

# UNIX - Command-Line Survival Guide

## Files, directories, commands, text editors

Simon Prochnik & Lincoln Stein

### Book Chapters

Learning Perl (6th ed.): Chap. 1

Unix & Perl to the Rescue (1st ed.): Chaps. 3 & 5

### Lecture Notes

- [What is the Command Line?](#)
  - [Logging In](#)
  - [Amazon Web Services](#)
  - [The Desktop](#)
  - [The Shell](#)
  - [Home Sweet Home](#)
  - [Getting Around](#)
  - [Running Commands](#)
  - [Command Redirection](#)
  - [Pipes](#)
- 

## What is the Command Line?

Underlying the pretty Mac OSX GUI is a powerful command-line operating system. The command line gives you access to the internals of the OS, and is also a convenient way to write custom software and scripts.

Many bioinformatics tools are written to run on the command line and have no graphical interface. In many cases, a command line tool is more versatile than a graphical tool, because you can easily combine command line tools into automated scripts that accomplish tasks without human intervention.

In this course, we will be writing Perl scripts that are completely command-line based.

---

## Logging into Your Workstation

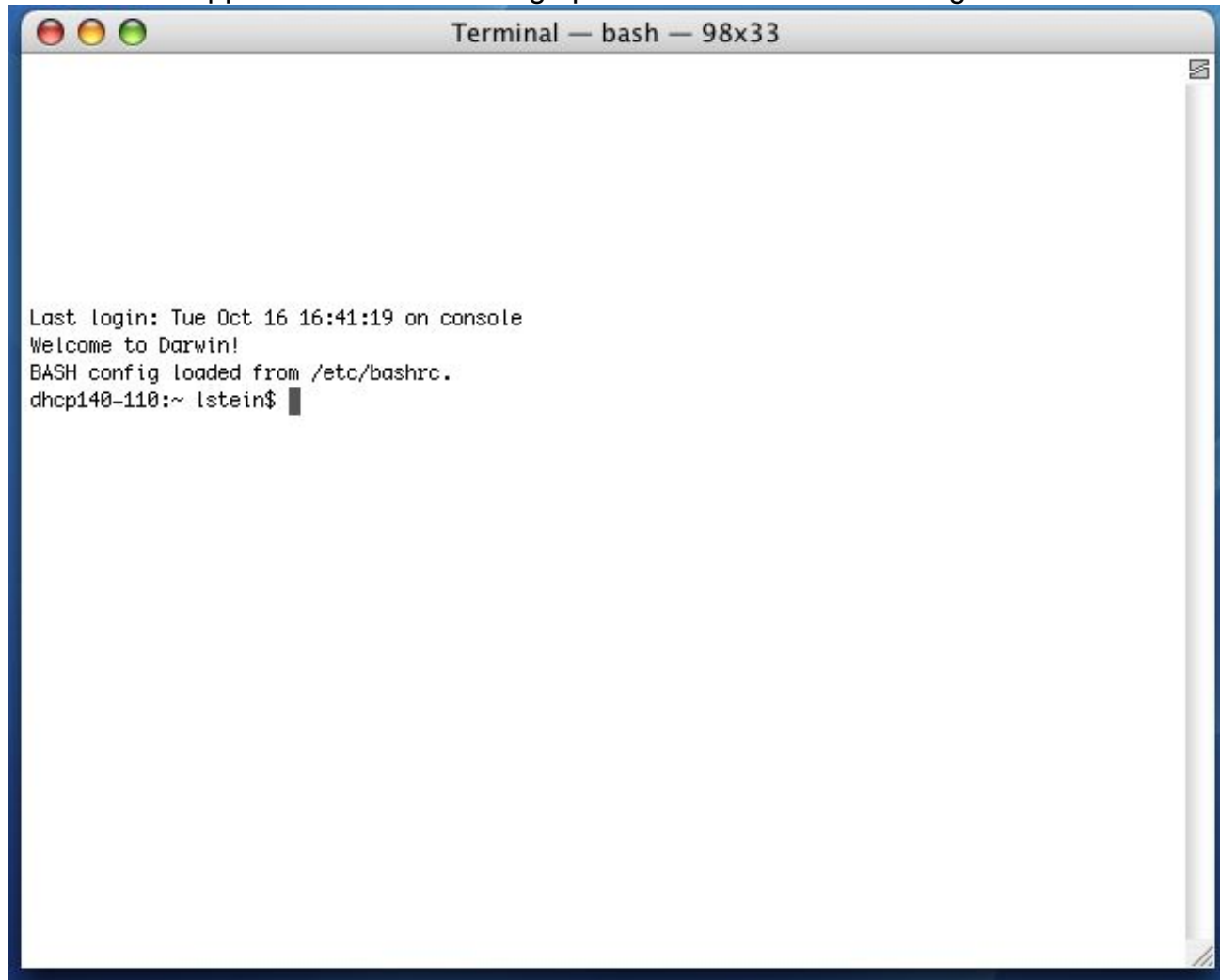
Your workstation is an iMac. To log into it, provide the following information:

*Your username:* the initial of your first name, followed by your full last name. For example, if your username is **srobb** for **sofia robb**

*Your password:* **pfb@forever**

# Bringing up the Command Line

To bring up the command line, use the Finder to navigate to *Applications->Utilities* and double-click on the *Terminal* application. This will bring up a window like the following:



*OSX Terminal*

You can open several Terminal windows at once. This is often helpful.

You will be using this application a lot, so I suggest that you drag the Terminal icon into the shortcuts bar at the bottom of your screen.

---

## Amazon Web Services cloud computing

The computers we will be using on the course are part of Amazon's cloud computing. Their system is called Amazon Web Services (AWS).

Everyone will have access to their own computer. Amazon refers to them as instances.

Different computers or instance types have different amounts of memory and CPUs. Here are the two types of instance we will be working on.

AWS instance type	CPUs (cores) and Memory
Small	1 CPU 1.7Gb RAM
Extra large	4 CPUs 15Gb

Later in the course when we try to assemble genomes for example, we will require computers with more memory and more cores

You need to log into an instance by using the ssh command in the Terminal window. 'ssh' stands for secure shell. This is an encrypted connection to another computer. You'll learn more about the 'shell' part in the next section.

Here's how you log in to an instance.

```
ssh srobb@ec2-107-22-31-168.compute1.amazonaws.com
```

This is confusing, so we made you an easier way to log in. There is a webpage with everyone's user name and a link. The webpage is here.

<http://ec2-54-205-98-165.compute-1.amazonaws.com/files/awslogins.html>

The links act as an ssh command, so if you click on the link, you will get logged in to your instance.

This might take a little getting used to, because you are really just using the iMac as a Terminal (hence the name of the Application) into a server somewhere else. In our case, this is an AWS virtual computer in the cloud. This is a very common way to work with UNIX. In a day or so, you will be used to it.

---

## OK. I've Logged in. What Now?

The terminal window is running a **shell** called "bash." The shell is a loop that:

1. Prints a prompt
2. Reads a line of input from the keyboard
3. Parses the line into one or more commands
4. Executes the commands (which usually print some output to the terminal)
5. Go back 1.

There are many different shells with bizarre names like **bash**, **sh**, **csch**, **tcsh**, **ksh**, and **zsh**. The "sh" part means shell. Each shell was designed for the purpose of confusing you and tripping you up. We have set up your accounts to use **bash**. Stay with **bash** and you'll get used to it, eventually.

---

## Command-Line Prompt

Most of bioinformatics is done with command-line software, so you should take some time to learn to use the shell effectively.

This is a command line prompt:

```
bush202>
```

This is another:

```
(~) 51%
```

This is another:

```
srobb@bush202 1:12PM>
```

What you get depends on how the system administrator has customized your login. You can customize yourself when you know how.

The prompt tells you the shell is ready to accept a command. When a long-running command is going, the prompt will not reappear until the system is ready to deal with your next request.

## Issuing Commands

Type in a command and press the <Enter> key. If the command has output, it will appear on the screen. Example:

```
(~) 53% ls -F
GNUstep/          cool_elegans.movies.txt  man/
INBOX             docs/                   mtv/
INBOX~           etc/                   nsmail/
Mail@            games/                 pcod/
News/            get_this_book.txt      projects/
axhome/          jcod/                 public_html/
bin/             lib/                   src/
build/           linux/                 tmp/
ccod/
(~) 54%
```

The command here is `ls -F`, which produces a listing of files and directories in the current directory (more on which later). After its output, the command prompt appears again.

Some programs will take a long time to run. After you issue their command name, you won't recover the shell prompt until they're done. You can either launch a new shell (from Terminal's File menu), or run the command in the background using the ampersand:

```
(~) 54% long_running_application&
(~) 55%
```

The command will now run in the background until it is finished. If it has any output, the output will be printed to the terminal window. You may wish to redirect the output as described later.

## Command Line Editing

Most shells offer command line editing. Up until the comment you press <Enter>, you can go back over the command line and edit it using the keyboard. Here are the most useful keystrokes:

## Backspace

Delete the previous character and back up one.

## Left arrow, right arrow

Move the text insertion point (cursor) one character to the left or right.

## control-a (^a)

Move the cursor to the beginning of the line. Mnemonic: A is first letter of alphabet

## control-e (^e)

Move the cursor to the end of the line. Mnemonic: <E> for the End (^Z was already taken for something else).

## control-d (^d)

Delete the character currently under the cursor. D=Delete.

## control-k (^k)

Delete the entire line from the cursor to the end. k=kill. The line isn't actually deleted, but put into a temporary holding place called the "kill buffer".

## control-y (^y)

Paste the contents of the kill buffer onto the command line starting at the cursor. y=yank.

## Up arrow, down arrow

Move up and down in the command history. This lets you reissue previous commands, possibly after modifying them.

There are also some useful shell commands you can issue:

## *history*

Show all the commands that you have issued recently, nicely numbered.

## !*<number>*

Reissue an old command, based on its number (which you can get from *history*)

## !!

Reissue the immediate previous command.

## !*<partial command string>*

Reissue the previous command that began with the indicated letters. For example *!!* would reissue the *ls -F* command from the earlier example.

**bash** offers automatic command completion and spelling correction. If you type part of a command and then the tab key, it will prompt you with all the possible completions of the command. For example:

```
(~) 51% fd<tab>
(~) 51% fd
fd2ps    fdesign  fdformat fdlist    fdmount  fdmountd fdrawcmd fdumount
(~) 51%
```

If you hit tab after typing a command, but before pressing <Enter>, **bash** will prompt you with a list of file names. This is because many commands operate on files.

## Wildcards

You can use wildcards when referring to files. "\*" refers to zero or more characters. "?" refers to any single character. For example, to list all files with the extension ".txt", run **ls** with the pattern "\*.txt":

```
(~) 56% ls -F *.txt
final_exam_questions.txt  genomics_problem.txt
genebridge.txt             mapping_run.txt
```

There are several more advanced types of wildcard patterns which you can read about in the **tcsh** manual page. For example, you can refer to files beginning with the characters "f" or "g" and ending with ".txt" this way:

```
(~) 57% ls -F [f-g]*.txt
final_exam_questions.txt  genebridge.txt  genomics_problem.txt
```

---

## Home Sweet Home

When you first log in, you'll be placed in a part of the system that is your personal domain, called the *home directory*. You are free to do with this area what you will: in particular you can create and delete files and other directories. In general, you cannot create files elsewhere in the system.

Your home directory lives somewhere way down deep in the bowels of the system. On our iMacs, it is a directory with the same name as your login name, located in **/Users**. The full directory path is therefore **/Users/username**. Since this is a pain to write, the shell allows you to abbreviate it as **~username** (where "username" is your user name), or simply as **~**. The weird character (technically called the "tilde" or "twiddle") is usually hidden at the upper left corner of your keyboard.

To see what is in your home directory, issue the command **ls -F**:

```
(~) % ls -F
INBOX          Mail/          News/          nsmail/        public_html/
```

This shows one file "INBOX" and four directories ("Mail", "News") and so on. (The "-F" in the command turns on fancy mode, which appends special characters to directory listings to tell you more about what you're seeing. "/" means directory.)

In addition to the files and directories shown with **ls -F**, there may be one or more hidden files. These are files and directories whose names start with a "." (technically called the "dot" character). To see these hidden files, add an "a" to the options sent to the **ls** command:

```
(~) % ls -aF
./              .cshrc         .login         Mail/
../            .fetchhost     .netscape/    News/
.Xauthority    .fvwmrc        .xinitrc*      nsmail/
.Xdefaults     .history       .xsession@     public_html/
.bash_profile  .less          .xsession-errors
.bashrc        .lessrc        INBOX
```

Whoa! There's a lot of hidden stuff there. But don't go deleting dot files willy-nilly. Many of them are essential configuration files for commands and other programs. For example, the *.profile* file contains configuration information for the **bash** shell. You can peek into it and see all of **bash**'s many options. You can edit it (when you know what you're doing) in order to change things like the command prompt and command search path.

---

# Getting Around

You can move around from directory to directory using the `cd` command. Give the name of the directory you want to move to, or give no name to move back to your home directory. Use the `pwd` command to see where you are (or rely on the prompt, if configured):

```
(~/docs/grad_course/i) 56% cd
(~) 57% cd /
(/) 58% ls -F
bin/          dosc/          gmon.out      mnt/          sbin/
boot/         etc/           home@         net/          tmp/
cdrom/        fastboot      lib/          proc/         usr/
dev/          floppy/       lost+found/   root/         var/
(/) 59% cd ~/docs/
(~/docs) 60% pwd
/usr/home/lstein/docs
(~/docs) 62% cd ../projects/
(~/projects) 63% ls
Ace-browser/   bass.patch
Ace-perl/      cgi/
Foo/           cgi3/
Interface/     computertalk/
Net-Interface-0.02/ crypt-cbc.patch
Net-Interface-0.02.tar.gz fixer/
Pts/           fixer.tcsh
Pts.bak/       introspect.pl*
PubMed/        introspection.pm
SNPdb/         rhmap/
Tie-DBI/       sbbox/
ace/           sbbox-1.00/
atir/          sbbox-1.00.tgz
bass-1.30a/    zhmapper.tar.gz
bass-1.30a.tar.gz
(~/projects) 64%
```

Each directory contains two special hidden directories named "." and "..". "." refers always to the directory in which it is located. ".." refers always to the parent of the directory. This lets you move upward in the directory hierarchy like this:

```
(~/docs) 64% cd ..
```

and to do arbitrarily weird things like this:

```
(~/docs) 65% cd ../../docs
```

The latter command moves upward to levels, and then into a directory named "docs".

If you get lost, the `pwd` command prints out the full path to the current directory:

```
(~) 56% pwd
/Users/lstein
```

---

# Essential Unix Commands

With the exception of a few commands that are built directly into the shell, all Unix commands are standalone executable programs. When you type the name of a command, the shell will search through all the directories listed in the PATH environment variable for an executable of the same name. If found, the shell will execute the command. Otherwise, it will give a "command not found" error.

Most commands live in `/bin`, `/usr/bin`, or `/usr/local/bin`.

## Getting Information About Commands

The **man** command will give a brief synopsis of the command:

```
(~) 76% man wc
Formatting page, please wait...
WC(1)                                                                 WC(1)

NAME
    wc - print the number of bytes, words, and lines in files

SYNOPSIS
    wc [-clw] [--bytes] [--chars] [--lines] [--words] [--help]
    [--version] [file...]

DESCRIPTION
    This manual page documents the GNU version of wc.  wc
    counts the number of bytes, whitespace-separated words,
    ...
```

## Finding Out What Commands are on Your Computer

The **apropos** command will search for commands matching a keyword or phrase:

```
(~) 100% apropos column
showtable (1)          - Show data in nicely formatted columns
colrm (1)              - remove columns from a file
column (1)            - columnate lists
fix132x43 (1)         - fix problems with certain (132 column) graphics
modes
```

## Arguments and Command Switches

Many commands take arguments. Arguments are often (but not inevitably) the names of one or more files to operate on. Most commands also take command-line "switches" or "options" which fine-tune what the command does. Some commands recognize "short switches" that consist of a single character, while others recognize "long switches" consisting of whole words.



The **wc** (word count) program is an example of a command that recognizes both long and short options. You can pass it the **-c**, **-w** and/or **-l** options to count the characters, words and lines in a text file, respectively. Or you can use the longer but more readable, **--chars**, **--words** or **--lines** options. Both these examples count the number of characters and lines in the text file `/var/log/messages`:

```
(~) 102% wc -c -l /var/log/messages
      23      941 /var/log/messages
(~) 103% wc --chars --lines /var/log/messages
      23      941 /var/log/messages
```

You can cluster short switches by concatenating them together, as shown in this example:

```
(~) 104% wc -cl /var/log/messages
      23      941 /var/log/messages
```

Many commands will give a brief usage summary when you call them with the **-h** or **--help** switch.

