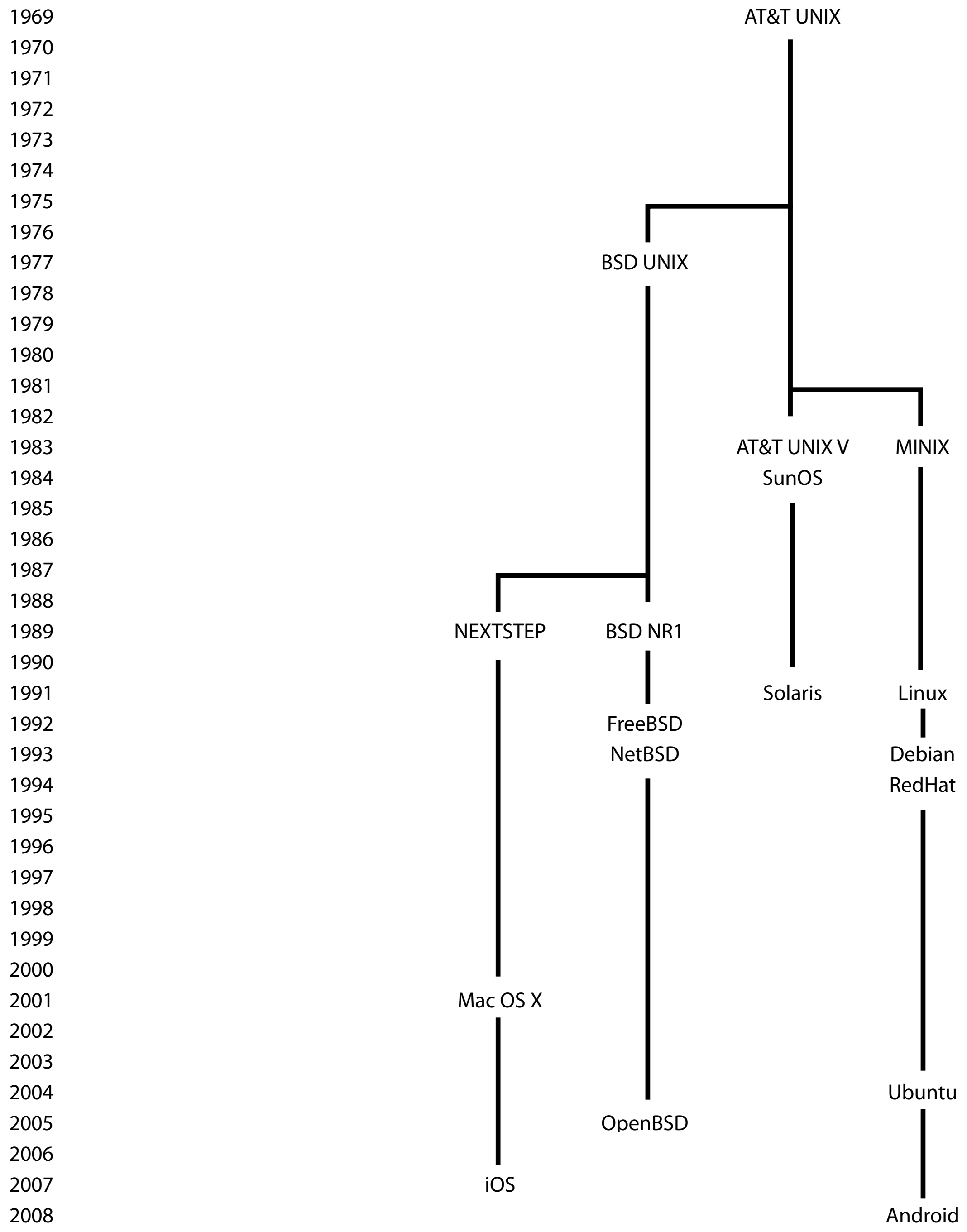
ICQB

Introduction to Computational & Quantitative Biology (G4120)
Spring 2017
Oliver Jovanovic, Ph.D.
Columbia University
Department of Microbiology & Immunology

