Lecture 11
Introduction to Automation & Programming
December 11, 2007

G4120: Introduction to Computational Biology

Oliver Jovanovic, Ph.D.
Columbia University
Department of Microbiology

Copyright © 2007 Oliver Jovanovic, All Rights Reserved.
Evolution of Programming Languages

1956
1958
1960
1962
1964
1966
1968
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002

Smalltalk 80
Ruby
SML
Caml
OCaml
Perl
PL/I
Algol 60
Fortran I
Lisp
Common Lisp
SML
OCaml
Common Lisp
Scheme R4RS
Scheme
Java
Java 2 (v1.2)
C++ (ISO)
C#
C++ (ISO)
Pascal
Fortran 77
Ada 83
Eiffel
Python
Perl
Ada 83
Tcl

History of Programming Languages

1831  Lady Ada Lovelace writes the first computer program, for Charles Babbage's Analytical Engine.
1936  Alan Turning develops the theoretical concept of the Turing Machine, forming the basis of modern computer programming.
1945  John von Neumann develops the theoretical concepts of shared program technique and conditional control transfer.
1946  Plankalkul, the first formal computer language, is developed by Konrad Zuse, a German engineer, which he later applies to, among other things, chess.
1949  Short Code, the first computer language actually used on an electronic computer, appears.
1951  A-O, the first widely used compiler, is designed by Grace Hopper at Remington Rand.
1954  FORTRAN (FORmula TRANslating system) language is developed by John Backus at IBM for scientific computing.
1958  ALGOL, the first programming language with a formal grammar, is developed by John Backus for scientific applications.
1958  LISP (LISt Processing) language is created by John McCarthy of MIT for Artificial Intelligence (AI) research.
1959  COBOL is created by the Conference on Data Systems and Languages (CODASYL) for business programming, and becomes widely used with the support of Admiral Grace Hopper.
1964  BASIC (Beginner's All-purpose Symbolic Instruction Code) is created by John Kemeny and Thomas Kurtz as an introductory programming language.
1965  Structured programming is defined by Edsger Dijkstra.
1968  Pascal is created by Niklaus Wirth as a teaching language.
1972  PROLOG (Programming Logic) is developed as result of logic theorem research. It has become the most generally used logic programming language, often used in developing expert systems.
1972  Smalltalk, which becomes the first popular object oriented programming language, is developed at Xerox PARC by Alan Kay.
1972  
C is created by Dennis Ritchie at Bell Labs for programming the Unix operating system. It is fast, widely used, and forms the basis of many other current procedural languages.

1975  
Bill Gates and Paul Allen write the first version of Microsoft BASIC.

1978  
Awk, a text-processing language named after the designers, Aho, Weinberger, and Kernighan, appears. It is often used with the sed stream editor to manipulate text data in Unix.

1979  
SQL (Structured Query Language) is developed at IBM based on work to simplify access to data stored in a relational database. It has become the most widely used database language.

1982  
PostScript, a language for graphics printing and display, appears.

1983  
C++, an object-oriented version of the C programming language, appears, based on earlier work on “C with Classes”. It is often used for large projects that require speed.

1986  
Objective C is a Smalltalk influenced object-oriented version of C. It first became widely used as the development language for NeXTstep, and is currently the principle programming language for Mac OS X.

1987  
Perl (Practical Extraction and Reporting Language) is developed by Larry Wall after he finds Unix text utilities limiting. It has become popular as a jack-of-all trades language, and in computational biology applications.

1991  
Python, a simple functional and object oriented language, is developed by Guido Van Rossum. It is often used for rapid development, and is well suited for computational biology applications.

1991  
Visual Basic is developed by Alan Cooper and Microsoft to allow for easy visual creation of Windows applications.

1994  
PHP (Personal Home Page tools) was a simple web server scripting language developed by Rasmus Lerdorf that has developed into a useful language for rapidly developing web applications.

1995  
Java, a simplified version of C++, originally developed by Sun Microsystems to control consumer appliances, is repurposed for web development. It has become popular for writing cross-platform and web applications.