## Spaces and Funny Characters

The shell uses whitespace (spaces, tabs and other nonprinting characters) to separate arguments. If you want to embed whitespace in an argument, put single quotes around it. For example:

```
mail -s 'An important message' 'Bob Ghost <bob@ghost.org>'
```

This will send an e-mail to the fictitious person Bob Ghost. The **-s** switch takes an argument, which is the subject line for the e-mail. Because the desired subject contains spaces, it has to have quotes around it. Likewise, my e-mail address, which contains embedded spaces, must also be quoted in this way.

Certain special non-printing characters have *escape codes* associated with them:

Escape Code	Description
<code>\n</code>	new line character
<code>\t</code>	tab character
<code>\r</code>	carriage return character
<code>\a</code>	bell character (ding! ding!)
<code>\nnn</code>	the character whose ASCII code in octal is <b>nnn</b>

## Useful Commands

Here are some commands that are used extremely frequently. Use **man** to learn more about them. Some of these commands may be useful for solving the problem set ;-)

### Manipulating Directories

#### **ls**

Directory listing. Most frequently used as **ls -F** (decorated listing) and **ls -l** (long listing).

#### **mv**

**mv** Rename or move a file or directory.

**cp** Copy a file.

**rm** Remove (delete) a file.

**mkdir** Make a directory

**rmdir** Remove a directory

**ln** Create a symbolic or hard link.

**chmod** Change the permissions of a file or directory.

## Manipulating Files

**cat** Concatenate program. Can be used to concatenate multiple files together into a single file, or, much more frequently, to send the contents of a file to the terminal for viewing.

**more** Scroll through a file page by page. Very useful when viewing large files. Works even with files that are too big to be opened by a text editor.

**less** A version of **more** with more features.

**head** View the head (top) of a file. You can control how many lines to view.

**tail** View the tail (bottom) of a file. You can control how many lines to view. You can also use **tail** to view a growing file.

**wc** Count words, lines and/or characters in one or more files.

**tr** Substitute one character for another. Also useful for deleting characters.

**sort** Sort the lines in a file alphabetically or numerically.

**uniq** Remove duplicated lines in a file.

**cut** Remove sections from each line of a file or files.

**fold** Wrap each input line to fit in a specified width.

**grep** Filter a file for lines matching a specified pattern. Can also be reversed to print out lines that don't match the specified pattern.

**gzip (gunzip)** Compress (uncompress) a file.

**tar** Archive or unarchive an entire directory into a single file.

**emacs** Run the Emacs text editor (good for experts).

## Networking

### ssh

A secure (encrypted) way to log into machines.

### ping

See if a remote host is up.

### ftp and the secure version **sftp**

Transfer files using the File Transfer Protocol.

### who

See who else is logged in.

### lp

Send a file or set of files to a printer.

---

## Standard I/O and Command Redirection

Unix commands communicate via the command line interface. They can print information out to the terminal for you to see, and accept input from the keyboard (that is, from *you*!)

Every Unix program starts out with three connections to the outside world. These connections are called "streams" because they act like a stream of information (metaphorically speaking):

### standard input

This is a communications stream initially attached to the keyboard. When the program reads from standard input, it reads whatever text you type in.

### standard output

This stream is initially attached to the command window. Anything the program prints to this channel appears in your terminal window.

### standard error

This stream is also initially attached to the command window. It is a separate channel intended for printing error messages.

The word "initially" might lead you to think that standard input, output and error can somehow be detached from their starting places and reattached somewhere else. And you'd be right. You can attach one or more of these three streams to a file, a device, or even to another program. This sounds esoteric, but it is actually very useful.

## A Simple Example

The **wc** program counts lines, characters and words in data sent to its standard input. You can use it interactively like this:

```
(~) 62% wc
Mary had a little lamb,
little lamb,
little lamb.

Mary had a little lamb,
whose fleece was white as snow.
^D
```

In this example, I ran the **wc** program. It waited for me to type in a little poem. When I was done, I typed the END-OF-FILE character, control-D (^D for short). **wc** then printed out three numbers indicating the number of lines, words and characters in the input.

More often, you'll want to count the number of lines in a big file; say a file filled with DNA sequences. You can do this by *redirecting* **wc**'s standard input from a file. This uses the **<** metacharacter:

```
(~) 63% wc <big_file.fasta
      2943      2998      419272
```

If you wanted to record these counts for posterity, you could redirect standard output as well using the **>** metacharacter:

```
(~) 64% wc <big_file.fasta >count.txt
```

Now if you **cat** the file *count.txt*, you'll see that the data has been recorded. **cat** works by taking its standard input and copying it to standard output. We redirect standard input from the *count.txt* file, and leave standard output at its default, attached to the terminal:

```
(~) 65% cat <count.txt
      2943      2998      419272
```

## Redirection Meta-Characters

Here's the complete list of redirection commands for **bash**:

<b>&lt;filename</b>	Redirect standard input to file
<b>&gt;filename</b>	Redirect standard output to file
<b>1&gt;filename</b>	Redirect just standard output to file (same as above)
<b>2&gt;filename</b>	Redirect just standard error to file
<b>&gt;filename 2&gt;&amp;1</b>	Redirect both stdout and stderr to file

These can be combined. For example, this command redirects standard input from the file named */etc/passwd*, writes its results into the file *search.out*, and writes its error messages (if any) into a file named *search.err*. What does it do? It searches the password file for a user named "root" and returns all lines that refer to that user.

```
(~) 66% grep root </etc/passwd >search.out 2>search.err
```

## Filters, Filenames and Standard Input

Many Unix commands act as filters, taking data from a file or standard input, transforming the data, and writing the results to standard output. Most filters are designed so that if they are called with one or more filenames on the command line, they will use those files as input. Otherwise they will act on standard input. For example, these two commands are equivalent:

```
(~) 66% grep 'gatttgc' <big_file.fasta  
(~) 67% grep 'gatttgc' big_file.fasta
```

Both commands use the **grep** command to search for the string "gatttgc" in the file *big\_file.fasta*. The first one searches standard input, which happens to be redirected from the file. The second command is explicitly given the name of the file on the command line.

Sometimes you want a filter to act on a series of files, one of which happens to be standard input. Many filters let you use "-" on the command line as an alias for standard input. Example:

```
(~) 68% grep 'gatttgc' big_file.fasta bigger_file.fasta -
```

This example searches for "gatttgc" in three places. First it looks in *big\_file.fasta*, then in *bigger\_file.fasta*, and lastly in standard input (which, since it isn't redirected, will come from the keyboard).

---

## Standard I/O and Pipes

The coolest thing about the Unix shell is its ability to chain commands together into pipelines. Here's an example:

```
(~) 65% grep gatttgc big_file.fasta | wc -l  
22
```

There are two commands here. **grep** searches a file or standard input for lines containing a particular string. Lines which contain the string are printed to standard output. **wc -l** is the familiar word count program, which counts words, lines and characters in a file or standard input. The **-l** command-line option instructs **wc** to print out just the line count. The **|** character, which is known as the "pipe" character, connects the two commands together so that the standard output of **grep** becomes the standard input of **wc**.

What does this pipe do? It prints out the number of lines in which the string "gatttgc" appears in the file *big\_file.fasta*.

## More Pipe Idioms

Pipes are very powerful. Here are some common command-line idioms.

### Count the Number of Times a Pattern does NOT Appear in a File

The example at the top of this section showed you how to count the number of lines in which a particular string pattern appears in a file. What if you want to count the number of lines in which a pattern does **not** appear?

Simple. Reverse the test with the **grep -v** switch:

```
(~) 65% grep -v gatttgc big_file.fasta | wc -l  
2921
```

## Uniquify Lines in a File

If you have a long list of names in a text file, and you are concerned that there might be some duplicates, this will weed out the duplicates:

```
(~) 66% sort long_file.txt | uniq > unique.out
```

This works by sorting all the lines alphabetically and piping the result to the **uniq** program, which removes duplicate lines that occur together. The output is placed in a file named *unique.out*.

## Concatenate Several Lists and Remove Duplicates

If you have several lists that might contain repeated entries among them, you can combine them into a single unique list by **cating** them together, then uniquifying them as before:

```
(~) 67% cat file1 file2 file3 file4 | sort | uniq
```

## Count Unique Lines in a File

If you just want to know how many unique lines there are in the file, add a **wc** to the end of the pipe:

```
(~) 68% sort long_file.txt | uniq | wc -l
```

## Page Through a Really Long Directory Listing

Pipe the output of **ls** to the **more** program, which shows a page at a time. If you have it, the **less** program is even better:

```
(~) 69% ls -l | more
```

## Monitor a Rapidly Growing File for a Pattern

Pipe the output of **tail -f** (which monitors a growing file and prints out the new lines) to **grep**. For example, this will monitor the */var/log/syslog* file for the appearance of e-mails addressed to *mzhang*:

```
(~) 70% tail -f /var/log/syslog | grep mzhang
```

---

# Beginning Perl Scripting

## Simple scripts, Expressions, Operators, Statements, Variables

Simon Prochnik & Lincoln Stein

### Suggested Reading

Learning Perl (6th ed.): Chap. 2, 3, 12,  
Unix & Perl to the Rescue (1st ed.): Chap. 4 Chapters 1, 2 & 5 of *Learning Perl*.

### Lecture Notes

1. [What is Perl?](#)
2. [Some simple Perl scripts](#)
3. [Mechanics of creating a Perl script](#)
4. [Statements](#)
5. [Literals](#)
6. [Operators](#)
7. [Functions](#)
8. [Variables](#)
9. [Processing the Command Line](#)

### Problems

---

## What is Perl?

### Perl is a Programming Language

Written by Larry Wall in late 80's to process mail on Unix systems and since extended by a huge cast of characters. The name is said to stand for:

1. Pathologically Eclectic Rubbish Lister
2. Practical Extraction and Report Language

### Perl Properties

1. Interpreted Language
2. "Object-Oriented"
3. Cross-platform
4. Forgiving
5. Great for text
6. Extensible, rich set of libraries
7. Popular for web pages
8. Extremely popular for bioinformatics

### Other Languages Used in Bioinformatics

C, C++

Compiled languages, hence very fast.  
Used for computation (BLAST, FASTA, Phred, Phrap, ClustalW)  
Not very forgiving.

Java

Interpreted, fully object-oriented language.

Built into web browsers.  
Supposed to be cross-platform, getting better.

Python , Ruby  
Interpreted, fully object-oriented language.  
Rich set of libraries.  
Elegant syntax.  
Smaller user community than Java or Perl.

---

## Some Simple Scripts

Here are some simple scripts to illustrate the "look" of a Perl program.

### Print a Message to the Terminal

Code:

```
#!/usr/bin/perl
# file: message.pl
use strict;
use warnings;
print "When that Aprill with his shoures soote\n";
print "The droghte of March ath perced to the roote,\n";
print "And bathed every veyne in swich licour\n";
print "Of which vertu engendered is the flour...\n";
```

Output:

```
(~) 50% perl message.pl
When that Aprill with his shoures soote
The droghte of March ath perced to the roote,
And bathed every veyne in swich licour
Of which vertu engendered is the flour...
```

### Do Some Math

Code:

```
#!/usr/bin/perl
# file: math.pl
use strict;
use warnings;
print "2 + 2 =", 2+2, "\n";
print "log(1e23)= ", log(1e23), "\n";
print "2 * sin(3.1414)= ", 2 * sin(3.1414), "\n";
```

Output:

```
(~) 51% perl math.pl
2 + 2 =4
log(1e23)= 52.9594571388631
2 * sin(3.1414)= 0.000385307177203065
```

### Run a System Command

Code:

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



```
# file: system.pl
use strict;
use warnings;
system "ls";
```

### Output:

```
(~/docs/grad_course/perl) 52% perl system.pl
index.html          math.pl~          problem_set.html~  what_is_perl.html
index.html~         message.pl       simple.html       what_is_perl.html~
math.pl            problem_set.html  simple.html~
```

## Return the Time of Day

### Code:

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
my $time = localtime;
print "The time is now $time\n";
```

### Output:

```
(~) 53% perl time.pl
The time is now Thu Sep 16 17:30:02 1999
```

# Mechanics of Writing Perl Scripts

Some hints to help you get going.

## Creating the Script

A Perl script is just a text file. Use any text (programmer's) editor. *Don't use word processors like Word.*

By convention, Perl script files end with the extension *.pl*.

I suggest Emacs, because it is already installed on almost all Unix machines, but there are many good options: vi, vim, Textwrangler, eclipse

The Emacs text editor has a *Perl mode* that will auto-format your Perl scripts and highlight keywords. Perl mode will be activated automatically if you end the script name with **.pl**.

GUI-based script writing tools (Aquamacs, xemacs, Textwrangler, Eclipse) are easier to use, but you may have to install them yourself.

Let's write a simple perl script. It'll be a simple text file called time.pl and will contain the lines above.

Let's try doing this in emacs

## Emacs Essentials

A GUI version is simpler to use e.g. Aquamacs, run it by adding the icon for the application to your Dock then clicking on the icon. You can also run emacs in a Terminal window. Emacs will be installed on almost every Unix system you encounter.

```
(~) 50% emacs
```

The same shortcuts you can use on the command line work in Emacs  
e.g.

control-a (^a)  
    move cursor to beginning of line  
etc

## The most important Emacs-specific commands

control-x control-f (^x ^f)  
    open a file  
control-x control-w (^x ^w)  
    save as...  
control-x control-c (^x ^c)  
    quit  
control-g (^g)  
    cancel command  
shift-control-\_ (^\_)  
    Undo typing  
control-h ?(^h ?)  
    Help!!  
option-; (M;)  
    Add comment  
option-/ (M/)  
    Variable/subroutine name auto-completion (cycles through options)

## Running the Script

Don't forget to save any changes in your script before running it. The filled red circle at the top left of the emacs GUI window has a dot in it if there are unsaved changes.

Option 1 (quick, not used much)

Run the **perl** program from the command line, giving it the name of the script file to run.

```
(~) 50% perl time.pl
The time is now Thu Sep 16 18:09:28 1999
```

Option 2 (as shown in examples above)

Put the magic comment

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

at the top of your script.

It's really easy to make a mistake with this complicated line and this causes confusing errors (see below). Double check, or copy from a friend who has it working.

And always add

```
use strict;
use warnings;
```

to the top of your script like in the example below

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
my $time = localtime;
print "The time is now $time\n";
```

Now make the script executable with *chmod +x time.pl*:

```
(~) 51% chmod +x time.pl
```

Run the script as if it were a command:

```
(~) 52% ./time.pl  
The time is now Thu Sep 16 18:12:13 1999
```

Note that you have to type `./time.pl` rather than `time.pl` because, by default, **bash** does not search the current directory for commands to execute. To avoid this, you can add the current directory (".") to your search PATH environment variable. To do this, create a file in your home directory named `.bashrc` and enter the following line in it:

```
export PATH=$PATH:.
```

The next time you log in, your path will contain the current directory and you can type `time.pl` directly.

## Common Errors

Plan out your script before you start coding. Write the code, then run it to see if it works. Every script goes through a few iterations before you get it right. Here are some common errors:

### Syntax Errors

Code:

```
#!/usr/bin/perl  
# file: time.pl  
use strict;  
use warnings;  
time = localtime;  
print "The time is now $time\n";
```

Output:

```
(~) 53% time.pl  
Can't modify time in scalar assignment at time.pl line 3, near "localtime;"  
Execution of time.pl aborted due to compilation errors.
```

### Runtime Errors

Code:

```
#!/usr/bin/perl  
# file: math.pl  
use strict;  
use warnings;  
  
$six_of_one = 6;  
$half_dozen = 12/2;  
$result = $six_of_one/($half_dozen - $six_of_one);  
print "The result is $result\n";
```

Output:

```
(~) 54% math.pl
```

Illegal division by zero at math.pl line 6.

## Forgetting to Make the Script Executable

```
(~) 55% test.pl
test.pl: Permission denied.
```

## Getting the Path to Perl Wrong on the #! line

Code:

```
#!/usr/local/bin/perl
# file: time.pl
use strict;
use warnings;
my $time = localtime;
print "The time is now $time\n";
```

```
(~) 55% time.pl
time.pl: Command not found.
```

This gives a very confusing error message because the command that wasn't found is 'perl' not time.pl

## Useful Perl Command-Line Options

You can call Perl with a few command-line options to help catch errors:

- c** Perform a syntax check, but don't run.
- w** Turn on verbose warnings. Same as  

```
use warnings;
```
- d** Turn on the Perl debugger.

Usually you will invoke these from the command-line, as in `perl -cw time.pl` (syntax check `time.pl` with verbose warnings). You can also put them in the top line: `#!/usr/bin/perl -w`.

# Perl Statements

A Perl script consists of a series of *statements* and *comments*. Each statement is a command that is recognized by the Perl interpreter and executed. Statements are terminated by the semicolon character (;). They are also usually separated by a newline character to enhance readability.

A *comment* begins with the # sign and can appear anywhere. Everything from the # to the end of the line is ignored by the Perl interpreter. Commonly used for human-readable notes. Use comments plentifully, especially at the beginning of a script to describe what it does, at the beginning of each section of your code and for any complex code.