Lecture 4: Unix and Scripting
February 27, 2017

Evolution of Unix



1969 AT&T UNIX

1970

1971

1972

1973

1974

1975

1976

1977 BSD UNIX

1978

1979

1980

1981

1982

1983 AT&T UNIX V MINIX

1984 SunOS

1985

1986

1987

1988

1989 NEXTSTEP BSD NR1

1990

1991 Solaris Linux

1992 FreeBSD

1993 NetBSD Debian

1994 RedHat

1995

1996

1997

1998

1999

2000

2001 Mac OS X

2002

2003

2004 Ubuntu

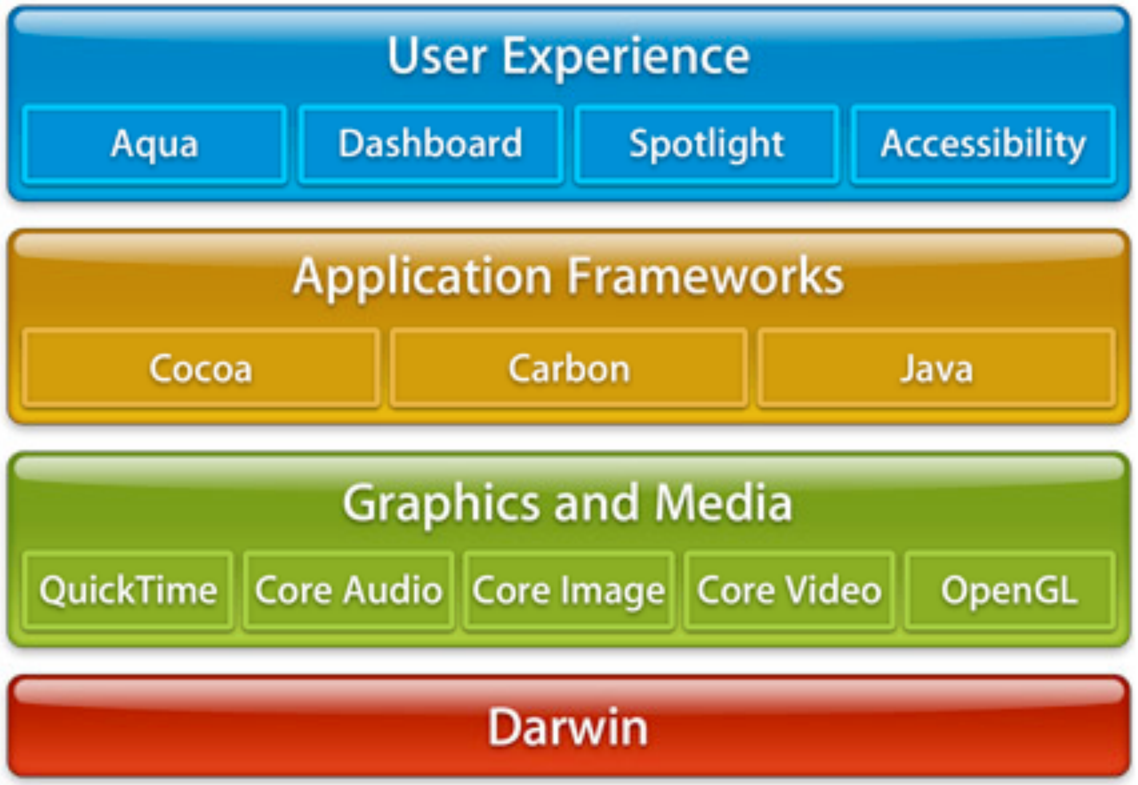
2005 OpenBSD

2006

2007 iOS

2008 Android

Macintosh OS X Architecture



User Experience
The layer with which most users interact with the Macintosh includes Aqua (the graphical user interface (GUI) of OS X), Dashboard (which manages and displays desktop widgets), Spotlight (which provides system wide search and indexing through the use of metadata) and Accessibility (assistive technology for the disabled).

Darwin
The Open Source Unix operating system that underlies OS X. It is derived from NeXTSTEP and BSD Unix, and is built around a XNU Mach3/BSD hybrid kernel and a I/O Kit device driver API.

Why Use Unix?