1995  
Ruby, a simple and elegant object oriented programming language, is developed by Yukihiro Matsumoto.
Macro
A single, user-defined command that executes a series of one or more commands.
Example: alias, Keyboard Shortcuts

Scripting Languages
A simple programming language that uses a syntax close to a natural language and sends commands to the operating system or other programs when executed.
Example: AppleScript, Automator, bash, JavaScript

Database Languages
A programming language tied closely to a database, allowing for easy queries and manipulation.
Example: SQL

Procedural Languages
A fully featured programming language in which variables can keep changing as the program runs. Most commonly used programming languages are procedural.
Examples: C, Perl

Logical Languages
These programming languages are collections of logical statements and questions.
Examples: Prolog, custom expert systems

Object Oriented Programming Languages
A programming language in which data and functions are encapsulated in objects. An object is a particular instance of a class. Each object can contain different data, but all objects belonging to a class have the same functions or methods. Objects can restrict or hide access to data within them.
Examples: C++, Objective C, Python, Java, Ruby
Automation

One of the tasks computers are best at is performing highly repetitive tasks at great speed with great accuracy. Learning a programming language such as C or Perl allows you to fully utilize the power of a computer, but becoming fluent in a programming language can take significant time and effort. With a fraction of that effort, however, it is possible to learn to use macros, regular expressions, simple scripting languages such as AppleScript, certain applications, Unix shell scripts and the Unix command line interface to automate repetitive tasks and save a great deal of time and effort.

Common Tasks that can be Automated

- Entering passwords (Keychain Access, ssh)
- Filling out forms (Safari or Internet Explorer AutoFill)
- Searching and reformatting files (BBEdit, grep, regular expressions, shell scripts)
- Tabulating data in files (BBEdit, grep, regular expressions)
- Renaming files (File Buddy, Renamer4Mac, AppleScript, shell scripts)
- Entering repetitive information (macros, AppleScript, shell scripts)
- Readback of repetitive information (AppleScript)
- Assembly of repetitive information (AppleScript, regular expressions, shell scripts)
- Repetitive tasks (cron, osascript, CronniX, shell scripts)
- Backups (Retrospect, Carbon Copy Cloner, AppleScript, rsync, shell scripts)
Automation Utilities

**Macros**
A macro is a single, user-defined command that executes a series of one or more commands.
- Typeit4me ([http://www.typeit4me.com/](http://www.typeit4me.com/))
- Keyboard Shortcuts (System Preferences > Keyboard & Mouse > Keyboard Shortcuts > +)
- alias (Unix shell command)

**Renaming Utilities**
- Renamer4Mac ([http://www.power4mac.com/renamer/](http://www.power4mac.com/renamer/))

**Backup Utilities**
- Retrospect ([http://www.dantz.com](http://www.dantz.com)) Commercial

**Text Processing Utilities**
- grep and regular expressions (Unix utilities)
- BBEdit ([http://www.barebones.com/](http://www.barebones.com/)) Commercial (TextWrangler is free)

**Scripting Languages**
A scripting language is a simple language, typically written in a series of English-like statements that, when executed, sends commands to applications or to the operating system itself.
- AppleScript
- Automator
- bash Unix shell scripts
Automating Chronological Tasks in OS X

**Startup**

It is possible to automatically launch applications (including AppleScripts) in OS X 10.4 when the computer starts up by adding them to the Login Items tab of the Accounts pane under System Preferences.

**cron**

Cron is a Unix system service that allows for scheduled (daily, weekly or monthly) execution of scripts, programs and applications. The lists of scheduled actions are kept in a formatted text file called a crontab. There is a system crontab, which normally should not be edited, and a personal crontab for each user, which is what you should edit to add a regularly scheduled task. Cron can be used with rsync or RsyncX to create an automated backup system.

By default, OS X 10.4 has a number of regular maintenance functions handled by another process called launchd, scheduled to run between 3:15 A.M. and 5:30 A.M. (which will not take place if the computer is shut off or asleep at that time).