## Some Statements

```
$sum = 2 + 2; # this is a statement

$f = <STDIN>; $g = $f++; # these are two statements

$g = $f
```

```
/
$sum;          # this is one statement, spread across 3 lines
```

The Perl interpreter will start at the top of the script and execute all the statements, in order from top to bottom, until it reaches the end of the script. This execution order can be modified by loops and control structures.

## Blocks

It is common to group statements into *blocks* using curly braces. You can execute the entire block conditionally, or turn it into a *subroutine* that can be called from many different places.

Example blocks:

```
{ # block starts
  my $EcoRI = 'GAATTC';
  my $sequence = <STDIN>;
  print "Sequence contains an EcoRI site" if $sequence=~/$EcoRI/;
} # block ends

my $sequence2 = <STDIN>;
if (length($sequence) < 100) { # another block starts
  print "Sequence is too small. Throw it back\n";
  exit 0;
} # and ends

foreach $sequence (@sequences) { # another block
  print "sequence length = ",length($sequence),"\n";
}
```

## Literals

A *literal* is a constant value that you embed directly in the program code. You can think of the value as being *literally* in the code. Perl supports both *string literals* and *numeric literals*. A string literal or a numeric literal is a *scalar* i.e. a single value.

Literals cannot be changed. If you want to change the value of some data, it needs to be a *variable*. Much, much more on this coming up, until you're really sick of the whole thing.

## String Literals

String literals are enclosed by single quotes (') or double quotes ("):

```
'The quality of mercy is not strained.'; # a single-quoted string
"The quality of mercy is not strained."; # a double-quoted string
```

The difference between single and double-quoted strings is that variables and certain special escape codes are interpolated into double quoted strings, but not in single-quoted ones. Here are some escape codes:

<code>\n</code>	New line
<code>\t</code>	Tab
<code>\r</code>	Carriage return
<code>\f</code>	Form feed
<code>\a</code>	Ring bell

<b>\040</b>	Octal character (octal 040 is the space character)
<b>\0x2a</b>	Hexadecimal character (hex 2A is the "*" character)
<b>\cA</b>	Control character (This is the ^A character)
<b>\u</b>	Uppercase next character
<b>\l</b>	Lowercase next character
<b>\U</b>	Uppercase everything until \E
<b>\L</b>	Lowercase everything until \E
<b>\Q</b>	Quote non-word characters until \E
<b>\E</b>	End \U, \L or \Q operation

```
"Here goes\n\tnothing!";
# evaluates to:
# Here goes
#      nothing!
```

```
'Here goes\n\tnothing!';
# evaluates to:
# Here goes\n\tnothing!
```

```
"Here goes \unothing!";
# evaluates to:
# Here goes Nothing!
```

```
"Here \Ugoes nothing\E";
# evaluates to:
# Here GOES NOTHING!
```

```
"Alert! \a\a\a";
# evaluates to:
# Alert! (ding! ding! ding!)
```

Putting backslashes in strings is a problem because they get interpreted as escape sequences. To include a literal backslash in a string, double it:

```
"My file is in C:\\Program Files\\Accessories\\wordpad.exe";
# evaluates to: C:\Program Files\Accessories\wordpad.exe
```

Put a backslash in front of a quote character in order to make the quote character part of the string:

```
"She cried \"Oh dear! The parakeet has flown the coop!\>";
# evaluates to: She cried "Oh dear! The parakeet has flown the coop!"
```

## Numeric Literals

You can refer to numeric values using integers, floating point numbers, scientific notation, hexadecimal notation, and octal. With some help from the `Math::Complex` module, you can refer to complex numbers as well:

```
123;          # an integer
```

```

1.23;      # a floating point number

-1.23;     # a negative floating point number

1_000_000; # you can use _ to improve readability

1.23E45;   # scientific notation

0x7b;      # hexadecimal notation (decimal 123)

0173;      # octal notation (decimal 123)

use Math::Complex; # bring in the Math::Complex module

12+3*i;    # complex number 12 + 3i

```

## Backtick Strings

You can also enclose a string in backticks (```). This has the helpful property of executing whatever is inside the string as a Unix system command, and returning its output:

```

`ls -l`;
# evaluates to a string containing the output of running the
# ls -l command

```

## Lists

The last type of literal that Perl recognizes is the *list*, which is multiple values strung together using the comma operator (,) and enclosed by parentheses. Lists are closely related to *arrays*, which we talk about later. *Lists* (and *arrays*) are composed from zero, one or more *scalars*, making an empty list, a list containing a single item or a more typical list containing many items, respectively.

```

('one', 'two', 'three', 1, 2, 3, 4.2);
# this is 7-member list contains a mixture of strings, integers
# and floats

```

# Operators

Perl has numerous *operators* (over 50 of them!) that perform operations on string and numeric values. Some operators will be familiar from algebra (like "+", to add two numbers together), while others are more esoteric (like the "." string concatenation operator).

## Numeric & String Operators

The "." operator acts on strings. The "!" operator acts on strings and numbers. The rest act on numbers.

Operator	Description	Example	Result
.	String concatenate	'Teddy' . 'Bear'	TeddyBear
=	Assignment	\$a = 'Teddy'	\$a variable contains 'Teddy'
+	Addition	3+2	5
-	Subtraction	3-2	1
-	Negation	-2	-2

!	Not	!1	0
*	Multiplication	3*2	6
/	Division	3/2	1.5
%	Modulus	3%2	1
**	Exponentiation	3**2	9
<FILEHANDLE>	File input	<STDIN>	Read a line of input from standard input
>>	Right bit shift	3>>2	0 (binary 11>>2=00)
<<	Left bit shift	3<<2	12 (binary 11<<2=1100)
	Bitwise OR	3 2	3 (binary 11 10=11)
&	Bitwise AND	3&2	2 (binary 11&10=10)
^	Bitwise XOR	3^2	1 (binary 11^10=01)

## Operator Precedence

When you have an expression that contains several operators, they are evaluated in an order determined by their *precedence*. The precedence of the mathematical operators follows the rules of arithmetic. Others follow a precedence that usually does what you think they should do. If uncertain, use parentheses to force precedence:

```
2+3*4;    # evaluates to 14, multiplication has precedence over addition
(2+3)*4;  # evaluates to 20, parentheses force the precedence
```

## Logical Operators

These operators compare strings or numbers, returning TRUE or FALSE:

Numeric Comparison		String Comparison	
3 == 2	equal to	'Teddy' eq 'Bear'	equal to
3 != 2	not equal to	'Teddy' ne 'Bear'	not equal to
3 < 2	less than	'Teddy' lt 'Bear'	less than
3 > 2	greater than	'Teddy' gt 'Bear'	greater than
3 <= 2	less or equal	'Teddy' le 'Bear'	less than or equal
3 >= 2	greater than or equal	'Teddy' ge 'Bear'	greater than or equal
3 <=> 2	compare	'Teddy' cmp 'Bear'	compare
		'Teddy' =~ /Bear/	pattern match

The <=> and cmp operators return:

- -1 if the left side is less than the right side
- 0 if the left side equals the right side
- +1 if the left side is greater than the right side

## File Operators

Perl has special *file operators* that can be used to query the file system. These operators generally return TRUE or FALSE.



Example:

```
print "Is a directory!\n" if -d '/usr/home';
print "File exists!\n" if -e '/usr/home/lstein/test.txt';
print "File is plain text!\n" if -T '/usr/home/lstein/test.txt';
```

There are many of these operators. Here are some of the most useful ones:

<b>-e filename</b>	file exists
<b>-r filename</b>	file is readable
<b>-w filename</b>	file is writable
<b>-x filename</b>	file is executable
<b>-z filename</b>	file has zero size
<b>-s filename</b>	file has nonzero size (returns size)
<b>-d filename</b>	file is a directory
<b>-T filename</b>	file is a text file
<b>-B filename</b>	file is a binary file
<b>-M filename</b>	age of file in days since script launched
<b>-A filename</b>	same for access time

## Functions

In addition to its operators, Perl has many *functions*. Functions have a human-readable name, such as **print** and take one or more arguments passed as a list. A function may return no value, a single value (AKA "scalar"), or a list (AKA "array"). You can enclose the argument list in parentheses, or leave the parentheses off.

A few examples:

```
# The function is print. Its argument is a string.
# The effect is to print the string to the terminal.
print "The rain in Spain falls mainly on the plain.\n";

# Same thing, with parentheses.
print("The rain in Spain falls mainly on the plain.\n");

# You can pass a list to print. It will print each argument.
# This prints out "The rain in Spain falls 6 times in the plain."
print "The rain in Spain falls ",2*4-2," times in the plain.\n";

# Same thing, but with parentheses.
print ("The rain in Spain falls ",2*4-2," times in the plain.\n");

# The length function calculates the length of a string,
# yielding 45.
length "The rain in Spain falls mainly on the plain.\n";

# The split function splits a string based on a delimiter pattern
# yielding the list ('The','rain in Spain','falls mainly','on the plain.')
split ' ','The/rain in Spain/falls mainly/on the plain.';
```

# Creating Your Own Functions

You can define your own functions or redefine the built-in ones using the **sub** function. This is described in more detail in the lesson on creating subroutines, which you'll be seeing soon..

## Often Used Functions (alphabetic listing)

For specific information on a function, use **perldoc -f *function\_name*** to get a concise summary.

<a href="#">abs</a>	absolute value
<a href="#">chdir</a>	change current directory
<a href="#">chmod</a>	change permissions of file/directory
<a href="#">chomp</a>	remove terminal newline from string variable
<a href="#">chop</a>	remove last character from string variable
<a href="#">chown</a>	change ownership of file/directory
<a href="#">close</a>	close a file handle
<a href="#">closedir</a>	close a directory handle
<a href="#">cos</a>	cosine
<a href="#">defined</a>	test whether variable is defined
<a href="#">delete</a>	delete a key from a hash
<a href="#">die</a>	exit with an error message
<a href="#">each</a>	iterate through keys & values of a hash
<a href="#">eof</a>	test a filehandle for end of file
<a href="#">eval</a>	evaluate a string as a perl expression
<a href="#">exec</a>	quit Perl and execute a system command
<a href="#">exists</a>	test that a hash key exists
<a href="#">exit</a>	exit from the Perl script
<a href="#">glob</a>	expand a directory listing using shell wildcards
<a href="#">gmtime</a>	current time in GMT
<a href="#">grep</a>	filter an array for entries that meet a criterion
<a href="#">index</a>	find location of a substring inside a larger string
<a href="#">int</a>	throw away the fractional part of a floating point number
<a href="#">join</a>	join an array together into a string
<a href="#">keys</a>	return the keys of a hash
<a href="#">kill</a>	send a signal to one or more processes
<a href="#">last</a>	exit enclosing loop
<a href="#">lc</a>	convert string to lowercase
<a href="#">lcfirst</a>	lowercase first character of string
<a href="#">length</a>	find length of string

<a href="#">local</a>	temporarily replace the value of a global variable
<a href="#">localtime</a>	return time in local timezone
<a href="#">log</a>	natural logarithm
<a href="#">m//</a>	pattern match operation
<a href="#">map</a>	perform an operation on each member of array or list
<a href="#">mkdir</a>	make a new directory
<a href="#">my</a>	create a local variable
<a href="#">next</a>	jump to the top of enclosing loop
<a href="#">open</a>	open a file for reading or writing
<a href="#">opendir</a>	open a directory for listing
<a href="#">pack</a>	pack a list into a compact binary representation
<a href="#">package</a>	create a new namespace for a module
<a href="#">pop</a>	pop the last item off the end of an array
<a href="#">print</a>	print to terminal or a file
<a href="#">printf</a>	formatted print to a terminal or file
<a href="#">push</a>	push a value onto the end of an array
<a href="#">q/STRING/</a>	generalized single-quote operation
<a href="#">qq/STRING/</a>	generalized double-quote operation
<a href="#">qx/STRING/</a>	generalized backtick operation
<a href="#">qw/STRING/</a>	turn a space-delimited string of words into a list
<a href="#">rand</a>	random number generator
<a href="#">read</a>	read binary data from a file
<a href="#">readdir</a>	read the contents of a directory
<a href="#">readline</a>	read a line from a text file
<a href="#">readlink</a>	determine the target of a symbolic link
<a href="#">redo</a>	restart a loop from the top
<a href="#">ref</a>	return the type of a variable reference
<a href="#">rename</a>	rename or move a file
<a href="#">require</a>	load functions defined in a library file
<a href="#">return</a>	return a value from a user-defined subroutine
<a href="#">reverse</a>	reverse a string or list
<a href="#">rewinddir</a>	rewind a directory handle to the beginning
<a href="#">rindex</a>	find a substring in a larger string, from right to left
<a href="#">rmdir</a>	remove a directory
<a href="#">s///</a>	pattern substitution operation
<a href="#">scalar</a>	force an expression to be treated as a scalar

<a href="#">seek</a>	reposition a filehandle to an arbitrary point in a file
<a href="#">select</a>	make a filehandle the default for output
<a href="#">shift</a>	shift a value off the beginning of an array
<a href="#">sin</a>	sine
<a href="#">sleep</a>	put the script to sleep for a while
<a href="#">sort</a>	sort an array or list by user-specified criteria
<a href="#">splice</a>	insert/delete array items
<a href="#">split</a>	split a string into pieces according to a pattern
<a href="#">sprintf</a>	formatted string creation
<a href="#">sqrt</a>	square root
<a href="#">stat</a>	get information about a file
<a href="#">sub</a>	define a subroutine
<a href="#">substr</a>	extract a substring from a string
<a href="#">symlink</a>	create a symbolic link
<a href="#">system</a>	execute an operating system command, then return to Perl
<a href="#">tell</a>	return the position of a filehandle within a file
<a href="#">tie</a>	associate a variable with a database
<a href="#">time</a>	return number of seconds since January 1, 1970
<a href="#">tr///</a>	replace characters in a string
<a href="#">truncate</a>	truncate a file (make it smaller)
<a href="#">uc</a>	uppercase a string
<a href="#">ucfirst</a>	uppercase first character of a string
<a href="#">umask</a>	change file creation mask
<a href="#">undef</a>	undefine (remove) a variable
<a href="#">unlink</a>	delete a file
<a href="#">unpack</a>	the reverse of pack
<a href="#">untie</a>	the reverse of tie
<a href="#">unshift</a>	move a value onto the beginning of an array
<a href="#">use</a>	import variables and functions from a library module
<a href="#">values</a>	return the values of a hash variable
<a href="#">wantarray</a>	return true in an array context
<a href="#">warn</a>	print a warning to standard error
<a href="#">write</a>	formatted report generation

Ok, now you know all the perl functions, so we're done. Thanks for coming.

## Variables

A variable is a symbolic placeholder for a value, a lot like the variables in algebra. These values can be changed. Compare literals whose values cannot be changed. Perl has several built-in variable types:

**Scalars: `$variable_name`**

A single-valued variable, always preceded by a \$ sign.

**Arrays: `@array_name`**

A multi-valued variable indexed by integer, preceded by an @ sign.

**Hashes: `%hash_name`**

A multi-valued variable indexed by string, preceded by a % sign.

**Filehandle: `FILEHANDLE_NAME`**

A file to read and/or write from. Filehandles have no special prefix, but are usually written in all uppercase.

We discuss arrays, hashes and filehandles later.

## Scalar Variables

Scalar variables have names beginning with \$. The name must begin with a letter or underscore, and can contain as many letters, numbers or underscores as you like. These are all valid scalars:

- \$foo
- \$The\_Big\_Bad\_Wolf
- \$R2D2
- \$\_\_\_\_\_A23
- \$Once\_Upon\_a\_Midnight\_Dreary\_While\_I\_Pondered\_Weak\_and\_Weary

You assign values to a scalar variable using the = operator (not to be confused with ==, which is numeric comparison). You read from scalar variables by using them wherever a value would go.

A scalar variable can contain strings, floating point numbers, integers, and more esoteric things. You don't have to predeclare scalars. A scalar that once held a string can be reused to hold a number, and vice-versa:

**Code:**

```
$p = 'Potato'; # $p now holds the string "potato"
$bushels = 3; # $bushels holds the value 3
$potatoes_per_bushel = 80; # $potatoes_per_bushel contains 80;

$total_potatoes = $bushels * $potatoes_per_bushel; # 240

print "I have $total_potatoes $p\n";
```

**Output:**

```
I have 240 Potato
```

## Scalar Variable String Interpolation

The example above shows one of the interesting features of double-quoted strings. If you place a scalar variable inside a double quoted string, it will be interpolated into the string. With a single-quoted string, no interpolation occurs.

To prevent interpolation, place a backslash in front of the variable:

```
print "I have \$total_potatoes \$p\n";

# prints: I have $total_potatoes $p
```

## Operations on Scalar Variables

You can use a scalar in any string or numeric expression like `$hypotenuse = sqrt($x**2 + $y**2)` or `$name = $first_name . ' ' . $last_name`. There are also numerous shortcuts that combine an operation with an assignment:

### **\$a++**

Increment \$a by one

### **\$a--**

Decrement \$a by one

### **\$a += \$b**

Modify \$a by adding \$b to it.

