Introduction to Perl

Kirrily Robert

Introduction to Perl

by Kirrily Robert

Open Publications License 1.0

Table of Contents

List of Tables

Chapter 1. Introduction

Welcome to Netizen's Introduction to Perl training module. This is a one-day training module in which you will learn how to program in the Perl programming language.

1.1. Course outline

- What is Perl? (30 minutes)
- Creating and running a Perl program (45 minutes)
- *Morning tea* (15 minutes)
- Variable types (45 minutes)
- Operators and Functions (60 minutes)
- *Lunch break* (60 minutes)
- Conditional constructs (45 minutes)
- Subroutines (30 minutes)
- *Afternoon tea* (15 minutes)
- Regular expressions (45 minutes)
- Practical exercises (until finish)

1.2. Assumed knowledge

To gain the most from this course, you should:

- Be able to use the Unix operating system

    - Move around the file system
    - Create and edit files
    - Run programs

- Have programmed in least one other language and
    - Understand variables, including data types and arrays
    - Understand conditional and looping constructs
    - Understand the use of subroutines and/or functions

1.3. Module objectives

- Understand the history and philosophy behind the Perl programming language
- Know where to find additional information about Perl
- Write simple Perl scripts and run them from the Unix command line
- Use Perl's command line options to enable warnings
- Understand Perl's three main data types and how to use them
- Use Perl's `strict` pragma to enforce lexical scoping and better coding
- Understand Perl's most common operators and functions and how to use them
- Understand and use Perl's conditional and looping constructs
- Understand and use subroutines in Perl
- Understand and use simple regular expressions for matching and substitution

1.4. Platform and version details

This module is taught using Unix or a Unix-like operating system. Most of what is learnt will work equally well on Windows NT or other operating systems; your instructor will inform you throughout the course of any areas which differ.

All Netizen's Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs signficantly from previous versions of Perl, so you will need a Perl 5 interpreter to use what you have learnt. However, older Perl programs should work fine under Perl 5.

1.5. The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what ap-
pears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appears *`like this`*

Notes and tips appear offset from the text like this.

Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

1.6. Other materials

In addition to these notes, you should have a copy of the required text book for this course: Programming Perl (2nd edition) by Larry Wall, Tom Christiansen and Randal L. Schwartz, more commonly referred to as "the Camel book". The Camel book will be used throughout the day, and will be a valuable reference to take home and keep next to your computer.

You will also have received a floppy disk containing these notes in HTML form (with working links to external resources etc) and all the exercises and data used in this course.

Chapter 2. What is Perl

2.1. In this chapter...

This section describes Perl and its uses. You will learn about this history of Perl, the main areas in which it is commonly used, and a little about the Perl community and philosophy. Lastly, you will find out how to get Perl and what software comes as part of the Perl distribution.

2.2. Perl’s name

Perl has been said to stand for "Practical Extraction and Reporting Language" (by it’s fans) or "Pathologically Eclectic Rubbish Lister" (by its detractors). In fact, Perl is not an acronym; it’s a shortened version of the program’s original name, "pearl", and when you’re talking about the language it’s spelt with a capital "P" and lowercase "erl", not all capitals as is sometimes seen (especially in job advertisements posted by contract agencies). When you’re talking about the Perl interpreter, it’s spelt in all lower case: **perl**.

Perl has been described as everything from "line noise" to "the Swiss-army chainsaw of programming languages". The latter of these nicknames gives some idea of how programmers see Perl - as a very powerful tool that does just about everything.

2.3. Typical uses of Perl

2.3.1. Text processing

Perl’s original main use was text processing. It is exceedingly powerful in this regard,

and can be used to manipulate textual data, reports, email, news articles, log files, or just about any kind of text, with great ease.

2.3.2. System administration tasks

System administration is made easy with Perl. It's particularly useful for tying together lots of smaller scripts, working with file systems, networking, and so on.

2.3.3. CGI and web programming

Since HTML is just text with built-in formatting, Perl can be used to process and generate HTML. Perl is probably the most popular language around for web development, and there are many tools and scripts available for free.

2.3.4. Database interaction

Perl's DBI module makes interacting with all kinds of databases --- from Oracle down to comma-separated variable files --- easy and portable. Perl is increasingly being used to write large database applications, especially those which provide a database backend to a website.

2.3.5. Other Internet programming

Perl modules are available for just about every kind of Internet programming, from Mail and News clients, interfaces to IRC and ICQ, right down to lower level Socket programming.

2.3.6. Less typical uses of Perl

Perl is used in some unusual places as well. The Human Genome Project relies on Perl for DNA sequencing, NASA use Perl for satellite control, PDL (Perl Data Language, pron. "piddle") makes number-crunching easy, and there is even a Perl Object Environment (POE) which is used for event-driven state machines.

2.4. What is Perl like?

The following (somewhat paraphrased) article, entitled "What is Perl", comes from The Perl Journal (http://www.tpj.com/) (Used with permission.)

Perl is a general purpose programming language developed in 1987 by Larry Wall. It has become the language of choice for WWW development, text processing, Internet services, mail filtering, graphical programming, and every other task requiring portable and easily-developed solutions.

Perl is interpreted. This means that as soon as you write your program, you can run it - there’s no mandatory compilation phase. The same Perl program can run on Unix, Windows, NT, MacOS, DOS, OS/2, VMS and the Amiga.

Perl is collaborative. The CPAN software archive contains free utilities written by the Perl community, so you save time.

Perl is free. Unlike most other languages, Perl is not proprietary. The source code and compiler are free, and will always be free.

Perl is fast. The Perl interpreter is written in C, and a decade of optimisations have resulted in a fast executable.

Perl is complete. The best support for regular expressions in any language, internal support for hash tables, a built-in debugger, facilities for report generation, networking functions, utilities for CGI scripts, database interfaces, arbitrary-precision arithmetic - are all bundled with Perl.

Perl is secure. Perl can perform "taint checking" to prevent security breaches. You can also run a program in a "safe" compartment to avoid the risks inherent in executing unknown code.

Perl is open for business. Thousands of corporations rely on Perl for their information processing needs.

Perl is simple to learn. Perl makes easy things easy and hard things possible. Perl handles tedious tasks for you, such as memory allocation and garbage collection.

Perl is concise. Many programs that would take hundreds or thousands of lines in other programming languages can be expressed in a pageful of Perl.

Perl is object oriented. Inheritance, polymorphism, and encapsulation are all provided by Perl's object oriented capabilities.

Perl is flexible The Perl motto is "there's more than one way to do it." The language doesn't force a particular style of programming on you. Write what comes naturally.

Perl is fun. Programming is meant to be fun, not only in the satisfaction of seeing our well-tuned programs do our bidding, but in the literary act of creative writing that yields those programs. With Perl, the journey is as enjoyable as the destination.

2.5. The Perl Philosophy

2.5.1. There's more than one way to do it

The Perl motto is "there's more than one way to do it" - often abbreviated TMTOWTDI. What this means is that for any problem, there will be multiple ways to approach it using Perl. Some will be quicker, more elegant, or more readable than others, but that doesn't make them *wrong*.

2.5.2. A correct Perl program...

"... is one that does the job before your boss fires you." That's in the preface to the Camel book, which is highly recommended reading.

Of course, some Perl programs are more correct than others, but while elegance is a fine thing to strive for, most Perl people realise that soemtimes you just have to write a quick and dirty hack that'll keep things running for the mean time. If you get the time to make it beautiful later, so much the better.

2.5.3. Three virtues of a programmer

The Camel book contains the following entries in its glossary:

2.5.3.1. Laziness

The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it. Hence, the first great virtue of a programmer.

2.5.3.2. Impatience

The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to. Hence, the second great virtue of a programmer.

2.5.3.3. Hubris

Excessive pride, the sort of thing Zeus zaps you for. Also the quality that makes you write (and maintain) programs that other people won't want to say bad things about. Hence, the third great virtue of a programmer.

2.5.4. Three more virtues

In his "State of the Onion" keynote speech at The Perl Conference 2.0 in 1998, Larry Wall described another three virtues, which are the virtues of a community of programmers. These are:

- Diligence
- Patience
- Humility

You may notice that these are the opposites of the first three virtues. However, they are equally necessary for Perl programmers who wish to work together, whether on a software project for their company or on an Open Source project with many contributors around the world.

2.5.5. Share and enjoy!

Perl is Open Source software, and most of the modules and extensions for Perl are also released under Open Source licenses of various kinds (Perl itself is released under dual licenses, the GNU General Public License and the Artistic License, copies of which are distributed with the software).

The culture of Perl is fairly open and sharing, and thousands of volunteers worldwide have contributed to the current wealth of software and knowledge available to us. If you have time, you should try and give back some of what you've received from the Perl community. Contribute a module to CPAN, help a new Perl programmer to debug her programs, or write about Perl and how it's helped you. Even buying books written by the Perl gurus (like many of the O'Reilly Perl books) helps give them the financial means to keep supporting Perl.

2.6. Parts of Perl

2.6.1. The Perl interpreter

The main part of Perl is the interpreter. The interpreter is available for Unix, Windows, and many other platforms. The current version of Perl is 5.005, which is available from the Perl website (http://www.perl.com/) or any of a number of mirror sites (the Windows version is available from Activestate (http://www.activestate.com/). The next release of Perl will be version 5.6; the jump in version numbers is because it was felt that the number of additional features between releases warranted a larger difference between version numbers.

2.6.2. Manuals

Along with the interpreter come the manuals for Perl. These are accessed via the **perldoc** command or, on Unix systems, also via the **man** command. More than 30 manual pages come with the current version of perl. These can be found by typing **man perl** (or **perldoc perl** on non-Unix systems). The Perl FAQs (Frequently Asked Questions files) are available in perldoc format, and can be accessed by typing **perldoc perlfaq**

Watch while this is demonstrated; you'll get a chance to try it soon.