**CronniX**

CronniX provides an OS X Aqua graphical user interface to cron, which makes editing crontabs to schedule repetitive tasks easier. It can be downloaded from the Mac OS X section of VersionTracker ([http://www.versiontracker.com/](http://www.versiontracker.com/)).

**osascript**

Osascript is a shell utility that allows you to run AppleScripts from the Unix command line (and thus use cron to run an AppleScript at scheduled intervals).
Automator is an automation application introduced in OS X 10.4 (found in /Applications). It allows users to assemble a series of steps (called actions) into a single sequence (called a workflow), without any programming required. Instead, it uses a simple drag-and-drop graphical user interface. Once created, workflows can be saved and modified or run again at a later time.

A number of predefined Finder and application actions are included (stored in /System/Library/Automator). Other actions can be added, and third-party applications can be written in such a way that their own actions automatically become available for use in Automator.

Behind the scenes, Automator uses the AppleScript scripting language to create and carry out its actions. Although Automator can be used to create useful workflows, more powerful scripting requires the use of a scripting language and techniques such as regular expressions.
iNquiryXTE is an example of an Automator based bioinformatics application. It is an OS X Open Source application that functions both as a web services client to an iNquiry enabled Apple Workgroup Cluster and an Automator action generator.
Regular Expressions

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

Wildcards
- A period matches any character except a line break (i.e. a carriage return).
- A caret matches the beginning of a line (unless used in a character class).
- A dollar sign matches the end of line (unless used in a character class).

Character Classes
- To match a set of characters, place square brackets around them. \[agct\] will match an a, g, c or t.
- To exclude a set of characters, place a caret after the opening bracket. \[^agct\] will not match an a, g, c or t, but will match any other character.
- To specify a range of characters, use a hyphen within the brackets. \[0-9\] will match any digit.

Quantifiers
- An asterisk matches zero or more occurrences of the specified class or character that precedes it. .* will match no or any characters.
- A plus sign matches one or more occurrences of the specified class or character that precedes it. \[0-9\]+ will match at least one digit.

Escape Character
- A backslash acts as an escape character, allowing you to search for wildcard or special characters, e.g. \. will actually match a period, and \ will match a backslash.
Searching for Regular Expressions with grep

grep (Globally search for Regular Expression and Print)
Grep is a powerful Unix text searching utility that is available at both the command line in OS X and in certain applications such as BBEdit. Grep can search the input for lines containing a match to a given pattern, using regular expressions if necessary, then output the lines that matched. Grep can thus greatly automate searching for information in text files.

The Unix grep utility can be invoked from the Terminal in OS X using the following syntax:
grep -[options] 'pattern' filename(s)

Useful grep Options
- **-c** print count of matching lines, rather than the matching lines themselves
- **-i** ignore case distinctions in pattern and file(s)
- **-l** print filenames containing matching lines, but not the matching lines

grep Results
grep normally prints a list of every line within the file(s) searched containing a match.

In Terminal, typing grep 'RNA' sars.txt will find all lines containing RNA within the sars.txt file. To automatically output those lines to a file called sarsrna.txt, one would type grep 'RNA' sars.txt > sarsrna.txt (in Unix, > redirects output).

grep '>' sequences.fasta will return the name of every sequence in the sequences.fasta file (in a fasta file, the sequence name is on a line beginning with >).

grep -c '>' sequences.fasta will only return the number of sequences in the file.
Grep with BBEdit and TextWrangler

BBEdit and TextWrangler have a built-in grep utility which can be used through their Find & Replace function. To use it, simply make sure the Find & Replace Use Grep option is checked.

For example, to strip out all non-DNA characters (such as line numbers or spaces) from a text file containing a DNA sequence (such as `r751.dna`) in BBEdit or TextWrangler, one can simply enter `[^acgt]` in Search For, make sure Use Grep is checked, and Replace With nothing, then select Replace All (by default, it is case insensitive).

To strip all numbering and blank spaces from a text file containing a protein sequence (such as `gag.aa`), one can first enter `\d` in Search For and Replace With nothing, then enter a space in Search For and Replace With nothing.