### **\$a -= \$b**

Modify \$a by subtracting \$b from it.

### **\$a \*= \$b**

Modify \$a by multiplying \$b to it.

### **\$a /= \$b**

Modify \$a by dividing it by \$b.

### **\$a .= \$b**

Modify the **string** in \$a by appending \$b to it.

### Example Code:

```
$potatoes_per_bushel = 80; # $potatoes_per_bushel contains 80;

$p = 'one';
$p .= ' ';          # append a space
$p .= 'potato';     # append "potato"

$bushels = 3;
$bushels *= $potatoes_per_bushel; # multiply

print "From $p come $bushels.\n";
```

### Output:

```
From one potato come 240.
```

## String Functions that Come in Handy for Dealing with Sequences

### Reverse the Contents of a String

```
$name = 'My name is Lincoln';
$reversed_name = reverse $name;
print $reversed_name, "\n";
# prints "nlocniL si eman yM"
```

### Translating one set of letters into another set

```
$name = 'My name is Lincoln';
# swap a->g and c->t
$name =~ tr/ac/gt/;
print $name, "\n";
```

```
# prints "My ngme is Lintoln"
```

*Can you see how a combination of these two operators might be useful for computing the reverse complement?*

---

## Processing Command Line Arguments

When a Perl script is run, its command-line arguments (if any) are stored in an automatic array called **@ARGV**. You'll learn how to manipulate this array later. For now, just know that you can call the **shift** function repeatedly from the main part of the script to retrieve the command line arguments one by one.

### Printing the Command Line Argument

Code:

```
#!/usr/bin/perl
# file: echo.pl
use strict;
use warnings;
$argument = shift;
print "The first argument was $argument.\n";
```

Output:

```
(~) 50% chmod +x echo.pl
(~) 51% echo.pl tuna
The first argument was tuna.
(~) 52% echo.pl tuna fish
The first argument was tuna.
(~) 53% echo.pl 'tuna fish'
The first argument was tuna fish.
(~) 53% echo.pl
The first argument was.
```

### Computing the Hypotenuse of a Right Triangle

Code:

```
#!/usr/bin/perl
# file: hypotenuse.pl
use strict;
use warnings;
$x = shift;
$y = shift;
$x>0 and $y>0 or die "Must provide two positive numbers";

print "Hypotenuse=",sqrt($x**2+$y**2),"\n";
```

Output:

```
(~) 82% hypotenuse.pl
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 83% hypotenuse.pl 1
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 84% hypotenuse.pl 3 4
Hypotenuse=5
(~) 85% hypotenuse.pl 20 18
```

Hypotenuse=26.9072480941474

(~) 86% hypotenuse.pl -20 18

Must provide two positive numbers at hypotenuse.pl line 6.

---



# Perl II

Logic and control structures

Sofia Robb

# What is truth?

`0` the number 0 is false

`"0"` the string 0 is false

`""` and `' '` an empty string is false

`my $x;` an undefined variable is false

everything else is **true**

# Examples of truth

```
my $a;           # FALSE (not yet defined)
$x = 1;          # TRUE
$x = 0;          # FALSE
$x = "\n";       # TRUE (a single newline is non-empty)
$x = 'true';     # TRUE
$x = 'false';    # TRUE (!!!! "false" is a nonempty string)
$x = ' ';        # TRUE (a single space is non-empty)
$x = 0.0;        # FALSE (converts to string "0")
$x = '0.0';      # TRUE (watch out! The string "0.0" is not the
                  # same as "0")
```

# defined

`defined` lets you test whether a variable is defined.

```
if ( defined($x) ) {  
    print "$x is defined\n";  
}
```

# Control structures

Control structures allow you to control if and how a line of code is executed.

You can create alternative branches in which different sets of statements are executed depending on the circumstances.

You can create various types of repetitive loops.

# Control structures

So far you've seen a basic program, where every line is executed, in order, and only once.

```
my $x = 1;  
my $y = 2;  
my $z = $x + $y;  
print "$x + $y = $z\n";
```

# Control structures

Here, the print statement is only executed some of the time.

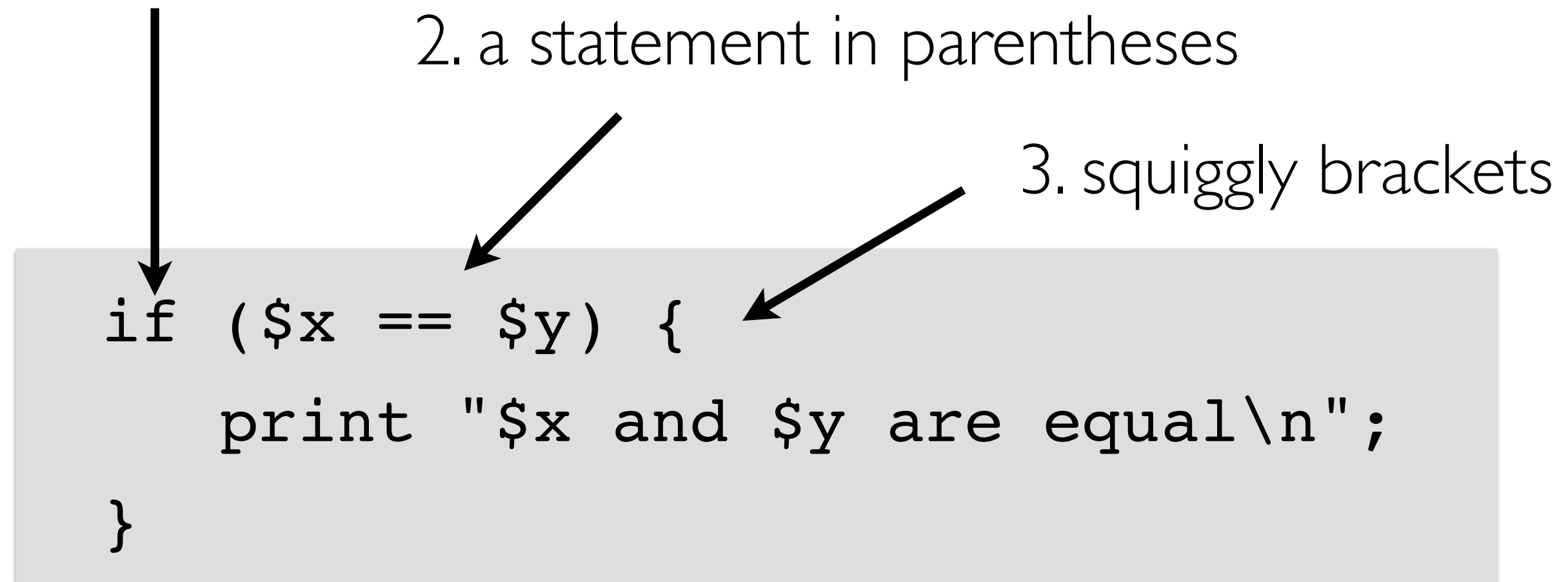
```
my $x = 1;  
my $y = 2;  
if ($x == $y) {  
    print "$x and $y are equal\n";  
}
```

# Components of a control structure

1. a keyword

2. a statement in parentheses

3. squiggly brackets



```
if ($x == $y) {  
    print "$x and $y are equal\n";  
}
```

The diagram illustrates the components of a control structure using an example code snippet. Three arrows point from labels to specific parts of the code: an arrow from '1. a keyword' points to 'if', an arrow from '2. a statement in parentheses' points to '(\$x == \$y)', and an arrow from '3. squiggly brackets' points to '{'.

The part enclosed by the squiggly brackets is called a block.



# Components of a control structure

When you program, build the structure first and then fill in.

1. a keyword

2. a statement in parentheses

3. squiggly brackets

```
if ($x == $y) {  
    print "$x and $y are equal\n";  
}
```

4. now add the print statement

# if

```
if ($x == $y) {  
    print "$x and $y are equal\n";  
}
```

If `$x` is the same as `$y`, then the print statement will be executed.

or said another way:

If `($x == $y)` is true, then the print statement will be executed.

# if — a common mistake

What happens if we write it this way?

```
if ($x = $y) {  
    print "$x and $y are equal\n";  
}
```

# if — a common mistake

- 1 equals sign to make the left side equal the right side.
- 2 equals signs to test if the left side is equal to the right.

```
my $x;           # x is undefined
my $x = 1;       # x is now defined
if ($x == 1)     # is $x equal to 1?
if ($x = 1)      # (wrong)
```

use `warnings` will catch this error.

# else

If the if statement is false, then the first print statement will be skipped and only the second print statement will be executed.

```
if ($x == $y) {  
    print "$x and $y are equal\n";  
}  
else {  
    print "$x and $y aren't equal\n";  
}
```

# elseif

Sometimes you want to test a series of conditions.

```
if ($x == $y) {  
    print "$x and $y are equal\n";  
}  
elseif ($x > $y) {  
    print "$x is bigger than $y\n";  
}  
elseif ($x < $y) {  
    print "$x is smaller than $y\n";  
}
```

# elseif

What if more than one condition is true?

```
if (1 == 1) {  
    print "$x and $y are equal\n";  
}  
elseif (2 > 0) {  
    print "2 is positive\n";  
}  
elseif (2 < 10) {  
    print "2 is smaller than 10\n";  
}
```

# Logical operators

Use **and** and **or** to combine comparisons.

Operator	Meaning
<b>and</b>	TRUE if left side is TRUE and right side is TRUE
<b>or</b>	TRUE if left side is TRUE or right side is TRUE



# Logical operator examples

```
if ($i < 100 and $i > 0) {  
    print "$i is the right size\n";  
}  
else {  
    print "out of bounds error!\n";  
}
```

```
if ($age < 10 or $age > 65) {  
    print "Your movie ticket is half price!\n";  
}
```

# while

As long as `( $x == $y )` is true, the print statement will be executed over and over again.

```
while ( $x == $y ) {  
    print "$x and $y are equal\n";  
}
```

# Perl III

File input and output

Sofia Robb

# Recap of UNIX I/O

STDIN - Reads in the text you type or from a file using redirection or pipes.

STDOUT - Prints to your screen, but can be redirected to a file or other program in the shell using redirection or pipes.

STDERR Standard error, used for diagnostic messages. Also prints to your screen, and also can be redirected in the shell.

# Recap of UNIX I/O

The UNIX way of reading from and writing to files is redirection.

If we use output redirection on the UNIX command line, the standard output goes to a file and we see only the standard error on the screen:

```
$ ls  
kubrick.txt
```

```
$ cat kubrick.txt  
Barry Lyndon  
The Shining
```

```
$ cat kubrick.txt fincher.txt > films.txt  
ls: fincher.txt: No such file or directory
```

# Perl I/O

The Perl way of reading from or writing to a file is the function `open`.

```
open(IN, '<', 'myfile.txt') or die "can't open  
myfile.txt: $!\n";
```

# open's first argument

open is a function, which takes 3 arguments:

```
open(IN, '<', 'myfile.txt')
```

↑  
First argument

1st argument, or the  
filehandle, can be any  
name.

Examples:  
FASTA, FILE, INFASTA,  
OUTFASTA, TAB, TEXT

The first argument is a filehandle. Filehandles are how you refer to a file within Perl.

STDOUT and STDERR are filehandles.

When you open a file yourself, you make your own filehandle and give it a name (here, I chose IN).

# Filehandles

Reading and writing to the filesystem is very complicated, involving bits, buffers, and memory.

Perl provides a 'handle' to the file and takes care of all the complicated parts for us so we can interact with a file more simply.

**Filename, filehandles, and the data in a file are three different things!!!!**

Filename	the name of the file
Filehandle	a way to get access to a file's contents
File contents	the actual data inside the file



# open's second argument

```
open(IN, '<', 'myfile.txt')
```



Second argument

The second argument is a mode. The modes are borrowed from redirection on the command line.

- < for reading from a file
- > for writing to a file
- >> for appending to a file

# open's third argument

```
open(IN, '<', 'myfile.txt')
```

↑  
Third argument

The third argument is the name of a file to open. It can either be a literal name:

```
open(IN, '<', 'myfile.txt')
```

or a variable containing a filename:

```
open(IN, '<', $file)
```

# Catch errors with `die`

If you're going to read from a file, that file must exist and be readable.

Since it rarely makes sense to continue when it's not possible to read the file, we want the program to stop. We do this with `die`.

```
open(IN, '<', 'myfile.txt')  
    or die "can't open myfile.txt: $!\n";
```

**`open or die`** is a Perl idiom. `die` is a function that exits the program immediately and prints the specified string to `STDERR`.

# Capturing system errors with \$!

Perl can also tell us what the filesystem said about why the file couldn't be opened.

`$!` is a special Perl variable that contains error messages from the system. If there was a problem with opening your file, there will be an error message in `$!`, and we can include it in our error string.

```
or die "can't open $file: $!\n";
```

↑  
contains error  
message from  
the filesystem

# Open a file for writing

Open also can be used to open files for writing by using '>' as the second argument to open.

```
my $out = 'out.txt';  
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

Now specify that filehandle when you print

```
print OUT "I'm writing to a file!\n";
```

and the output will go into a file instead of the screen:

```
$ perl myprog.pl          ← no redirection on the command line  
$ cat out.txt  
I'm writing to a file!
```

# Open a file for writing

If you open a file for writing and the file doesn't exist, it will be created.

```
$ ls
```

← look! no file there!

```
$ perl myprog.pl
```

```
$ ls
```

```
out.txt
```

← out.txt has been created by myprog.pl

Be careful! If you open an existing file for writing, you will erase everything inside that file!

# Opening multiple files

You can open more than one file in a script — just give them different filehandles.

```
my $in  = 'in.txt';  
my $out = 'out.txt';  
open(IN, '<', $in ) or die "can't open $in: $!\n";  
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

# Open files from user input

Instead of hardcoding filenames inside your program, you can read them in from the command line:

```
my $in  = shift;  
my $out = shift;  
open(IN, '<', $in ) or die "can't open $in: $!\n";  
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

On the command line, you'd type this:

```
$ perl test.pl myinfile.txt myoutfile.txt
```



# Open files from user input

## Command line

```
$ perl test.pl myinfile.txt myoutfile.txt
```

## Inside our Perl program

```
my $in  = shift;  
my $out = shift;  
open(IN, '<', $in ) or die "can't open $in: $!\n";  
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

Which are the filehandles and which are the filenames?

`myinfile.txt` and `myoutfile.txt` are filenames.

`IN` and `OUT` are filehandles.

`$in` and `$out` are variables containing the filenames.

# <> to get contents out of a file

Perl reads files one line at a time.

To read a line from a file, you put the filehandle inside <>, like this:

```
my $in = 'in.txt';  
open(IN, '<', $in ) or die "can't open $in: $!\n";  
  
print "This is the first line from the file $in:  
\n";  
my $line = <IN>;  
print $line;
```

# <> to get contents out of a file

This code reads the first two lines from a file:

```
my $in = "in.txt";  
open(IN, '<', $in ) or die "can't open $in: $!\n";  
  
print "This is the first line from the file $in:\n";  
my $line = <IN>;  
print $line;  
  
print "This is the 2nd line from the file $in:\n";  
$line = <IN>;  
print $line;
```

# <> to get contents out of a file

Most files have lots of lines, and we often want to read all the lines in a file one by one. We can do that using a while loop.

To read from a filehandle line by line, put

```
my $line = <IN>
```

into a while loop, like this:

```
my $in = shift;
open(IN, '<', $in) or die "can't open $in: $!\n";

while (my $line = <IN>) {
    chomp $line;
    print "This line is from the file $in:\n";
    print $line\n";
}
```

# Removing newlines with chomp

chomp removes the newline from the end of a string (if there is a newline).

```
my $string = "hey there!\n";  
print "my string is: ", $string, "\n";  
chomp $string;  
print "after chomp : ", $string, "\n";
```

When you read a line from a file, the first thing you always want to do is chomp.

# Counting lines in a file

Let's do something more interesting than printing the line back out. Let's count how many lines there are in the file.

```
my $line_count;
while (my $line = <IN>) {
    chomp $line;
    $line_count = $line_count + 1;
}
print "There are $line_count lines\n";
```

# Why we read a file with `while`

Let's step back for a moment and think about why this works. What exactly is going on on this line?

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

`<IN>` returns a line from a file.

We assign that line to a variable, `$line`.

`while` tests that assignment for truth:

"Can we assign a value to `$line` ?"

If we've hit the end of the file, there are no more lines to read,  
and so the answer is "no", or FALSE.

When the expression in parentheses is false, we exit the loop.