- Historically, Unix has been used as a free academic and research operating system (BSD, FreeBSD, etc.), and many forms of Unix are Open Source
- Unix is extremely stable
- Unix is very efficient
- Unix has powerful free scripting and automation tools (bash, tcsh, grep, sed, awk, rsync, etc.)
- Unix has excellent free programming tools (Perl, Python, Ruby, **bioperl.org, biopython.org, biojava.org,** etc.)
- All the bioinformatics tools needed to do complex analysis are available for free on Unix (BLAST, FASTA, CLUSTAL, PHYLIP, PHRED, PHRAP, CONSED, EMBOSS, etc.)
- New algorithms in computational biology are generally first implemented in Unix
- Unix is easy to program and network (HTML, HTTP, Apache, CGI, etc.)
- Many other useful programs originated on Unix, and the majority are available for free, complete with source code (Open Source licenses, **bioinformatics.org, open-bio.org,** etc.)

Unix Shells

Shells

The command line interface used to interact with Unix systems is known as a shell. A shell interprets and executes the commands you give it, and can also run text files called shell scripts,which allow for commands to be strung together, manipulated, and automated. A number of Unix shells exist, generally with slight variations in syntax and features:

tcsh
The TC shell, an enhanced version of the csh shell.

csh
The Berkley Unix C shell, from which tcsh is derived.

sh
The Bourne shell, the original Unix shell.

bash
The Bourne-again shell, a successor to sh. It is the default shell for OS X and cygwin.

Terminal

Terminal
Terminal is the application which gives an OS X or cygwin user command line shell access with which to directly interact with the underlying Unix operating system.

Terminal Startup
When Terminal starts up, it runs the **bash** shell by default. The **bash** shell is a program which begins by executing commands in the system file **/etc/profile**. It then looks for an optional list of commands to execute in a file in the home directory called **.bash_profile** (if not found, it will also look for files called **.bash_login** and **.profile**). It then looks for a file in the home directory called **.bash_history**, from which it loads a list of previously executed commands.

Terminal Preferences
In OS X, Terminal is located in the **/Applications/Utilities** folder, and its preferences can be adjusted in Terminal > Preferences. For cygwin, right click on a Terminal window and select Options… to adjust preferences. You can adjust the color of the text, the color and transparency of the Terminal window background, and other shell behavior.

In OS X you can activate **Use Option as Meta Key** under Preferences > Profiles > Keyboard, which means the cursor will go where you **click** while holding down the **option** key, instead of only allowing the cursor to move with the arrow keys. Cygwin offers an option for **Clicks place command line cursor** under Options… > Mouse.

Terminal Shortcuts

Saving Sessions
All the text in the Terminal window can be saved as a text file in OS X, or copied and pasted. This can be useful if you're carrying out a complex procedure you might want to repeat later, or to copy and paste commands from to cut down on typographical errors.

Multiple Windows
One can open and work in multiple Terminal windows. This is particularly useful if you are working in more than one directory, or have started a process that may take a while to complete.

Dragging to Reveal Pathnames
Dragging a folder or file onto the Terminal window enters its path. This is an extremely useful shortcut when using commands that require a pathname.

Prompt
By default, the Terminal prompt tells you the name of the computer, what directory you are currently in (~ for your home directory), and in OS X, what user you are logged in as.

exit
Type `exit` to logout of a Terminal session. You can adjust Terminal's Window Settings to automatically close upon a successful logout.

Unix Commands

Commands
A command in Unix consists of a **program** that is executed by typing its name. That program then generates output, by default to the Terminal window, based on the options and arguments it was provided with. Options follow the command, and arguments follow options, all separated by single spaces,
e.g. **`command -option(s) arguments(s)`**

Program
A command line program is executed by typing its name on the command line and pressing return, e.g. typing **`ls`** and pressing return will list files in the current directory. Hundreds of command programs are built-in.

Options
Options modify the behavior of a program. Traditionally, options are represented by a hyphen followed by a single letter. Adding the **`-l`** option to **`ls`**, i.e. **`ls -l`** results in long file listings. Multiple options can be used together by typing more than one letter after the hyphen.

Arguments
Arguments are the input the program acts upon. Typically, they are names of files or directories, but can be nearly anything, including the output of other programs. Unix wild card characters can be useful here, such as `*`, which stands for any group of characters, so **`ls *.txt`** would list only files with names ending in **.txt**.