Useful BBEdit Special Character Matches

- `\r` Line break (carriage return)
- `\n` Unix line break (line feed)
- `\t` Tab
- `\f` Page break (form feed)
- `\d` Any digit [0-9]
- `\D` Any non-digit character (including carriage return)
- `\s` Any whitespace character (space, tab, carriage return, line feed, form feed)
- `\S` Any non-whitespace character (any character not included by `\s`)
**Subpatterns**

A subpattern consists of a simple or complex pattern enclosed in a pair of parentheses. Subpatterns allow you to reorder the data as it is replaced. This is a feature of certain grep implementations such as the one in BBEdit or in the Perl programming language.

- \& The entire matched pattern (replacement only).
- (x) The pattern x is remembered (search only).
- \1, \2, ..., \99 The nth subpattern in the entire search pattern.

**If you have:**

- ccc
- gcg
- atg
- cga

**Search For:** [acgt$]+
**Replace With:** Codon: &

**You will get:**
- Codon: ccc
- Codon: gcg
- Codon: atg
- Codon: cga

**If you have:**

- 2152aa
- 623aa
- 15aa

**Search For:** (\d)aa
**Replace With:** \1nt

**You will get:**
- 2152nt
- 623nt
- 15nt
AppleScript in OS X

**AppleScript**

AppleScript is a scripting language that provides direct control of scriptable Macintosh applications, including many parts of the Mac OS itself. Instead of using a mouse or keyboard to manipulate applications, you create scripts to automate tasks, control applications on local or remote computers, and even to access web services. AppleScript allows scripters to extend and integrate features supplied by the operating system and by scriptable applications.

To provide scripters with increased flexibility and power, developers make their applications scriptable, or capable of responding to Apple events. Applications can also execute scripts or send individual Apple events to take advantage of features of other applications. For example, a script that executes when the system boots could run a mail program, scan messages in the in box for URLs of a certain form, and then open those URLs in a web browser.

<table>
<thead>
<tr>
<th>Aqua</th>
<th>AppleScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cocoa</td>
<td>Java 2</td>
</tr>
<tr>
<td>Quartz</td>
<td>OpenGL</td>
</tr>
<tr>
<td>Darwin - Open</td>
<td>QuickTime</td>
</tr>
<tr>
<td></td>
<td>Audio</td>
</tr>
<tr>
<td>Desktop</td>
<td></td>
</tr>
</tbody>
</table>
In AppleScript, users write a series of English-like statements or scripts, that, when executed, send commands to scriptable applications or to the Mac OS itself. A scriptable application is one that can respond to Apple events (implemented through Mach kernel messaging) sent by the Mac OS or by another application. For example, the following script tells the application TextEdit to speak a line using the voice Fred:

```plaintext
tell application "TextEdit"
    activate
    try
        set this_string to ("Fitter, happier, more productive") as string
        set default_voice to ("Fred") as string
        say this_string using default_voice
    end try
end tell
```

**Script Editor**
A user can compile and run (execute) this script with Script Editor (distributed with Mac OS X and found in `/Applications/AppleScript`) or with any of several third-party script editing applications. From Script Editor, a script can be saved in five different formats with various options (Run Only, Startup Screen, and Stay Open):

- as a text file (can be opened and edited)
- as a script (can be opened and edited or executed as a script)
- as a script bundle (can include additional resources, such as images or sounds)
- as an application (automatically running the script when opened)
- as an application bundle (can include additional resources, such as images or sound)
AppleScript and ScriptMenu

Script Menu
Script Menu is a menulet that allows you to conveniently launch numerous scripts from the menu bar. It supports not only AppleScripts, but other kinds of scripts, such as Perl scripts or shell scripts.

Installing Script Menu and Adding Scripts
To activate Script Menu in OS X 10.4, use AppleScript Utility, located in /Applications/AppleScript (to remove Script Menu, uncheck “Show Script Menu in menu bar”).

A number of useful scripts come preinstalled. Scripts can be added to Script Menu by choosing the Open Scripts Folder menu item and placing script files in the appropriate Scripts folder. Sub-menus are automatically created when folders are added within the Scripts folder.