**What happens if the input file contains a blank line?**

# Example Sequence File Parser

HDAC.nt

```
CTTTTTTAAATTTAGTAAATGGAAACCCAGTTCGTAAGAAAGTGTGCTATTATTATGATGGAGACATT
GGGAATTATTATTATGGCCAGGGTCATCCCATGAAACCTCACCGTATTCGCATGACGCACAGTTTACTTT
TAAATTATGGCCTATATAGAAAGATGGAAGTATACAAGCCTTCAAAGGCTACTGCAGACGACATGACTAT
GTTTCATACTGATGAGTATATCCAATTTCTTCAGAGAATCCATCCCGATAATATGCACGAATACAACAAA
GAAATGCAAAGGTTCAACGTTGGAGAAGATTGTCCTGTATTTGATGGGTTATTCGAATTTGTCAATTAT
```

Notice that the sequence is not contiguous. There are new lines at the end of each line.

```
my $in = shift;
open(IN, '<', $in) or die "can't open $in: $!\n";
my $seq = '';
while (my $line = <IN>) {
    chomp $line;
    $seq .= $line;
}
print $seq , "\n";
```



# Arrays and Loops

Sofia Robb

# An array is a Named Ordered List.

- What is a list?
  - ('cat', 'dog', 'narwhal')
- Why is it named?
  - @animals = ('cat', 'dog', 'narwhal');
- Why is it ordered?
  - each element has an ordered numerical index or position

Arrays are denoted with '@' symbol

0	1	2
cat	dog	narwhal

# Arrays

- Each element of an array has to be a scalar variable
- These are all scalar variables
  - number
  - letter
  - word
  - sentence
  - \$scalar\_variable

# Example array

```
my @colors = ('red', $favorite_color,  
             'cornflower blue', 5);
```

# The elements of the array are stored in a specific order.

```
my @colors = ('red', $favorite_color,  
             'cornflower blue', 5);
```

0	1	2	3
'red'	\$favorite_color	'cornflower blue'	5

\$colors[0]

\$colors[1]

\$colors[2]

\$colors[3]

Each element of an array can be accessed by its position, or index, in the array.

```
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);
```

GET

```
my $first    = $colors[0];  
my $second   = $colors[1];  
my $third    = $colors[2];  
my $last     = $colors[-1];
```



negative numbers can be used to access from the end

The value of each element can be reassigned with use of its index.

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

```
$colors[0] = 'green';  
$colors[2] = 'gray';
```

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
green	\$favorite_ color	gray	5

# Assign values to indices that are far away

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

```
$colors[0] = 'green';  
$colors[2] = 'gray';  
$colors[8] = 'black';
```

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]	\$colors[4]	\$colors[5]	\$colors[6]	\$colors[7]	\$colors[8]
green	\$favorite_ color	gray	5	undefined	undefined	undefined	undefined	black

@colors now contains 9  
elements.  
4 of the elements are  
undefined.



# GET/SET: Mirror Images

```
#GET:
```

```
$first    = $colors[0];
```

```
$second   = $colors[1];
```

```
#SET:
```

```
$colors[0] = 'green';
```

```
$colors[2] = 'gray';
```

A common MISTAKE is to try to access an element in array context  
( meaning using the '@').

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```

This is wrong:

```
my $first = @colors[0];
```

This is correct:

```
my $first = $colors[0];
```

# Length of an array

```
scalar(@array)
```

The scalar() function can be used to return the scalar attribute of an array. Its scalar attribute is the length, or in other words, the number of elements in the array.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```

```
my $length = scalar @colors;  
print "len of array: $length\n";
```

## Output:

```
len of array: 4
```

A common MISTAKE is to use the `length()` function to get the number of elements in an array

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```

## WRONG:

```
my $length = length @colors;  
print "len of array: $length\n";
```

## Output:

```
len of array: 1
```

## CORRECT:

```
my $length = scalar @colors;  
print "len of array: $length\n";
```

## Output:

```
len of array: 4
```

# Quick print of an array

When an array is printed with use of double quotes ("@array"), a single white space is automatically inserted between each element. This allows for a quick way to visualize the contents of your array.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);  
print "@colors";
```

## Output

```
red purple cornflower blue 5
```

Notice that the print out of the array looks like it has 5 elements while our array actually has 4 elements. Printing within quotes may not always be helpful in cases when a white space is included within a single element, such as 'cornflower blue'.

# Array to a String

```
my $new_string = join(string , @array);
```

join() can be used to combine all the individual elements of a list or an array into a string on a set of characters. A string is returned.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```

```
my $new_string = join ('--' , @colors);  
print "$new_string\n";
```

## Output

```
red--purple--cornflower blue--5
```

'--' is used here to clearly differentiate the elements of @colors. A tab ("\t") is a common character to use with the join() function.

# Arrays are Dynamic

Not only can values be reassigned but  
Arrays can grow and shrink.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```

**unshift**

**push**



\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_color	cornflower blue	5

**shift**

**pop**

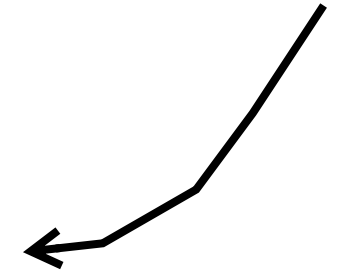
shift() has been used in  
previous lectures to get user  
command line arguments

## Add elements to the end with push();

```
push (@array, list of values);
```

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

push



```
#add one element to the end  
push (@colors, 'black');  
print join ('--', @colors) , "\n";
```

## Output

```
red--purple--cornflower blue--5--black
```

push() is changing the  
actual array

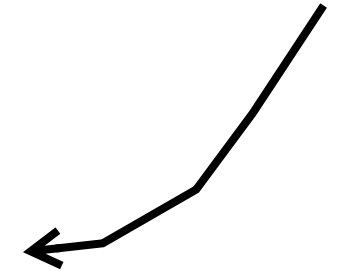


## Add elements to the end with push();

```
push (@array, list of values);
```

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

push



```
#add multiple elements to the end  
push (@colors, 'black','blue');  
print join ('--', @colors) , "\n";
```

## Output

```
red--purple--cornflower blue--5--black--blue
```

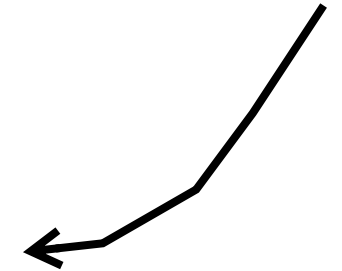
push() is changing the  
actual array

# Add elements to the end with push();

```
push (@array, list of values);
```

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

push



```
#add an array of elements  
my @more_colors = ('yellow','pink','white','orange');  
push (@colors, @more_colors);  
print join ('--', @colors) , "\n";
```

## Output

```
red--purple--cornflower blue--5--black--yellow--pink--white--orange
```

push() is changing the  
actual array

## Remove an element from the end with pop();

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

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5



**pop**

```
my $last_element = pop @colors;  
  
print "last: $last_element\n";  
print join ('--', @colors) , "\n";
```

## Output

```
last: 5  
red--purple--cornflower blue
```

pop() is changing the  
actual array

# Remove an element from the beginning with shift();

```
$first = shift(@array);
```

**shift** ←

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_color	cornflower blue	5

```
my $first_element = shift (@colors);
```

```
print "first: $first_element\n";  
print join ('--', @colors) , "\n";
```

## Output

```
first: red  
purple--cornflower blue--5
```

shift() is changing the actual array

# Add elements to the beginning with unshift();

```
unshift (@array, list of values);
```

unshift



\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

```
#add one element to the beginning  
unshift (@colors, 'black');  
print join ('--', @colors) , "\n";
```

## Output

```
black--red--purple--cornflower blue--5
```

# Add elements to the beginning with unshift();

```
unshift (@array, list of values);
```

unshift



\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

```
#add two elements to the beginning  
unshift (@colors, 'black' , 'blue');  
print join('--',@colors), "\n";
```

## Output

```
black--blue--red--purple--cornflower blue--5
```

# Add elements to the beginning with unshift();

```
unshift (@array, list of values);
```

unshift



\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
red	\$favorite_ color	cornflower blue	5

```
#add an array of elements to the beginning  
my @more_colors = ('yellow','pink','white','orange');
```

```
unshift (@colors, @more_colors);  
print join('--',@colors) , "\n";
```

## Output

```
yellow--pink--white--orange--red--purple--cornflower blue--5
```

# Dynamic Arrays

Function	Meaning
<code>push(@array, a list of values)</code>	add value(s) to the end of the list
<code>\$popped_value = pop(@array)</code>	remove a value from the end of the list
<code>\$shifted_value = shift(@array)</code>	remove a value from the front of the list
<code>unshift(@array, a list of values)</code>	add value(s) to the front of the list
<code>splice(...)</code>	everything above and more!



# String to an Array

```
my @array = split(/pattern/ , string);
```

The `split()` function can be used to create an array from a string by providing a delimiter of any set of characters or any pattern. `split()` is similar to Excel's "Text to columns" feature that allows you to indicate which characters separate each field, such as tabs (`\t`) and commas (`,`). Just like in Excel, the `split()` function will remove the delimiter, and it will not be present in the returned data.

```
my $string = "I do not like green eggs and ham";
```

```
#'/ /' sets the delimiter to a single white space  
my @words = split(/ /,$string);
```

```
print join('--',@words), "\n";  
I--do--not--like--green--eggs--and--ham
```

Notice that there are no white spaces in the printed array. The delimiter was removed.

# Using `qw()` to create a list of words

```
my @array = ('one', 'two', 'three', 'four');
```

It is a lot of work to type all the quotes and commas.  
Use `qw()` instead:

```
my @array = qw( one two three four );
```

`qw()` will produce a list of quoted words:  
('one', 'two', 'three', 'four')  
that can now be saved as an array

# Sorting

```
my @sorted_array = sort (@array)
```

The sort() function is used to sort a list. The default behavior is to sort in ascii order. A sorted list is returned.

```
my @words = qw(I do not like green eggs and ham);
```

```
my @sorted_words = sort @words;  
print join('--' , @sorted_words), "\n";
```

## Output

```
I--and--do--eggs--green--ham--like--not
```

ascii sort order:

0-9

A-Z

a-z

# Default Sort: `sort {$a cmp $b}`

```
my @sorted_array = sort {$a cmp $b} (@array)
```

The `sort()` function performs a series of pairwise comparisons of all the elements in the list. For example it compares the first (`$a`) and second (`$b`) elements, tests if `$a` is less than `$b`, then it makes another pairwise comparison and so on until the list is sorted.

```
my @words = qw(I do not like green eggs and ham);
```

```
##sort {$a cmp $b} is default sort() behavior  
my @sorted_words = sort {$a cmp $b} @words;  
print join('--' , @sorted_words), "\n";
```

`sort @array`  
is equivalent to  
`sort {$a cmp $b}`

## Output

```
I--and--do--eggs--green--ham--like--not
```

`$a` and `$b` are special Perl variables and do not need to be declared. If you use these elsewhere in the same scope, the sort function won't work. This is another reason not to use uninformative variable names (like `a` and `b`) when you're writing your scripts!

# Quick Review:

## The comparison operator and strings

```
my $x = 'sid';
```

```
my $y = 'nancy';
```

```
my $result = $x cmp $y;
```

\$result is:

- 1 if the left side is less than the right side
- 0 if the left side equals the right side
- +1 if the left side is greater than the right side

# Quick Review:

## The comparison operator and numbers

```
my $x = 2;
```

```
my $y = 3.14;
```

```
my $result = $x <=> $y;
```

\$result is:

- 1 if the left side is less than the right side
- 0 if the left side equals the right side
- +1 if the left side is greater than the right side

# The comparison operator

use cmp to compare two strings

```
my $x = 'sid';
```

```
my $y = 'nancy';
```

```
my $result = $x cmp $y;
```

use `<=>` to compare two numbers

```
my $x = 2;
```

```
my $y = 3.14;
```

```
my $result = $x <=> $y;
```

# Modify sort behavior for Numeric Sorting

The default sort can be modified by specifying the sort behavior in {} using Perl reserved variables \$a and \$b.

```
my @numbers = (15,2,10,20,11,1);
```

```
## default sorting is ascii  
my @sorted_numbers = sort @numbers;  
print "@sorted_numbers\n";
```

## Output

```
1 10 11 15 2 20
```

```
## modify to sort numerically  
@sorted_numbers = sort {$a <=> $b} @numbers;  
print "@sorted_numbers\n";
```

## Output

```
1 2 10 11 15 20
```



# More sort() customization with uc()

Before \$a and \$b are compared they are uppercased. This only changes the temporary copy of the array elements stored in \$a and \$b during the sort. The actual array elements are not being changed. It is a sorted list of the original list that is being returned

```
my @sorted = sort { uc($a) cmp uc($b) } @array;
```

```
my @words = qw(I do not like green eggs and ham);  
my @sorted_words = sort {uc($a) cmp uc($b)} (@words);  
print join('--', @sorted_words), "\n";
```

## Output

```
and--do--eggs--green--ham--I--like--not
```

The returned list is in the same format as the original list. The uc() used on \$a and \$b did not change the @array

# Sort based on the length of each element

```
my @sorted = sort { length($a) <=> length($b) } @array;
```

```
my @words = qw(I do not like green eggs and ham);  
my @sorted_words = sort {length($a) <=> length($b)} (@words);  
print join('--',@sorted_words), "\n";
```

## Output

```
I--do--not--and--ham--like--eggs--green
```

The returned list is in the same format as the original list. The `length()` used on `$a` and `$b` did not change the `@array`

Noticed that the `<=>` is used. The lengths of the words are being compared.

Warning Advanced!! Sort on length then on alphabet.

```
my @sorted_words = sort {length($a) <=> length($b) || uc($a) cmp uc($b)} (@words);
```

# Reverse Sorting

Reverse the order of \$a and \$b to reverse the results of sort.

```
my @words = qw(I do not like green eggs and ham);  
my @sorted_words = sort {$b cmp $a} @words;  
print join('---', @sorted_words), "\n";
```

## Output

```
not---like---ham---green---eggs---do---and---I
```

# Swapping the values of 2 elements

```
my @words = qw(I do not like green eggs and ham);  
print "Before Swap : w5=$words[5] w7=$words[7]\n";
```

```
my $val_1 = $words[5];  
my $val_2 = $words[7];  
$words[5] = $val_2;  
$words[7] = $val_1;
```

```
print "After Swap : w5=$words[5] w7=$words[7]\n";  
print join('--', @words), "\n";
```

## Output

```
Before Swap : w5=eggs w7=ham  
After Swap : w5=ham w7=eggs  
I--do--not--like--green--ham--and--eggs
```

What is wrong with this?

```
$words[5] = $words[7];  
$words[7] = $words[5];  
print join('--', @words), "\n";
```

```
I--do--not--like--green--ham--and--ham
```

# Swapping values

```
my @words = qw(I do not like green eggs and ham);  
print "Before Swap : w5=$words[5] w7=$words[7]\n";  
  
($words[5], $words[7]) = ($words[7], $words[5]);  
  
print "After Swap : w5=$words[5] w7=$words[7]\n";  
print join('---', @words), "\n";
```

## Output

```
Before Swap:  
w5:eggs w7:ham  
after swap:  
w5:ham w7:eggs  
I---do---not---like---green---ham---and---eggs
```

- Loops
  - `foreach()` : perfect for arrays
  - `for()` : good for arrays and much more
  - `while()` : perfect for many things other than arrays as well as lines of files

# foreach loop

The foreach loop is especially equipped to iterate through each element of a list. It retrieves the value of each element of the list, one at a time, in the order of indices, and stores it in a variable for use within the foreach code block.

<code>\$array[0]</code>	<code>\$array[1]</code>	<code>\$array[2]</code>	<code>\$array[3]</code>	<code>\$array[4]</code>	<code>\$array[5]</code>
15	2	10	20	11	1

```
my @array = (15,2,10,20,11,1);
```

```
foreach my $one_element (@array) {  
    ##do something to each $one_element  
  
}
```

# foreach loop

The foreach loop is especially equipped to iterate through each element of a list. It retrieves the value of each element of the list, one at a time, in the order of indices, and stores it in a variable for use within the foreach code block.

\$array[0]	\$array[1]	\$array[2]	\$array[3]	\$array[4]	\$array[5]
15	2	10	20	11	1

```
my @array = (15,2,10,20,11,1);
```

```
foreach my $one_element (@array) {  
    ##do something to each $one_element  
    print "Number: $one_element\n";  
}
```

## Output

```
Number: 15  
Number: 2  
Number: 10  
Number: 20  
Number: 11  
Number: 1
```

A foreach loop  
**knows** everything  
about your array.



# foreach() code block

All the lines within the foreach code block will be executed on each array element, one at a time.

```
my @words = qw(I do not like green eggs and ham);

foreach my $word (@words) {
    my $uc_word = uc($word);
    my $len = length($word);
    print "word: $uc_word($len) \n";
}
```

## Output

```
word: I (1)
word: DO (2)
word: NOT (3)
word: LIKE (4)
word: GREEN (5)
word: EGGS (4)
word: AND (3)
word: HAM (3)
```

start at \$index=0;

- 1.The value of the index \$index is retrieved from @words and a copy is stored in \$word.
2. \$word is uppercased and the result is stored in \$uc\_word.
- 3.The length of \$word is calculated and stored in \$len.
4. \$uc\_word and \$length are printed.
5. Increment to the next index (\$index++).
- 6.Go to Step **1**, repeat until the foreach code block is executed on all elements

# for loop

The for loop is especially equipped to keep count and for repeating a block of code until a numerical condition is met.