Output
The result of a program is its output, which in OS X by default goes to the Terminal window, and can be copied and pasted. However, a program's output can be sent to a file, or even to another program.

Unix Command Line Tips

Unix Wild Card Characters

`*` an asterisk stands for any group of characters (including none)

`?` a question mark stands for any one character

`[]` square brackets can be used to wrap a choice of single characters, e.g. **`[Aa]`** or can be used to indicate a range of consecutive characters, e.g. **`[1-5]`**

Unix is Case Sensitive

Although OS X is not case sensitive, the underlying Unix operating system is case sensitive, and therefore one should try to use the correct case when typing from the command line, or feeding input to command line programs. Entering **`ls a*`** would list only files with names beginning with a lowercase **a**, and not files beginning with an uppercase **A**.

man (manual)

The **man** program provides documentation for nearly every installed Unix program. Simply type **`man`** ***`nameofprogram`*** to view the documentation for that program. While viewing the documentation, hold down the **down arrow** key or **return** key to reveal more of the documentation line by line, hit the **spacebar** to display the next page, or type **`q`** to quit. To view a one page summary of documentation for **`man`**, use the **`-h`** option, i.e. **`man -h`** (for more information about the **man** command, enter **`man man)`**.

Limits

Some Unix commands may be limited to handling a maximum of 256 files at once, and having command lines no longer than 2,048 characters.

Unix Shell Shortcuts

Pathname Tab Autocompletion
If you press **tab** after partially typing a pathname, the shell will attempt to complete it for you.

Repeating Commands
The **up arrow** key will cycle forwards through all the commands you typed recently (and the down arrow will go back), which can be considerably faster than typing, or even copying and pasting a previously entered command. One can also use the **back** and **forward arrow** and **delete** keys to modify a recalled previously entered command.

Typing two exclamation marks, i.e. `!!` will repeat the last command. Typing **`history`** will show a numbered list of previous commands, any of which can be executed by typing **`!`** ***`numberofcommand`***. Pressing **control** and **R**, then typing a few letters will do a reverse search for previous commands that start with those letters.

Multiple Commands
Multiple commands may be executed sequentially on the same line if they are separated by semicolons, e.g. **`cd;ls`**.

Break
Press **control** and **C** or **command** and **.** to stop any program. This is useful if you start getting more output than you anticipated. If you just want to pause output, press **control** and **S** to stop it, then press **control** and **Q** to start again.

Unix Pathnames

/ (root directory)
The Unix file system organizes files and directories in a hierarchical inverted tree structure. The root directory is the highest level directory in Unix, represented by a frontslash, e.g. with a default OS X installation, this corresponds to Macintosh HD. The **frontslash** key (/) is located next to the right **shift** key.

Absolute Pathnames
An absolute pathname explicitly tells which directories you must travel to get from the root to the directory or file you want, e.g. **`/Applications/Utilities/Console.app`**. Frontslashes are used to separate directory names in a pathname, and an absolute pathname ***always*** starts with a frontslash.

~ (home directory)
The home directory is your user directory, and its path is represented by the tilde character. Instead of typing **`/Users/`*`myuserdirectory`*`/Documents`**, you can type **`~/Documents`**. The tilde character (~) is located on the **backtick** key underneath the **esc** key.

Relative Pathnames
Relative pathnames give the location relative to your current directory. If you are currently in the OS X Applications directory, the relative pathname to Utilities below is simply **`Utilities`**. Two periods (**`..`**) in a relative pathname refer to the directory directly above the one that follows. In **`../Utilities`**, the two periods refer to the Applications directory above. A relative pathname ***never*** starts with a frontslash.

Unix Navigation Commands

ls (list)
Lists the current directory's files. Adding the **`-a`** option, i.e. **`ls -a`** lists all files, including invisible files (files whose names start with a period are normally invisible). The **`-l`** option, i.e. **`ls -l`** lists long information about files: type, permissions, links, owner, group, size, modification date & time and name. They can be used together, i.e. **`ls -la`**

The wild card character (`*`) is often useful in arguments for this command, e.g. **`ls *.doc`** lists all Word files in a directory, based on the **.doc** file name extension Word files should have.

cd (change directory)
By itself, **`cd`** takes you to your home directory. If you add a pathname argument to the command, e.g. **`cd /Applications`**, it will take you to that directory instead.

