitted. We assign the result of the sort to a new array, but we could have also overwritten the original array, e.g.

```
my @list = sort @list;
```

Looking at the output, it should be clear that Perl sorts by ASCII value by default. It is using the `cmp` operator we saw earlier. What if you want to sort numerically? Then you would have to use the numeric comparison operator `<=>`. To specify this, you use an unfamiliar syntax:

```
8.  @sorted_list = sort {$a <=> $b} @list;
9.  print "numeric: @sorted_list\n";
```

In general, sorting routines compare pairs of values. In Perl, these values are held by the magic variables `$a` and `$b`. For this reason, you should not use these variable names in your own programs. Line 8 shows that `$a` and `$b` are compared numerically. The default sort is simply `{$a cmp $b}`.

Because we have the `use warnings;` statement, this code should produce a few warning messages. This is because we are asking to sort values numerically but 'A', 'B', 'C' etc are not numbers. As we saw previously, text has a numeric value of zero. So if you compare text as numbers, it does not sort alphabetically (and Perl warns us of this fact).

If you want to sort in reverse direction, you simply exchange the variables `$a` and `$b`.

```
10. @list = qw (2 34 -1000 1.6 8 121 73.2 0);
11. @sorted_list = sort {$b <=> $a} @list;
```

```
12; print "reversed numeric: @sorted_list\n";
```

What if you want to sort both numerically and alphabetically and you want no differentiation between capitals and lowercase? Perl can do this, of course, but the explanation will be left for later.

```
@sorted_list = sort {$a <=> $b or uc($a) cmp uc($b)} @list;
print "combined: @sorted_list\n";
```

P12. Loops

Loops are one of the most important constructs in programming. Once you have mastered loops, you can do some really useful programming. Loops allow us to do things like count from 1 to 100, or cycle through each element in an array, or even, process every line in an input file. There are three main loops that you will use in programming, the `for` loop, the `foreach` loop, and the `while` loop.

The `for` Loop

The `for` loop generally iterates over integers, usually from zero to some other number. You can think of the integer as a 'loop counter' which keeps track of how many times you have been through the loop (just like a lap counter during a car race). The `for` loop has 3 components:

1. initialization — provide some starting value for the loop counter
2. validation — provide a condition for when the loop should end
3. update — how should the loop counter be changed in each loop cycle

If we return to the car race analogy, we can imagine a car having to drive 10 laps around a circular track. At the start of the race the car has not completed any laps so the loop counter would be initialized to zero. The race is clearly over when the counter reaches 10 and each lap of the track updates the counter by 1 lap.

Task P12.1

Create and run the following program.

```perl
#!/usr/bin/perl
# loop.pl
use strict; use warnings;

for (my $i = 0; $i < 10; $i = $i + 1) {
    print "$i\n";
}
```

The syntax for a `for` loop requires the three loop components to be placed in parentheses and separated with semi-colons. Curly braces are then used to write the code that will be executed during each iteration of the loop. This code is usually indented in the same way that we indent blocks of code following if statements. In this loop we first declare a new variable `my $i` to act as our loop counter. It is a convention in programming to use `$i` as a loop variable name because of the use of `i` as a counter in mathematical notation, e.g.

$$\sum_{i=0}^{n} x_i$$

mathematical summation

You could name your loop counter anything that you wanted to, but we suggest that for now you just use `$i`. Let's see what the three components of our loop are doing:

- `$i = 0` — performs initialization, i.e. start our loop with `$i` equal to zero
- `$i < 10` — performs validation, i.e. keep the loop going as long as `$i` is less than 10
- `$i = $i + 1` — performs the update, `$i` is incremented by one during each loop iteration

It is very common in Perl that you want to take a number and just add 1 to it. In fact, it is so common that Perl has its own operator to do it, the increment operator. Here's how you could increment the value of `$i` by 1:

```
$i++
```

This is more succinct and is the common way to increment a variable by one. Not surprisingly, you can also decrement a variable by one:

```
$i--
```

Note that the 'update' component of the loop should describe a way of increasing (or decreasing) the value of `$i` otherwise the loop would never end.

Task P12.2

Try looping backwards and skipping:

```
8.  for (my $i = 50; $i >= 45; $i--) {print "$i\n"}
9.  for (my $i = 0; $i < 100; $i += 10) {print "$i\n"}
```

Since the blocks of code following these `for` loops are only one line long, they do not need a semi-colon at the end. We saw this earlier when making conditional statements tidy.

Line 9 uses the `+=` operator. This is a useful shortcut and in this case the result is exactly the same as if we had typed `$i = $i + 10`. Similar operators exist for subtraction `-=`, multiplication `*=` etc. Note how the loop in line 9 is counting in tens and not incrementing by one at a time.

Task P12.3

Let's do something a little bit useful with a loop. This program computes the sum of integers from 1 to n, where n is some number on the command line. Of course you could compute this as (n+1) * n / 2, but what is the point of having a computer if not to do brute force computations?

```
#!/usr/bin/perl
# sumint.pl
use strict; use warnings;

die "usage: sumint.pl <limit>\n" unless @ARGV == 1;
my ($limit) = @ARGV;
```

```
7.  my $sum = 0;
8.  for (my $i = 1; $i <= $limit; $i++) {$sum += $i}
9.  print "$sum\n";
```

Line 5 contains a usage statement. We saw this earlier in Project 1 and in this script it is just adding a check to ensure that we specify one (and only one) command-line argument when we run the script. Line 6 assigns the specified command-line argument to `$limit`. Line 7 creates a variable to hold the sum. Line 8 uses a loop to add the latest value of `$i` to the `$sum` variable.

Task P12.4

Write a program, factorial.pl, that computes the factorial of a number. Structurally, it will be very similar to sumint.pl, but of course you will be multiplying values instead of adding.

Task P12.5

One of the most common operations you will do as a programmer is to loop over arrays. Let's do that now. To make it interesting, we will loop over two arrays simultaneously:

```
#!/usr/bin/perl
# loops.pl
use strict; use warnings;

my @animals = qw(cat dog cow);
my @sounds  = qw(Meow Woof Moo);
for (my $i = 0; $i < @animals; $i++) {
    print "$i) $animals[$i] $sounds[$i]\n";
}
```

The `for` loop starts at 0, which is where all arrays start, and continues as long as the loop variable `$i` is less than the length of the array (which is found from the scalar context of an array).

The `foreach` Loop

The `foreach` loop allows you to iterate through the contents of an array without a numeric index. Instead, a temporary variable is set to the contents of each element. Add the following code to your program.

```
foreach my $animal (@animals) {
    print "$animal\n";
}
```

In this loop, `$animal` is the temporary variable. It changes from cat to dog to cow with each iteration of the loop. It is common to name the temporary variable as a singular form of the array name. E.g.

```
foreach my $protein (@proteins) { ... }
foreach my $car (@cars) { ... }
foreach my $knight_who_say_Ni (@knights_who_say_Ni) { ... }
```

You can also use the `foreach` loop in a numeric manner. If you are a lazy typist (potentially an admirable quality if you are concerned about RSI), you can even use `for` rather than `foreach`. Line 13 shows how to create to create a numeric list with the range operator `..`. This can be used to loop over letters or numbers:

```
13. for my $i (0..5) {print "$i\n"}
```

The `while` Loop

The `while` loop continues to iterate as long as some condition is met, where the condition is some notion of True or False. The 'condition' part of a while loop can be as simple or as complex as you want it to be. Here is an example of a very simple while loop which keeps doubling a number until some limit is reached:

```
14. my $x = 1;
15. while($x < 1000){
16.     print "$x\n";
17.     $x += $x;
18. }
```

In this example the code will continue to loop while the value of `$x` is less than 1000, and `$x` is doubled for each iteration of the loop. It is important that the test condition will be testing something that is going to change. But Perl will allow you to write code which contains a pointless test condition.

Task P12.6

Add these lines to your program and run it:

```
19. while (0) {
20.     print "this statement is never executed because 0 is false\n";
21. }
22. while (1) {
23.     print "this statement loops forever\n";
24. }
```

The first while loop will never print anything at all because a zero value is always treated as false by Perl. So the loop will run only while the value of zero evaluates to true which is never going to happen.

The second loop will start but never end because the test condition ('while 1 is true') is always true. In fact, anything which isn't a zero or the null string ("") will always evaluate as true. To stop this program, press Control+c in your terminal. This sends the Unix 'interrupt' signal to the program (you might want to commit that trick to memory).

Let's try looping through an array with a `while` loop. Replace lines 10–12 with these.

```
10. while (@animals) {
11.     my $animal = shift @animals;
12.     print "$animal\n";
13. }
```

In each iteration through the loop, the array `@animals` is shortened by removing one item from the front of the list (using the shift function). The loop ends when the length of the array is 0 (empty). There are times when this kind of array-deletion construct is useful, but most of the time you will be looping through arrays with for or foreach.

The `do` Loop

The `do` loop is a variation of the `while` loop. Unlike the `while` loop, it always executes at least once. `do` loops are not so common.

```
do {
    print "hello\n";
} while (0);
```

Congratulations! You have now learned about variables, numbers, math, strings, conditionals, arrays, and loops. Even though there is still a lot to learn, you have come a long way. You can now write some very useful programs.

Loop Control

There are times when you will want a little more control in your loops. The next keyword keyword immediately restarts the loop at the top and advances the loop variable. The redo keyword restarts the loop also, but does not advance the loop variable. The last keyword terminates the entire loop.

P12.7

Here is a program that illustrates `redo` and `last`. It computes the prime numbers between 100 and 200.

```
1.  #!/usr/bin/perl
2.  # primes.pl
3.  use strict; use warnings;
4.
5.  my $n = 0;
6.  while (1) {
7.      $n++;
8.      redo if $n < 100;
9.      last if $n > 200; # breaks out of while loop
10.
11.     my $prime = 1; # assumed true
12.     for (my $i = 2; $i < $n; $i++) {
13.         if ($n % $i == 0) {
14.             $prime = 0; # now known to be false
15.             last; # breaks out of for loop
16.             }
17.     }
18.
19.     print "$n\n" if $prime;
20. }
```

Line 8 uses the `redo` keyword. This short-circuits the `while` loop as long as `$n` is less than 100. You could have used `next` here also because there is no loop variable.

Line 9 uses the `last` function to terminate the while loop, effectively ending the program, if `$n` is greater than 200.

Lines 11–17 determine if a number is prime. This method starts off assuming `$n` is prime. It then checks all the numbers between 2 and `$n - 1` to determine if `$i` is a factor of `$n`. If `$i` is a factor of `$n` (line 11) then there is no point in calculating any further because `$n` is not prime. So `$prime` is set to false (line 14) and the `for` loop is terminated (line 15).

When to use each type of loop?

There will be situations where you can use different types of loop structure to achieve exactly the same goal for a program. Conversely there are times when only one type of loop will do. It might not always be clear to you how to make the correct choice, but with practice it becomes more obvious. Feel free to experiment with different loop structures to see what works and what doesn’t.

Project 2: Descriptive statistics

In this project, you will write a program that computes typical descriptive statistics for a set of numbers. Here are the first few lines. Your program should compute the count, sum, minimum, maximum, median, mean, variance, and standard deviation. It should report these in some pleasing format. Of course, it should have a usage statement like all good programs. Here are the first few lines.

```perl
#!/usr/bin/perl
# stats.pl by ___
use strict; use warnings;

die "usage: stats.pl <number1> <number2> <etc>\n" unless @ARGV > 1;
```

Count, Sum, and Mean

We already know how to do these.

Min, Max, and Median

The median value is at the middle of the sorted list of values. If the list has an even number of elements, then the median is the average of the two at the middle. The minimum and maximum are easily found from the sorted array.

Variance

Variance is the average squared difference from the mean. So compute the mean first and then go back through the values, find the difference from the mean, square it, and add it all up. In the end, you divide by `n` or by `n - 1` depending on if you are computing the population or sample variance.

Standard Deviation

Simply the sqrt() of the variance.

Project 3: Sequence shuffler

In this project, you will create a program that randomly shuffles a DNA sequence (or any text really). Shuffling is a really useful way to provide a null model. For example, suppose you do a sequence alignment and get a score of 30. Is this a good score? How often does a score of 30 happen by chance? To determine this, you could randomly shuffle your sequence and perform the search again (and again, and again…).