2.6.3. Perl Modules

Perl also comes with a collection of modules. These are Perl programs which carry out certain common tasks, and can be included as common libraries in any Perl script. Less commonly used modules aren't included with the distribution, but can be downloaded from (CPAN (http://www.perl.com/CPAN)) and installed separately.

2.7. Chapter summary

- Common uses of Perl include
  - text processing
  - system administration
  - CGI and web programming
  - other Internet programming
- Perl is a general purpose programming language, distributed for free via the Perl website (http://www.perl.com/) and mirror sites
- Perl includes excellent support for regular expressions, object oriented programming, and other features
- Perl allows a great degree of programmer flexibility - "There’s more than one way to do it".
- The three virtues of a programmer are Laziness, Impatience and Hubris. Perl will help you foster these virtues
- The three virtues of a programmer in a group environment are Diligence, Patience, and Humility.
- Perl is a collaborative language - everyone is free to contribute to the Perl software and the Perl community
- Parts of Perl include:
  - the Perl interpreter
  - documentation in several formats
  - library modules

Chapter 3. Creating and running a Perl program

3.1. In this chapter...

In this chapter we will be creating a very simple "Hello, world" program in Perl and exploring some of the basic syntax of the Perl programming language.

3.2. Logging into your account

Your username and password will have been given to you with these course notes.

Table 3-1. Details required to connect to the Netizen training server

Hostname or IP address	
Your username	
Your password	

1. Open the telnet program
2. Connect to the training server at the hostname or IP number given above
3. Login using the username and password you were given

You will find yourself at a Unix shell prompt. Hopefully (if you met the pre-requisites of this course) you will now be able to see that your account has a subdirectory called `exercises/` which are the example scripts and exercises given in these course notes.

If you're not quite up to speed with Unix, there's a cheat-sheet in Appendix A of these notes.

3.3. Using perldoc

On the command line, type **perldoc perl**. You will find yourself in the Perl documentation pages. Here's how to get around inside the documentation:

Table 3-2. Getting around in perldoc

Action	Keystroke
Page down	**SPACE**
Page up	**b**
Quit	**q**

3.4. Using the editor

A Perl script is just a normal text file, which means that you can edit it using a normal text editor.

The system you are using has several editors available for your use, including **vi**, **pico**, and others. Those who are not already familiar with **vi** should probably use **pico**, as it has a simpler interface. If you're an **emacs** user, sorry, our server doesn't have the resources to run it, but we do have other editors which have an **emacs** emulation mode.

To edit a file using pico, type:

```
% pico filename
```

(Note that the percent sign is your unix command line prompt - you don't have to type it.)

To edit a file using vi, type:

```
% vi filename
```

For other editors, just type the name of the editor followed by the name of the file you wish to edit.

A summary of editor commands appears in Appendix B in the back of these coursenotes, just in case you need them.

Incidentally, Appendix C contains a guide to pronouncing ASCII characters, especially punctuation. This will help you translate perl into spoken language, for ease of communication with other programmers.

3.5. Our first Perl program

We’re about to create our first, simple Perl script: a "hello world" program. There are a couple of things you should know in advance:

- Perl programs (or scripts --- the words are interchangeable) consist of a series of statements
- When you run the program, each statement is executed in turn, from the top of your script to the bottom. (There are two special cases where this doesn’t occur, one of which --- subroutine declarations --- we’ll be looking at later today)
- Each statement ends in a semi-colon
- Statements can flow over several lines
- Whitespace (spaces, tabs and newlines) are ignored most places in a Perl script.

Now, just for practice, open a file called `hello.pl` in your text editor. Type in the following one-line Perl program:

```
print "Hello, world!\n";
```

This one-line program calls the `print` function with a single parameter, the *string literal* "Hello, world!" followed by a newline character.

Save it and exit.

3.6. Running a Perl program from the command line

We can run the program from the command line by typing in:

```
perl hello.pl
```

You should see this output:

```
Hello, world!
```

This program should, of course, be entirely self-explanatory. The only thing you really need to note is the `\n` ("backslash N") which denotes a new line.

3.7. The "shebang" line

So what if we want to run our program from the command line without having to type in the name of the Perl interpreter first?

You can make a file executable by typing:

```
% chmod +x hello.pl
```

at the command line. (For more information about the **chmod** command, type **man chmod**).

In order to let the shell know what to do with our program when we try to run it with **./hello.pl** from the command line, we put the following line at the top of our program:

```
#!/usr/bin/perl
```

That's what we call a "shebang" line (because the # is a "hash" sign, and the `!` is referred to as a "bang", hence "hashbang" or "shebang"). It tells the system what to use to interpret our script. Of course, if the Perl interpreter were somewhere else on our system, we'd have to change the shebang line to reflect that.

3.8. Comments

Incidentally, comments in Perl start with a hash sign (#), either on a line on their own or after a statement. Anything after a hash is a comment.

```
# This is a hello world program
print "Hello, world!\n";        # print the message
```

3.9. Command line options

Perl has a number of command line options, which you can specify on the command line by typing **perl *`options`* hello.pl** or which you can include in the shebang line. Let's say you want to use the `-w` command line option to turn on warnings:

```
#!/usr/bin/perl -w
```

(Incidentally, it's always a good idea to turn on warnings while you're developing something.)

Setting the special variable `$^W` to a true value will locally disable warnings (i.e. in the current block).

A full explanation of command line options can be found in the Camel book on pages 330 to 337, or by typing **perldoc perlrun**.

3.10. Chapter summary

Here’s what you know about Perl’s operation and syntax so far:

- Perl programs typically start with a "shebang" line
- statements (generally) end in semicolons
- statements may span multiple lines; it’s only the semicolon that ends a statement
- comments are indicated by a hash (#) sign. Anything after a hash sign on a line is a comment.
- `\n` is used to indicate a new line
- whitespace is ignored almost everywhere
- command line arguments to Perl can be indicated on the shebang line
- the `-w` command line argument turns on warnings

Chapter 4. Perl variables

4.1. In this chapter...

In this section we will explore Perl’s three main variable types --- scalars, arrays, and hashes --- and learn to assign values to them, retrieve the values stored in them, and manipulate them in certain ways.

4.2. What is a variable?

A variable is a place where we can store data. Think of it like a pigeonhole with a name on it indicating what data is stored in it.

The Perl language is very much like human languages in many ways, so you can think of variables as being the "nouns" of Perl. For instance, you might have a variable called "total" or "employee".

4.3. Variable names

Variable names in Perl may contain alphanumeric characters in upper or lower case, and underscores. A variable name may not start with a number, though - that means something special, which we’ll encounter later. Likewise, variables that start with anything non-alphanumeric are also special, and we’ll discuss that later, too.

It’s standard Perl style to name variables in lower case, with underscores separating words in the name. For instance, `employee_number`. Upper case is usually used for constants, for instance `LIGHT_SPEED` or `PI`. Following these conventions will help make your Perl more maintainable and more easily understood by others.

Lastly, variable names all start with a punctuation sign depending on what sort of variable they are:

Table 4-1. Variable punctuation

Variable type	Starts with	Pronounced
Scalar	$	dollar
Array	@	at
Hash	%	Percent

(Don’t worry if those variable type names don’t mean anything to you. We’re about to cover it.)

4.4. Variable scoping and the strict pragma

Many programming languages require you to "pre-declare" variables -- that is, say that you’re going to use them before you use them. Variables can either be declared as global (that is, they can be used anywhere in the program) or local (they can only be used in the same part of the program in which they were declared).

In Perl, it is not necessary to declare your variables before you begin. You can summon a variable into existence simply by using it, and it will be globally available to any routine in your program. If you’re used to programming in C or any of a number of other languages, this may seem odd and even dangerous to you. This is, in fact, the case.

4.4.1. Arguments in favour of strictness

- avoids accidental creation of unwanted variables when you make a typing error

- avoids scoping problems, for instance when a subroutine uses a variable with the same name as a global variable
- allows for warnings if values are assigned to variables and never used

4.4.2. Arguments against strictness

- takes a while to get used to, and may slow down development until it becomes instinctual
- enforces a nasty, fascist style of coding which isn't nearly as much fun

Sometimes a little bit of fascism is a good thing, like when you want the trains to run on time. Because of this, Perl lets you turn strictness on if you want it, using something called the *strict pragma*. A pragma, in Perl-speak, is a set of rules for how your code is to be dealt with.

Other effects of the strict pragma are discussed on page 500 of the Camel.

4.4.3. Using the strict pragma

In the interests of bug-free code and teaching better Perl style, we're going to use the strict pragma throughout this training course. Here's how it's invoked:

```
#!/usr/bin/perl -w
use strict;
```

That typically goes at the top of your program, just under your shebang line and introductory comments.

Once we use the strict pragma, we have to explicitly declare new variables using `my`. You'll see this in use below, and it will be discussed again later when we talk about blocks and subroutines.

Try running the program `exercises/strictfail.pl` and see what happens. What needs to be done to fix it? Try it and see if it works. By the way, get used to this error message - it's one of the most common Perl programming mistakes, though it's easily fixed.

There's more about use of `my` on page 189 of the Camel.

4.5. Scalars

The simplest form of variable in Perl is the scalar. A scalar is a single item of data such as:

- Arthur
- Just Another Perl Hacker
- 42
- 0.000001
- 3.27e17

Here's how we assign values to scalar variables:

```
my $name = "Arthur";
my $whoami = 'Just Another Perl Hacker';
my $meaning_of_life = 42;
my $number_less_than_1 = 0.000001;
my $very_large_number = 3.27e17;          # 3.27 by 10 to the power of 17
```

There are other ways to assign things apart from the = operator, too. They're covered on pages 92-93 of the Camel.