If a directory name in the path contains spaces, make sure to wrap it in double quotes, e.g. **`cd /Applications/"DNA Strider"`**. If a name already contains double quotes, wrap it in single quotes, and vice-versa.

Using an argument of two periods, i.e. **`cd ..`**, moves you to the directory directly above the current directory, while **`cd /`** moves you to the root directory. Again, one can drag a folder or application to the Terminal window to get its pathname, so it is often easier to type **`cd`** followed by a space and drag the folder you want to make your current directory onto the Terminal window than try to type the entire path.

pwd (print working directory)
Displays the pathname of the current directory by printing it to the screen. The first front slash in the pathname represents the root directory, with another front slash separating subsequent directories. The last directory listed is the current directory.

Unix File System Commands

cp (copy)
Copies files. The **`-R`** option, i.e. **`cp -R`** copies directories and their contents.

mkdir (make directory)
Makes a new directory (creates a new folder) with the provided name. Multiple directories can be made at once if the names are separated by a space.

mv (move)
Allows you to **rename** or **move** a file, e.g. **`mv oldname.txt newname.txt`**.

rm (remove)
Removes/deletes the specified file(s). ***This command should be used with great precision,*** particularly when combined with wild card characters, used in recursive mode or when used with **`sudo`.** Used carelessly it can delete nearly every file on a computer. The **`-r`** option, i.e. **`rm -r`** runs **`rm`** in **r**ecursive mode, where it will delete directories as well as files, including everything inside a specified directory. The **`-i`** option, i.e. **`rm -i`** runs **`rm`** in **i**nteractive mode, where it will ask if you really want to delete each file or directory before it is deleted, which can be a wise precaution.

rmdir (remove directory)
Removes/deletes a specified empty directory.

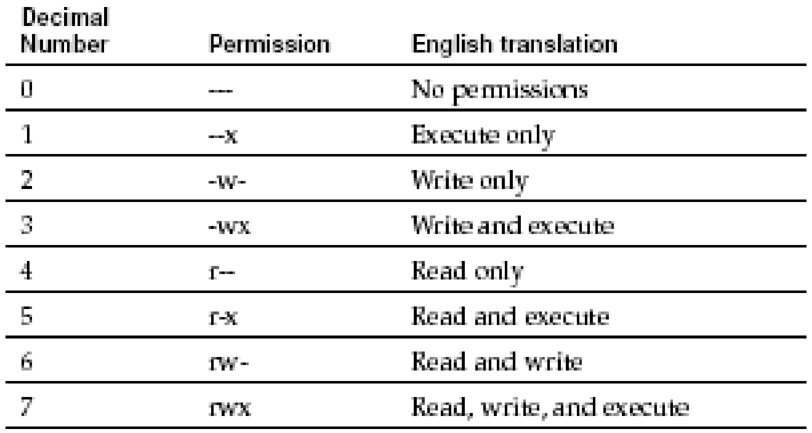
File Ownership and Permissions

File Ownership

A file is typically owned by the account that created it. There are three levels of file access: **owner, group** and **other**. Each can have different levels of access. A user can belong to more than one group, and a file can have more than one owner, with different access levels for each. The default group in OS X is **staff**. Users with administrative privileges also belong to the group **admin**. The **`chown`** command is used to change ownership permissions.

File Permissions

In Unix, file permissions are commonly noted in the order: **owner, group, other.** Thus, a file with permissions **755** can be read, written and executed by the owner, but only read and executed by members of the group or others. The **`chmod`** command is used to change file permissions.



Decimal Number	Permission	English translation
0	---	No permissions
1	--x	Execute only
2	-w-	Write only
3	-wx	Write and execute
4	r--	Read only
5	r-x	Read and execute
6	rw-	Read and write
7	rwx	Read, write, and execute

Ownership and Permission Commands

chown (change ownership)
Allows you to change the owner and/or group a file belongs to. The syntax is **owner:group,** i.e. **`chown simon file.txt`** will change the owner of **file.txt** to **simon**, while **`chown simon:developer file.txt`** will change the owner to **simon** and the group to **developer**. On occasion, **`sudo`** may have to be used with **`chown`**.

chmod
The command changes file or directory permissions, which can first be checked with **`ls -l`**. The simplest **`chmod`** syntax to use is the numerical syntax **owner/group/everyone**, e.g. **`chmod 660 file.txt`** to give the owner and group of **file.txt** read and write permissions (**6**), or **`chmod 777 test.sh`** to give full read, write and execute permissions (**7**) on **test.sh**.