There are a variety of ways to shuffle a sequence. One way is to repeatedly exchange randomly selected pairs of letters. Another way is to remove a random letter from one sequence to build up another sequence. Random numbers can be generated with the `rand()` function which we first saw back in lesson P4.2.

As usual, your program should have a usage statement.

Strategy 1

1. Turn the DNA string into an array with `split()`
2. Use a `for` loop to perform the following procedure many times
    1. Select a random position A with `rand()`
    2. Select a random position B with `rand()`
    3. Exchange the letters at indices A and B
3. Print the now shuffled array

Strategy 2

1. Create a new, empty array to hold the shuffled sequence T
2. urn the DNA string into an array with `split()`
3. Use a `while` loop for as long as the original array exists
    1. Select a random position A in the original array with `rand()`
    2. Remove the letter at index A from the original array with `splice()`
    3. Add the letter to the shuffled array with `push()`
4. Print the new shuffled array

P13. File I/O

Our programs so far have taken arguments on the command line. But that is not a very common way to receive data. People generally hand you a *file* (or more commonly lots of files). Fortunately, reading files in Perl is incredibly simple.

Task P13.1

Create the program below and when you run it, specify the name of a text file on the command line after the program name. E.g.

```
$ cd
$ linecount.pl Data/Misc/oligos.txt
```

Remember, you can be in *any* directory on your computer when you run this script, but you will always need to specify where the oligos.txt file is *in relation to where you are* (using a relative or absolute path). After you run this script with the oligos.txt file, try running it against several files at once (by putting multiple file names on the command line).

```
1.  #!/usr/bin/perl
2.  # linecount.pl
3.  use strict; use warnings;
4.
5.  my $lines = 0;
6.  my $letters = 0;
7.  while (<>) {
8.      $lines++;
9.      $letters += length($_);
10. }
11. print "$lines\t$letters\n"; # \t is a tab character
```

Line 7 contains something you haven't seen before. This is the `<>` file operator. By default, this reads *one line at a time* from the file specified on the command line. If there are multiple files on the command line, it will read them all in succession. It even reads from STDIN (standard input) if you include it in a pipe. True Perl magic!

The default variable `$_`

Line 9 is our first introduction to the default variable `$_`. Perl automatically assigns data to this this variable in some settings. In this example, `$_` will contain each line of the file. Although you don't see it, Perl is actually performing the following operation.

```
5.  while ($_ = <>) {
```

If you find this confusing, you can use a named variable of your choice instead of using `$_`:

```
while (my $line = <>) {
```

But you should get used to using `$_` because it is so common among Perl programs. Perl can also retrieve `$_` by default in many functions. For example, if you try using the `print()` function without any arguments, it will report the contents of `$_`. The following one-line program simply prints out the the contents of any file specified on the command-line:

```
while (<>) {print}
```

You can use `$_` in loops too, but we prefer not to. Here is another one-liner in which `$_` is used in place of a named loop variable:

```
for (0..5) {print}
```

Confusing? Yes, a little. But you do get used to it. For now, feel free to name all your variables. By the way, in addition to `$_`, there are a large number of other *special variables* with equally strange symbols.

The `open()` Function

There are times when you have several files and you don't want to read them all one after the other. For example, one might be a FASTA file and the other GFF. You wouldn't want to process both files with the same code. To open and read a single file, you use the open() function. This will open a file for reading or writing, but not both. Let's see how we can use it.

Task P13.2

Create the following program. This will read the contents of a file that you specify on the command line and then create a second file with slightly altered contents.

```
1.  #!/usr/bin/perl
2.  # filemunge.pl
3.  use strict; use warnings;
4.
5.  open(IN, "<$ARGV[0]") or die "error reading $ARGV[0] for reading";
6.  open(OUT, ">$ARGV[0].munge") or die "error creating $ARGV[0].munge";
7.  while (<IN>) {
8.      chomp;
9.      my $rev = reverse $_;
10.     print OUT "$rev\n";
11. }
12. close IN;
13. close OUT;
```

Lines 5 and 6 contain `open()` statements for reading and writing. IN and OUT are called filehandles. These are special variables used only for file operations. The second argument determines if the `open()` statement is for reading or writing. `<` is for reading and `>` is for writing. This should look familiar from your Unix lessons. If you do not include `<` or `>`, then the file is opened for reading. Both of the `open()` statements include an additional `or` clause in case of failure. We will talk more about this later. Line 7 should look a little familiar. Instead of using `<>` by itself, there is a named filehandle inside the brackets. Only the file associated with `IN` will be read. The file in question does not need to be stored in a variable, but usually is (e.g. you could open a file called 'sesame.txt' with `open(IN, "sesame.txt")`.

Line 8 introduces the chomp() function. This removes a `\n` character from the end of a line if present. It is quite common to chomp your `$_`.

Line 9 uses the reverse() function to reverse the contents of `$_`. We haven't seen the `reverse()` function before. It reverses both strings and arrays.

Lines 12 and 13 close the two filehandles. You should always get into the habit of making sure that every `open()` function has a matching `close()` function. It is possible that bad things will happen if you don't close a filehandle. You should also try to close a filehandle at the first opportunity when it is safe to do so, i.e. as soon as you are finished with reading from, or writing to, a file.

Naming filehandles - part 1

Filehandles are typically given upper-case names. You can use lower-case names and your script will probably still work but Perl will also print out a warning. If you only ever read from one input file and write to one output file then IN and OUT are typical filehandle names, though feel free to name them whatever you feel is most suitable (INPUT, DATA etc.). If you need to read from multiple files then it might be a good idea to use filehandle names that describe the type of data, e.g. GFF or FASTA.

Naming filehandles - part 2

The way in which we have just explained how to name and create filehandles is an older style which has become less common in recent years due to some changes that Perl made (yes, even programming languages have styles that come and go!). Perl now allows you to use a regular scalar variable as a filehandle. This may or may not be more intuitive to you. Here is the code from task P13.2 rewritten to use the newer style of filehandle:

```perl
#!/usr/bin/perl
# filemunge.pl
use strict; use warnings;

open(my $in, "<$ARGV[0]") or die "error reading $ARGV[0] for reading";
open(my $out, ">$ARGV[0].munge") or die "error creating $ARGV[0].munge";
while (<$in>) {
    chomp;
    my $rev = reverse $_;
    print $out "$rev\n";
}
close $in;
close $out;
```

As you can see, the only differences are that `IN` and `OUT` have been replaced by the scalar variables `$in` and `$out`. If you work with other people's Perl code, you might see examples of the older style filehandle so it is good to know about them both, though we prefer the newer style.

Different ways of creating filehandles

As well as having two different ways of naming filehandles, Perl also allows you to create them using a couple of different methods. So far we have seen the *two-argument* method, where the two arguments in question were:

1. the filehandle name (e.g. `IN` or `my $in`)
2. the file name and the read/write status (using `<` or `>`)

If you are open a file to read from it, then the `<` part (technically known as the *file mode*) is not actually necessary. This means that the following are considered identical by Perl:

```
open(my $in, "<input.txt");  # with file mode (<)
open(my $in,  "input.txt");  # without file mode
```

Some people prefer it when things are more explicit (i.e. when you always have to specify the read/write permission) and so Perl also allows you to use a *three-argument* mode for creating filehandles. In this syntax, you *must* specify the read/write permission as the second, of three, arguments. E.g.

```
open(my $in,  "<", "input.txt");    # read from file
open(my $out, ">", "output.txt");   # write to file
```

This is the syntax we suggest you use as it more obvious when you are reading from, and when you are writing to, a file. It can be bad to accidentally write to a file when you were expecting to read from it, as you will overwrite the file!

P14. Hashes

A *hash* is also called a dictionary or associative array. It is very similar to the kind of array we saw earlier except that instead of indexing the array with integers, the array is indexed with text. The dictionary analogy is fitting. A word is an index to its definition. A hash can be created in list context, just like an array. But since we need to provide the text index, it is necessary to provide *key, value* pairs.

Task P14.1

Create the following program. We have not seen the `%` sign in front of a variable before. This is symbol for a hash variable. If we are including the `use strict;` statement (which we *always* should be doing) then we will also need to declare any hashes with `my`.

```
1.  #!/usr/bin/perl
2.  # hash.pl
3.  use strict; use warnings;
4.
5.  my %genetic_code = ('ATG', 'Met', 'AAA', 'Lys', 'CCA', 'Pro');
6.  print "$genetic_code{'ATG'}\n";
```

Notice that when you want to access a value from a hash, you use curly brackets (`{` and `}`) rather than square brackets (`[` and `]`). Curly brackets lets Perl know you are accessing a hash rather than an array. You could therefore have variables named `$A`, `@A`, and `%A`, and they would all be different variables. Note that using the same name for different things in this way, would be considered bad programming style. `$A` is scalar. `$A[0]` is the first element of the `@A` array. `$A{'cat'}` is the value for the 'cat' key of the `%A` hash.

When declaring hashes, there is an alternative syntax that makes the assignments more obvious. Here, we replace the comma between the key and the value with a kind of arrow `=>`. This reads as 'gets'. Or alternatively 'says'. So `'cat' => 'meow'` reads as "cat says meow". Let's change line 5 to use this alternative syntax:

```
5.  %genetic_code = ('ATG' => 'Met', 'AAA' => 'Lys', 'CCA' => 'Pro');
```

This syntax looks even more logical when the hash assignment is split across multiple lines:

```
5.  %genetic_code = (
6.      'ATG' => 'Met',
7.      'AAA' => 'Lys',
8.      'CCA' => 'Pro',
9.  );
10. print "$genetic_code{'ATG'}\n";
```

The last comma in line 8 is unnecessary, but it does no harm, and we like tidy, consistent code. It turns out that when using the `=>` syntax, Perl knows that the you are assigning a hash, so the quotes around the keys are actually unnecessary:

```
5.  %genetic_code = (
6.      ATG => 'Met',  # single quotes now removed from keys
7.      AAA => 'Lys',
8.      CCA => 'Pro',
9.  );
10. print "$genetic_code{'ATG'}\n";
```

The quotes on the values are absolutely required in this example because the values are strings. You would not need them if the values were numbers.

Keys and Values

It's a simple matter to iterate through arrays because they have numeric indices from 0 to one less than the array size. For hashes, we must iterate over the keys, and for that, we need the various strings. Not surprisingly, this is performed with the keys() function.

Task P14.2

Add the following code to your program to report the keys and corresponding values from your hash. It is sometimes common to use the variable name `$key` in a `foreach` loop, although in this example `$codon` may also be a suitable choice.

```
11. foreach my $key (keys %genetic_code) {
12.     print "$key $genetic_code{$key}\n";
13. }
```

The `keys()` function returns an array of keys. Each key in turn is then assigned to a temporary variable as part of a `foreach` loop (see section P12 for a refresher on loops). A function that is related to the `keys()` is values(). This function returns an array of values. Add the following lines to your program to observe this more explicitly:

```
14. my @keys = keys(%genetic_code);
15. my @vals = values(%genetic_code);
16. print "keys: @keys\n";
17. print "values: @vals\n";
```

Hashes store key-value pairs in a semi-random order (it's not random, but you have no control over it). So you will often want to use the `sort()` function to sort the keys that you extract from the hash. Replace line 11 with the following.

```
11. foreach my $key (sort keys %genetic_code) {
```

Adding, Removing, and Testing

Recall that for arrays, you generally either `push()` or `unshift()` to add new values to an array. You can also assign a value at an arbitrary index such as `$array[999] = 5`. Adding pairs to a hash is similar to assigning an arbitrary index. If you assume the key exists, Perl will create it for you. But watch out, if you use a key that previously existed, the value will be overwritten.

Task P14.3

Modify your program to include the following statements:

```
18. $genetic_code{CCG} = 'Pro';
19. $genetic_code{AAA} = 'Lysine';
```

Line 18 adds a new key `CCG` to the hash. Note that the value of this key `Pro` already exists as the value to *another* hash key `CCA` (this is not a problem). Line 19 reassigns the value that the `AAA` key points to (`Lysine` instead of `Lys`). Sometimes you may want to ask if a particular key already exists in a hash, for example, before overwriting something. To do this, you use the exists() function:

```
20. if (exists $genetic_code{AAA}) {print "AAA codon has a value\n"}
21. else                           {print "No value set for AAA codon\n"}
```

To remove both a key, and its associated value, from a hash, you need to use the delete() function:

```
22. delete $genetic_code{AAA};
```

Use suitable `print` statements to check that you correctly added and removed new key-value pairs in your hash.