As you can see, a scalar can be text of any length, and numbers of any precision (machine dependent, of course). Perl magically converts between them when it needs to. For instance, it's quite legal to say:

```
# adding an integer to a floating point number
my $sum = $meaning_of_life + $number_less_than_1;

# here we're putting the int in the middle of a string we
# want to print
print "$name says, 'The meaning of life is $mean-
ing_of_life.'\n";
```

This may seem extraordinarily alien to those used to strictly typed languages, but believe it or not, the ability to transparently convert between variable types is one of the great strengths of Perl. Some people say that it's also one of the great weaknesses.

You can explicitly cast scalars to various specific data types. Look up `int()` on page 180 of the camel, for instance.

4.6. Double and single quotes

While we're here, let's look at the assignments above. You'll see that some have double quotes, some have single quotes, and some have no quotes at all.

In Perl, quotes are required to distinguish strings from the language's reserved words or other expressions. Either type of quote can be used, but there is one important difference: double quotes can include other variable names inside them, and those variables will then be interpolated - as in the last example above - while single quotes do not interpolate.

```
# single quotes don't interpolate...
my $price = '$9.95';

# double quotes interpolate...
my $invoice_item = "24 widgets at $price each\n";

print $invoice_item;
```

The above example is available in your directory as `exercises/interpolate.pl` so you can experiment with different kinds of quotes.

Note that special characters such as the `\n` newline character are only available within double quotes. Single quotes will fail to expand these special characters just as they fail to expand variable names.

When using either type of quotes, you must have a matching pair of opening and closing quotes. If you want to include a quote mark in the actual quoted text, you can escape it by preceding it with a backslash:

```
print "He said, \"Hello!\"\n";
```

You can also use a backslash to escape other special characters such as dollar signs within double quotes:

```
print "The price is \$300\n";
```

To include a literal backslash in a double-quoted string, use two backslashes: `\\`

There are special quotes for executing a string as a shell command (see "Input operators" on page 52 of the Camel), and also special quoting functions (see "Pick your own quotes" on page 41).

4.6.1. Exercises

1. Write a script which sets some variables:
    a. your name
    b. your street number
    c. your favourite colour

2. Print out the values of these variables using double quotes for variable interpolation
3. Change the quotes to single quotes. What happens?

4. Write a script which prints out `C:\WINDOWS\SYSTEM\` twice -- once using double quotes, once using single quotes. How do you have to escape the backslashes in each case?

You'll find answers to the above in `exercises/answers/scalars.pl`.

4.7. Arrays

If you think of a scalar as being a singular thing, arrays are the plural form. Just as you have a flock of sheep or a wunch of bankers, you can have an array of scalars.

An array is a list of (usually related) scalars all kept together. Arrays start with an `@` (at sign), and are initialised thus:

```
my @fruit = ("apples", "oranges", "guavas", "passion-
fruit", "grapes");
my @magic_numbers = (23, 42, 69);
my @random_scalars = ("mumble", 123.45, "willy the wombat", -
300);
```

As you can see, arrays can contain any kind of scalars. They can have just about any number of elements, too, without needing to know how many before you start. *Really* any number - tens or hundreds of thousands, if you've got the memory.

Arrays are discussed on page 6 of the Camel or by typing **perldoc perldata**.

So if we don't know how many items there are in an array, how can we find out? Well, there are a couple of ways.

First of all, Perl's arrays are indexed from zero. We can access individual elements of the array like this:

```
print $fruits[0];                # prints "apples"
print $random_scalars[2];        # prints "willy the wombat"
```

Wait a minute, why are we using dollar signs in the example above, instead of at signs? The reason is this: we only want a scalar back, so we show that we want a scalar. There's a useful way of thinking of this, which is explained in chapter 1 of the Camel: if scalars are the singular case, then the dollar sign is like the word "the" - "the name", "the meaning of life", etc. The `@` sign on an array, or the `%` sign on a hash, is like saying "those" or "these" - "these fruit", "those magic numbers". However, when we only want one element of the array, we'll be saying things like "the first fruit" or "the last magic number" - hence the scalar-like dollar sign.

If we wanted what we call an *array slice* we could say:

```
@fruits[1,2,3];          # oranges, guavas, passionfruit
@magic_numbers[0..1];    # 23, 42
```

You just learnt something new, by the way: the `..` ("dot dot") range operator (see pages 90-91 of your Camel or **perldoc perlop**) which creates a temporary list of numbers between the two you specify - in this case 0 and 1, but it could have been 1 and 100 if we'd had an array big enough to use it on. You'll run into this operator again and again, so remember it.

Another thing you can do with arrays is insert them into a string, the same as for scalars:

```
print "My favourite fruits are @fruits\n";      # whole array
print "Two types of fruit are @fruits[0,2]";    # array slice
```

Returning to the point, how do we find the last element in an array? Well, there's a special variable called `$#array` which is the index of the last element, so you can say:

```
@fruit[0..$#fruit];
```

and you'll get the whole array. If you print `$#fruit` you'll find it's 4, which is not the same as the number of elements - 5. Remember that it's the *index of the last element* and that the index *starts at zero*, so you have to add one to it to find out how many elements in the array.

But wait! There's More Than One Way To Do It - and an easier way, at that. If you evaluate the array in a scalar context - that is, do something like this:

```
my $fruit_count = @fruits;
```

... you'll get the number of elements in the array.

There's more than two ways to do it, as well - `scalar(@fruits)` and `int(@fruits)` will also tell us how many elements there are in the array.

4.7.1. A quick look at context

There's a term you've heard used just recently but which hasn't been explained: *context*.

All Perl expressions are evaluated in a context. The two main contexts are:

- scalar context, and
- list context

Here's an example of an expression which can be evaluated in either context:

```
my $howmany = @array;           # scalar context
my @newarray = @array;          # list context
```

If you look at an array in a scalar context, you'll see how many elements it has; if you look at it in list context, you'll see the contents of the array itself.

4.7.2. What's the difference between a list and an array?

Not much, really. A list is just an unnamed array. Here's a demonstration of the difference:

```
# printing a list of scalars
print ("Hello", " ", $name, "\n");

# printing an array
@hello = ("Hello", " ", $name, "\n");
print @hello;
```

If you come across something that wants a LIST, you can either give it the elements of list as in the first example above, or you can pass it an array by name. If you come across something that wants an ARRAY, you have to actually give it the name of an array.

List values and Arrays are covered on page 47 of the Camel.

4.7.3. Exercises

1. Create an array of your friends' names
2. Print out the first element
3. Print out the last element
4. Print out the array from within a double-quoted string using variable interpolation
5. Print out an array slice of the 2nd to 4th items using variable interpolation

Answers to the above can be found in `exercises/answers/arrays.pl`

4.7.4. Advanced exercises

1. Print the array without putting quotes around its name. What happens?
2. Set the special variable `$,` to something appropriate and try the previous step again (see page 132 of your Camel for this variable's documentation)

3. What happens if you have a small array and then you assign a value to `$array[1000]`?

Answers to the above can be found in `exercises/answers/arrays_advanced.pl`

4.8. Hashes

A hash is a two-dimensional array which contains keys and values. Instead of looking up items in a hash by an array index, you can look up values by their keys.

Hashes are covered in the Camel on pages 7-8, then in more detail on page 50 or in **perldoc perldata**.

4.8.1. Initialising a hash

Hashes are initalised in exactly the same way as arrays, with a comma separated list of values:

```
my %monthdays = ("January", 31, "Febru-
ary", 28, "March", 31, ...);
```

Of course, there's more than one way to do it:

```
my %monthdays = (
        "January"       =>      31,
        "February"      =>      28,
        "March"         =>      31,
        ...
);
```

The spacing in the above example is commonly used to make hash assignments more readable.

The => operator is syntactically the same as the comma, but is used to distinguish hashes more easily from normal arrays. Also, you don't need to put quotes on the item which comes immediately before the => operator:

```
my %monthdays = (
        January       =>      31,
        February      =>      28,
        March         =>      31,
        ...
);
```

4.8.2. Reading hash values

You get at elements in a hash by using the following syntax:

```
print $monthdays{"January"};    # prints 31
```

Again you'll notice the use of the dollar sign, which you should read as "the monthdays belonging to January".

4.8.3. Adding new hash elements

You can also create elements in a hash on the fly:

```
my %monthdays = ();
$monthdays{"January"} = 31;
$monthdays{"February"} = 28;
...
```

4.8.4. Other things about hashes

- Hashes have no internal order
- There is no equivalent to `$#array` to get the size of a hash
- However, there are functions such as `each()`, `keys()` and `values()` which will help you manipulate hash data. We look at these later, when we deal with functions.

You may like to look up the following functions which related to hashes: `keys()`, `values()`, `each()`, `delete()`, `exists()`, and `defined()`.

4.8.5. What's the difference between a hash and an associative array?

Back in the days of Perl version 4 (and earlier), hashes were called associative arrays. The name "hash" is now preferred because it's much quicker to type. If you consider all the times that hashes are talked about in the newsgroup comp.lang.perl.misc (news:comp.lang.perl.misc) and other Perl newsgroups, the renaming of associative arrays to hashes has resulted in a major saving of bandwidth.

4.8.6. Exercises

1. Create a hash of people and something interesting about them
2. Print out a given person's interesting fact
3. Change an person's interesting fact
4. Add a new person to the hash
5. What happens if you try to print an entry for a person who's not in the hash?

Answers to these exercises are given in `exercises/answers/hash.pl`

4.9. Special variables

Perl has many special variables. These are used to set or retrieve certain values which affect the way your program runs. For instance, you can set a special variable to turn interpreter warnings on and off, or read a special variable to find out the command line arguments passed to your script.

Special variables can be scalars, arrays, or hashes. We'll look at some of each kind.