Scripts must have execute permission (**7, 5** or **1**) to work. Commonly, **755** permissions are used for scripts and programs, e.g. **`chmod 755 stat.sh`**, to allow the owner full access (**7**) to **stat.sh**, and everyone else execute and read permissions (**5**). On occasion, **`sudo`** may have to be used with **`chmod`**.

sudo (superuser do)
Allows you to execute a single command as a root user, or superuser, who has no limitations. It can only be used from an account with administrative privileges. *It should always be used with care and precision.*

Other Unix File Commands

more
Reads a file and outputs its content to the screen a page at a time (press the **spacebar** to move to the next page, or press **q** to quit). A related utility called **less** allows for more control.

cat (concatenate)
Reads files and outputs their entire content (use **control** and **s** to pause then press **control** and **q** to restart).

find
Finds files starting in the specified directory, then recursively descending. A period (**`.`**) specifies starting in the current directory, and the **`-name`** option finds files with the specified name, e.g. **`find . -name *.pdf`** will find all files ending in **.pdf** in and below the current directory.

sort
Sort arranges lines of text alphabetically or numerically, as specified by its options. The option **`-r`** (**`sort -r`**) sorts in **r**everse order.

Other Useful Unix Commands

grep
Globally search a **r**egular **e**xpression and **p**rint. Searches text for lines matching a regular expression.

tr
Translate. Replaces specified characters in input with other specified characters in output.

sed
A simple **s**tream **ed**itor for parsing and transforming text.

awk
A simple programming language for searching and processing text files. Other good options are the Perl or Python programming languages.

vi
A text editor with a modal interface and keystroke commands. Another option is **emacs,** which uses key combinations. A simple option is **ed,** a line editor.

Unix Process Commands

Processes

When running, Unix programs run as processes, which can be displayed in OS X using the Activity Monitor utility (located in /Applications/Utilities), or directly from the Unix command line using **`top`**.

top

Displays all currently running processes, identifying each with a unique PID (process ID). COMMAND is the name of the program, and %CPU is the percentage of the processor that process is using. The option **`-u`**, i.e. **`top -u`** will list the top processes using the most processor power in order from first to last. Press **Q** or issue a break (**control** and **C** or **command** and .) to quit **`top`**.

kill

Kills a particular process when a PID is provided as an argument, e.g. **`kill 2323`**. The option **`-9`**, i.e. **`kill -9`** tells it to terminate a process with extreme prejudice, e.g. **`kill -9 2323`**. Use this command carefully, but it can sometimes be useful – for example, if an application crashes, it may leave a process running, which can slow the computer unless it is killed. On OS X is also possible to kill processes by quitting them with Activity Monitor.

Command Redirection

| (pipe)
Send output directly to the command line program that follows for it to use as input. This allows chaining a series of simple programs together to perform complex tasks. For example, **`ls | more`** sends output from the **`ls`** program to the **`more`** program, which displays the output only one screen at a time, while **`ls -l | grep 'Apr 16'`** (sends the output of the **`ls`** program to **`grep`**, which will result in a long information list of every file in the directory last modified on April 16th). The pipe character (|) is the vertical bar on the **backslash** key below the **delete** key.

`` ` ` `` (command substitution)
Enclosing a program command with backticks results in the output from that program being sent back to the command line for use an an argument by another program, e.g. **``ls -l `grep -l 'virus' *` ``** (takes any files **`grep`** finds with the word **virus** inside them and outputs them as a long information list). The **backtick** key is directly below the **esc** key.

Input and Output Redirection

< (input redirection)

Takes input from the file specified as an argument, rather than from standard input (normally, the keyboard).