Summary of hash-related functions

Function	Meaning
`keys %hash`	returns an array of keys
`values %hash`	returns an array of values
`exists $hash{key}`	returns true if the key exists
`delete $hash{key}`	removes the key and value from the hash

Hash names

If you work with a lot of hashes, it can sometimes help to make the hash name explain something about the data it contains. Hashes typically link pairs of connected data, e.g. name of sequence, and GC% content of that sequence; name of a politician, and the number of votes that they received. Based on these examples, which of the following hash names do you think best describe the data that they contain?

```
%seq;
%sequences;
%sequence_details;
%sequence2gc;
%sequence_to_gc;

%vote;
%names;
%name2votes;
%name_to_votes;
```

Bad names for hashes include:

```
%hash;
%data;
%stuff;
%things;
%Perl_is_awesome; # but bonus points for enthusiasm!
```

P15. Organizing with hashes

Task P15.1

Examine the `Data/Misc/oligos.txt` file. This is a file containing the names and sequences of some oligos separated by tabs. Suppose you want to calculate the melting temperature (Tm) of each oligo and then print out the oligos ordered by their Tm. We can do this by using two hashes, one will store the sequences and the other will store the Tms. Both hashes will be indexed by the oligo name (i.e we will use the same keys for both hashes).

Note the use of comments and whitespace in this script. As we discussed previously, these help the readability of the program. The “header” is lines 1–3. Line 5 is by itself because it is functionally distinct. Line 7 declares the two hashes that we will use. Lines 9–24 are the main body of the program. Whitespace and comments further refine these sections to their purpose. Lines 26–29 are for output. Try to follow a similar logical structure in your own programs. Line 27 may be incomprehensible at first. If you don’t get it, don’t sweat it.

```
1.  #!/usr/bin/perl
2.  # oligo.pl by ___
3.  use strict; use warnings;
4.
5.  die "usage: oligo.pl <file of oligos>\n" unless @ARGV == 1;
6.
7.  my (%sequences, %tm); # declare two hashes
8.
9.  # process file line-by-line
10. while (<>) {
11.     chomp;
12.
13.     # store sequence
14.     my ($name, $seq) = split("\t", $_);
15.     $sequences{$name} = $seq; # first hash assignement
16.
17.     # calculate and store Tm
18.     my $A = $seq =~ tr/A/A/;
19.     my $C = $seq =~ tr/C/C/;
20.     my $G = $seq =~ tr/G/G/;
21.     my $T = $seq =~ tr/T/T/;
22.     my $tm = 2 * ($A + $T) + 4 * ($C + $G); # simple Tm formula
23.     $tm{$name} = $tm; # second hash assignment
24. }
25.
26. # report oligos sorted by Tm
27. foreach my $name (sort {$tm{$a} <=> $tm{$b}} keys %tm) {
28.     print "$name\t$tm{$name}\t$sequences{$name}\n"; # $name as used as key in 2 hashes
29. }
```

P16. Counting codons with `substr()`

Task P16.1

The substr() function is useful for extracting a sub-string from a string. In bioinformatics we often want to extract a part of an amino acid or nucleotide sequence. Here is a little program that shows how `substr()` works. Note the 3 arguments that the `substr()` function requires:

1. the string that you want to extract from,
2. an offset (starting from zero)
3. the length for how many characters to extract.

This program should just print `MRVLK ... TVLSAPAKIT`:

```perl
 #!/usr/bin/perl
 # substr.pl
 use strict; use warnings;

my $seq = "MRVLKFGGTSVANAERFLRVADILESNARQGQVATVLSAPAKIT";
my $first5 = substr($seq, 0, 5);
my $last10 = substr($seq, length($seq) - 10, 10);
print "$first5 ... $last10\n";
```

Task P16.2

Now let's do something useful and determine the codon usage for a sequence that is provided on the command line.

```perl
#!/usr/bin/perl
# codon_usage.pl by ___
use strict; use warnings;

die "usage: codon_usage.pl <sequence>\n" unless @ARGV == 1;
my ($seq) = @ARGV;

my %count = ();  # individual codons
my $total = 0;   # total codons

# extract each codon from the sequence and count it
for (my $i = 0; $i < length($seq); $i += 3) {
    my $codon = substr($seq, $i, 3);
    if (exists $count{$codon}) {$count{$codon}++}
    else                       {$count{$codon} = 1}
    $total++;
}

# report codon usage of this sequence
foreach my $codon (sort keys %count) {
    my $frequency = $count{$codon}/$total;
    printf "%s\t%d\t%.4f\n", $codon, $count{$codon}, $frequency;
}
```

Note that on line 8 we use a slightly different way of introducing a hash. The following lines of code are similar:

```
my %count;
my %count = ();
```

The first example declares a new hash, and the second example additionally initializes the hash which means it will empty the hash of any data (if any existed).

You may have noticed that line 20 adds a second `my $codon =` statement to the script. It is important to realize that the `$codon` within this second `foreach` loop is completely different to the `$codon` that exists in the previous `for` loop (lines 12–17). If this seems confusing, then you will have to wait a little longer before we give the full explanation for this. If it bothers you, then feel free to rename the second `$codon` variable to something else.

Line 22 introduces the printf() function to format the output. The `printf()` function has a somewhat arcane syntax handed down from the C programming language, and uses the following special characters:

- `%s` means string
- `%d` means digit (integer)
- `%f` means floating point

When using this function, you first specify how you want to format the list of things that you want to print (this part is between quotation characters). You then specify a list of variables that contain the data (or you could put the actual data here, rather than use variables). So on line 22 we print three things (separated by tab characters) as follows:

- `%s` will print the string contained in `$codon`
- `%d` will print the digit contained in `$count{$codon}`
- `%.4f` will print the floating point number in `$frequency` to 4 decimal places.

Project 4: The name game

Have you ever wondered which protein sequences might contain your name? If you have a short name, and your name does not contain the letters B, J, O, U, X, or Z, then it may occur in many different protein sequences. If not, you may need to alter your name a little or try searching for questions (e.g. 'WHATISTHEANSWER').

Goal

Write a program that reports the names of sequences matching your name (or some other word). The usage statement for your program should look something like this:

```
usage: name_search.pl <protein file> <name>
```

Details

The file of proteins you will use is called `At_proteins.fasta`. You will find this in the `Data/Arabidopsis` directory. This is a typical FASTA formatted file and contains entries with sequences that span multiple lines. Ideally, a good Perl script would be able to look for patterns across multiple lines, but to make things a little easier we will first modify this file in order to make the sequence for each entry only span one line. Later on, you might want to find out about Perl modules like FAlite.pm that can help you more easily read FASTA files, but for now we will revisit some of our Unix skills and make a new version of this file:

```
cat At_proteins.fasta | sed 's/\(^>.*\)/!\1!/' | tr '\n' '@' | tr '!' '\n' | sed 's/@//g' > At_pro
```

This is a bit of hack and not something that you should assume will work on all FASTA files (particularly if they include characters like `!` and `@`). The result of this operation should be that we create a new file (`At_proteins_v2.fasta`). The first line of this new file is a FASTA sequence definition. The second line now has the entire protein sequence and this pattern repeats for subsequent lines.

If your program works as intended, then you should be able to run it as follows and see the following output:

```
$ name_search.pl At_proteins_v2.fasta KEITH
AT5G58710.1
```

Project 5: K-mer analysis

In this project, you will write a program to investigate nucleotide patterns in introns. This project is inspired by the Rose *et al.* 2008 paper which showed that introns near the promoter are compositionally distinct from introns farther downstream. This leads to a gene expression phenomenon known as 'intron-mediated enhancement' or IME.

We will make a program that will compare *Arabidopsis* introns that occur at different positions within a gene in order to determine just how different 'early' introns are compared to 'late' introns, in terms of their nucleotide composition.

K-mers

K-mer is another name for oligo. It's just a string/word of some fixed length. If we have the sequence `ATGCGA` there are four possible k-mers of length 3: ATG, TGC, GCG, and CGA. Unlike codons, which skip every 3 nt, k-mers generally step by ones rather than threes. Codons are also strand-specific, while k-mers can be counted on one or both strands.

Coding

This project will use the `intron_IME_data.fasta` file in the `Data/Arabidopsis` directory. However, this is a multi-line FASTA file and you will first need to make a new FASTA file that rearranges each sequence to occupy only one line (see Project 4 for how to do this).

Your program should have the following structure:

1. Provide a typical command-line interface allowing the user to choose the value for k.
2. Create two hashes to store the k-mer counts for 1st introns and other introns. You might name these `%count1` and `%count2`.
3. Read a definition line from your new FASTA file. Extract the intron number from the definition line. First introns will be labeled `i1`, second introns `i2`, and so on.
4. If it is the first intron, count all of the k-mers in the intron and add the counts to the `%count1` hash. If it is a more distant intron, add the counts to `%count2`.
5. After all the counting is done, create two new hashes, call them `%freq1` and `%freq2` to hold the frequencies for every k-mer.
6. Report the log-odds ratio of the frequency of each k-mer occurring in 1st introns vs. other introns.

The last part of the program asks you to report a log-odds ratio. Many programs report a *score* which is a base–2 log-odds ratio of observed over expected. A positive value indicates more observations than expected, and a negative value is fewer observations than expected. For example, let's say we have made 40 observations when we expected 20. The *score* is 1.0 because 40/20 = 2, and the base–2 log of 2 is 1.0. Similarly, if we made 5 observations and expected 20, the *score* would be –2. Log-odds ratios are just positive and negative powers of 2.

In our program, we don't have an expectation, but rather two observations (first introns, and more distant introns). You can use the code below to calculate and report the log-odds ratio.

```
my $odds_ratio = $freq1{$kmer} / $freq2{$kmer};
my $lod = log($odds_ratio);  # Perl's log() function uses base e, so...
my $lod2 = $lod / log(2);    # ...need to convert base e to base 2
printf "%s %.3f\n", $kmer, $lod2;
```

If you like brevity, the above four lines can be simplified as:

```
printf "%s %.3f\n", $kmer, log($freq1{$kmer} / $freq2{$kmer}) / log(2);
```

Which k-mers are most differently distributed? To help you observe this, you might want to pipe your output to the Unix `sort` command. Are some of the k-mers statistically or biologically significant? Those can be difficult questions to answer. You could try a Poisson model (see Project 0) to ensure that the counts are statistically robust. You might also randomly shuffle the intron sequences before counting (see Project 3) to determine if the k-mers are simply compositional biases or the result of some more interesting biological signal. In the paper, Rose et al. add up all the kmer scores for experimentally validated introns (they call this the IMEter score) and show there is a good correlation between IMEter score and gene expression.

P17. Regular expressions 101

Previously we learned about the matching and substitution operators (see P6.4: the former lets you see whether a variable contains some text, the latter lets you substitute one string for another. These operators are much more powerful than they first appear. This is because you can use them to search for *patterns* rather than strings.

Task P17.1

Create the simple program below. We will be modifying it quite a bit in this section. Let's say we want to see whether a particular DNA sequence contains a codon for proline. There are four codons that encode for proline (CCA, CCC, CCG, or CCT). If we have a coding sequence which is already separated into codons, then one (tedious) way to do this would be with multiple conditional statements:

```
1.  #!/usr/bin/perl
2.  # codonsearch.pl
3.  use strict; use warnings;
4.
5.  my $seq = "ACG TAC GAA GAC CCA ACA GAT AGC GCG TGC CAG AAA TAG ATT";
6.  if    ($seq =~ m/CCA/) {print "Contains proline (CCA)\n"}
7.  elsif ($seq =~ m/CCC/) {print "Contains proline (CCC)\n"}
8.  elsif ($seq =~ m/CCG/) {print "Contains proline (CCG)\n"}
9.  elsif ($seq =~ m/CCT/) {print "Contains proline (CCT)\n"}
10. else                   {print "No proline today. Boo hoo\n"}
```

Imagine doing this for all possible codons… tiresome. Ideally, we want a solution which would search for 'CCN' where N is A, C, G, or T. This is where regular expressions (also known as regexes) come in. Simply put, a regular expression defines a single pattern which describes a finite range of possibilities. Unix, Perl and other programming languages use a fairly standard way of implementing regular expressions (so anything you learn about them in Perl, will be very useful if you use Unix commands like '`grep`' or `sed`).