Folder Actions
Folder Actions Setup is an AppleScript application which allows scripts to be “attached” to folders so that the attached scripts may be executed when items are placed into or removed from a folder, a folder is opened, or a folder window closed or moved. With Folder Actions, one can create “hot folders” and “drop boxes” that respond to external actions to trigger a workflow or procedure.
Example of an AppleScript

(*
This script will speak the text entered in the dialog box, with a pause between each letter. *)

tell me to activate
display dialog "Enter the sequence to speak." buttons {"Cancel", "Speak"} default button 2 default answer ""
copy result to dialogResult
set textReturned to text returned of dialogResult
set dialogChoice to button returned of dialogResult
if dialogChoice = "Speak" then
  try
    set default_voice to ("Victoria") as string
    set this_string to textReturned as string
    set AppleScript's text item delimiters to ""
    set the this_list to every text item of this_string
    set AppleScript's text item delimiters to "",""
    set this_string to the this_list as string
    set AppleScript's text item delimiters to ""
    say this_string using default_voice
  end try
end if
AppleScript Studio

**AppleScript Studio, Project Builder and Interface Builder**

AppleScript Studio is an integrated development environment (IDE) for rapid creation of Aqua AppleScript applications. It is installed as part of XCode and allows one to use Interface Builder to quickly add an Aqua graphical user interface (GUI) to an AppleScript application, so that it can be controlled with buttons, sliders, checkboxes, popup menus and other interface elements. Project Builder allows one to easily construct complex AppleScripts, including droplets, which are AppleScript applications configured to support the dropping of files on the application.
Sample of a File Name
Trimming AppleScript

(*
Trim File Names
This script is designed to trim the names of files in the front window, or if no window is open, the files on the Desktop.
It is included with Script Menu.
*)

try
tell application "Finder" to set the source_folder to (folder of the front window) as alias
on error -- no open folder windows
    set the source_folder to path to desktop folder as alias
end try
repeat
display dialog "Text to trim from every file name:" default answer "" buttons {"Cancel", "Trim Start", "Trim End"}
copy the result as list to {the text_to_trim, the button_pressed}
    --if the button_pressed is "Cancel" then return "user cancelled"
    if the text_to_trim is not "" then exit repeat
end repeat
set the character_count to the number of characters of the text_to_trim
etc.
Sample of a Unix Shell Script

Maintenance AppleScript

(*)

This script will completely destroy the specified user account (and all files associated with it) under OS X 10.2. Use with caution. Copyright © 2003 Oliver Jovanovic, Ph.D.

*)

tell me to activate
display dialog "This script will completely destroy the specified user account." buttons
   {"Cancel", "Destroy"} default button 2 default answer "" with icon caution

copy result to dialogResult
set userDestroyed to text returned of dialogResult
set dialogChoice to button returned of dialogResult

try
   set userID to do shell script "niutil -readprop . /users/" & userDestroyed & " uid"
on error errorID
      set userID to errorID
end try

try
   if (userID > 501 and userID < 63000) and dialogChoice = "Destroy" then

      set rMail to "sudo rm -rf /var/mail/" & userDestroyed & "; "
      set rTmp to "sudo rm -rf /tmp/" & userDestroyed & "; "

   etc.
Why Use Unix?

- Historically, Unix has been used as a free academic and research operating system (BSD, FreeBSD, etc.)
- 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 scripts, called shell scripts. 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. It is the default shell for OS X 10.2 (Jaguar).

**csh**
The Berkley Unix C shell, from which tcsh is derived, available as an option in OS X.

**sh**
The Bourne shell, the original Unix shell, available as an option in OS X, though no longer used much.

**bash**
The Bourne-again shell, a successor to sh. It is the default shell for OS X 10.3 (Panther) and 10.4 (Tiger).
Terminal

Terminal, located in the Applications folder, is the application which gives an OS X user command line shell access with which to directly interact with the underlying Unix operating system.