```
for(initialization; test; increment){  
    statements;  
}
```

```
for (my $i=0; $i<5 ; $i++) {  
    print "\$i is: $i\n";  
}
```

## Output

```
$i is: 0  
$i is: 1  
$i is: 2  
$i is: 3  
$i is: 4
```

A for loop **does not know** anything about your array.

# for loop

The for loop can also be used with @arrays. The \$i can be used to retrieve each the value of each index.

\$array[0]	\$array[1]	\$array[2]	\$array[3]	\$array[4]	\$array[5]
15	2	10	20	11	1

```
my @array = (15,2,10,20,11,1);  
for (my $i=0; $i<scalar @array ; $i++) {  
    my $value = $array[$i];  
    print "value of $i is $value\n";  
}
```

## Output

```
value of 0 is 15  
value of 1 is 2  
value of 2 is 10  
value of 3 is 20  
value of 4 is 11  
value of 5 is 1
```

Loops are similar to the steps in a thermocycler program

start at \$i=0;

1. if \$i is less than the length of the @array (scalar @array) then the code in the for block will be executed.

2. \$value is set to contain the contents of \$array[\$i].

3. \$value is printed

4. \$i is auto incremented. (\$i=\$i+1);

5. Go to Step 1, repeat as long as the test (\$i<scalar @array) remains true.

# Thermocycler Program and loops

## Standard PCR program

1. 94 °C 3 min : Initial Denature
2. 94 °C 30 sec : Denature
3. 57 °C 30 sec : Annealing
4. 72 °C 1 min : Extension
5. Go to step 2, for additional 29 times
6. 72 °C 5 min
7. 4 °C for ever

```
my ($temp, $time);
my $cycles = 30;

($temp,$time) = (94,"3min");
doDenature($temp,$time);

for (my $i=0 ; $i<$cycles ; $i++) {

    ($temp,$time) = (94, "30sec");
    doDenature($temp,$time);
    ($temp,$time) = (57, "30sec");
    doAnnealing($temp,$time);
    ($temp,$time) = (72, "1min");
    doExtension($temp,$time);
}

($temp,$time) = (72, "5min");
doAnnealing($temp,$time);

while (1){
    ($temp,$time) = (4, "forever");
    doChilling($temp,$time);
}
```

# while loop

while loops continue to execute the while code block until the while conditional statement is false.

```
while (condition) {  
    code;  
}
```

A while loop **does not know** anything about your array.

# while loops and <FILEHANDLES>

while loops are great for getting lines from a file one by one and executing code on each line. It will continue until the condition is false.

```
open (IN, "<", "file.txt") or die "Can't open file.txt $!";

while (my $line = <IN>) {
    chomp $line;
    print "$line\n";
}
```

## Output

```
file line 1
file line 2
file line 3
```

start at line 1;

1. <> is an operator that returns the contents of one line from a file. If the end of the file is reached, nothing is returned, and nothing is false.

2. if the contents of \$line is true the code block is executed.

3. \$line is chomped, then printed.

4. Go to Step 1.

# while loop

while loops can also be used for counting.

```
my $i = 0;
while ($i<5) {
    print "\$i is $i\n";
    $i++;
}
```

This instance of the while loop functions like a for loop.

1. A counter is initialized
2. there is a test that incorporates the counter
3. the counter is changed in each iteration of the loop.

## Output

```
$i is: 0
$i is: 1
$i is: 2
$i is: 3
$i is: 4
```

# for and while loop can do the same thing

for and while loops can be used to do the same things, the format is just different. Neither way is better, just different

## for(;;){

```
for (my $i=0; $i<5 ; $i++) {  
    print "$i\n";  
}
```

## while(){

```
my $i = 0;  
while ($i<5) {  
    print "$i\n";  
    $i++;  
}
```

\$i	\$i<5	print "\$i\n";	\$i++
0	yes	0	1
1	yes	1	2
2	yes	2	3
3	yes	3	4
4	yes	4	5
5	no		



# Different loops can do the same things

foreach and for loops with arrays	for and while loops with counters
<pre>my @array = (15,2,10,20,11,1);</pre>	
<pre>foreach my \$ele(@array){     print "\$ele\n"; }</pre>	<pre>for (my \$i=0; \$i&lt;5 ; \$i++){     print "\$i\n"; }</pre>
<pre>for (my \$i=0; \$i&lt;scalar @array ; \$i++){     my \$ele = \$array[\$i]     print "\$ele\n"; }</pre>	<pre>my \$i = 0; while (\$i&lt;5){     print "\$i\n";     \$i++; }</pre>

# Loop Control: next()

execution of next() will cause the loop to jump to the next iteration. Any code, in the loop block, that falls after the next will be skipped. The next iteration of the loop will commence. All code after the loop block will also be executed.

```
my @words = qw(I do not like green eggs and ham);

foreach my $word (sort {uc($a) cmp uc($b)} @words) {
    if ($word eq 'and') {
        next;
    }
    print "$word\n";
}
```

Every element but  
'and' is printed.

## Output

```
do
eggs
green
ham
I
like
not
```

# Loop Control: last

execution of `last()` will cause the loop to exit the loop. Any code, in the loop block, that falls after the `last` will be skipped. No further iterations will be attempted. All code that falls after the loop block will also be executed.

```
my @words = qw(I do not like green eggs and ham);  
  
foreach my $word (@words) {  
    if ($word eq 'and') {  
        last;  
    }  
    print "$word\n";  
}
```

```
I  
do  
not  
like  
green  
eggs
```

Every word before 'and' in @words is printed. When the element is equal to 'and' the current iteration ends, the loop block is exited and no other words are printed

# Sorting, Arrays and Loops

```
my @words = qw(I do not like green eggs and ham);  
#my @sorted = sort {uc($a) cmp uc($b)} @words;
```

Previously, the array was sorted and the sorted results were stored in a new array

```
foreach my $word (sort {uc($a) cmp uc($b)} @words) {  
    print "$word\n";  
}
```

## Output

```
and  
do  
eggs  
green  
ham  
I  
like  
not
```

Here,

1. the array is sorted
2. the final sorted results are returned to the foreach loop.
3. Then one element at a time, in the sorted list will be stored in \$word
4. Each element stored in \$word will be processed in the loop code block

# Example usage of a foreach loop

```
my @seqs = qw(TTT CGG ATG TAA CCC ACC TGA);

my $count = 0;
foreach my $seq (@seqs) {
    if ($seq eq 'TAA' or $seq eq 'TGA' or $seq eq 'TAG') {
        print "*\n";
    } else {
        $count++;
    }
}
print "$count non-stop codons\n";
```

## Output

```
*
*
5 non-stop codons
```

# @ARGV

@ARGV is a special Perl array that automatically contains the list of arguments that follow the script name on the command line.

```
./sample_usr_input.pl 5 five
```

\$ARGV[0]	\$ARGV[1]
5	five

```
print "\@ARGV: @ARGV\n";
```

```
print "\$ARGV[0]: $ARGV[0]\n";
```

```
print "\$ARGV[1]: $ARGV[1]\n";
```

```
my $arg1 = shift @ARGV;
```

```
my $arg2 = shift @ARGV;
```

```
print "arg1: $arg1\n";
```

```
print "arg2: $arg2\n";
```

## Output

```
@ARGV: 5 five
```

```
$ARGV[0]: 5
```

```
$ARGV[1]: five
```

```
arg1: 5
```

```
arg2: five
```

# Code Example: Same analysis on multiple files

You have multiple files you want to process in the same way. Use loops!!

```
./processManyFiles.pl *txt

my @files = @ARGV;

foreach my $file (@files){
    print $file , "\n";
    open (INFILE, "<" , $file) or die "Can't open $file: $!\n";
    while (my $line = <INFILE>){
        chomp $line;
        print "\t", length $line , "\n";
    }
}
```

# Hashes

Sofia Robb



# First a few words about when to use: Arrays and Hashes

- Use an array when ordered values matter or when you have an arbitrary list of values.
  - Sort list:  
( 0.001, 0.01 , 0.1 , 1 )  
( 'apple', 'bee', 'cedar' , 'deer' )
  - States:  
( 'MD', 'UT', 'CA', 'MO', 'OR', 'NM' );
- Use a hash when order does not matter but you have paired information
  - dictionaries : Word => Definition
  - FASTA : Gene => Sequence

# Hashes

- Perl hashes are denoted with a '%' symbol like this:  
%data
- Each key and each value contains a scalar value for example this could be
  - a number
  - a letter
  - a word
  - a sentence
  - a scalar variable like \$scalar\_variable
  - a gene ID
  - a sequence

# What is a hash?

- A hash is an associative array made up of key/value pairs.
- Like a dictionary
- And unlike an array a hash is unordered.
- Keys need to be Unique!!

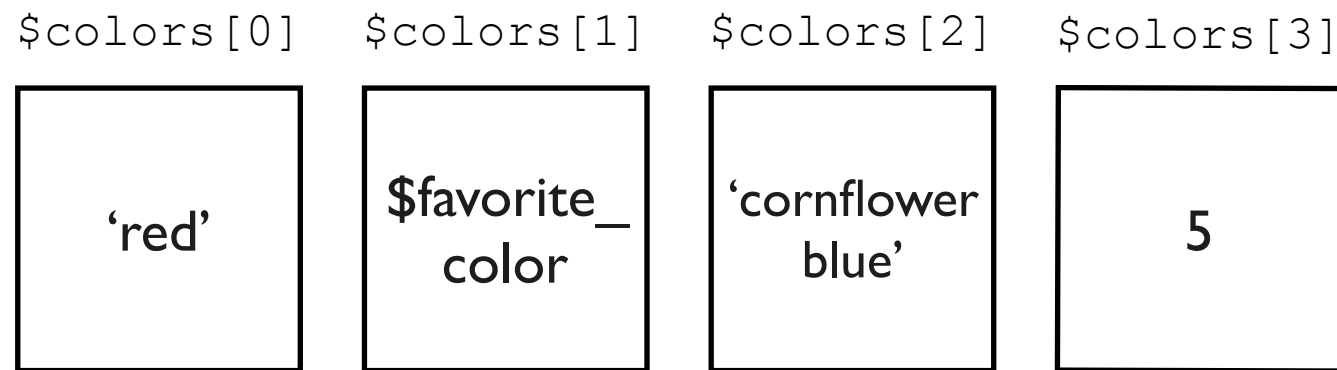
key 

value
-------

'ATG'	Met
'AAA'	Lys
'CCA'	Pro

# A key is like a descriptive array index.

## An array

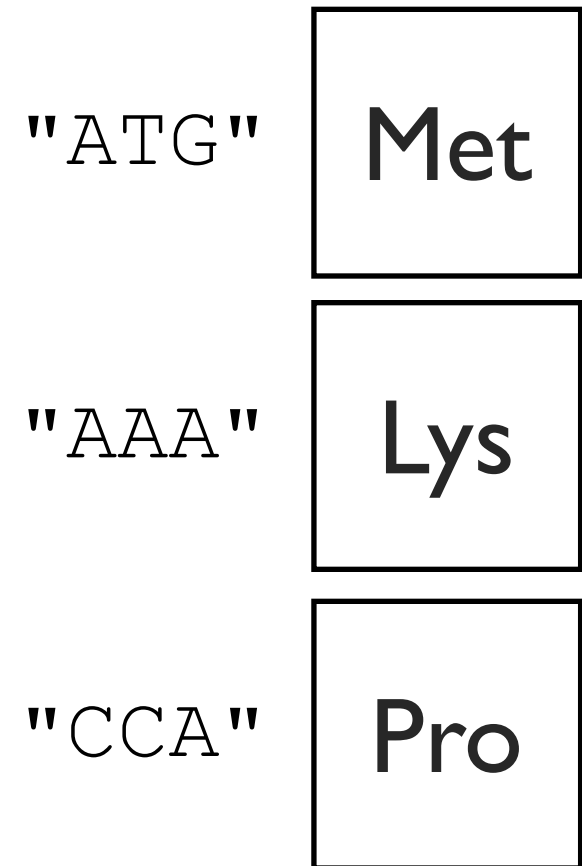


The array index `[0]` is similar to the key `"ATG"`.

The key `"ATG"` is used to access the value `"Met"`, just as `[0]` is used to access `"red"`

But the key/value pairs are not stored in order

## A hash



## Creating a Hash *All At Once*

The hash %genetic\_code is built with key/value pairs

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

key	value
"ATG"	Met

# Creating a Single Hash *One Key/Value Pair at a time.*

```
$hash{key} = "value";
```

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
);
```

**SAME AS:**

```
$genetic_code{"ATG"} = "Met";  
$genetic_code{"AAA"} = "Lys";  
$genetic_code{"CCA"} = "Pro";
```

**Even though there are 3 lines, this is still only 1 hash**

# Accessing a hash value using a key

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

Look!! GET/SET are mirror images!!

SET: \$genetic_code{"ATG"} = \$value; GET: \$value = \$genetic_code{"ATG"};
--------------------------------------------------------------------------------

```
my $aa = $genetic_code{"ATG"};  
print "ATG translates to $aa\n";  
ATG translates to Met
```

Each value of the hash is a scalar therefore we use the '\$' when we refer to an individual value.

Hash keys are surrounded by squiggly brackets {}

**keys()** returns an unordered list of the keys of a hash

```
@array_of_keys = keys (%hash);
```

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

```
my @codons = keys (%genetic_code);  
print join("--", @codons), "\n";  
CCA--AAA--ATG
```



# values() returns an unordered list of values

```
@array_of_values = values(%hash);
```

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

You can use `sort values` to be sure that the order of the values is always the same.

```
my @amino_acids = values(%genetic_code);  
print join("--", @amino_acids), "\n";  
Pro--Lys--Met
```

# Iterating through a hash by looping through an list of hash keys.

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

Remember: the key is used to access  
the value

```
$value = $hash{$key}
```

```
foreach my $codon (keys %genetic_code) {  
  my $aa = $genetic_code{$codon};  
  print "$codon translates to $aa\n";  
}
```

*CCA translates to Pro*

*AAA translates to Lys*

*ATG translates to Met*

# Sorting and iterating through the keys of a hash

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

Remember: hash keys are unordered so we use `sort` to be sure that the order is always the same.

```
foreach my $codon (sort(keys %genetic_code)) {  
  my $aa = $genetic_code{$codon};  
  print "$codon translates to $aa\n";  
}
```

*AAA translates to Lys*  
*ATG translates to Met*  
*CCA translates to Pro*

**Nested Functions:**  
(sort (keys %hash))

1. keys() returns a list of all the keys in %hash
2. This list is passed on to sort().
3. sort() will then proceed to sort the list

# Iterating through a hash and sorting by the values

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

Remember: the key is used to access  
the value

`$value = $hash{$key}`

`$a` and `$b` are keys, the value of key `$a` is being sorted in  
comparison to the value of key `$b`

evaluates to a value

evaluates to a value

```
foreach my $codon (sort { $genetic_code{$a} cmp $genetic_code{$b} } keys %genetic_code) {
```

```
  my $aa = $genetic_code{$codon};  
  print "$codon translates to $aa\n";  
}
```

```
AAA translates to Lys  
ATG translates to Met  
CCA translates to Pro
```

we can create a custom  
sort function using  
`{ $a cmp $b }`

# Adding additional key/value pairs

```
$hash{key} = "value";
```

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
);
```

```
$genetic_code{"TGT"} = "Cys";
```

```
foreach my $codon (keys %genetic_code) {  
    print "$codon -- $genetic_code{$codon} \n";  
}
```

```
CCA -- Pro
```

```
AAA -- Lys
```

```
ATG -- Met
```

```
TGT -- Cys
```

# Over-writing values

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

```
$hash{key} = "value";
```

```
print "Before/ATG: " , $genetic_code{"ATG"} , "\n";
```

```
$genetic_code{"ATG"} = "start_codon";
```

```
print "After/ATG: " , $genetic_code{"ATG"} , "\n";
```

*Before/ATG: Met*

*After/ATG: start\_codon*

Since keys need to be unique a value is over-written when using a preexisting key.

# Deleting key/value pairs

```
delete $hash{"KEY"};
```

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
);
```

```
delete $genetic_code{"AAA"};
```

```
foreach my $codon (keys %genetic_code) {  
    print "$codon -- $genetic_code{$codon} \n";  
}
```

*CCA -- Pro*

*ATG -- Met*

# Use exists() to test if a key exists.

```
exists $hash{"KEY"}
```

```
my %genetic_code = (  
  "ATG" => "Met",  
  "AAA" => "Lys",  
  "CCA" => "Pro",  
);
```

key exists?	return value
yes	1
no	'' empty string is false

```
my $codon = "ATG";  
if (exists $genetic_code{$codon}) {  
  print "$codon -- $genetic_code{$codon}\n";  
}else{  
  print "key: $codon does not exist\n";  
}
```

*ATG -- Met*

*##when \$codon= "TTT", code prints "key: TTT does not exist"*



# Auto increment hash values

Auto increment scalars:

```
my $num = 1;
print $num , "\n";    #prints 1
$num++;               #same as $num=$num +1;
print $num , "\n";    #prints 2
```

Auto increment hash values:

```
my %hash;
$hash{books} = 0;
print $hash{books}, "\n";    #prints 0
$hash{books}++; #same as $hash{books} = $hash{books} + 1
print $hash{books} , "\n";    #prints 1
```

# nothing + 1 equals 1

```
my %hash;
```

```
$hash{books} = 0;  
print $hash{books}, "\n";
```

```
$hash{books}++;  
print $hash{books} , "\n"; # prints 1
```

When we first start, the key 'books' doesn't exist.  
We try to add 1 to nothing, so the total is 1.