Task P17.2

Delete lines 6–9 from the previous program and replace them with the folowing:

```
6.  if ($seq =~ m/CC./){
7.      print "Contains proline ($&)\n";
8.  }
```

Well that was easy! In the context of regular expressions, the dot `.` on line 6 represents *any single character*. It should not be confused with the use of a dot as the concatenate operator (see P6.1).

Line 7 contains a funny variable called `$&`. This is another one of Perl's "special" variables (like `$_`). Perl sets `$&` to be the string matched by the most recent regular expression match.

Task P17.3

Change the regex to now see whether the sequence contains an arginine codon.

```
6.  if ($seq =~ m/CG./){
7.      print "Contains arginine ($&)\n";
8.  }
```

If you copied the sequence exactly as above, your script should be telling you that the `$seq` variable contains an arginine codon, even though it doesn't. Can you see why?

The dot character will match *any* single character, including a space. So the last two letters of the ACG codon *plus* the space that follows matches the pattern, i.e. it matches `CG `. A better solution is to restrict the match to any character that is within a specified set of allowed characters.

Task P17.4

Replace line 6 with this more specific pattern.

```
6.  if ($seq =~ m/CG[ACGT]/) {
```

The square brackets allow you denote a number of possible characters, any of which can match (this is known as specifying a *character class*). This is a much better solution when we have a limited range of characters and our regular expression can now only match four different strings (CGA, CGC, CGG, or CGT). Note though, that even when you have many characters inside the square brackets, you are only ever matching *one* character in the target sequence.

Sometimes biological sequences (DNA, RNA, and proteins) are sometimes represented as uppercase characters (ACG), and sometimes as lowercase characters (acg). It's also possible to download sequence files which use a mixture of upper- and lowercase (e.g. representing exons in uppercase, and introns in lowercase). How do you handle situations like this when trying to match patterns?

Task P17.5

Go back to line 5 and substitute some of the capital letters in our DNA string for lowercase characters, as in the example below.

```
5.  my $seq = "ACG TAC GAA GAC ccA ACA GAT AGC gcg TGC CAG aaa TAG ATT";
```

There are two solutions to matching both upper- and lowercase. The first option is to make character classes (using the square brackets) that describe out all possible combinations of upper or lowercase letters that specify CCN:

```
6.  if ($seq =~ m/[Cc][Cc][ACGTacgt]/){
```

The second option is much simpler. Use the ignore case functionality of the matching operator. This just involves appending an `i` after the second forward slash, and this will now mean that ccc, ccG, cCa, CaT, etc. will all count as a valid match.

```
if ($seq =~ m/GG[ACGT]/i){
```

Because there is no uppercase or lowercase standard for sequence files, it is good to always use the ignore-case option when working with sequences. This option also works with the substitution operator (introduced in P6.5). An alternative is to always convert a sequence to upper- or lowercase *before* you start processing it. The uc() and lc()[] functions perform these operations. If the first thing you do with a new sequence is make it all uppercase or all lowercase, then you don't need to use the ignore-case option later on in your script.

Character ranges

Another useful option when specifying a character class is to use a dash to specify a range of characters or numbers.

```
if ($seq =~ m/[a-z]/){
    print "Contains at least one lower case letter\n";
}
```

Perl defines several symbols for common character classes. Two of the most useful ones are `\s` and `\S` which are used to match whitespace and non-whitespace respectively. Matching whitespace allows you to match spaces that might be the result of space and/or tab characters (not always obvious when you are processing data from a text file).

Anchors

To ensure that a pattern matches the beginning or ending of a string, you can use the `^` and `$` symbols. This is the same as when using regexes in Unix:

```
if ($dna =~ m/^ATG/){ ...} # matches if $seq started with 'ATG'
if ($dna =~ m/TGA$/){ ...} # matches if $seq ended with 'TGA'
if ($pep =~ m/^M[ST]G$/){ ...} # matches if $pep started with M, followed by S or T, and ended wit
```

Negated character classes

You can also specify character classes that should *not* occur as part of a input string that you are trying to match. Unfortunately, the symbol that is used to specify this is the same symbol as the anchor character we just showed you `^`. This can be confusing, but the character is used a little differently when making negated character classes. Here is how you could find whether a protein sequence either contained polar amino acids (D, E, R, K, or H) or whether it *didn't* contain any:

```
if ($pep =~ m/[DERKH]/){ ...}  # would only match if $pep has at least one polar residue
if ($pep =~ m/[^DERKH]/){ ...} # would only match if $pep has no polar residues
```

The basic rule here is that if `^` is the first character inside the square brackets of a character class, then it becomes a negated character class.

Repetition

If you want to match several characters or character classes in a row, you can use various repetition symbols `+`, `?`, and `*` which let you match '1 or more', 'zero or 1' and 'zero or more' characters respectively:

```
if ($text =~ m/A+/) { ... } #  matches 1 or more As.
if ($text =~ m/A?/) { ... } #  matches zero or 1 As.
if ($text =~ m/A*/) { ... } #  matches zero or more As.
if ($text =~ m/^A+B+$/) { ... } #  matches 1 or more As at start of string & 1 or more Bs at end
```

If you want to match a character between 'm' and 'n' times or to match exactly 'm' times, then you can use the following syntax:

```
if ($text =~ m/A{1,3}/) { ... } #  matches between 1 and 3 As
if ($text =~ m/C{42}/) { ... } #  matches exactly 42 Cs
if ($text =~ m/T{6,}/) { ... } #  matches at least 6 Ts
```

Alternation

You can match more than one pattern at once if you separate them with pipe symbols `|` (in this context, the pipe character is known as the alternation operator). E.g. to match any of the three stop codons you could use:

```
if ($seq =~ m/TAA|TAG|TGA/) { ... } # matches TAA *or* TAG *or* TGA
```

Parentheses can be used to clarify what text is part of the alternation:

```
if ($text =~ m/Super(man|girl|boy|rabbit)/) { ... } # matches various superheroes
```

Combining metaharacters

It can be confusing to learn about regular expressions in Perl because there are so many of these special *metacharacters* that you can use inside patterns. It is even more confusing when some of these characters have completely different meanings outside of their use in regular expressions. All of the metacharacters we have seen can be combined to produce very powerful — and often very confusing — regular expressions. Consider the following regular expression:

```
if ($seq =~ m/^ATGCC[ACGT]GG[ACGT]N{6,9}(TAG|TGA|TAA)$/) { ... }
```

This would produce a match as long as:

1. `$seq` started with a start codon `^ATG`
2. was followed by a single proline codon `CC[ACGT]`
3. was followed by a single glycine codon `GG[ACGT]`
4. was followed by between 6–9 N nucleotides (unknown bases) `N{6,9}`
5. ended with one of the three valid stop codons `(TAG|TGA|TAA)$`

Backslash

We have just seen many special regular expression metacharacters. But what if you wanted to actually match one of these characters? E.g. you wanted to see whether `$species` contained the text ‘A. thaliana’. The following would probably work, though not as you might expect:

```
if ($species =~ m/A. thaliana/) { ... }
```

The `.` is `A. thaliana` is being used as a regular expression metacharacter, meaning that it will match any single character. Therefore if `$species` contained the text `AB thaliana`, then this would produce an erroneous match. The special meaning of any metacharacter can be “escaped” by prefixing it with a backslash. Therefore, you can match a dot `.` with `\.` and it will only match a dot:

```
if ($species =~ m/A\. thaliana/) { ... }
```

The backslash is also used to escape the special meaning of other characters in Perl. What if you wanted to print the value of the variable `$answer` but also include the text `$answer` in the output string? Or what if you wanted to print `\n` but not have it print a newline?

```
my $answer = 3;
print "\$answer is $answer\n";
print "This is a newline character: \\n\n";
```

Perl regular expression metacharacters

Symbol	Meaning
`.`	any character
`\w`	alphanumeric and _
`\W`	any non-word character
`\s`	any whitespace
`\S`	any non-whitespace
`\d`	any digit character
`\D`	any non-digit character
`\t`	tab
`\n`	newline
`*`	match 0 or more times
`+`	match 1 or more times
`?`	match 0 or 1 times
`{n}`	match exactly n times
`{n,m}`	match n to m times
`^`	match from start
`$`	match to end

P18. Extracting text

You will often want to extract some strings from a large file. For example, you may be processing a GFF or GenBank file to retrieve specific coordinates or features. Hopefully you will not be parsing HTML for email addresses! The power of regexes not only fuels bioinformatics, but also spam…. There are a number of ways to pull out specific patterns from a file. We have seen a couple of these already; e.g. in P17.2 we saw how `$&` contains the string of the last pattern match. But this doesn't let you extract multiple strings at once, so it has limited use. If you happen to be parsing a tab-delimited file, rejoice because you can just use the `split()` function.

```
while (<>) {
    my @fields = split("\t", $_); # each line gets placed in $_
}
```

If the file happens to be space-delimited, you can even abbreviate even further.

```
2.  my @fields = split; # \s+ and $_ are assumed
```

But you won't always have tab-delimited text. Some files are much more complex.

Task P18.1

Let's retrieve all the gene names and coordinates from a GenBank file. Take a look (using `less`) at the file `Unix_and_Perl_course//Data/GenBank/E.coli.genbank` and scroll down (or search) until you find the 'gene' keyword in the 'FEATURES' section of the file:

```
FEATURES             Location/Qualifiers
     source          1..4686137
                     /organism="Escherichia coli str. K12 substr. DH10B"
                     /mol_type="genomic DNA"
                     /strain="K-12"
                     /sub_strain="DH10B"
                     /db_xref="taxon:316385"
     gene            190..255
                     /gene="thrL"
                     /locus_tag="ECDH10B_0001"
                     /db_xref="GeneID:6058969"
```

The coordinates of the gene are given on the same line `190..225`. One line below contains the gene name as `/gene="thrL`. Page down a bit and you will find another gene on the reverse (complement) strand:

```
     gene            complement(5683..6459)
                     /gene="yaaA"
                     /locus_tag="ECDH10B_0006"
                     /db_xref="GeneID:6061859"
```

In order to parse this file, we must deal with genes on the complement strand and also the fact that all the information isn't on the same line. The following program reports the name and coordinates of all genes. Remember to specify the name of the GenBank file when you run the script.

```
1.  #!/usr/bin/perl
2.  # parse_genes.pl
3.  use strict; use warnings;
4.
5.  while (my $line = <>) {
6.      if ($line =~ /^\s{5}gene/) {
7.          my ($beg, $end) = $line =~ /(\d+)\.\.(\d+)/;
8.          $line = <>;
9.          my ($name) = $line =~ /="(.+)"/;
10.         print "$name $beg $end\n";
11.     }
12. }
```

Lines 1–5 should look very familiar by now. Line 5 sets up the `while` loop which will loop over every line of the specified file.

Line 6 asks if `$line` starts with 5 spaces `^\s{5}` followed by the word 'gene'. GenBank format is very strict about how many spaces begin each line. Had we been lazy, we could have used `\s*` or `\s+` to match the spaces at the start of the line.

Line 7 matches the coordinates from `$line` *and* assigns them to the `$beg` and `$end` variables. We have seen matches like this before but have not tried assigning the results to anything. Regular expressions matches in list context return values from parenthesized patterns. You might want to repeat the phrase a dozen times or so. It's that important.

> ***Regular expressions in list context return values from parenthesized patterns***
> ***Regular expressions in list context return values from parenthesized patterns***
> ***Regular expressions in list context return values from parenthesized patterns…***

Line 8 gets another line of input from the GenBank file by using the file operator `<>` once again. We need to read another line because the gene's name is one line below the gene's coordinates. The logic of the `while` loop becomes "keep looping over lines and *if* I see something that looks like a pair of gene coordinates, then read one more line from the file".