> (output redirection)

Writes the output to the file name specified in an argument, either creating a new file, or overwriting an existing file with the same name, instead of sending it to standard output (on OS X, the Terminal window). **`ls > contents.txt`** will write a list of files in the current directory to a new file called **contents.txt**. ***Take care not to overwrite important files with the same name.***

>> (append output)

Appends the output to the file with the name specified in an argument. The command **`ls >> contents.txt`** will append a list of the files in the current directory to the end of an existing **contents.txt** file. The command **`cat body.txt >> contents.txt`** will concatenate the content of **body.txt** to the end of an existing **contents.txt** file.

Customizing the bash shell

.bash_profile
Once a **.bash_profile** file is added to your home directory, it is read by the Terminal on startup, and any commands you have entered into that text file are automatically executed, allowing you to significantly customize your shell environment. Use **mv** to rename a text file to **.bash_profile**, as OS X will not normally display or allow you to start a file name with a period, and make sure there is a return after the end of the text.

PATH
PATH allows you to add directories to the paths recognized by the shell so that you can execute commands by just typing the name of the command. **`PATH=$PATH .`** will allow you to execute a program located in your current directory without having to type **`./`** in front of the name. Multiple paths can be added, separated by a colon (**`:`**). Specifying particular directories that allow script execution is the safest approach to use.

alias
Alias lets you create a custom command or macro, e.g. **`alias dir "ls -la"`** would give a long format listing of all files when you typed **`dir`** at a Terminal prompt.

Sample .bash_profile

```
alias dir "ls -la"
alias desk "cd ~/Desktop"
PATH=$PATH:/Developer/Tools:~/bin
export PATH
```

Unix Shell Scripts

A shell script allows you to create a single new Unix command out of existing shell commands and programs. This is very useful for automating a process or running a series of complex commands. The shell script is saved in an executable text file, the first line of which must consist of a pound sign followed by a bang (together known as a shebang), then the full path to the shell in which the script runs. For the OS X **bash** shell, this is:

```
#!/bin/bash
```

The same procedure is used to execute programs as scripts by interpreted programming languages such as Python, except the full path to the programming language is specified instead.

hal.sh

```
#!/bin/bash
echo It can only be attributable to human error.
```

```
chmod 755 hal.sh
./hal.sh
```

Other Unix Scripts

Running Scripts

Running another kind of simple script, such as a Python script, simply involves downloading and uncompressing the script, making sure that the script has its permissions set to allow it to be executable, then running it. Some complex scripts may require that other scripts or libraries be installed before they can run.

Permissions

To check permissions on a script, use the command **`ls -l.`** If the permissions list for the script does not have an **`x`** in it, is is not executable (typically it will start **-rwx**).

Setting Permissions

To set the permissions of a script to be executable, use the program **`chmod`**. Issue the command **`chmod 700`** *filename* to give only the owner (yourself) permission to execute. To give any user the ability to execute a script (but not change it) issue the command **`chmod 755`** *filename.*

Run the Script Using Dot Slash Notation

You can execute a script in your current directory by using the command *./filename*. If you know the scripting language being used, you can call that directly, and pass a reference to the script name, e.g. issuing the command **`python`** *filename.py*. You can also add the directory to your path to allow you to execute scripts in that directory directly (e.g. **`PATH=$PATH:~/bin`**).

Running a Python Script

Running a Python Script

1) Open a new document in TextWrangler and type the following source code:

```
#!/usr/bin/python

print "Hello, world?"
```

2) Save the file in your home directory as **hello.py**

3) Try to run the script by typing **./hello.py** and pressing **return**

4) Make any necessary modifications

More Complex Shell Scripts

countfasta.sh

```
chmod 755 countfasta.sh
./countfasta.sh *.fasta
```

```
#!/bin/bash
for filename
do
     grep -cH '>' $filename
done
```

addfasta.sh

```
chmod 755 addfasta.sh
./addfasta.sh *.fasta
```

```
#!/bin/bash
total=0
for filename
do
     total=$(($total+`grep -ch '>' $filename`))
done
echo $total
```

Unix and Scripting Resources

Bash Guide for Beginners

A free guide to the bash Unix shell (in PDF format) is available at:
http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf

Books on Unix and Scripting

Mastering Regular Expressions by Jeffrey E. F. Friedl
Learning the UNIX Operating System by Grace Todino, John Strang & Jerry Peek
Think Unix by Jon Lasser
Learning Unix for Mac OS X by Dave Taylor
The Mac OS X Command Line by Ken McElhearn