# Using hashes for keeping count

```
my $seq = "ATGGGGCGTATGCAATT";  
my @nucs = split "", $seq;  
print "@nucs\n";  
#A T G G G C G T A T G C A A T T
```

```
my %nt_count;  
foreach my $nt (@nucs) {  
    $nt_count{$nt}++;  
}
```

```
foreach my $nt (keys %nt_count) {  
    my $count = $nt_count{$nt};  
    print "$nt\t$count\n";  
}
```

A	4
T	5
C	2
G	5

# Parse a tab-delimited file and store the file columns in a hash.

```
my $file = shift @ARGV;
open (INFILE, '<', $file)
    or die "can't open file $file $!\n";
my %hash;
while (my $line = <INFILE>) {
    chomp $line;
    my ($key, $value) = split /\t/, $line;
    $hash{$key} = $value;
}
foreach my $key (sort keys %hash) {
    my $value = $hash{$key};
    print "key:$key value:$value\n";
}
```

## input:

genes.txt (tab delimited file):

```
geneA\tATGC
geneB\tTGCA
geneC\tAACT
```

## output:

STDOUT (on the screen):

```
key:geneA value:ATGC
key:geneB value:TGCA
key:geneC value:AACT
```

# Regular Expressions

Regular expressions is a language you can use within perl to identify patterns in text.

A regular expression is a string template, or pattern, against which you can match a piece of text. They are something like shell wildcard expressions, but **much** more powerful.

[Preview of Regular Expressions](#)

[Atoms](#)

[Quantifiers](#)

[Binding Operator](#)

[Variable Pattern](#)

[Alternatives and Grouping](#)

[Subpatterns](#)

[Using Subpatterns Inside the Regular Expression Match](#)

[Using Subpatterns Outside the Regular Expression Match](#)

[Extracting Subpatterns Using Arrays](#)

[Subpatterns and Greediness](#)

[String Substitution](#)

[Translating Character Ranges](#)

[Regular Expression Options](#)

[Global Matches](#)

## Preview of Regular Expressions

---

### A brief introduction of what a regular expression can do.

This bit of code loops through each line of a file. Finds all lines containing an EcoRI site, and bumps up a counter:

Code:

```
#!/usr/bin/perl -w
#file: EcoRI1.pl

use strict;

my $filename = "example.fasta";
open (FASTA , '<' , $filename ) or print "$filename does not exist\n";
my $sites;

while (my $line = <FASTA> ) {
    chomp $line;

    if ($line =~ /GAATTC/){
        print "Found an EcoRI site!\n";
        $sites++;
    }
}

if ($sites){
    print "$sites EcoRI sites total\n";
}else{
    print "No EcoRI sites were found\n";
}

#note: if $sites is declared inside while loop you would not be able to
#print it outside the loop
```

Output:

```
~]$ ./EcoRI1.pl
Found an EcoRI site!
Found an EcoRI site!
.
.
.
Found an EcoRI site!
Found an EcoRI site!
34 EcoRI sites total
```

**This does the same thing, but counts one type of methylation site (Pu-C-X-G) instead.**

(Pu-C-X-G) Methylation Site:

```
/[GA]C. ?G/
```

G or an A

[GA]

followed by a C

C

followed by one of anything, but could be nothing

.?

followed by a G

G

Code:

```
#file:methy.pl
while (my $line = <FASTA>) {
    chomp $line;

    if ($line =~ /[GA]C.?[G/]){
        $sites++;
    }
}
if ($sites){
    print "$sites Methylation Sites total\n";
}else{
    print "No Methylation Sites were found\n";
}
```

Output:

```
~]$ ./methy.pl
723 Methylation Sites total
```

## Atoms

### Atoms are the content, or the bits and peices of a regular expression

A regular expression is normally delimited by two slashes ("/"). Everything between the slashes is a pattern to match. Patterns can be made up of the following **Atoms**:

1. Ordinary characters: a-z, A-Z, 0-9 and some punctuation. These match themselves.
2. The "." character, which matches everything **except the newline**.

3. A bracket list of characters, such as [AaGgCcTtNn], [A-F0-9], or [^A-Z] (the last means anything BUT A-Z).

4. Certain predefined character sets:

**\d**

The digits [0-9]

**\w**

A word character [A-Za-z\_0-9]

**\s**

White space [ \t\n\r]

**\D**

A non-digit

**\W**

A non-word

**\S**

Non-whitespace

5. Anchors:

**^**

Matches the beginning of the string

**\$**

Matches the end of the string

**\b**

Matches a word boundary (between a \w and a \W)

Examples:

- `/g..t/` matches "gaat", "goat", and "gotta get a goat" (twice)
- `/g[ga@c][ga@c]t/` matches "gaat", "gttt", "gatt", and "gotta get an agatt" (once)
- `/\d\d\d-\d\d\d\d/` matches 376-8380, and 5128-8181, but not 055-98-2818.
- `/^\d\d\d-\d\d\d\d/` matches 376-8380 and 376-83801, but not 5128-8181.
- `/^\d\d\d-\d\d\d\d$/` only matches telephone numbers
- `/\bcat/` matches "cat", "catsup" and "more catsup please" but not "scat".
- `/\bcat\b/` only text containing the word "cat".

## Quantifiers

---

Quantifiers quantify how many atoms are to be found.



By default, an atom matches once. This can be modified by following the atom with a quantifier:

**?**

atom matches zero or exactly once

**\***

atom matches zero or more times

**+**

atom matches one or more times

**{3}**

atom matches exactly three times

**{2,4}**

atom matches between two and four times, inclusive

**{4,}**

atom matches at least four times

Examples:

- `/goa?t/` matches "goat" and "got". Also any text that contains these words.
- `/g.+t/` matches "goat", "goot", and "grant", among others.
- `/g.*t/` matches "gt", "goat", "goot", and "grant", among others.
- `/^\d{3}-\d{4}$/` matches US telephone numbers (no extra text allowed).

## Quick Exercises:

1. Design a pattern to recognize an email address.
2. Design a pattern to recognize the id portion of a sequence in a FASTA file

```
>SEQ1
ATGCTGCGCGTGCATGATGCT
>SEQ2
CGCGTGCATGATGCTGCGCGT
```

## Binding operator

---

### Specifying the String to Match

The Binding operator (`=~`) is used to "bind" the string to be searched and the pattern.

```
$h = "Who's afraid of Virginia Woolf?";
$h =~ /Woo?lf/;
```

The one line version of the 'if statement' can be combined with a regular expression:

```
$h = "Who's afraid of Virginia Woolf?";  
print "I'm afraid!\n" if $h =~ /Woo?lf/;
```

There's also an equivalent "not match" operator `!~`, which reverses the sense of the match:

```
$h = "Who's afraid of Virginia Woolf?";  
print "I'm not afraid!\n" if $h !~ /Woo?lf/;
```

## Quick Exercises:

1. What happens if you do not use the `~` symbol in your pattern match?
2. Create a script with a regular expression within an if-statement.
3. Design the regular express to match an entire sentence, up to the ending period in the provided string.

```
my $str = "This is a paragraph. A Paragraph is usually made up of more than one  
sentence.";
```

4. Modify your regular expression to take '.', '?', and '!'; into account as ending punctuation.

## Variable Patterns

### Matching with a pattern stored in a variable

You can use a scalar variable for all or part of a regular expression. For example:

```
$pattern = '/usr/local';  
print "matches" if $file =~ /^$pattern/;
```

## Quick Exercises:

1. Save your pattern from the last exercise in a variable. Replace the pattern portion of the regular expression with the variable.
2. Does your match still work?

## Alternatives and Grouping

### Sometimes you want to match "this" or "that" with your pattern

A set of alternative patterns can be specified with the pipe `|` symbol:

This pattern matches "wolf" or "sheep"

```
/wolf|sheep/;
```

## Use parenthesis to group the alternative patterns:

This pattern matches "big bad wolf" or "big bad sheep"

```
/big bad (wolf|sheep)/;
```

## Quick Exercises

1. Create a regular expression that will match a string with this pattern:
  - ATG followed by a C or a T.
2. Test your regular expression with these two strings:
  - GCTGATGCGTTA
  - GCTATGGCT

## Subpatterns

**Sometimes a subset of the pattern is important and you would like to save it for later use.**

You can extract and manipulate subpatterns in regular expressions.

To designate a subpattern, surround its part of the pattern with parenthesis (same as with the grouping operator).

This example has just one subpattern, `(.+)` :

```
/Who's afraid of the big bad w(.+)f/
```

You can combine parenthesis and quantifiers to quantify entire subpatterns:

```
/Who's afraid of the big (bad )?wolf\?/;
```

This matches "Who's afraid of the big bad wolf?" as well as "Who's afraid of the big wolf?"

This also shows how to literally match the special characters, **put a backslash** `\` **in front of them.**

## Quick Exercises

1. A FASTA has the following format:  
>ID Optional Description

SEQUENCE

SEQUENCE

SEQUENCE

2. Create a regular expression that captures the sequence ID as a subpattern.

## Using Subpatterns Inside the Regular Expression Match

---

**This is helpful when you want to find a subpattern and then match the contents again**

Once a subpattern matches, you can refer to it within the same regular expression. The first subpattern becomes `\1`, the second `\2`, the third `\3`, and so on.

```
while ($line = <FILE>) {  
    chomp $line;  
    print "I'm scared!\n" if /Who's afraid of the big bad w(.)\1f/;  
}
```

This loop will print "I'm scared!" for the following matching lines:

- Who's afraid of the big bad woof
- Who's afraid of the big bad weef
- Who's afraid of the big bad waaf

but not

- Who's afraid of the big bad wolf
- Who's afraid of the big bad wife

In a similar vein, `/\b(\w+)s love \1 food\b/` will match "dogs love dog food", but not "dogs love monkey food".

## Using Subpatterns Outside the Regular Expression Match

---

**Using the captured subpattern in code that follows the regular expression.**

Outside the regular expression match statement, the matched subpatterns (if any) can be found in the variables `$1`, `$2`, `$3`, and so forth.

Example. Extract 50 base pairs upstream and 25 base pairs downstream of the TATTAT consensus transcription start site:

```
while (my $line = <FILE>) {
    chomp $line;
    next unless $line =~ /(.{50})TATTAT(.{25})/;
    my $upstream = $1;
    my $downstream = $2;
}
```

## Quick Exercises

1. A FASTA header has the following format:  
 >ID Optional Description  
 SEQUENCE  
 SEQUENCE  
 SEQUENCE
2. Create a regular expression that captures the sequence ID as a subpattern.
3. If an ID is found, print it.

## Extracting Subpatterns Using Arrays

### Storing all the captured subpatterns in an array

If you assign a regular expression match to an **array**, it will return a list of all the subpatterns that matched.

Alternative implementation of previous example:

```
while ($line = <FILE>) {
    chomp $line;
    my ($upstream,$downstream) = $line =~ /(.{50})TATTAT(.{25})/;
}
```

Notice the `=` to the left of the string being searched. The subpatterns are returned and are assigned to the list of variables. This is another way to write it

```
my @nts = $line =~ /(.{50})TATTAT(.{25})/;
```

@nts will contain two values:

```
$nts[0] will have the upstream sequence
$nts[1] will have the downstream sequence
```

If the regular expression doesn't match at all, then it returns an empty list. Since an empty list is FALSE, you can use it in a logical test:

```
while (my $line = <FILE>) {
  chomp $line;
  next unless my($upstream,$downstream) = $line =~ /(.{50})TATTAT(.{25})/;
  print "upstream = $upstream\n";
  print "downstream = $downstream\n";
}
```

## Subpatterns and Greediness

By default, regular expressions are "greedy". They try to match as much as they can. For example:

```
$h = 'The fox ate my box of doughnuts';
$h =~ /(f.+x)/;
$subpattern = $1;
```

Because of the greediness of the match, **\$subpattern** will contain "fox ate my box" rather than just "fox".

To match the minimum number of times, put a `?` after the qualifier, like this:

```
$h = 'The fox ate my box of doughnuts';
$h =~ /(f.+?x)/;
$subpattern = $1;
```

Now **\$subpattern** will contain "fox". This is called *lazy* matching.

Lazy matching works with any quantifier, such as `+`, `*` and `{2,50}`.

## String Substitution

String substitution allows you to replace a pattern or character range with another one using the **s///** and **tr///** functions.

### The s/// Function

**s///** has two parts: the regular expression and the string to replace it with: *s/ expression/ replacement/*.

```
$h = "Who's afraid of the big bad wolf?";
$i = "He had a wife.";

$h =~ s/w.+f/goat/; # yields "Who's afraid of the big bad goat?"
$i =~ s/w.+f/goat/; # yields "He had a goate."
```

If you extract pattern matches, you can use them in the replacement part of the substitution:

Remember, if using the saved subpattern *outside* the pattern, or expression, use **\$1**, **\$2**, etc.

```
$h = "Who's afraid of the big bad wolf?";  
  
$h =~ s/(\w+) (\w+) wolf/$2 $1 wolf/;  
# yields "Who's afraid of the bad big wolf?"
```

## Quick Exercises

1. Create a regular expression with the `s///` function to find all ATG and replace with '-M-'.

```
GCAGAGGTGATGGACTCCG  
TAATGGCCAAATGACACGT
```

## Using a Variable in the Substitution Part

Yes you can:

```
$h = "Who's afraid of the big bad wolf?";  
$animal = 'hyena';  
$h =~ s/(\w+) (\w+) wolf/$2 $1 $animal/;  
# yields "Who's afraid of the bad big hyena?"
```

## Translating Character Ranges

The `tr///` function allows you to translate one set of characters into another. Specify the source set in the first part of the function, and the destination set in the second part:

```
$h = "Who's afraid of the big bad wolf?";  
$h =~ tr/ao/AO/; # yields "WhO's AfrAid Of the big bAd wOlF?";
```

`tr///` returns the number of characters transformed, which is sometimes handy for counting the number of a particular character without actually changing the string.

This example counts N's in a series of DNA sequences:

Code:

```
while (my $line = <FILE>) {  
    chomp $line; # assume one sequence per line  
    my $count = $line =~ tr/Nn/Nn/;  
    print "Sequence $line contains $count Ns\n";  
}
```

Output:

```
(~) 50% count_Ns.pl sequence_list.txt
Sequence 1 contains 0 Ns
Sequence 2 contains 3 Ns
Sequence 3 contains 1 Ns
Sequence 4 contains 0 Ns
...
```

## Regular Expression Options

---

Regular expression matches and substitutions have a whole set of options which you can toggle on by appending one or more of the **i**, **m**, **s**, **g**, **e** or **x** modifiers to the end of the operation.

See Programming Perl Page 153 for more information. Some example:

```
$string = 'Big Bad WOLF!';
print "There's a wolf in the closet!" if $string =~ /wolf/i;
# i is used for a case insensitive match
```

### **i**

Case insensitive match.

### **g**

Global match (see below).

### **e**

Evaluate right side of `s///` as an expression.

### **m**

Treat string as multiple lines. `^` and `$` will match at start and end of internal lines, as well as at beginning and end of whole string. Use `\A` and `\Z` to match beginning and end of whole string when this is turned on.

### **s**

Treat string as a single line. `"."` will match any character at all, including newline.

### **o**

Defining that a variable used as a pattern will never change, so perl will not attempt to interpolate the variable.

## Global Matches

---

Adding the **g** modifier to the pattern causes the match to be global. Called in a scalar context (such as an **if** or **while** statement), it will match as many times as it can.