Special variables are discussed at length in chapter 2 of your Camel book (from page 127 onwards) and in the `perlvar` manual page. You may also like to look up the `English` module, which lets you use longer, more English-like names for special variables. You'll find this information in chapter 7, on page 403, or use **perldoc English** to read the module documentation.

4.9.1. The first special variable, $_

The first special variable, and possibly the one you'll encounter most often, is called `$_` ("dollar-underscore"), and it represents the current thing that your Perl script's working with - often a line of text or an element of a list or hash. It can be set explicitly, or it can be set implicitly by certain looping constructs (which we'll look at later).

The special variable `$_` is often the default argument for functions in Perl. For instance, the `print()` function defaults to printing `$_`

```
$_ = "Hello, world!\n";
print;
```

If you can think of Perl variables as being "nouns", then `$_` is the pronoun "it".

There's more discussion of using `$_` on page 131 of your Camel book.

4.9.1.1. Exercises

1. Set `$_` to a string like "Hello, world", then print it out by using the `print()`

command’s default argument

The answers to the above exercise are in `exercises/answers/scalars2.pl`.

4.9.2. @ARGV - a special array

Perl programs accept arbitrary arguments or parameters from the command line, like this:

```
perl printargs.pl foo bar baz
```

This passes "foo", "bar" and "baz" as arguments into our program, where they end up in an array called `@ARGV`. Try this script, which you’ll find in your directory. It’s called `exercises/printargs.pl`.

```
#!/usr/bin/perl -w

print "@ARGV\n";
```

To run the script, type:

```
% exercises/printargs.pl
```

You should see "foo bar baz" printed out.

4.9.2.1. Exercises

1. Modify your earlier array-printing script to print out the script’s command line arguments instead of the names of your friends. Call your script by typing **./scriptname.pl firstarg secondarg thirdarg** or similar.

The answers to the above exercise is in `exercises/answers/argv.pl`

4.9.3. %ENV - a special hash

Just as there are special scalars and arrays, there is a special hash called `%ENV`. This hash contains the names and values of environment variables. To view these variables under Unix, simply type **setenv** (C-type shells) or **export** (sh type shells) on the command line.

4.9.3.1. Exercises

1. A user's home directory is stored in the environment variable `HOME`. Print out your own home directory.

The answer to the above can be found in `exercises/answers/env.pl`

4.10. Chapter summary

- Perl variable names typically consist of alphanumeric characters and underscores. Lower case names are used for most variables, and upper case for global constants.
- The statement `use strict;` is used to make Perl require variables to be pre-declared and to avoid certain types of programming errors.
- There are three types of Perl variables: scalars, arrays, and hashes.
- Scalars are single items of data and are indicated by a dollar sign (`$`) at the beginning of the variable name.
- Scalars can contain strings, numbers, etc
- Strings must be delimited by quote marks. Using double quote marks will allow you to interpolate other variables and meta-characters such as `\n` (newline) into a string. Single quotes do not interpolate.

- Arrays are one-dimensional lists of scalars and are indicated by an at sign (`@`) at the beginning of the variable name.
- Arrays are initialised using a comma-separated list of scalars inside round brackets.
- Arrays are indexed from zero
- Item n of an array can be accessed by using `$arrayname[n]`
- The index of the last item of an array can be accessed by using `$#arrayname`.
- The number of elements in an array can be found by interpreting the array in a scalar context, eg `my $items = @array;`
- Hashes are two-dimensional arrays of keys and values, and are indicated by a percent sign (`%`) at the beginning of the variable name.
- Hashes are initialised using a comma-separated list of scalars inside curly brackets. Whitespace and the `=>` operator (which is syntactically identical to the comma) can be used to make hash assignments look neater.
- The value of a hash item whose key is foo can be accessed by using `$hashname{foo}`
- Hashes have no internal order
- `$_` is a special variable which is the default argument for many Perl functions and operators
- The special array `@ARGV` contains all command line parameters passed to the script
- The special hash `%ENV` contains information about the user's environment.

Chapter 5. Operators and functions

5.1. In this chapter...

In this chapter, we look at some of the operators and functions which can be used to manipulate data in Perl. In particular, we look at operators for arithmetic and string manipulation, and many kinds of functions including functions for scalar and list manipulation, more complex mathematical operations, type conversions, dealing with files, etc.

5.2. What are operators and functions?

Operators and functions are routines that are built into the Perl language to do stuff.

The difference between operators and functions in Perl is a very tricky subject. There are a couple of ways to tell the difference:

- Functions usually have all their parameters on the right hand side
- Operators can act in much more subtle and complex ways than functions
- Look in the documentation - if it's in **perldoc perlop**, it's an operator; if it's in **perldoc perlfunc**, it's a function. Otherwise, it's probably a subroutine.

The easiest way to explain operators is to just dive on in, so here we go:

5.3. Operators

There are lists of all the available operators, and what they each do, on pages 76-94 of the Camel. Precedence and associativity are also covered there.

5.3.1. Arithmetic operators

Arithmetic operators can be used to perform arithmetic operations on variables or constants. The commonly used ones are:

Table 5-1. Arithmetic operators

Operator	Example	Description
`+`	`$a + $b`	Addition
`-`	`$a - $b`	Subtraction
`*`	`$a * $b`	Multiplication
`/`	`$a / $b`	Division
`%`	`$a % $b`	Modulus (remainder when `$a is divided by $b, eg 11 % 3 = 2`)
`**`	`$a ** $b`	Exponentiation (`$a to the power of $b`)

Just like in C, there are some short cut arithmetic operators:

```
$a += 1;        # same as $a = $a + 1
$a -= 3;        # same as $a = $a - 3
$a *= 42;       # same as $a = $a * 42
```

(In fact, you can extrapolate the above with just about any operator - see page 17 of the Camel for more about this)

You can also use `$a++` and `$a----` if you're familiar with such things. `++$a` and `----$a` are also valid, but they do some slighty different things and you won't need them today (but you can read about them on pages 17 to 18 of the Camel if you are sufficiently interested).

5.3.2. String operators

Just as we can add and multiply numbers, we can also do similar things with strings:

Table 5-2. String operators

Operator	Example	Description
`.`	`$a . $b`	Concatenation (puts `$a` and `$b` together as one string)
`x`	`$a x $b`	Repeat (repeat `$a` `$b` times --- eg `"foo" x 3` gives us `"foofoofoo"`

There's more about the concatenation operator an the top of page 16 of the Camel.

5.3.3. Exercises

1. Calculate the cost of 18 widgets at $37.00 each and print the answer (Answer: `exercises/answers/widgets.pl`)
2. Print out a line of dashes without using more than one dash in your code (except for the `-w`). (Answer: `exercises/answers/dashes.pl`)
3. Use `exercises/operate.pl` to practice using arithmetic and string operators.

5.3.4. File operators

We can use file test operators to test various attributes of files and directories:

Table 5-3. File test operators

Operator	Example	Description
`-e`	`-e $a`	Exists - does the file exist?
`-r`	`-r $a`	Readable - is the file readable?
`-w`	`-w $a`	Writable - is the file writable?
`-d`	`-d $a`	Directory - is it a directory?
`-f`	`-f $a`	File - is it a normal file?
`-T`	`-T $a`	Text - is the file a text file?

There are examples of these in use on pages 19-20 of the Camel. There is a complete list of the file operators in the Camel on page 85. There are lots!

5.3.5. Other operators

You'll encounter all kinds of other operators in your Perl career, and they're all described in the Camel from page 76 onwards. We'll cover them as they become necessary to us -- you've already seen operators such as the assignment operator (=), the => operator which behaves a bit like the comma operator, and so on.

While we're here, let's just mention what "unary" and "binary" operators are.

A unary operator is one that only needs something on one side of it, like the file operators or the autoincrement (++) operator.

A binary operator is one that needs something on either side of it, such as the addition operator.

A trinary operator also exists, but we don't deal with it in this course. C programmers will probably already know about it, and can use it if they want.

5.4. Functions

A function is like an operator - and in fact some functions double as operators in certain conditions - but with the following differences:

- longer names
- can take any kinds of arguments
- arguments always come *after* the function name

The only real way to tell whether something is a function or an operator is to check the `perlop` and `perlfunc` manual pages and see which it appears in.

There's an introduction to functions on page 8 of the Camel, labeled 'Verbs'.

5.4.1. Types of arguments

Functions typically take the following kind of arguments:

SCALAR -- Any scalar variable - `42`, `"foo"`, or `$a`

LIST -- Any named or unnamed list (remember that a named list is an array)

ARRAY -- A named array; usually results in the array being modified

HASH -- Any named or unnamed hash

PATTERN -- A pattern to match on - we'll talk more about these later on, in Regular Expressions

FILEHANDLE -- A filehandle indicating a file that you've opened or one of the pseudo-files that is automatically opened, such as STDIN, STDOUT, and STDERR

There are other types of arguments, but you're not likely to need to deal with them in this module.

In chapter 3 of the Camel (starting on page 141) you'll see how the documentation describes what kind of arguments a function takes.

5.4.2. Return values

Just as a function can take arguments of various kinds, they can return various things for you to use - though they don't have to, and you don't have to use them if you don't want.

If a function returns a scalar, and we want to use it, we can say something like:

```
my $age = 29.75;
my $years = int($age);
```

and `$years` will be assigned the returned value of the `int()` function when given the argument `$age` - in this case, 29, since `int()` truncates instead of rounding.

If we just wanted to do something to a variable and didn't care what value was returned, we could just say:

```
my $input = <STDIN>;
chomp($input);
```

While we're at it, you should also know that the brackets on functions are optional if it's not likely to cause confusion. What's likely to cause confusion varies from one person to the next, but it's a pretty safe bet to use brackets as much as possible when you're starting out, and then drop them off if you see that other people are usually doing it. Seriously. You can learn a lot about Perl style by looking at other people's code, especially code found on CPAN or given as examples in Perl books, newsgroups, etc.