Line 9 matches the gene name using the regular expression pattern `="(.+)"`. The parentheses ensure that any text between the quotation marks is captured (in `$name`). We use this very generalized pattern (match one or more of any single character) because gene names sometimes contain strange characters and spaces, even though they don't in *E. coli*. Nearly all CDSs in the GenBank file have a gene name, but because a few don't we also have to be able to capture lines that match either a `/gene=` or a `/locus_tag=` pattern.

More Info

We've only scratched the surface of regular expressions. For more information, read the Perl man pages.

```
$ man perlrequick
$ man perlre
```

P19. Boolean logic

Back in P13.2 we saw the following statement:

```
open(my $in, "<$ARGV[0]") or die "error reading $ARGV[0] for reading";
```

The meaning of this is pretty clear: open the file or die trying. We understood the `or` part as something that only happens if the file doesn't open. How exactly does this work? All Perl functions return a True or False value. False values are 0 and the empty string `""`. All other values are true. This point bears repeating:

> ***False values are 0 and the empty string `""`. All other values are true.***

So we can understand the `open()` statement above as a more concise version of the following:

```
$return_value = open(my $in, "< $ARGV[0]);
if ($return_value == 0) {
    die "can't open file $ARGV[0]\n";
}
```

But why doesn't the `die()` statement get executed if the return value is True? The answer it is because the *whole* statement — from the `open()` function through to the semicolon — that is evaluated with Boolean logic. The Boolean operators are `and`, `or`, `not`. Let's review how `and` and `or` behave:

```
True  and True  = True
True  and False = False
False and True  = False
False and False = False
True  or  True  = True
True  or  False = True
False or  True  = True
False or  False = False
```

If the `open()` function works then that function returns true, which then means that the entire Boolean expression `open() or die` must be True. Perl does not attempt to evaluate more than it needs to, so once `open()` succeeds, it short-circuits the rest of the statement. I.e. if the left-hand side of any `or` statement evaluates as True, Perl doesn't need to evaluate what's on the right-hand side of the `or` statement. Likewise, is the left-hand side of any `and` statement evaluates as False, then there is no need to evaluate the right-hand side.

Back in P11.1 we saw the following statement:

```
@list = sort {$a <=> $b or uc($a) cmp uc($b)} @list
```

What’s going on here? The sorting function first compares `$a` and `$b` numerically. If their numeric values are zero (e.g. because they are strings), the expression `$a <=> $b` returns zero. Perl must then evaluate the right side `uc($a) cmp uc($b)` to determine if the whole expression is true or false. So numbers get compared first, and if they are equal, they are further compared by ASCII value.

Project 6: Codon usage of a GenBank file

The goal of this project is to create a codon usage table for any bacteria in GenBank. For example, we could calculate the codon usage of *E. coli*, *B. subtilis*, or *Y. pestis*. You will find GenBank files for several bacteria in the `Data/GenBank` directory. You will use write a program and use it to analyze their codon usage. Later, we will compare them with Information Theory! But that is for another day.

Use the Unix command `less` to look at the *E. coli* file. GenBank files have a well-defined structure and each file may contain thousands of sequences, but this file contains just one. After the header section which describes details of the entry in GenBank, and various bibliographic details you will see the 'FEATURES' section which describes all of the features that have been annotated on this sequence. In this section you will see many 'CDS' (coding sequence) features along with their corresponding protein translations. Take a look at several of these and see if you can determine how GenBank distinguishes those genes which are on the reverse DNA strand. At the end of the file you will find the DNA sequence of the genome. Note that GenBank files provide sequence coordinates which appear at the start of each sequence line. Your tasks are:

1. Extract the nucleotide sequences that correspond to these proteins
2. Count the codons in each coding sequence
3. Print a summary of the codon usage from all genes combined

This might sound challenging, and it is. But we have the knowledge to do it. The strategy you will use for this program is outlined below:

1. Create a `$genome` variable that will store the *E. coli* genome sequence
2. Open the GenBank file
3. Skip all lines until you get to the sequence (look for the text 'ORIGIN')
4. Process each line of sequence as you capture it:
   i. Remove the digits that start each line
   ii. Remove the spaces
   iii. Remove the newline at the end of each line
   iv. Add each line of processed sequence to `$genome`
5. Close the GenBank file
6. Re-open the GenBank file
7. Process each line
   i. Find lines which describe CDS features
   ii. Extract the start and end coordinates of each CDS
   iii. Use these coordinates to extract the corresponding DNA sequence of the CDS from `$genome`
   iv. Reverse-complement CDS sequence if necessary
   v. Count codons of current CDS, add to running totals for all CDSs
8. Report frequencies of all codons

Tips

- Use a named filehandle for steps 2 and 6 (e.g. `open(my $genbank, "<", $ARGV[0])`. Don't use the file operator on its own `<>`.
- Make sure that the coding sequences are correct. Most should start with ATG and end with a stop codon. If they

do not, you may need to improve your code.

- Remember that sequence coordinates start at ‘1’ but the `substr()` function starts counting positions at ‘0’.
- Some proteins may contain the peptide ‘CDS’ (cysteine-aspartate-serine). So be careful when searching for the text ‘CDS’ in order to find CDS features.
- Don’t use this program with eukaryotic GenBank sequences which will have multi-exon CDS features. Describing the joins of the various exons can take several lines, which makes parsing the file a little more difficult.
- Be careful about making assumptions about how biology works (biology has a nasty habit of kicking you in the butt just to surprise you!). I.e. don’t assume that all DNA characters in this file will be A, C, G, and T. Rather than assume, check!
- Test each section of code as you write it. If you can’t correctly extract the genome sequence, then there is no point going any further with your script!

P20. Functions (a.k.a. subroutines)

We've seen several built-in functions already such as `print()` and `rand()`. Programming would be pretty difficult if we didn't have these. Wouldn't it be great to make your own functions? This is where the real power of programming lies. In Perl, we can make our own functions by using *subroutines*. A subroutine is like a little parcel of code that usually performs one focused task. E.g. calculate the minimum value from a set of numbers, or translate a DNA sequence into a protein sequence. Subroutines are ideally suited to small coding tasks that you might need to perform multiple times within a single script.

Task P20.1

Create the following program. It takes a DNA sequence that you specify on the command-line and runs three simple checks to see whether the sequence might represent a valid CDS. If the sequence fails any check, an error message is printed and we will simply use a subroutine to print the error message:

```
1.  #!/usr/bin/perl
2.  # sequencecheck.pl
3.  use strict; use warnings;
4.
5.  # take sequence from command-line and make upper case
6.  my $seq = uc($ARGV[0]);
7.
8.  if ($seq !~ m/^ATG/){           # test for start codon
9.      print_error();
10.  }
11. elsif($seq !~ m/(TGA|TAG|TAA)$/){ # test for stop codon
12.     print_error();
13. }
14. elsif($seq !~ m/^[ACGTN]+$/){   # test for non DNA characters
15.     print_error();
16. }
17. else{
18.     print "$seq looks likes a valid CDS\n";
19. }
20.
21. sub print_error {
22.     print "$ARGV[0] is not a valid sequence for a CDS\n";
23.     print "It may not start with an ATG start codon\n";
24.     print "It may not end with a stop codon\n";
25.     print "It may contain non ATCGN DNA characters\n";
26. }
```

Lines 9, 12, and 15 all call the `print_error()` subroutine which is declared on line 21. Subroutines behave just like any other Perl function, but unlike built-in functions like `print()`, you must include parentheses. To declare a function/subroutine, you use the `sub` keyword. This is immediately followed by the name of the function and a block structure delimited by curly braces.

Any time Perl sees the subroutine name `print_error()` it immediately jumps to the block of code that starts `sub print_error`. When Perl finishes processing the code in the subroutine it immediately returns back to where it originally was in the main body of the script. Depending on the sequence that is provided to the script, this code might jump back and forth between the subroutine three times (if all three errors are present). Because Perl will jump straight to the subroutine no matter where it is located, you can define the subroutine anywhere in the script. However, there is a convention to putting subroutines at the end of a Perl script (though in other languages you might find them at the beginning). You can do it either way, but try to be consistent.

This script is not a very good script, it prints the same error message regardless of what error is found in the sequence. However, you should see that by using a subroutine we only need to write the code to produce the error message in one place. If we wanted to add or change the error message, we only need to edit the code in one place.

Task P20.2

When we use subroutines it is far more common to pass the subroutine one or more variables and get the subroutine to do something useful with those variables. Create the following program. It reads a file of sequences and computes the GC% of each one.

```
1.  #!/usr/bin/perl
2.  # gc.pl
3.  use strict; use warnings;
4.
5.  while (my $seq = <>) {
6.      chomp($seq);
7.      gc($seq);
8. }
9.
10. sub gc {
11.     my ($seq) = @_;
12.     $seq = uc($seq); # convert to upper case to be sure
13.     my $g = $seq =~ tr/G/G/;
14.     my $c = $seq =~ tr/C/C/;
15.     my $gc = ($g + $c) / length($seq);
16.     print "GC% = $gc\n";
17. }
```

Before you run this script you will need to create a text file which contains a few lines of DNA sequence. Use the name of the file when you run the script e.g. `gc.pl dna_file.txt`

Line 5 sets up a `while` loop to loop over any (and every) file specified on the command-line.

Line 7 calls the `gc()` subroutine and passes it the `$seq` variable. To pass a variable to a subroutine, include it between the parentheses that follow the subroutine name. The `gc()` subroutine will be called for every line that is present in your input file.

Lines 10-–17 contain the `gc()` function. Because line 7 passes a variable to the subroutine, we must add code to receive it. Subroutines receive arguments via the special `@_` array. Any variables that are passed to the subroutine will be stored in this array. Just as we don't like to use `@ARGV` throughout our script (because the name isn't very meaningful) we also want to extract any variables from `@_` and assign them to variables with new names.

Line 11 shows a typical list assignment, the first element of the `@_` array is copied to `$seq`. You may see some programs using the `shift()` function to remove elements of the `@_` array in one of two ways:

```
my $seq = shift(@_);
my $seq = shift;
```

In the second example, `shift()` is used without specifying an array name. If no array is specified the shift function uses the `@_` array by default.

Note that we use `$seq` again as a variable name within the subroutine, we'll explain why in the next section. For now, just accept that the `$seq` in the subroutine is unrelated to the other `$seq`.

Anything passed to the `@_` array is copied. This means that line 7 is effectively sending a copy of `$seq` to the `gc()` subroutine. In other words, `$seq` is unchanged by anything that happens in `gc()`. You can test this by adding a `print "$seq\n"` statement after line 7.

Task P20.3

The previous program demonstrated a much better use of subroutines, but it is still not ideal. Maybe we don't always want to print the value of GC% as soon as we calculate it. In general, we often want a subroutine to calculate something and send that back to wherever we called the subroutine from. We can do this in Perl by using `return values` within a subroutine. Let's make a script that uses the melting temperature code that we saw earlier in P15.1, but that now puts this code in its own subroutine:

```
1.  #!/usr/bin/perl
2.  # tm.pl
3.  use strict; use warnings;
4.
5.  while (my $seq = <>) {
6.      chomp($seq);
7.      my $tm = tm($seq);
8.      print "Tm = $tm\n";
9.  }
10.
11. # calculate Tm
12. sub tm{
13.     my $seq = shift;
14.     my $A = $seq =~ tr/A/A/;
15.     my $C = $seq =~ tr/C/C/;
16.     my $G = $seq =~ tr/G/G/;
17.     my $T = $seq =~ tr/T/T/;
18.     my $tm = 2 * ($A + $T) + 4 * ($C + $G); # simple Tm formula
19.     return($tm)
20. }
```

First of all, let's look at line 19. This line returns a value from the subroutine using the return() function. You can use return anywhere in the subroutine and it will exit at that point and return to wherever the subroutine was called from. I.e. if there was a `print` statement on line 20, it would never be performed. You can return multiple values in a return statement or even none. Sometimes we just return 1 or 0 to indicate success or failure.

So what does Perl do with returned values? If we now look at line 7 we can see that the output of the `tm()` function is assigned to a variable. If we had wanted to make our code more concise (which is not always a good thing) we could have replaced lines 7 and 8 with:

```
7.  print "TM = ", tm($seq), "\n";
```

When Perl evaluates this code, it knows that the first thing that has to be dealt with is the call to the `tm()` subroutine. When that code is finished, the subroutine will return a value (or potentially a list of values) and in this case we plug that value straight into a `print()` function rather than saving it in a variable. We could also have replaced lines 18 and 19 with the following:

```
18.     return(2 * ($A + $T) + 4 * ($C + $G));
```

In this case, Perl will first make the calculation of the melting temperature and then return the resulting value. Most people find it easier to first store this result into a variable and then return the variable.