This will match all codons in a DNA sequence, printing them out on separate lines:



Code:

```
$sequence = 'GTTGCCTGAAATGGCGGAACCTTGAA';
while ( $sequence =~ /(.{3})/g ) {
    print $1, "\n";
}
```

Output:

```
GTT
GCC
TGA
AAT
GGC
GGA
ACC
TTG
```

If you perform a global match in a **list** context (e.g. assign its result to an array), then you get a list of all the subpatterns that matched from left to right. This code fragment gets arrays of codons in three reading frames:

```
@frame1 = $sequence =~ /(.{3})/g;
@frame2 = substr($sequence,1) =~ /(.{3})/g;
@frame3 = substr($sequence,2) =~ /(.{3})/g;
```

The position of the most recent match can be determined by using the **pos** function. The pos function returns the position where the next attempt begins. Remember that pos will return in 0-base notation, the first position is 0 not 1. Code:

```
#file:pos.pl
my $seq = "XXGGATCCXX";

if ( $seq =~ /(GGATCC)/gi ){
    my $pos = pos($seq);
    print "Our Sequence: $seq\n";
    print '$pos = ', "1st position after the match: $pos\n";
    print '$pos - length($1) = 1st position of the match: ', ($pos-length($1)), "\n";
    print '($pos - length($1))-1 = 1st position before the the match: ',
        ($pos-length($1)-1), "\n";
}
```

Output:

```
~]$ ./pos.pl
Our Sequence: XXGGATCCXX
$pos = 1st postion after the match: 8
$pos - length(GGATCC) = 1st postion of the match: 2
($pos - length(GGATCC))-1 = 1st postion before the the match: 1
```

## Quick Exercises

1. Create a regular expression that matches ATG.
2. Print "Found ATG" when an ATG is found.
3. Did you use a while loop?
  - If not, try it with a while loop. How many ATGs do you find?
  - If so, try it without a while loop. How many ATGs do you find?
4. Did you use the 'g' modifier? What happens in your while loop with it and without it?
5. Print the position of each of each of the ATGs.

SEQUENCE:

TAATGGCCAAATGACACGTGCAGAGGTGATGGACTCCG

# Scope

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

```
use strict;  
use warnings;
```

```
my $x = 100;  
my $y = 20;
```

```
if ($x > $y) {  
    my $z = 10;  
    $x = 30;  
    print "x (inside if block): $x\n";  
    print "y (inside if block): $y\n";  
    print "z (inside if block): $z\n";  
}
```

```
print "x (outside if block): $x\n";  
print "y (outside if block): $y\n";  
print "z (outside if block): $z\n";
```

Global symbol "\$z" requires explicit package name at ./scope.pl line 19.

Execution of ./scope.pl aborted due to compilation errors.

# Blocks

That's because `$z` was declared inside the if block, so it's only accessible inside that block.

Any time we see `{ }`, we're creating a block.

Blocks are like boxes that have one way mirrors – you can see outside the box from inside, but not inside the box from the outside.

To fix that error, we need to declare `$z` outside the if block.

# Blocks

Variables declared inside of a block using “my” only exist inside the block – once the block is finished, they will be destroyed.

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

```
use strict;  
use warnings;
```

```
my $x = 100;  
my $y = 20;  
my $z = 5;
```

```
if ($x > 10) {  
    my $z = 10;  
    $x = 30;  
    print "x (inside if block): $x\n";  
    print "y (inside if block): $y\n";  
    print "z (inside if block): $z\n";  
}
```

```
print "x (outside if block): $x\n";  
print "y (outside if block): $y\n";  
print "z (outside if block): $z\n";
```

Output:

```
$x (inside of block):30  
$z (inside of block):10  
$x (outside if block): 30  
$z (outside if block): 5
```

# Scope

Does the program give the expected behavior?

By declaring “`my $z = 10;`” inside the if block, we’re creating a new variable called `$z` only accessible within the block.

This new variable will not modify the outside variable!

Note that we can create a new `$z` variable inside the block with no problems – if we do it outside, we’ll get a warning.

# Scope

- If we remove “my” from that line, the modification to \$z will show outside the block.

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

```
use strict;  
use warnings;
```

```
my $x = 100;  
my $z = 5;
```

```
if ($x > 10) {  
    $z = 10;  
    $x = 30;  
    print "x (inside if block): $x\n";  
    print "z (inside if block): $z\n";  
}
```

```
print "x (outside if block): $x\n";  
print "z (outside if block): $z\n";
```

Output:

```
$x (inside if block): 30  
$z (inside if block): 10  
$x (outside if block): 30  
$z (outside if block): 10
```

# References and multidimensional data

Simon Prochnik,  
with input from Dave Messina, Lincoln Stein, Steve Rozen

## Why do we need references?

Sometimes you need a more complex data structure than a scalar or a list.

What if you want to work with several related pieces of information? How would you represent this data in perl?

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

# Working with related data

To represent the table of data below, you could imagine working with 4 arrays (one for each column of data)  
@gene, @seq, @organism, @expression  
or perhaps 3 hashes (with a common key of the gene name)  
%sequence, %organism, %expression

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

## Representing tables of data in perl

Perl lets you work with multidimensional arrays and hashes very easily. You just string keys or indices together.  
For data with named columns, hashes are the most natural way to work

```
my %gene;  
$gene{HOXB2}{sequence} = 'ATCAGCAATATTT';  
$gene{HOXB2}{org} = 'mouse';  
$gene{HOXB2}{expr} = 45.33;  
$gene{HDAC1}{sequence} = 'GAGCGGAGCCGGGC';  
$gene{HDAC1}{org} = 'human';  
$gene{HDAC1}{expr} = 8.91;
```

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91



## Two-dimensional arrays

You could also represent this table with a two-dimensional array. The first index will be the row number, the second index will be the column number (both starting with 0). It would look like this

```
my @data;  
$data[0][0] = 'HOXB2';  
$data[0][1] = 'ATCAGCAATATTT';  
$data[0][2] = 'mouse';  
$data[0][3] = 45.33;  
$data[1][0] = 'HDAC1';  
$data[1][1] = 'GAGCGGAGCCGCGG';  
$data[1][2] = 'human';  
$data[1][3] = '8.91';
```

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

## More complex data structures

You can use more dimensions

```
$data[0][3][45] = 'human';  
$expression{human}{BRCA1}{liver} = 45.98;  
and you can mix and match hashes and arrays  
$assay{HDAC1}[0]{human}{liver}[3] = 62.95;
```

# How does perl store two-dimensional data?

You can only store one item of data in a scalar or an element of a hash or an element of array. So to make a two-dimensional array, perl stores a **reference** to an array in an element of the first array.

The debugger can help you explore and understand this

```
DB<100>$data[0][0] = 'HOXB2'
```

```
DB<101> p $data[0][0]
```

```
HOXB2
```

```
DB<102> p $data[0]
```

```
ARRAY(0x7fd02a245490)
```

↪ This is how perl displays a **reference** to an array

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GCGGACGGAC	human	8.91

## What is a reference?

Well first, what is a variable?

A variable is a labeled memory address that holds a value.

The location's label is the name of the variable.

0x84048ec

hexadecimal  
memory  
location

`$x=1;` *really means* SCALAR `x:`

1
---

`@y = (1, 'a', 23);`

*really means*

0x82056b4

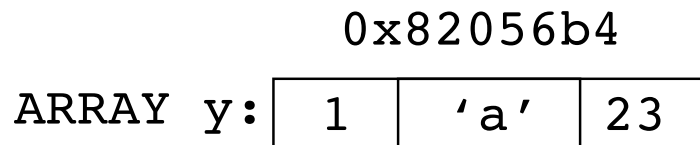
ARRAY `y:`

1	'a'	23
---	-----	----

# How is a reference different from a variable?

A variable is a labeled memory address.

When we read the contents of the variable, we are reading the contents of the memory address.



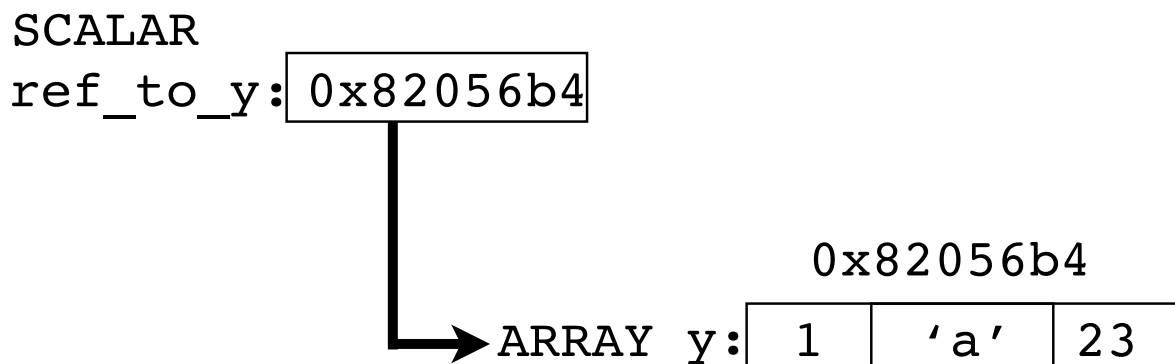
In contrast, a **reference** contains the memory address where some data is stored; it does not contain the data itself.

## Creating references yourself

Because a reference is a scalar, it is useful to create a reference to a more complicated data structure if you want to pass it to a subroutine.

We can create a reference to named variable `@y`. We use a backslash character `\` to say 'a reference to' like this:

```
my $ref_to_y = \@y;
```



# What's stored in a reference

SCALAR ref\_to\_y: 0x82056b4

If we print out \$ref\_to\_y, we see the raw hexadecimal memory address where the array @y is stored:

```
print $ref_to_y, "\n";  
ARRAY(0x82056b4)
```

## Dereferencing = the opposite of making a reference

You can create references to scalars, arrays and hashes

```
# create some references  
my $scalar_ref = \$count;  
my $array_ref  = \@array;  
my $hash_ref   = \%hash;
```

To dereference a reference, place the appropriate symbol (\$ for scalar references, @ for array references, % for hash references) in front of the reference.

This makes a new scalar, array or hash that is a copy of the one the reference pointed to.

```
# dereference your references:  
my $count = ${$scalar_ref};  
my @new_array = @{$array_ref};  
my %new_hash = %{$hash_ref};
```

A reference is a pointer to the data. It isn't a copy of the data.

When you make a reference to a variable, you have only created another way to get at the data.

There is still only one copy of the data.

```
my @y          = (1, 'a', 23);
my $ref_to_y    = \@y;
print join ' ', @{$ref_to_y};
1 a 23
```

```
push @{$ref_to_y}, 'new1', 'new2';

print join ' ', @y;
1 a 23 new1 new2
```

This is in contrast to assigning a variable to be equal to another, which creates a new data structure in a new memory location.

```
my @y = (1, 'a', 23);
my @z = @y;           # copy @y into @a
push @y, 'new1', 'new2'; # add to @y only

print join ' ', @y;
1 a 23 new1 new2

print join ' ', @z;
1 a 23
```

If you have a reference to an array or a hash, you can access any element.

```
my $value = $y[2];
```

*directly access the 3rd element in @y*

```
$value = ${$ref_to_y}[2];
```

*dereference the reference, then access the 3rd element in @y*

```
${$ref_to_y}[2] = 'new';  
print join ' ', @y;  
1 a new
```

*change the value of the 3rd element in @y*

```
my %z = ('dog' => 'animal',  
        'potato' => 'vegetable',  
        'quartz' => 'mineral',  
        'tomato' => 'vegetable');
```

```
my $ref_to_z = \%z;
```

```
my $value = $z{'dog'};
```

*directly access the value associated with the key 'dog' in the hash %z*

```
$value = ${$ref_to_z}{'dog'};
```

*dereference the reference, then get the value associated with the key 'dog' in the hash %z*

```
${$ref_to_z}{'tomato'} = 'fruit';  
print join ' ', values %z;  
animal vegetable mineral fruit
```

*change the value associated with the key 'tomato' in the hash %z*

# Anonymous Hashes and Arrays

You will not usually make references to existing variables. Instead you will create anonymous hashes and arrays. These have a memory location, but no symbol or name, i.e. you can't write `@my_data`. The reference is the only way to address them.

To create an anonymous array use the form:

```
my $array_ref = ['item1', 'item2'...];
```

To create an anonymous hash, use the form:

```
my $hash_ref = {key1=>'value1', key2=>'value2', ...};
```

```
my $y_gene_families = ['DAZ', 'TSPY', 'RBM', 'CDY1',  
  'CDY2' ];
```

```
$y_gene_family_counts = { 'DAZ'   => 4,  
                          'TSPY' => 20,  
                          'RBM'   => 10,  
                          'CDY2' => 2 };
```

```
my $third_item_of_array = $y_gene_families->[2];  
my $daz_count           = $y_gene_family_counts->{'DAZ'};
```

`$y_gene_families` gets (i.e. is assigned) a reference to an array, and  
`$y_gene_family_counts` gets a reference to a hash.

# Making a Hash of Hashes

The beauty of anonymous arrays and hashes is that you can nest them:

```
my %y_gene_data = ( 'DAZ' => {'family_size' => 4,  
    'description' => 'deleted in azoospermia' },  
    'TSPY' => {'family_size' => 20,  
    'description' => 'testis specific protein Y-linked'  
},  
    'RBMV' => {'family_size' => 10,  
    'description' => 'RNA-binding motif Y'},  
    'CDY2' => {'family_size' => 2,  
    'description' => 'chromodomain protein, Y-linked' }  
);  
  
# what is the size of the RBMY family?  
my $size = $y_gene_data{'RBMV'}{'family_size'};  
  
# what is the description of TSPY?  
my $desc = $y_gene_data{'TSPY'}{'description'};
```

# Making an Array of Arrays

```
my @spotarray = (  
    [0.124, 43.2, 0.102, 80.4],  
    [0.113, 60.7, 0.091, 22.6],  
    [0.084, 112.2, 0.144, 35.3]  
);  
  
print $spotarray[1][0];  
0.113  
  
my $cell_1_0 = $spotarray[1][0];  
print $cell_1_0;  
0.113
```



# Examining References

Inside a Perl script, the `ref` function tells you what kind of value a reference points to:

```
<DB> print ref($y_gene_data), "\n";  
HASH
```

```
<DB> print ref($spotarray), "\n";  
ARRAY
```

```
<DB> $x = 1;  
<DB> print ref($x), "\n";  
(empty string)
```

## Examining complex data structures in the debugger

Inside the Perl debugger, the "x" command will print the contents of a complex reference nicely formatted like so:

```
DB<3> x $y_gene_data  
0  HASH(0x8404bb0)  
    'CDY2' => HASH(0x8404b80)  
        'description' => 'chromodomain protein, Y-linked'  
        'family_size' => 2  
    'DAZ' => HASH(0x84047fc)  
        'description' => 'deleted in azoospermia'  
        'family_size' => 4  
    'RBMV' => HASH(0x8404b50)  
        'description' => 'RNA-binding motif Y'  
        'family_size' => 10  
    'TSPY' => HASH(0x8404b20)  
        'description' => 'testis specific protein Y-linked'  
        'family_size' => 20
```

# Scripting Example: Creating a Hash of Hashes

We are presented with a table of sequences in the following format: the ID of the sequence, followed by a tab, followed by the sequence itself.

```
2L52.1      atgtcaatggtaagaaatgtatcaaatacagagcgaaaaattggaagtaag...
4R79.2      tcaaatacagcaccagctccttttttatagttcgaattaatgtccaact...
AC3.1       atggctcaaactttactatcacgtcatttcogtggtgtcaactgttattt...
...
```

For each sequence calculate the length of the sequence and the count for each nucleotide. Store the results into hash of hashes in which the outer hash's key is the ID of the sequence, and the inner hashes' keys are the names and counts of each nucleotide.

```
#!/usr/bin/perl
use warnings;
use strict;

# tabulate nucleotide counts, store into %sequences

my %seqs; # initialize hash
while (my $line = <>) {
    chomp $line;
    my ($id,$sequence) = split "\t",$line;
    my @nucleotides    = split '', $sequence; # array of base pairs
    foreach my $n (@nucleotides) {
        $seqs{$id}{$n}++; # count nucleotides and keep tally
    }
}

# print table of results
print join("\t",'id','a','c','g','t'),"\n";

foreach my $id (sort keys %seqs) {
    print join("\t",$id,
                $seqs{$id}{a},
                $seqs{$id}{c},
                $seqs{$id}{g},
                $seqs{$id}{t},
                ),"\n";
}
```

The output will look something like this:

id	a	c	g	t
2L52.1	23	4	12	11
4R79.2	15	12	5	18
AC3.1	11	11	8	20
...				

## When do you really need references? For passing complex data to subroutines

```
#!/usr/bin/perl
use warnings;
use strict;

my @scores = (1,2,3,4);
my @students = qw(bob karen emily john);

# you can't use this next form. Why not?
#my $smartest_student=see_who_is_best(@scores,@students); #WRONG

my $smartest = see_who_is_best(\@scores,\@students);
print "$smartest\n";

sub see_who_is_best {
    # this next line doesn't work
    # my (@scores,@students) = @_; # WRONG!
    # you have to use this
    # can you see why?
    my ($score_ref,$student_ref) = @_;
    my @scores = @{$score_ref};
    my @students = @{$student_ref};
    # some more code goes here
    #
    #
}
```