5.5. More about context

Many different functions and operators behave differently depending on whether they're called in *scalar context* or *list context*. Each one will be noted in its documentation, either in the Camel or in the manual pages.

Here are some Perl operators and functions that care about context:

Table 5-4. Context-senstive functions

What?	Scalar context	List context
`reverse()`	Reverses characters in a string	Reverses the order of the elements in an array
`each()`	Returns the next key in a hash	Returns a two-element list consisting of the next key and value pair in a hash
`gmtime()` and `localtime()`	Returns the time as a string in common format	Returns a list of second, minute, hour, day, etc
`keys()`	Returns the number of keys (and hence the number of elements) in a hash	Returns a list of all the keys in a hash
`readdir()`	Returns the next filename in a directory, or undef if there are no more	Returns a list of all the filenames in a directory

There are many other cases where an operation varies depending on context. Take a look at the notes on context at the start of **perldoc perlfunc** to see the official guide to this: "anything you want, except consistency".

You can also use **perldoc -f *`functionname`*** to get the documentation for just a single function.

5.6. Some easy functions

Starting on page 143 of the Camel book, there is a list of every single Perl function, their arguments, and what they do.

5.6.1. String manipulation

5.6.1.1. Finding the length of a string

The length of a string can be found using the `length()` function:

```
#!/usr/bin/perl -w

use strict;

my $string = "This is my string";
print length($string);
```

5.6.1.2. Case conversion

You can convert Perl strings from upper case to lower case, or vice versa, using the `lc()` and `uc()` functions, respectively.

```
#!/usr/bin/perl -w

print lc("Hello, World!");              # prints "hello, world!"
print uc("Hello, World!");              # prints "HELLO, WORLD!"
```

The `lcfirst()` and `ucfirst()` functions can be used to change only the first letter of a string.

```
#!/usr/bin/perl -w

print lcfirst("Hello, World!");         # prints "hello, World!"
print lcfirst(uc("Hello, World!"));     # prints "hELLO, WORLD!"
```

Notice how, in the last line of the example above, the `lcfirst()` operates on the result of the `uc()` function.

5.6.1.3. chop() and chomp()

The `chop()` function removes the last character of a string and returns that character.

```
#!/usr/bin/perl -w

use strict;

my $char = chop("Hello");               # $char is now equal to "o"

my $string = "Goodbye";

$char = chop $string;
print $char . "\n";                     # "e"
print $string . "\n";                   # "Goodby"
```

The `chomp()` works similarly, but *only* removes the last character if it is a newline. This is very handy for removing extraneous newlines from user input.

5.6.1.4. String substitutions with substr()

The `substr()` function can be used to return a portion of a string, or to change a portion of a string.

```
#!/usr/bin/perl -w

use strict;

my $string = "Hello, world!";
print substr($string, 0, 5);             # prints "Hello"

substr($string, 0, 5) = "Greetings";
print $string;                           # prints "Greet-
ings, world!"
```

5.6.2. Numeric functions

There are many numeric functions in Perl, including trig functions and functions for dealing with random numbers. These include:

- `abs()` (absolute value)
- `cos()`, `sin()`, and `atan2()`
- `exp()` (exponentiation)
- `log()` (logarithms)
- `rand()` and `srand()` (random numbers)
- `sqrt()` (square root)

5.6.3. Type conversions

The following functions can be used to force type conversions (if you really need them):

- `oct()`
- `int()`
- `hex()`
- `chr()`
- `ord()`
- `scalar()`

5.6.4. Manipulating lists and arrays

5.6.4.1. Stacks and queues

Stacks and queues are special kinds of lists.

A stack can be thought of like a stack of paper on a desk. Things are put onto the top of it, and taken off the top of it.

A queue, on the other hand, has things added to the end of it and taken out of the start of it. Queues are also referred to as "FIFO" lists (for "First In, First Out").

We can simulate stacks and queues in Perl using the following functions:

- `push()` -- add items to the end of a list
- `pop()` -- remove items from the end of a list
- `shift()` -- remove items from the start of a list
- `unshift()` -- add items to the start of a list

A queue can be created by `push`ing items onto the end of a list and `shift`ing them off the front.

A stack can be created by `push`ing items on the end of a list and `pop`ping them off.

5.6.4.2. Sorting lists

The `sort()` function, when used on a list, returns a sorted version of that list. It *does not* sort the list in place.

The `reverse()` function, when used on a list, returns the list in reverse order. It *does not* reverse the list in place.

```
#!/usr/bin/perl -w

my @list = ("a", "z", "c", "m");
my @sorted = sort(@list);
```

```
my @reversed = reverse(sort(@list));
```

5.6.4.3. Converting lists to strings, and vice versa

The `join()` function can be used to join together the items in a list into one string. Conversely, `split()` can be used to split a string into elements for a list.

5.6.5. Hash processing

The `delete()` function deletes an element from a hash.

The `exists()` function tells you whether a certain key exists in a hash.

The `keys()` and `values()` functions return lists of the keys or values of a hash, respectively.

5.6.6. Reading and writing files

The `open()` function can be used to open a file for reading or writing. The `close()` function closes a file after you’re done with it.

5.6.7. Time

The `time()` function returns the current time in Unix format (that is, the number of seconds since 1 Jan 1970).

The `gmtime()` and `localtime()` functions can be used to get a more friendly representation of the time, either in Greenwich Mean Time or the local time zone. Both can be used in either scalar or list context.

5.6.8. Exercises

These exercises range from easy to difficult. Answers are provided in the exercises directory (filenames are given with each exercise).

1. Create a scalar variable containing the phrase "There's more than one way to do it" then print it out in all upper-case (Answer: `exercises/answers/tmtowtdi.pl`)
2. Print a random number
3. Print a random item from an array (Answer: `exercises/answers/quotes.pl`)
4. Print out the third character of a word entered by the user as an argument on the command line (There's a starter script in `exercises/thirdchar.pl` and the answer's in `exercises/answers/thirdchar.pl`)
5. Print out the date for a week ago (the answer's in `exercises/answers/lastweek.pl`
6. Print out a sentence in reverse
   a. reverse the whole sentence
   b. reverse just the words

   (Answer: `exercises/answers/reverse.pl`)

5.7. Chapter summary

- Perl operators and functions can be used to manipulate data and perform other necessary tasks
- The difference between operators and functions is blurred; most can behave in either way

- Chapter 3 of your Camel book, **perldoc perlop**, **perldoc perlfunc**, and **perldoc -f *`functionname`*** can be used to find out detailed information about operators and functions.
- Functions can accept arguments of various kinds
- Functions may return scalars, lists etc
- Return values may differ depending on whether a function is called in scalar or list context

Chapter 6. Conditional constructs

6.1. In this chapter...

In this section, we look at Perl's various conditional constructs and how they can be used to provide flow control to our Perl programs. We also learn about Perl's meaning of Truth and how to test for truth in various ways.

6.2. What is a block?

The simplest block is a single statement, for instance:

```
print "Hello, world!\n";
```

Sometimes you'll want several statements to be grouped together logically. That's what we call a block. A block can be executed either in response to some condition being met, or as an independent chunk of code that's given a name.

Blocks always have curly brackets ({ and }) around them. In C and Java, curly brackets are optional in some cases - not so in Perl.

```
{
        $fruit = "apple";
        $howmany = 32;
        print "I'd like to buy $howmany $fruit" . "s.\n";
}
```

You'll notice that the body of the block is indented from the brackets; this is to improve readability. Make a habit of doing it.

The Camel book refers to blocks with curly braces around them as BLOCKs (in capitals). It discusses them on page 97.

6.2.1. Scope

Something that needs mentioning again at this point is the concept of variable scoping. You will recall that we use the `my` function to declare variables when we're using the `strict` pragma. The `my` also scopes the variables so that they are local to the *current block*

```
#!/usr/bin/perl -w

use strict;

my $a = "foo";

{                               # start a new block
        my $a = "bar";
        print "$a\n";           # prints bar
}

print $a;                       # prints foo
```

Now, onto the situations in which we'll encounter blocks.

6.3. What is a conditional statement?

A conditional statement is one which allows us to test the truth of some condition. For instance, we might say "If the ticket price is less than ten dollars..." or "While there are still tickets left..."

You've almost certainly seen conditional statements in other programming languages, so we'll just assume that you get the general idea.

Perl's conditional statements are listed and explained on pages 95-106 of the Camel.

6.4. What is truth?

Conditional statements invariably test whether something is true or not. Perl thinks something is true if it doesn't evaluate to zero (`0`), an empty string (`""`), or undefined.

```
42              # true
0               # false
"0"             # false, because perl switches it to a num-
ber when it
                # needs to
"wibble"        # true
$new_variable   # false (if we haven't set it to anything, it's
                # undefined)
```

The Camel discusses Perl's idea of truth on pages 20-21, including some odd cases.

6.5. Comparison operators

We can compare things, and find out whether our comparison statement is true or not. The operators we use for this are:

Table 6-1. Comparison operators

Operator	Example	Meaning
`==`	`$a == $b`	Equality (same as in C and other C-like languages)
`!=`	`$a != $b`	Inequality (again, C-like)
`<`	`$a < $b`	Less than
`>`	`$a > $b`	Greater than
`<=`	`$a <= $b`	Less than or equal to
`>=`	`$a >= $b`	Greater than or equal to

If we're comparing strings, we use a slightly different set of comparison operators, as

follows:

Table 6-2. String comparison operators

Operator	Meaning
`eq`	Equality
`ne`	Inequality
`lt`	Less than (in "asciibetical" order)
`gt`	Greater than
`le`	Less than or equal to
`ge`	Greater than or equal to

Some examples:

```
69 > 42                          # true
"0" == 3 - 3                     # true
’apple’ gt ’banana’              # false; apple is alphabeti-
cally before
                                 # banana
1 + 2 == "3com"                  # true -
3com is evaluated in numeric
                                 # context be-
cause we used == not eq
```

Assigning `undef` to a variable name undefines it again, as does using the `undef` function with the variable’s name as its argument.