Task 20.4

So far we have only ever passed one variable to a subroutine and returned just one thing back to the calling function. It is very common to pass and return multiple arguments. It is also common to have multiple return statements which are all dependent on the outcome of some logical test.

Modify the GC% script in order to pass two things to the subroutine: the sequence plus a GC% threshold (a floating point number which will be stored in a `$threshold` variable within the subroutine). If the GC content is above the value of `$threshold` then we will return “High GC” else we will return “Low GC”.

To simplify things, you can specify the sequence and the threshold value on the command-line (instead of reading a file). We also want the script to print out whether each sequence is high or low GC, but that print statement must not be in the subroutine! You will have to look up how to pass two things to a subroutine. The end of the subroutine will look like the following:

```
sub gc {
    #
    # missing code to go here
    #

    $seq = uc($seq);
    # convert to upper case to be sure
    my $g = $seq =~ tr/G/G/;
    my $c = $seq =~ tr/C/C/;
    my $gc = ($g + $c) / length($seq);

    if($gc > $threshold){
        return("High GC");
    } else{
        return("Low GC");
    }
}
```

Note that if you use multiple return statements (as in this example), they should always be part of a logical test such that only one return statement is ‘seen’ by the code. E.g. the following subroutine would be pointless:

```
sub pointless {
    my ($some_value) = @_;
    return("Yay!"); # subroutine exits at this point
    return("Boo!"); # this line will *never* be evaluated by Perl
}
```

Why use subroutines?

As your programs get longer you might find yourself wanting to do the same thing more than once in your program. Maybe part of your program takes two input sequences and calculates their percentage similarity. Your program might then modify those sequences and then recalculate the percentage similarity. Without subroutines you would have to have the same lines of code in two places in your script. This is a bad idea. Where possible, code should be reused. As soon as you find yourself writing the same code in more than one place, you should think about putting that code in a subroutine.

Subroutines can also help improve the readability of your code. Rather than see all of the details of how you calculate some mathematical function, it might be cleaner to keep that code in a subroutine and this keeps it hidden from the main body of the code.

P21. Lexical variables and scope

By default Perl lets you create variables whenever you need them and they are then available throughout your entire program. We call this global scope. Many people would argue that using global variables is dangerous and certainly not the best way of programming. Instead, we could (and should) use *local* variables. This is something that we have already made you do in nearly all of these Perl scripts. That's because when we include the `use strict;` statement this requires that we need to declare all variables, arrays, and hashes with the `my` keyword. This means that we are declaring a local variable.

Consider the following program that deliberately does not contain the `use strict;` statement. The program takes a sequence and counts how many codons it contains (using a subroutine). Run the program and observe the output.

```
1.  #!/usr/bin/perl
2.  # no_strict.pl
3.  use warnings;
4.
5.  my $seq = "atg att gaa cca tga";
6.  $codons = count_codons($seq);
7.  print "$seq contains $codons codons\n";
8.
9.  sub count_codons {
10.     $seq = shift;
11.     $seq = uc($seq);  # convert to upper case to be sure
12.     $seq =~ s/\s+//g; # remove all whitespace from sequence
13.     $codons = length($seq) / 3;
14.     return($codons);
15. }
```

Line 7 prints `$seq` but it now prints the version of `$seq` that was modified in the subroutine (on lines 11–12). Without the `my` declarations, the two `$seq` variables are no longer separate entities. This is probably not the behavior that we wanted. When we *don't* declare variables with `my`, they become *global variables*. Changing a global variable in any one part of the program changes it everywhere else. We should never do this, it is just about the worst thing you can do as a programmer.

To make sure we do not affect other parts of a program, we will always choose to make variables inside a function exist only within that function. The `my` keyword does this for us and it creates what is known as a lexical variable. These are variables that live and die within a set of curly braces (a block). This means that we can reuse variable names to store different things as long as they exist within different blocks of code.

Task P21.1

The following program demonstrates the use of lexical variables:

```
1.  #!/usr/bin/perl
2.  # lexical.pl
3.  use strict; use warnings;
4.
5.  my $x;                  # variable declaration without assignment
6.  $x = 1;                 # variable assignment
7.  my ($y, $z) = (2, 3); # you can declare and assign variables as a list
8.  if ($x < $y) {
9.      my $z = 10;
10.     print "inside:  X = $x, Y = $y, Z = $z\n";
11. }
12. print "outside: X = $x, Y = $y, Z = $z\n";
```

It is critical that you completely understand the concept of the *scope* of a variable. The scope is that part of your code that is allowed “to see” your variable. A variable’s scope starts from the point it is first declared and ends at the next enclosing curly bracket that’s at the same logical level as the variable. In this script, there is a `$z` variable that is ‘born’ at line 9 and ‘dies’ at line 11. Importantly, this `$z` is not the same as the `$z` on line 7. The inner `$z` effectively hides the outer `$z` as soon as it is declared. As soon as we reach line 12, we are no longer in the scope of the inner `$z` and revert to the scope of the outer `$z`.

Variables in a wider scope are visible in a narrower scope. So we can see `$x` and `$y` at line 11. Variables in a narrower scope do not exist in a wider scope. To see this more clearly, try changing lines 9 & 13 to to the following:

```
9.  my ($z, $q) = (10,15);

13. print "outside: $x $y $z $q\n";
```

This should produce an error because `$q` doesn’t exist outside the loop so we shouldn’t be able to print it on line 13. After line 11, `$q` no longer exists.

Loop Variables

Lexical variables in loops look a little strange because they are declared outside the curly braces:

```
for (my $i = 0; $i < 10; $i++) {
    # code inside for loop
}

foreach my $seq (@seq) {
    # code inside foreach loop
}
```

The loop counter variable `$i` is declared on line 1 and dies at line 3. So even though it appears outside the curly braces, its scope is actually the entire loop. Likewise, `$seq` is born anew with each iteration of the foreach loop at line 4 and dies each time at line 6. This means that if you have multiple `for` loops in a script you could, and indeed *should*, reuse the same loop counter variable name. The exception to this is when you have nested `for` loops where the convention is to use `$i`, then `$j`, then `$k` etc. E.g.

```
for (my $i = 0; $i < 10; $i++) {
    # outer loop, using $i
    for (my $j = 0; $j < 10; $j++) {
        # inner loop, using $j
    }
}
```

Safer programming: `use strict`

All variables should be lexical variables. To ensure this behavior, include `use strict` in *all* your programs. In fact, your programs should always contain a line like this:

```
use strict; use warnings;
```

You may run into someone who thinks that `strict` and `warnings` are a hassle. Feel free to talk to, dine with, or even marry this person, but in no circumstances should you share code with them!

P22. Sliding window algorithms

One of the most common sequence analysis scenarios is to look at the local composition of a sequence rather than the global composition. For example, a genome might be 45% GC, but it might be more GC-rich in CpG islands and less GC-rich in introns. Similarly, a protein may have hydrophobic and hydrophilic regions, and you might want to identify these.

Task P22.1

Create the following program and run it with a few window sizes to observe the smoothing effect of larger window sizes. In this algorithm, there are two loops. The outer loop moves the window along the sequence. The inner loop counts the nucleotides inside the window.

```
1.  #!/usr/bin/perl
2.  # sliding.pl
3.  use strict; use warnings;
4.
5.  die "usage: sliding.pl <window> <seq>" unless @ARGV == 2;
6.
7.  my ($window, $seq) = @ARGV;
8.
9.  # outer loop
10. for (my $i = 0; $i < length($seq) - $window +1; $i++) {
11.     my $gc_count = 0;
12.
13.     # inner loop
14.     for (my $j = 0; $j < $window; $j++) {
15.         my $nt = substr($seq, $i + $j, 1);
16.         $gc_count++ if $nt =~ /[GC]/i;
17.     }
18.
19.     printf "%d\t%.3f\n", $i, $gc_count/$window;
20. }
```

Note that to make sure all the windows are the same size and that the last few windows do not run off the end of the sequence, we have to stop the sliding before it gets to the end. The conditional part of line 10 might look a bit strange `$i < length($seq) - $window +1`, but it's ensuring that we don't go past the end of our sequence.

Task P22.2

Here is an alternative approach using a single loop that reuses our `gc()` function. Replace lines 14–19 with the following two lines and then copy the `gc()` subroutine into the script. This strategy is slightly less efficient because there is some overhead in every function call. But we think you will agree that it reads much better!

```
14. my $subseq = substr($seq, $i, $window);
15. printf "%d\t%.3f\n", $i, gc($subseq);
```

Task P22.3

Did you notice that both of the previous sliding window algorithms recount the same bases? Imagine a window of 1000 bases. The total number of Cs and Gs is not going to change much as the window slides over one more position. In fact, the number of Gs or Cs can only change by plus or minus 1. Why count 1000 letters when you only need to change one value? You don't have to. If you count the Cs and Gs in the initial window, you can then update the counts as you slide along. This algorithm turns out to much more efficient for large windows. You might want to come back to this task at a later time. It's doesn't introduce any new concepts, but the code is definitely more complicated:

```perl
#!/usr/bin/perl
# sliding_fast.pl
use strict; use warnings;

die "usage: sliding_fast.pl <window> <seq>" unless @ARGV == 2;
my ($window, $seq) = @ARGV;

# initial window
my $gc_count = 0;
for (my $i = 0; $i < $window; $i++) {
    my $nt = substr($seq, $i, 1);
    if ($nt =~ /[CG]/i) {$gc_count++}
}
printf "%d\t%.3f\n", 0, $gc_count/$window;

# all other windows
my $limit = length($seq) - $window + 1;
for (my $i = 1; $i < $limit; $i++) {
    my $prev = substr($seq, $i -  1, 1);
    my $next = substr($seq, $i + $window -1, 1);
    if ($prev =~ /[CG]/i) {$gc_count--}
    if ($next =~ /[CG]/i) {$gc_count++}
    printf "%d\t%.3f\n", $i -$window +1, $gc_count/$window;
}
```

Sometimes you must choose between readability and speed. Most of the time, you should let readability take precedence. Why? Because readable code is easier to debug and maintain. If you absolutely need something to run faster, there are a variety of possible solutions including (a) buying a faster computer (b) changing the structure of the algorithm (c) programming in a compiled language such as C.

P23. Function libraries

Once you develop some useful functions like `gc()`, you will find that you want to use them again and again. One way to re-use code is to simply copy-paste your functions from one program to another. Since functions are like mini programs, this usually works just fine. But what if you discover an error in the function and now you want to fix all the programs that use it? You'll have to search all your programs and fix each one. Wouldn't it be better if the programs all used the exact same code? Absolutely!

A function library is a file where you keep a group of related functions. Any program you write can use these functions. Having your own personal library makes programming much simpler. But the real power of libraries comes when you use other people's libraries. *The only thing better than your function library is someone else's*. Enough talk, let's create our first library.

Perl uses the term package or module for function library. They (mostly) mean the same thing. All Perl modules are saved with the .pm suffix (for Perl module). The first line of a module uses the `package` statement and the last line is simply 1;. All of the functions go between those statements. There is no limit to the number of functions you can place in a library.

Task P23.1

Save the following code in your 'Code' directory as `Library.pm`. This is not a particularly descriptive name, but it will do for now.

```
package Library;
use strict; use warnings;

sub gc {
    my ($seq) = @_;
    $seq = uc($seq); # convert to upper case to be sure
    my $g = $seq =~ tr/G/G/;
    my $c = $seq =~ tr/C/C/;
    my $gc = ($g + $c) / length($seq);
    return $gc;
}

1;
```

The `gc()` function can now be used in any program you write as long as Library.pm is in the same directory as the script that wants to use it. Now let's see how we use libraries.