6.5.1. Existence and Defined-ness

We can also check whether things are defined (something is defined when it’s had a value assigned to it), or whether an element of a hash exists.

To find out if something is defined, use Perl's `defined` function. You can't just use the name of the variable because the variable can be defined an still evaluate to false - for example, if you assign it the value `0`.

```
$skippy = "bush kangaroo";
if (defined($skippy)) {
        print "Skippy is defined.\n";
} else {
        print "Skippy is undefined.\n";
}
```

The `defined` function is described in the Camel on page 155.

To find out if an element of a hash exists, use the `exists` function:

```
my %animals = (
        "Skippy"        =>      "bush kangaroo",
        "Flipper"       =>      "faster than lighting",
);

if (exists($animals{"Blinky Bill"}) {
        print "Blinky Bill exists.\n";
} else {
        print "Blinky Bill doesn't exist.\n";
}
```

There's a bit of explanation of the difference between a hash key "existing" and being "defined" on page 164 of the Camel.

One last quick example to clarify existence, definedness and truth:

```
my %miscellany = (
        "apple"         =>      "red",          # exists, de-
fined, true
```

```
        "howmany"       =>      0,              # exists, de-
fined, false
        "koala"         =>      undef,          # exists, unde-
fined, false
);

if (exists($miscellany("wombat")) {             # doesn't exist
        print "Wombat exists\n";
} else {
        print "We have no wom-
bats here.\n";     # this will happen
}
```

6.5.2. Boolean logic operators

Boolean logic operators can be used to combine two or more Perl statements, either in a conditional test or elsewhere.

The short circuit operators come in two flavours: line noise, and English. Both do similar things but have different precedence. This causes great confusion. There are two ways of avoiding this: use lots of brackets, or read page 89 of the Camel book very, very carefully.

Alright, if you insist: `and` and `or` operators have very low precendence (i.e. they will be evaluated after all the other operators in the condition) whereas `&&` and `||` have quite high precedence and may require parentheses in the condition to make it clear which parts of the statement are to be evaluated first.

Table 6-3. Boolean logic operators

English-like	C-style	Example	Result

English-like	C-style	Example	Result
and	&&	$a && $b	True if both $a and $b are true; acts on $a then if $a is true, goes on to act on $b.
or	||	$a || $b	True if either of $a and $b are true; acts on $a then if $a is false, goes on to act on $b.

Here's how you can use them to combine conditions in a test:

```
$a = 1;
$b = 2;

$a == 1 and $b == 2                    # true
$a == 1 or $b == 5                     # true
$a == 2 or $b == 5                     # false
($a == 1 and $b == 5) or $b == 2       # true (parenthe-
sized expression
                                       # evaluated first)
```

6.5.3. Using boolean logic operators as short circuit operators

These operators aren't just for combining tests in conditional statements --- they can be used to combine other statements as well.

Here's a real, working example of the || short circuit operator:

```
open(INFILE, "input.txt") || die("Can't open input file: $!");
```

What is it doing?

The `open()` function can be found on page 191 of the Camel, if you want to look at how it works.

The `&&` operator is less commonly used outside of conditional tests, but is still very useful. Its meaning is this: If the first operand returns true, the second will also happen. As soon as you get a false value returned, the expression stops evaluating.

```
($day eq 'Friday') && print "Have a good weekend!\n";
```

The typing saved by the above example is not necessarily worth the loss in readability, especially as it could also have been written:

```
print "Have a good weekend!\n" if $day eq 'Friday';

if ($day eq 'Friday') {
        print "Have a good weekend!\n";
}
```

...or any of a dozen other ways. That's right, there's more than one way to do it.

The most common usage of the short circuit operators, especially `||` (or `or`) is to trap errors, such as when opening files or interacting with the operating system.

Short circuit operators are covered from page 89 of the Camel book.

6.6. Types of conditional constructs

You'll have noticed that we snuck in something new in the last section -- the `if` construct. It probably didn't surprise you much - you'll have seen something similar in just about every programming language. (Bonus points will *not* be given for naming programming languages which have no "if" construct.)

6.6.1. if statements

The `if` construct goes like this:

```
if (conditional statement) {
        BLOCK
} elsif (conditional statement) {
        BLOCK
} else {
        BLOCK
}
```

Both the `elsif` and `else` parts of the above are optional, and of course you can have more than one `elsif`. `elsif` is also spelt differently to other languages' equivalents - C programmers should take especial note to not use `else if`.

If you're testing for something negative, it can sometimes make sense to use the similar-but-opposite construct, `unless`.

```
unless (conditional statement) {
        BLOCK
}
```

There is no such thing as an "elsunless" (thank the gods!), and if you find yourself using an `else` with `unless` then you should probably have written it as an `if` test in the first place.

There's also a shorthand, and more English-like, way to use `if` and `unless`:

```
print "We have apples\n" if $apples;
print "Yes, we have no bananas\n" unless $bananas;
```

6.6.2. while loops

We can repeat a block while a given condition is true:

```
while (conditional statement) {
```

```
        BLOCK
}

while ($hunger) {
        print "Feed me!\n";
        $hunger--;
}
```

The logical opposite of this is the "until" construct:

```
until ($full) {
        print "Feed me!\n";
        $full++;
}
```

6.6.3. for and foreach

Perl has a `for` construct identical to C and Java:

```
for ($count = 0; $count <= $enough; $count++) {
        print "Had enough?\n";
}
```

However, since we often want to loop through the elements of an array, we have a special "shortcut" looping construct called `foreach`, which is similar to the construct available in some Unix shells. Compare the following:

```
# using a for loop

for ($i = 0; $i <= $#array; $i++) {
        print $array[$i] . "\n";
}

# using foreach

foreach (@array) {
```

```
        print "$_\n";
}
```

There are some examples of foreach in `exercises/foreach.pl`

`foreach(n..m)` can be used to automatically generate a list of numbers between `n` and `m`.

We can loop through hashes easily too, using the `keys` function to return the keys of a hash as an list that we can use:

```
foreach $key (keys %monthdays) {
        print "There are $monthdays{$key} days in $key.\n";
}
```

We'll look at hash functions later.

6.6.4. Exercises

1. Set a variable to a numeric value, then create an `if` statement as follows:
    a. If the number is less than 3, print "Too small"
    b. If the number is greater than 7, print "Too big"
    c. Otherwise, print "Just right"

2. Set two variables to your first and last names. Use an `if` statement to print out whichever of them comes first in the alphabet.

3. Use a `while` loop to print out a numbered list of the elements in an array
4. Now do it with a `foreach` loop
5. Now do it with a hash, printing out the keys and values for each item (hint: look up the `keys` function in your Camel book)

Answers are given in `exercises/answers/loops.pl`

6.7. Practical uses of `while` loops: taking input from STDIN

STDIN is the standard input stream for any Unix program. If a program is interactive, it will take input from the user via STDIN. Many Unix programs accept input from STDIN via pipes and redirection. For instance, the Unix **cat** utility prints out any file it has redirected to its STDIN:

```
% cat < hello.pl
```

Unix also has STDOUT (the standard output) and STDERR (where errors are printed to).

We can get a Perl script to take input from STDIN (standard input) and do things with it by using the line input operator, which is a set of angle brackets with the name of a filehandle in between them:

```
my $user_input = <STDIN>;
```

The above example takes a single line of input from STDIN. The input is terminated by the user hitting Enter. If we want to repeatedly take input from STDIN, we can use the line input operator in a `while` loop:

```
#!/usr/bin/perl -w

while ($_ = <STDIN>) {
```

```
        # do some stuff here, if you want...
        print;     # remember that print takes $_ as its de-
fault argument
]
```

Conveniently enough, the `while` statement can be written more succinctly, because in these circumstances, the line input operator assigns to `$_` by default:

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

Better yet, the default filehandle used by the line input operator is STDIN, so we can shorten the above example yet further:

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

As always, there's more than one way to do it.

The above example script (which is available in your directory as `exercises/cat.pl`) will basically perform the same function as the Unix **cat** command; that is, print out whatever's given to it through STDIN.

Try running the script with no arguments. You'll have to type some stuff in, line by line, and type **CTRL-D** (a.k.a. `^D`) when you're ready to stop. `^D` indicates end-of-file (EOF) on most Unix systems.

Now try giving it a file by using the shell to redirect its own source code to it:

```
perl exercises/cat.pl < exercises/cat.pl
```

This should make it print out its own source code.

6.8. Named blocks

Blocks can be given names, thus:

```
#!/usr/bin/perl -w

LINE: while (<STDIN>) {
        ...
}
```

By tradition, the names of blocks are in upper case. The name should also reflect the type of thing you are iterating over -- in this case, a line of text from STDIN.

6.9. Breaking out of loops

You can break out of loops using `next`, `last` and similar statements.

```
#!/usr/bin/perl -w

LINE: while (<STDIN>) {
        chomp;                          # remove newline
        next LINE if $_ eq ”;          # skip blank lines
        last LINE if lc($_) eq ’q’;    # quit
}
```

The `LINE` indicating the block to break out of is optional (it defaults to the current smallest loop), but can be very useful when you wish to break out of a loop higher up the chain:

```
#!/usr/bin/perl -w

LINE: while (<STDIN>) {
        chomp;                                    # re-
move newline
        next LINE if $_ eq ”;                    # skip blank lines
```

```
        # we split the line into words and check all of them
        foreach (split $_) {
                last LINE if lc($_) eq 'quit';   # quit
        }
}
```

6.10. Chapter summary

- A block in Perl is a series of statements grouped together by curly brackets. Blocks can be used in conditional constructs and subroutines.
- A conditional construct is one which executes statements based on the truth of a condition
- Truth in Perl is determined by testing whether something is NOT any of: numeric zero, the null string, or undefined
- The `if - elsif - else` conditional construct can be used to perform certain actions based on the truth of a condition
- The `while`, `for`, and `foreach` constructs can be used to repeat certain statements based on the truth of a condition.
- A common practical use of the `while` loop is to read each line of a file.
- Blocks may be named using the `NAME:` convention
- You can break out of blocks using `next`, `last` and similar statements

Chapter 7. Subroutines

7.1. In this chapter...

In this chapter, we look at subroutines and how they can be used to simplify your code.

7.2. Introducing subroutines

If you have a long Perl script, you'll probably find that there are parts of the script that you want to break out into subroutines. In particular, if you have a section of code which is repeated more than once, it's best to make it a subroutine to save on maintenance (and, of course, linecount).

A subroutine is basically a little self-contained mini-program in the form of block which has a name, and can take arguments and return values:

```
# the general case

sub name {
        BLOCK
}

# the specific case

sub print_headers {
        print "Programming Perl, 2nd ed\n";
        print "by\n";
        print "Larry Wall et al.\n";
}
```

7.3. Calling a subroutine

A subroutine can be called in either of the following ways:

```
&print_headers;
print_headers();
```

If (for some reason) you've got a subroutine that clashes with a reserved function or something, you will need to prefix your function name with & (ampersand) to be perfectly clear. You should avoid doing this anyway; overloading built-in functions can cause more confusion than it's worth.

There are other times when you need to use an ampersand on your subroutine name, such as when a function needs a SUBROUTINE type of parameter, or when making an anonymous subroutine reference.

7.4. Passing arguments to a subroutine

You can pass arguments to a subroutine by including them in the brackets when you call it. The arguments end up in an array called @_ which is only visible inside the subroutine.

```
print_headers("Programming Perl, 2nd ed", "Larry Wall et al");

# we can also pass variables to a subroutine by name...
my $fiction_title = "Lord of the Rings";
my $fiction_author = "J.R.R. Tolkein";
print_headers($fiction_title, $fiction_author);

sub print_headers {
        my ($title, $author) = @_;
        print "$title\n";
        print "by\n";
        print "$author\n";
}
```

You can take any number of scalars in as arguments - they'll all end up in `@_` in the same order you gave them.

You could also use `$title = shift; $author = shift;` to get the same result. See the entry for `shift` on page 215 of the Camel book.

7.5. Returning values from a subroutine

To return a value from a subroutine, simply use the `return` function.

```
sub print_headers {
        my ($title, $author) = @_;
        return "$title\nby\n$author\n\n";
}

sub sum {
        my $total;
        foreach (@_) {
                $total = $total + $_;
        }
        return $total;
}
```

You can also return lists from your subroutine:

```
# subroutine to return the first three arguments passed to it
sub firstthree {
        return @_[0..2];
}

my @three_items = firstthree("x", "y", "z", "a", "b");
# sets @three_items to ("x", "y", "z");
```

7.6. Exercises

1. Write a subroutine which prints out its first argument
2. Modify the above subroutine to also print out the last argument
3. Now change it to compare the first and last arguments and return the one which is numerically larger (you'll want to use an `if` statement for that)

7.7. Chapter summary

- A subroutine is a named block which can be called from anywhere in your Perl program
- Subroutines can accept parameters, which are available via the special array `@_`
- Subroutines can return scalar or list values.

Chapter 8. Regular expressions

8.1. In this chapter...

In this chapter we begin to explore Perl’s powerful regular expression cababilities, and use regular expressions to perform matching and substitution operations on text.

8.2. What are regular expressions?

The easiest way to explain this is by analogy. You will probably be familiar with the concept of matching filenames under DOS and Unix by using wildcards - `*.txt` or `/usr/local/*` for instance. When matching filenames, an asterisk can be used to match any number of unknown characters, and a question mark matches any single character. There are also less well-known filename matching characters.

Regular expressions are similar in that they use special characters to match text. The differences are that any kind of text can be matched, and that the set of special characters is different.

Regular expressions are also known as REs, regexes, and regexps.

If you have a mathematical background, you may like to think of a regexp as a definition of a set of strings. For instance, a regexp may describe the set of all strings which begin with the letter "a".

8.3. Regular expression operators and functions

8.3.1. m/PATTERN/ - the match operator

The most basic regular expression operator is the matching operator, `m/PATTERN/`.

- Works on `$_` by default.
- In scalar context, returns true (`1`) if the match succeeds, or false (the empty string) if the match fails.
- In list context, returns a list of any parts of the pattern which are enclosed in parentheses. If there are no parentheses, the entire pattern is treated as if it were parenthesized.
- The `m` is optional if you use slashes as the pattern delimiters.
- If you use the `m` you can use any delimiter you like instead of the slashes. This is very handy for matching on strings which contain slashes, for instance directory names or URLs.
- Using the `/i` modifier on the end makes it case insensitive.

```
while (<>) {
        print if m/foo/;        # prints if a line con-
tains "foo"
        print if m/foo/i;       # prints if a line con-
tains "foo", "FOO", etc
        print if /foo/i;        # ex-
actly the same; the m is optional
        print if m!http://!;    # using ! as an alterna-
tive delimiter
```

8.3.2. s/PATTERN/REPLACEMENT/ - the substitution operator

This is the substitution operator, and can be used to find text which matches a pattern and replace it with something else.

- Works on `$_` by default.
- In scalar context, returns the number of matches found and replaced.
- In list context, behaves the same as in scalar context and returns the number of matches found and replaced.
- You can use any delimiter you want, the same as the `m//` operator.
- Using `/g` on the end of it matches globally, otherwise matches (and replaces) only the first instance of the pattern.
- Using the `/i` modifier makes it case insensitive.

```
# fix some misspelt text

while (<>) {
        s/freind/friend/g;
        s/teh/the/g;
        s/jsut/just/g;
        print;
}
```

The above example can be found in `exercises/spellcheck.pl`.

8.3.3. Binding operators

If we want to use `m//` or `s///` to operate on something other than `$_` we need to use binding operators to bind the match to another string.

Table 8-1. Binding operators

Operator	Meaning
`=~`	True if the pattern matches
`!~`	True if the pattern doesn't match

```
print "Please enter your homepage URL: ";
my $url = <STDIN>;
if ($url =~ /geocities/) {
        print "Ahhh, I see you have a geocities homepage!\n";
}
```

8.4. Metacharacters

The special characters we use in regular expressions are called *metacharacters*, because they are characters that describe other characters.

8.4.1. Some easy metacharacters

Table 8-2. Regular expression metacharacters

Metacharacter(s)	Matches...
`^`	Start of string
`$`	End of string
`.`	Any single character except `\n` (though special things can happen in multiline mode)

Metacharacter(s)	Matches...
`\n`	Newline (subtly different to `$ - when working in multiline mode, there may be newlines embedded in the multiline string you're working with.`
`\t`	Matches a tab
`\s`	Any whitespace character, such as space or tab
`\S`	Any non-whitespace character
`\d`	Any digit (0 to 9)
`\D`	Any non-digit
`\w`	Any "word" character - alphanumeric plus underscore (_)
`\W`	Any non-word character
`\b`	A word break - the zero-length point between a word character (as defined above) and a non-word character.

These and other metacharacters are all outlined in chapter 2 of the Camel book and in the `perlre` manpage - type **perldoc perlre** to read it.

Any character that isn't a metacharacter just matches itself. If you want to match a character that's normally a metacharacter, you can escape it by preceding it with a backslash

Some quick examples:

```
# Perl regular expressions are usually found within slashes -
the
# matching operator/function which we will see soon.

/cat/                                   # matches the three characters
                                        # c, a, and t in that order.
```

```
/^cat/                                  # matches c, a, t at start of line
/\scat\s/                               # matches c, a, t with spaces on
                                        # either side
/\bcat\b/                               # same as above, but won't
                                        # in-
clude the spaces in the text
                                        # it matches

# we can interpolate variables just like in strings:

my $ani-
mal = "dog"                       # we set up a scalar variable
/$animal/                               # matches d, o, g
/$animal$/                              # matches d, o, g at end of line

/\$\d\.\d\d/                            # matches a dol-
lar sign, then a
                                        # digit, then a dot, then
                                        # an-
other digit, then another
                                        # digit, eg $9.99
```

8.4.2. Quantifiers

What if, in our last example, we'd wanted to say "Match a dollar, then any number of digits, then a dot, then two more digits"? What we need are quantifiers.

Table 8-3. Regular expression quantifiers

Quantifier	Meaning
?	0 or 1
*	0 or more
+	1 or more

Quantifier	Meaning
`{n}`	match exactly n times
`{n,}`	match n or more times
`{n,m}`	match between n and m times

8.4.3. Greediness

Regular expressions are, by default, "greedy". This means that any regular expression, for instance `.*`, will try to match the biggest thing it possibly can. Greediness is sometimes referred to as "maximal matching".

To change this behaviour, follow the quantifier with a question mark, for example `.*?`. This is sometimes referred to as "minimal matching".

```
$string = "abracadabra";

/a.*a/                # greedy -- matches "abracadabra"
/a.*?a/               # not greedy -- matches "abra"
```

8.4.4. Exercises

1. You now know enough to work out the price example above. Work it through.
2. Another example: what regular expression would match the word "colour" with either British or American spellings?
3. How can we match any four-letter word?

8.5. Grouping techniques

8.5.1. Character classes

A character class can be used to find a single character that matches any one of a given set of characters.