Task P23.2

Go back to sliding.pl (P22.1) and insert the line `use Library;`. One generally puts such statements at the top of a program, but you can put them anywhere. This simple statement allows the program to use any of the functions in the library. To call `gc()`, we must prepend the function call with the library name `Library::gc()` (as in line 9 below). The reason for this is that we might be using several different libraries, each of which could have their own `gc()` function. Including the library name makes it clear that things such as `Library::gc()` and `OtherLibrary::gc()` are separate functions.

```perl
#!/usr/bin/perl
# sliding.pl use strict; use warnings;
use Library;

die "usage: sliding.pl <window> <seq>" unless @ARGV == 2;
my ($window, $seq) = @ARGV;
for (my $i = 0; $i < length($seq) - $window +1; $i++) {
    my $subseq = substr($seq, $i, $window);
    printf "%d\t%.3f\n", $i, Library::gc($subseq);
}
```

If you ended up writing several different scripts, all of which needed to calculate the GC content of a sequence, you could use this single piece of code in all of those scripts. This is much cleaner, and more efficient, than maintaining exactly the same code across many different scripts. As a general rule:

> ***If a single script has duplicate code that does the same thing, put the code in a subroutine***
> ***If multiple scripts contain the same subroutines, put the subroutine in a library***

Project 7: Useful functions

This project has a number of sub-projects, each of which is based around a new useful function that you should add to your library.

Project 7.1

Create a function that reverse-complements a DNA sequence. Ideally, this should even work if the sequence contains nucleotide ambiguity characters such as R, Y, M, K, etc.

Project 7.2

Write a function that computes the entropy of a sequence. The entropy is simply the sum of `$x * log($x)`, where `$x` is the frequency of each letter. Unbiased DNA has 2 bits of entropy. A biased composition results in less < 2 bits. To convert from nats to bits, divide by log(2). Use the entropy function in combination with a sliding window to find low entropy regions of a sequence.

Project 7.3

Using the standard genetic code, write a function that returns the translation of a nucleotide sequence. Use an 'X' for an ambiguous codon and a * for a stop codon. Write a program using this function finds the longest ORF in a sequence.

Project 7.4

Write a function that returns the codon frequencies in a GenBank file. The function should take a file name as the argument, and return the frequencies in either a hash or array. You should be able to modify your code from Project 6.

Project 7.5

Write a program that compares the codon usage of two bacteria. Use Kullback-Leibler distance (relative entropy) to compare the codon frequencies. K-L distance is the sum of `$x * log($x / $y)` where `$x` is a codon frequency in one genome and `$y` is the the frequency of the same codon in another genome. Experiment with several bacteria.

Project 7.6

Membrane-spanning regions of proteins are hydrophobic. To find potential trans-membrane domains, create a Kyte-Doolittle hydropathy function. For this you will need to look up the hydrophobicity of each amino acid and calculate the average hydrophobicity in a sliding window. The function should print all the hydrophobicity values. Alternatively (and better), the function can return an array of values.

P24. Interacting with other programs

Let's say you want to run BLAST 1000 times and retrieve the output. No problem, there are a number of ways to get information from other programs into your program. The simplest one is the backticks operator ``. This looks like an apostrophe but is actually a different character and will be hiding somewhere on your keyboard. Whatever you put in backticks will be executed in the Unix shell, and the output will be returned to you in an array or scalar depending on how you asked for it.

Task P24.1

Let's try an example of capturing the output of the `ls` command in two different ways:

```
1.  #!/usr/bin/perl
2.  # system.pl
3.  use strict; use warnings;
4.
5.  my @files = `ls`;
6.  print "@files\n";
7.  my $file_count = `ls | wc`;
8.  print "$file_count\n";
```

One line 5, we capture the program output in list context, and you should see that each element of `@files` is a separate file or directory that was returned by the `ls` command.

On line 7 we capture program output in scalar context and all output is assigned to a single variable.

Another way to run an external program is with a system() function call. Whatever you put into a `system()` call is run just like the Unix command line. Unlike the `open()` function which returns 0 when it fails, the `system()` function returns 0 when it succeeds (there are good reasons for this, but for now let's just be angry about it). It is generally preferable to use the `system()` function rather than backticks as this gives you more control of testing whether the Unix command that you run actually worked or not. Add the following line to your program:

```
9.  system("ls > foo") == 0 or die "Command failed\n";
```

You will now have a file called foo in the directory where you ran the script that contains the contents of the `ls` command. To get this into your program we can use the `open()` function as we have seen before. But this time, we will do something slightly different with it:

```
10. open(my $in, "<", "foo") or die "Can't open foo\n";
11. my @files = <$in>; # reads the entire file into @files
12. close $in;
13.
13. foreach my $file (@files) {
14.     print "$file\n"
15. }
```

On line 11 we introduce a shorthand for reading all the lines of a file at once. Be careful with this because you could run out of memory if you slurp up a big genome.

You will most commonly use filehandles to read from files, or write to files. However, filehandles can also be used in connection with ‘pipes’ which act just like pipes in Unix. So you can establish a filehandle which acts as a pipe that receives input from a Unix command (go back to the Unix lesson U34 if you need a reminder).

The following code connects a filehandle to the output of the `ls` command and then read the output, one line at a time:

```perl
# filehandle '$in' will now receive output from the 'ls' command
open(my $in, "ls |") or die "Can't open pipe from ls command";
while (my $line = <$in>) {
    print "file: ", $line;
}
close($in);
```

If we reverse things and put the pipe *before* the command/script name, then we can even use `open()` to send commands to a program!

```perl
# the filehandle '$out' will now connect to the Unix wc command
open(my $out, "| wc") or die "Can't open pipe to wc command";
print $out "this sentence has 1 line, 10 words, and 51 letters\n";
close $out;
```

P25. Options processing

It is useful for your programs to have command-line options that allow them to behave in different ways. For example, `ls` lists the current directory, but if you want to see which files sorted by date, you type `ls -lt`. Your Perl programs can have this same behavior. There are two built-in modules for processing command line options, `Getopt::Std` and `Getopt::Long`.

Do you wonder what the `::` means in `Getopt::Std`? This is a scope divider. It's like a sub-folder. So `Getopt::Std` and `Getopt::Long` both exist inside a hierarchy with `Getopt` as the parent. It happens that there is a folder called `Getopt` and inside this are files called `Std.pm` and `Long.pm`. If you like hierarchy, you can make your libraries have this kind of structure, but it is not usually necessary.

Take a minute to view the documentation for `Getopt::Std` and `Getopt::Long`. You don't have to read them in depth, but just know that you can read the Perl documentation for most modules with a quick command-line:

```
$ perldoc Getopt::std
$ perldoc Getopt::Long
```

To use `Getopt::Std`, you must first define global variables called `$opt_something` where the *something* is a single letter. For example if you wanted a command-line option `-v` to indicate that the program should display its version number, you need a global variable called `$opt_v`. To define a global variable, you can use the `use vars` method or the `our` method (sort of like the `my` keyword except for global rather than lexical variables). Both syntaxes are displayed below.

You also have to tell `Getopt::Std` that you want to parse the command-line. You do this with the `getopts()` function. The syntax is a little strange. If the option takes arguments, you follow the letter with a colon. So, `getopts('x')` signals that `-x` takes no arguments while `getopts('x:')` signals that `-x` requires an argument. The example below shows how you can mix both kinds.

Try running the program below with a bunch of different options and see what happens. Note that the options are removed from the command-line. So `@ARGV` never contains the options.

```perl
#!/usr/bin/perl
# getopt.pl
use strict; use warnings;
use Getopt::Std;
use vars qw($opt_h $opt_v);
our $opt_p; # alternative to 'use vars'
getopts('hvp:');

my $VERSION = "1.0"; # it's a good idea to version your programs

my $usage = "
usage: getopt.pl [options] <arguments...>
options:
    -h  help
    -v  version
    -p  <some parameter>
";

```

```
19. if ($opt_h) {
20.     print $usage; # it's common to provide a -h to give help
21.     exit;
22. }
23.
24. if ($opt_v) {
25.     print "version ", $VERSION, "\n";
26.     exit;
27. }
28.
29. if ($opt_p) {
30.     print "Parameter is: $opt_p\n"
31. }
32.
33. print "Other arguments were: @ARGV\n";
```

Line 11 does something with a variable assignment that we have not seen before. The `$usage` variable is assigned the contents of several lines of text. The closing `"` that ends the assignment does not appear until line 17. This is an easier way of getting newlines into a variable.

Lines 21 and 26 introduce the exit() function. This terminates the program immediately without producing an error message (unlike the `die()` function). We use the `exit()` function at places when we deliberately want to stop the program.

P26. References and complex data structures

One of the reasons Perl is so powerful is that you can create complex data structures very easily. In this final section, we give a brief introduction to references, which are the foundation of complex data structures and other advanced programming concepts. Once you start to use references, you will find that they open up a whole new level of programming.

Multi-dimensional Arrays

So far, all of our arrays have been one-dimensional. But you can make them multi-dimensional with ease. If you assume the extra dimensions exist, Perl will create them for you.

```
my @matrix;
$matrix[0][0] = 1;
$matrix[0][1] = 5;
$matrix[1][0] = 3;
$matrix[1][1] = 2;
for (my $i = 0; $i <= 1; $i++) {
    for (my $j = 0; $j <= 1; $j++) {
        print "value at $i, $j is $matrix[$i][$j]\n";
    }
}
```

References

Up to now, we have never passed two arrays or hashes to a subroutine. Why? Because the arrays would get damaged by passing through `@_`. Consider the following code:

```
sub compare_two_arrays {
    my (@a, @b) = @_;
}
```

The intent is to fill up `@a` and `@b` from `@_`. Unfortunately, in list context, Perl cannot determine the size of the arrays. So what happens is that `@a` gets all of the data and `@b` gets none. But surely, we want to be able to make comparisons of arrays. To do this, we must turn an array into a scalar value. This is done quite simply with the backslash `\` operator. To dereference a particular element of the array, we use the arrow operator `->` and square brackets.

```
my @array = qw(cat dog cow);
my $array_ref = \@array;
print $array_ref->[0], "\n"; # prints cat
```

We can also create references to hashes and dereference them with the arrow operator. Note that we use curly brackets here to show that the scalar value is a reference to a hash:

```
my %hash = (cat => 'meow', dog => 'woof', cow => 'moo');
my $hash_ref = \%hash;
print $hash_ref->{cat}, "\n"; # prints meow
```

To dereference the entire array or hash, rather than a specific element, we use the `{}` operator as follows:

```
print join("\t", @{$array_ref}), "\n";
foreach my $key (keys %{$hash_ref}) {
    print $key, "\t", $hash_ref->{$key}, "\n";
}
```

Anonymous Data

You can create a multi-dimensional array in a single statement:

```
my @matrix = ( [1, 5], [3, 2], );
```

Here, the arrays in square brackets are references to arrays. But these arrays have no names, so they are called anonymous arrays.

In a multi-dimensional array, the first dimension is a *reference* to other dimensions. References are scalar values, but they *point* to arrays, hashes, and some other types. To *dereference* a scalar, you use the `->` notation. In a multi-dimensional context, the `->` symbols are implied. Previously, we used `$matrix[0][0]`, but this can also be understood more explicitly as `$matrix[0]->[0]`. But use the former syntax, not the latter.

When constructing multi-dimensional structures, the various dimensions can be hashes or arrays, or a mixture. The dimensions need not even be the same size:

```
my @matrix = (
    [1, 2],
    {cat => 'meow', dog => 'woof', cow => 'moo'},
    [{hello => 'world'}, {foo => 'bar'}],
);
print $matrix[0][1], "\n";
print $matrix[1]{cat}, "\n";
print $matrix[2][1]{foo}, "\n";
```

Records

One of the most common places you will see a reference is a hash reference. These are used to store record-like data.

```
my @authors = (
    {first => 'Ian',   last => 'Korf',    middle => 'F'},
    {first => 'Keith', last => 'Bradnam', middle => 'R'},
);

foreach $author (@authors) {
    print $author->{last}, ", ", $author->{first}, " ", $author->{middle}, ".\n";
}
```

What next?

If you’ve come this far, you’ve done very well. You can now pick up a variety of Perl books and start to learn more advanced and specialized topics. As always you will learn Perl much more quickly if you have some real-world problems that you need to write a script for. This doesn’t have to be work related, if you have any text files that contain data of some sort, then you can probably think of a Perl script to do something with that data. E.g. you could work out the average rating of each artist in your iTunes library by writing a script to parse the ‘iTunes Music Library.xml’ file that is produced by iTunes.