Let’s say you’re looking for occurences of the word "grey" in text, then remember that the American spelling is "gray". The way we can do this is by using character classes. Character classes are specified using square brackets, thus: `/gr[ea]y/`

We can also use character sequences by saying things like `[A-Z]` or `[0-9]`. The sequences `\d` and `\w` can easily be expressed as character classes: `[0-9]` and `[a-zA-Z0-9_]` respectively.

We can negate a character class by putting a caret at the start of it. That’s right, the same character that we used to match the start of the line. Larry Wall has written that Perl does anything you want -- unless you want consistency, and it has also been said that consistency is the hobgoblin of small minds. Therefore, we’ll learn about these character class inconsistencies, learn to love them, and flatter ourselves that we do not have small minds.

Here are some of the special rules that apply inside character classes. I make no guarantee that this is a complete list; additions are always welcome.

- `^` at the start of a character class negates the character class, rather than specifying the start of a line.
- `-` specifies a range of characters.
- `$ . () \{\ * +}` and other metacharacters taken literally.

8.5.1.1. Exercises

Your trainer will help you do the following exercises as a group.

1. How would we find any word starting with a letter in the first half of the alphabet, or with X, Y, or Z?
2. What regular expression could be used for any word that starts with letters *other* than those listed in the previous example.
3. There's almost certainly a problem with the regular expression we've just created - can you see what it might be?

8.5.2. Alternation

The problem with character classes is that they only match one character. What if we wanted to match any of a set of longer strings, like a set of words?

The way we do this is to use the pipe symbol `|` for alternation:

```
/cat|dog|budgie/                # matches any of our pets
```

Now we come up against another problem. If we write something like:

```
/^cat|dog|budgie$/
```

...to match any of our pets on a line by itself, what we're actually matching is: "the start of the string followed by cat; or dog; or budgie followed by the end of the string". This is not what we originally intended. To fix this, we enclose our alternation in round brackets:

```
/^(cat|dog|budgie)$/
```

```
# a simple matching program to get some email head-
ers and print them out

while (<>) {
        print if /^(From|Subject|Date):\s/;
}
```

The above email example can be found in `exercises/mailhdr.pl`.

8.5.3. The concept of atoms

Round brackets bring us neatly into the concept of atoms. The word "atom" derives from the Greek *atomos* meaning "indivisible" (little did they know!). What we use it to mean is "something that is a chunk of regular expression in its own right" -- as opposed to "something that can wipe out cities with a single blast".

Atoms can be arbritrarily created by simply wrapping things in round brackets - handy for indicating grouping, using quantifiers for the whole group at once, and for indicating which bit(s) of a matching function should be the returned value (but we'll deal with that later).

In the example above, there are three atoms:

1. start of line
2. cat or dog or budgie
3. end of line

How many atoms were there in our dollar prices example earlier?

Atomic groupings can have quantifiers attached to them. For instance:

```
# match a consonant followed by a vowel twice in a row
# eg "tutu"
/([^aeiou][aeiou]){2}/

# match three or more words starting with "a" in a row
# eg "all angry animals"
/(\ba\w+\b\s*){3,}
```

8.6. Exercises

1. Determine whether your name appears in a string (an answer's in `exercises/answers/namere.pl`).
2. Remove footnote references (like [1]) from some text (see `exercises/footnote.txt` for some sample text, and `exercises/answers/footnote.pl` for an answer).
3. Split tab-separated data into an array then print out each element using a `foreach` loop.

8.7. Chapter summary

- Regular expressions are used to perform matches and substitutions on strings
- Regular expressions can include meta-characters (characters with a special meaning, which describe sets of other characters) and quantifiers
- Character classes can be used to specify any single instance of a set of characters
- Alternation may be used to specify any of a set of sub-expressions
- The matching operator is `m/PATTERN/` and acts on `$_` by default
- The substitution operator is `s/PATTERN/REPLACEMENT/` and acts on `$_` by default
- Matches and substitutions can be performed on strings other than `$_` by using the `=~` binding operator
- Functions such as `split()` and `grep()` use regular expression patterns as one of their arguments

Chapter 9. Practical exercises

This chapter provides you with some broader exercises to test your new Perl skills. Each exercise requires you to use a mixture of variables, operators, functions, conditional and looping constructs, and regular expressions.

There are no right or wrong answers. Remember, "There’s More Than One Way To Do It."

1. Write a simple menu system where the user is repeatedly asked to choose a message to display or Q to quit.
    a. Consider case-sensitivity
    b. Handle errors cleanly

2. Write a "chatterbox" program that holds a conversation with the user by matchings patterns in the user’s input.
3. Write a program that gives information about files.
    a. use file test operators
    b. offer to print the file out if it’s a text file
    c. how will you cope with files longer than a screenful?

Chapter 10. Conclusion

10.1. What you've learnt

Now you've completed Netizen's Introduction to Perl module, you should be confident in your knowledge of the following fields:

- What is Perl? Perl's features; Perl's main uses; where to find information about Perl online
- Creating Perl scripts and running them from the Unix command line, including the use of the `-w` flag to enable warnings
- Perl's three main variable types: scalars, arrays and hashes
- The `strict` pragma, lexical scoping, and their benefits
- Perl's most common operators and functions, and their use
- Perl's concept of truth; existence and definedness of variables
- Conditional and looping constructs: `if`, `while`, `foreach` and others.
- Regular expressions: the matching and substitution operators; simple metacharacters; quantifiers; alternation and grouping

10.2. Where to now?

To further extend your knowledge of Perl, you may like to:

- Borrow or purchase the books listed in our "Further Reading" section (below)
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme"

- Install Perl on your home or work computer
- Practice using Perl from day to day
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)
- Extend your knowledge with further Netizen courses such as:
  - Intermediate Perl
  - CGI Programming in Perl
  - Web enabled databases with Perl and DBI

  Information about these courses can be found on Netizen's website (http://netizen.com.au/services/training/). A diagram of Netizen's courses and the careers they can lead to is included with these training materials.

10.3. Further reading

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)

Appendix A. Unix cheat sheet

A brief run-down for those whose Unix skills are rusty:

Table A-1. Simple Unix commands

Action	Command
Change to home directory	**cd**
Change to *directory*	**cd *directory***
Change to directory above current directory	**cd ..**
Show current directory	**pwd**
Directory listing	**ls**
Wide directory listing, showing hidden files	**ls -al**
Showing file permissions	**ls -al**
Making a file executable	**chmod +x *filename***
Printing a long file a screenful at a time	**more *filename* or less *filename***
Getting help for *command*	**man *command***

Appendix B. Editor cheat sheet

This summary is laid out as follows:

Table B-1. Layout of editor cheat sheets

Running	Recommended command line for starting it.
Using	Really basic howto. This is not even an attempt at a detailed howto.
Exiting	How to quit.
Gotchas	Oddities to watch for.

B.1. vi

B.1.1. Running

```
% vi filename
```

B.1.2. Using

- `i` to enter insert mode, then type text, press **ESC** to leave insert mode.
- `x` to delete character below cursor.
- `dd` to delete the current line
- Cursor keys should move the cursor while *not* in insert mode.
- If not, try `hjkl`, `h` = left, `l` = right, `j` = down, `k` = up.

- `/`, then a string, then **ENTER** to search for text.
- `:w` then **ENTER** to save.

B.1.3. Exiting

- Press **ESC** if necessary to leave insert mode.
- `:q` then **ENTER** to exit.
- `:q!` **ENTER** to exit without saving.
- `:wq` to exit with save.

B.1.4. Gotchas

vi has an insert mode and a command mode. Text entry only works in insert mode, and cursor motion only works in command mode. If you get confused about what mode you are in, pressing **ESC** twice is guaranteed to get you back to command mode (from where you press i to insert text, etc).

B.1.5. Help

`:help` **ENTER** might work. If not, then see the manpage.

B.2. pico

B.2.1. Running

```
% pico -w filename
```

B.2.2. Using

- Cursor keys should work to move the cursor.
- Type to insert text under the cursor.
- The menu bar has `^X` commands listed. This means hold down **CTRL** and press the letter involved, eg **CTRL-W** to search for text.
- **CTRL-O**to save.

B.2.3. Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

B.2.4. Gotchas

Line wraps are automatically inserted unless the -w flag is given on the command line. This often causes problems when strings are wrapped in the middle of code and similar. \\ \hline

B.2.5. Help

CTRL-G from the main menu, or just read the menu bar.

B.3. joe

B.3.1. Running

```
% joe filename
```

B.3.2. Using

- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-K** then **S** to save.

B.3.3. Exiting

- **CTRL-C** to exit without save.
- **CTRL-K** then **X** to save and exit.

B.3.4. Gotchas

Nothing in particular.

B.3.5. Help

CTRL-K then **H**.

B.4. jed

B.4.1. Running

```
% jed
```

B.4.2. Using

- Defaults to the emacs emulation mode.
- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-X** then **S** to save.

B.4.3. Exiting

CTRL-X then **CTRL-C** to exit.

B.4.4. Gotchas

Nothing in particular.

B.4.5. Help

- Read the menu bar at the top.
- Press **ESC** then **?** then **H** from the main menu.

Appendix C. ASCII Pronunciation Guide

Table C-1. ASCII Pronunciation Guide

Character	Pronunciation
!	bang, exlamation
*	star, asterisk
$	dollar
@	at
%	percent
&	ampersand
"	double-quote
'	single-quote, tick
( )	open/close bracket, parentheses
<	less than
>	greater than
-	dash, hyphen
.	dot
,	comma
/	slash, forward-slash
\	backslash, slosh
:	colon
;	semi-colon
=	equals
?	question-mark
^	caret (pron. carrot)
_	underscore

Character	Pronunciation
[ ]	open/close square bracket
{ }	open/close curly brackets, open/close brace
|	pipe, or vertical bar
~	tilde (pron. “til-duh”, wiggle, squiggle)
‘	backtick