Sometimes the best way of improving your Perl is when you have to fix or improve someone else’s script. Seeing how other people code will give you ideas and make you realize what works well and what doesn’t. There is a lot of freely available Perl code on the web (just search Google for "Perl script to do x, y, and z) and you will often find that you can adapt from other people’s code. But it usually is much more fun to write your own!

Troubleshooting guide

Introduction

The next few pages list many of the common error messages that you might see if you are having problems with your Perl script. They are broadly divided into three categories:

1. Errors that are caused before your Perl code is even evaluated
2. Errors in the code itself (most commonly, very simple syntax errors)
3. Other mistakes (sometimes achieved by great feats of stupidity)

If there is a problem with your script, you will sometimes see a lot of errors appear when you try to run it. It pays to try to understand these error messages. With time, you will become quicker at fixing errors, or at least knowing where to look first.

Many code editors are specifically designed for working with programming languages, and they can help you hear and see problems as you create them. E.g. they might beep to warn you if you have entered too many closing brackets or parentheses. They almost certainly will color code that appears between pairs of quotation marks, so you quickly see if you have typed one quotation mark too many.

How to troubleshoot

Programming languages like Perl have sophisticated, and therefore complicated, debugging tools. But for simple scripts, these tools can be overkill. Here is some simpler advice to how to go about fixing your scripts:

1. Stay calm and don’t blame the computer. In nearly all cases, the computer is only ever doing what you have told it to do. Keeping a clear head will help you find the problem.
2. Check and re-check your code. Most errors are due to simple typos in your script, and sometimes you will be looking at the error without realizing that it is the error.
3. Start with the *first* error message that you see. Subsequent error messages often all stem from the first problem in your script. Fix one, and you may fix them all.
4. If you think a problem is due to an error on a single line of code, then you can comment out that line (by adding a `#` character to the start of the line). Then save and re-run your script to see if it now works. If it does, then you have confirmed which line contained the problem. Note that this is not appropriate for commenting out a single line of a block of code, e.g. the first line of an `if` statement.
5. Sometimes a program will partly work, but fail at some point within your code. Consider adding simple `print()` statements to work out where the program is failing.

Pre-Perl error messages

Permission denied

Do you have the permission to run your script, i.e. have you run `chmod` to add executable permissions? This won't affect scripts located on a USB flash drive, but in most 'real world' situations, you will always need to run the `chmod` command after creating your Perl script.

bad interpreter: No such file or directory

Most commonly caused by a typo in the first line of your script. The first line of a Perl script should let the Unix system know where it can find a copy of the Perl program that will understand your code. This will usually be `/usr/bin/perl`. If you miss the first forward slash, then the Unix system will try finding Perl (which is the interpreter of your code) in the wrong place, and hence things will fail.

command not found

You've either mistyped the name of the program or your program is not in a directory that the Unix system knows about (technically speaking the directory is not in your path). Make sure that scripts are kept in the Code directory (otherwise you will need to run them by using the perl command itself, e.g. `perl myscript.pl`).

Within-Perl error messages

Missing right curly or square bracket at script.pl line X , or… Unmatched right curly bracket at script.pl line X

Hopefully these two error messages are both very obvious. '['These are square brackets']' and '{'these are curly brackets'}'. Unless you are using them as text characters (e.g. within a print statement), then they always come in pairs. Make sure that yours are in pairs.

syntax error at script.pl line X, near YYY

Syntax errors are among the most frequent errors that you will see. On the plus side, they are usually very easy to fix. On the negative side, they can sometimes be very hard to spot as they frequently involve a single character that is either missing or surplus to requirements. Most commonly they might be because of:

- unmatched parentheses — like brackets, items that are in (parentheses) should always be a double act.
- missing semi-colon — If you start writing some code, then it has to end (at some point) with a semi-colon. The main exceptions to this rule are for the very first line of a script `#!/usr/bin/perl` or when a line ends in a closing curly bracket `}`. Also note that you can write one line of Perl code across several lines of your text editor, but this is still one line of code, and so needs one semi-colon.
- missing comma - Perl uses commas in many different ways, have you forgotten to include one in a place where Perl requires one?
- inventing new Perl commands and operators - if you write `if ($a === $b)`, then you have invented a new operator `===` which will cause a syntax error as Perl will have no idea what you mean.

Can't find string terminator '"' anywhere before EOF at script.pl line X

Did you make sure that you have pairs of quotation mark characters? If you have an odd number of single or double quotes characters, then you might see this error.

use of uninitialized variable in…

Your scripts will do many things with variables. You will add their values, calculate their lengths, and print their contents to the screen. But what if the variable doesn't actually contain any data? Maybe you were expecting to fill it with data from the command-line or from processing a file, but something went wrong? If you try doing something with a variable that contains no data, you will see this error.

Global symbol `$variable` requires explicit package name at

You wouldn't happen to be using the strict package and not declaring a variable with `my` would you? If you definitely have included `use strict;` then maybe check that all your variable names are spelled correctly. You might have introduced a variable as my `$apple` but then later incorrectly referred to it as `$appple`.

Other errors

Program changes not saved

If you make changes to your program but don't save them, then those changes will not be applied when you run the script. Always check that the script you are running is saved before you run it. If you are using a graphical text editor on an Apple computer, then you will always see a black dot within the red 'close window' icon on the top left of a window when there are any unsaved changes.

Program that you are editing is not the same as program you are running.

Occasionally, you might make copies of your programs and your directory might end up with programs named things like script1.pl, script1b.pl, script2.pl, new_script2.pl. This is a bad habit to get into and you might find yourself editing one script but trying to run another script. You will become very frustrated when every change you make to your script has seemingly no effect.

Program runs with no errors but doesn't print any output

It might seem mysterious when your Perl program which you so carefully wrote, doesn't seem to do anything. It is therefore worth asking yourself the question 'did I ask it to do anything?'. More specifically, have you made sure your program is printing out any output. Making your program calculate the answer to the life, the universe, and everything is one thing…but if you don't print out the answer, then it will remain a mystery.

Table of common Perl error messages

Error message	Description/Solution
Argument “xyz” isn’t numeric…	Perl is expecting a number, and you have given Perl something else, e.g. some text, or a variable containing text.
Array found where operator expected…	An operator (e.g. `+`, `==`, `>`, `eq`.) is missing and an array name has been used instead
bad interpreter: No such file or directory…	Check the 1st line of your script `#!/usr/bin/perl`. You have probably made a typo?
Bareword found where operator expected…	Most likely due to a simple typo in a Perl operator. e.g. typing ‘eqq’ rather than ‘eq’
Can’t find string terminator “"’ anywhere before EOF…	Probably a mismatched pair of quotation marks. These should come in pairs.
Can’t locate xyz.pm in …	You’ve added a ‘use’ statement, but the module name you are trying to use does not exist. Typo?
command not found	Possible typo when you typed the script name in the terminal. Or the script might not be in the Code directory.
Global symbol `$variable` requires explicit package name at…	You probably forgot to include the `use strict`
Permission denied… _ | Have you run the chmod command to give your script executable permission? | print () on unopened filehandle…_	If your script is printing output to a file, you have to first open a filehandle for the output file
Scalar found where operator expected… | An operator (e.g. `+`, `==`, `>`, `eq`.) is missing and a variable or array/hash element has been used instead | Search pattern not terminated…_	When you use the matching operator `=~`, there should be a pair of forward slashes surrounding the search pattern
String found where operator expected…	An operator (e.g. `+`, `==`, `>`, `<=`, `eq`, etc.) is missing, and some text has been added in it’s place
syntax error at…	Often due to a missing semi-colon/comma, or other typo (e.g. typing ‘iff’ instead of ‘if’)
use of uninitialized variable in…	You are working with a variable (or array/hash element) that doesn’t contain any data, even though it probably should. This is more common when the data is coming from a file or is specified on the command-line.

Version history

3.1.2 - 2016–10–16 - Many more small changes to keep links working

- Updated every broken external URL
- Updated references to unixandcode.com to use rescuedbycode.com
- Removed some references to text editors which are no longer in use
- Added a new shameless plug for the new book that we're currently writing

3.1.1 - 10/30/12 - Mostly small changes:

- Correction to Unix pipeline in Project 4
- New First steps section to clarify what should be done first with the downloaded files
- Removed part that wrongly stated that Factorial function will be used more than once in Project 0
- Fixed typo in Project 1
- Clarified example P13.1
- Clarified example P14.3

3.1.0 - 9/27/12 - Three new projects added. Updated information about book. Various small typos fixed.

3.0.0 - 3/5/12 - Major update to switch to using Markdown as the preferred underlying text format for this primer. This has also allowed us to tidy up some smaller typos which have been reported to us and make some of the styles that we use a little more consistent. Other changes:

- updated section on code editors
- changed text to use 'we' rather than 'I' (there are two authors!)
- added more hyperlinks to Perl functions
- added more hyperlinks between documents sections (where relevant)
- reworded many examples to make things clearer
- added Monty Python reference (a Python reference in Perl!)
- added small section to introduce newer style of filehandle in addition to older style
- added small section on the (preferred) 3-argument form of opening files
- several new snippets of example code to help explain regular expression metacharacters in a bit more detail
- Expanded the advice for Project 4 and clarified the steps involved
- Expanded explanations regarding subroutines P20 and lexical scope P21.

2.3.7 - 2/21/10 - Some minor changes (thanks to Claudius Kerth for raising most of these):

- removed unnecessary square brackets from `tr` example of task P6.12
- fixed a link to a wrong section number in Project 3
- simplified code in task P18.1
- fixed bugs in tasks [P20.1] and P22.3.

2.3.6 - 10/27/10 - Fixed some small typos, corrected some line numbers for a script and added one more clue in a section
which seems to confuse lots of people (Task 6.12). Also added links to Fraise (the successor to Smultron) and explained
that Smultron is only Mac OS 10.5 compatible whereas Fraise is only Mac OS 10.6 compatible.

2.3.5 - 5/24/10 - Fixed a single typo in exercise P20 which would prevent the script working properly.

2.3.4 - 11/13/09 - a couple of typo fixes and slight restructuring to transliteration routine

2.3.3 - 10/30/09 - Expanded on arrays and loops sections. More explanatory text is given with more examples.

2.3.2 - 10/16/09 - Added a new section on how to trouble-shoot problematic Perl scripts, with explanations of common
error messages. Plus some more minor typo fixes.

2.3.1 - 10/9/09 - Minor typo fixes

2.3 - 9/29/09 - One new Perl task added to introduce the die function slightly earlier. Added new Unix task to learn about converting newline characters.

2.2.1 - 8/2/09 - Fixed incorrect numbering for list of projects in Project 5 section

2.2 - 7/28/09 - Big change in that all examples (apart from first few) now have ‘use strict’. Changed some examples to
be more biologically relevant. Added more hyperlinks for Perl functions. Added graphical example of arrays. Expanded
explanations in many examples, particularly the section on subroutines which gains many new examples.

2.1 - 7/22/09 - Added Preamble section to explain how to go about this course on a Windows machine. Added author bios.
Changed directory structure for course files so that everything is contained within one parent directory (‘Unix_and_Perl_course’). Broke several of the Unix sections into smaller sections

2.05 - 7/17/09 - Some minor typos fixed

2.04 - 7/16/09 - Fixed minor typos. Expanded section on variables, and offered advice on variable names. Simplified some
print examples.

2.03 - 7/15/09 - Fixed minor typos. Expanded table of useful commands. Expanded explanation of tr operator, @ARGV, and
escaping via backslash character. Fixed E.coli project example. Mentioned how to spot unsaved files in Mac editors.
Moved tables inline.

2.02 - 7/14/09 - ‘use warnings’ reinstated to all scripts. Table of contents added. Hash bang line also included in all scripts. A few new sections added to Unix part of course, including a table of commonly used Unix commands. Lots of
small of formatting changes to stop sections splitting over pages where possible.

2.01 - 7/13/09 - Miscellaneous typos fixed and text reworded to clarify.

2.00 - 7/12/09 - Revision based on feedback from course. Switched to PDF documentation rather than HTML. Smaller, more
focused exercises.

1.00 - First taught course to grad students in UC Davis in Fall 2008

0.5 - Brief Unix/Perl training material written to help new students who join our lab