Terminal Startup

When Terminal starts up, OS X 10.4 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

The most important preferences for Terminal are located in the Terminal menu, under Window Settings. At a minimum, make sure Buffer is set to 10,000 lines.

Another useful Window Setting preference is to activate **Option click to position cursor** under Emulation, 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.

You can also adjust the color of the text, the color and transparency of the Terminal window background, and other shell behavior.
Terminal Shortcuts

Saving Sessions
All the text in the Terminal window can be saved as a text file. 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 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. `ls`, which lists files in the current directory. Hundreds of command programs are included in OS X.

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` (`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 by default goes to the Terminal window, and can be copied and pasted. However, it is possible to have a program’s output 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 (`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 Paths

/ (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. 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 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.
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 Navigation Commands

**ls (list)**
Lists the current directory’s files. Adding the **-a** option (``ls -a``) lists all files, including invisible files (files whose names start with a period are normally invisible). The **-l** option (``ls -l``) lists long information about files: type, permissions, links, owner, group, size, modification date & time and name. They can be used together (``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, but will not properly copy Macintosh files with resource forks, for which ditto should be used instead. The -R option (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. It will properly rename, but not properly move Macintosh files with resource forks. MvMac should be used to move Macintosh files with resource forks.

**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 (rm -r) runs rm in recursive mode, where it will delete directories as well as files, including everything inside a specified directory. The -i option (rm -i) runs rm in interactive 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.
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 reverse order.

**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**.
Useful OS X Unix Commands

**open**
Lets you launch an Cocoa OS X application from the command line.

**ditto**
Ditto copies files. It uses the first file name as the source of the copy, then copies it to the specified destination (which can have the same name or a different one), creating a duplicate, e.g. `ditto original.txt copy.txt`. As of OS X 10.4, by default `ditto` properly copies Macintosh files with resource forks and HFS meta-data (the standard Unix `cp` command does not).

**CpMac and MvMac (Copy Macintosh and Move Macintosh)**
Copies or moves Macintosh files with the resource fork intact. Included with XCode.

**m2u and u2m (Mac to Unix and Unix to Mac)**
Replaces Macintosh style carriage returns with Unix style carriage returns in text files, and vice-versa.

**passwd (password)**
Allows you to change the password for that account. Make sure you choose something memorable, but difficult for anyone to guess. Good passwords are at least 8 characters in length, mix upper and lowercase characters, and include numbers or punctuation symbols, so they cannot simply be guessed from a dictionary of common words.

**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.
Unix Process Commands

Processes
When running, Unix programs run as processes, which can be displayed using the Activity Monitor utility (located in /Applications/Utilities), or directly from the 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 (**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 (**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 a Classic application crashes, it may leave a TruBlueEnv process running, which can slow the computer unless it is killed. It 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, e.g. `ls | more` (sends output from the `ls` program to the `more` program, which displays the output only one screen at a time), or `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 as 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 (normally, 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 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 (:).

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 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 10.4 bash shell, this is:

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

The same procedure is used to execute programs as scripts by interpreted programming languages such as Perl, 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
```
More Complex Shell Scripts

countfasta.sh

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

addfasta.sh

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

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

chmod 755 addfasta.sh
./addfasta.sh *.fasta
Scripting and Unix Resources

**AppleScript for Absolute Starters (AS4AS)**
A free primer by Bert Altenburg on AppleScript for beginners is available in electronic format (as a PDF file) at:
http://files.macsriptet.net/sourcebook/AS4ASb2.pdf

**Bash Guide for Beginners**
A free guide to the bash Unix shell (in PDF format) is available at:

**Recommended Books**
*Mastering Regular Expressions* by Jeffrey E. F. Friedl
*AppleScript: The Definitive Guide* by Matt Neuburg
*The Complete AppleScript Handbook* by Danny Goodman
*Running Mac OS X Tiger* by Jason Deraleau & James Duncan Davidson
*Learning Unix for Mac OS X Tiger* by Dave Taylor
*Mac OS X Tiger Unleashed* by John Ray & William C. Ray
*The Mac OS X Command Line* by Ken McElhearn
*Learning the UNIX Operating System* by Grace Todino, John Strang & Jerry Peek
*Think Unix* by Jon Lasser
Structured Programming

In *structured programming*, programs are created using combinations of four constructs: (1) instruction sequences, (2) branches, (3) loops and (4) modules. The program uses these constructs to perform certain operations on data, which it can input and output.

**Instruction Sequence**
A sequential series of instructions.

**Branch**
A branch, also known as a conditional construct, occurs whenever a program’s flow can divide into two or more streams, depending on whether a particular condition is true or false, such as whether a stop codon has been reached or not.

**Loop**
A loop repeats an instruction or series of instructions a variable number of times, which can be controlled by a test, such as whether the end of a DNA sequence has been reached.

**Modules**
Modules are a way to combine several operations (consisting of one or more of the other three constructs) into a single, reusable component. That component can then be reused throughout the program, or even used by other programs.
Program Elements
A program consists of two elements: (1) data definitions and (2) statements.

Data Definitions
Defines what type of data is used: integers, characters, dates, etc. Types can be static (a variable is always associated with a fixed type) or dynamic, strong (type is known at all times) or weak.

Statements
A program’s statements are typically broken down by function: initialization (preparing the program for its main function), core functionality (the program's main function) and finalization (cleaning up after the program has run).

Batch Oriented Programs
These are programs that are normally started from a command line (or run automatically by a scheduler such as cron). A batch program can simply consist of a text file with a list of programs it runs, or be more complex. When started, a batch program typically initializes the data inside it, reads in what data is specified as input, processes it, and outputs the result.

Event Driven Programs
There are programs that react to certain events sent to it by the operating system. This is typical of graphical user interfaces (GUIs), where an event might be a MouseUp (user moving the mouse up), or MouseClick (user clicks the mouse), which the program then responds to.
Programming Languages for Bioinformatics

Perl
The Practical Extraction and Report Language (PERL) is currently the most heavily used programming language in bioinformatics. It is particularly adept at handling arbitrary strings of text and detecting patterns within data, which makes it particularly well suited to working with protein and DNA sequences. In addition, it features a very flexible grammar which allows one to write in a variety of syntaxes, ranging from simple to complex. Perl has been widely used in genomics, including by the human genome project and TIGR. It is distributed under a free open source Artistic License and has become widely adopted by the open source programming community, resulting in numerous useful add on modules for Perl.
http://www.perl.org/

PHP
PHP originated as a set of simple scripting tools called Personal Home Page (PHP), but has become a powerful general-purpose free open source scripting language that is particularly well suited for Web development. PHP code can be embedded into HyperText Markup Language (HTML) and processed by a hypertext preprocessor module for a web server to create dynamic Web pages with minimal effort. Yahoo runs on PHP.
http://www.php.net/

C
The C programming language is one of the oldest programming languages still in wide use. A compiled C program offers excellent performance, and the syntax used by C has influenced most other programming languages in current use.
http://www.lysator.liu.se/c/
Object Oriented Programming Languages for Bioinformatics

Python
A simple object oriented scripting language that is well suited for developing bioinformatics applications and available under a free open source license. It is particularly easy to read and understand, and has been used in a number of bioinformatics applications, particularly in structural biology.
http://www.python.org/

Ruby
Ruby is a simple and particularly elegant object oriented programming language developed and made available for free by Yukihiro Matsumoto under an open source license. It is designed to easy to use and understand, and is gaining popularity for use in bioinformatics applications. A framework called Ruby on Rails makes it particularly well suited for web development.
http://www.ruby-lang.org/

Java
Java is a powerful object oriented cross-platform programming language developed and made available for free by Sun. It was originally developed for controlling consumer appliances, but repurposed for web development, then expanded. It is particularly well suited for developing complex projects. Although it is simpler than C++, the object oriented version of C, it still takes significant effort to master, but is very powerful, and has been used in a number of major bioinformatics projects.
http://java.sun.com/
Programming in Perl

**Statement (;)**
A statement in Perl is the basic unit of execution, and ends in a semicolon.

**Block ({}])**
A block is a series of statements enclosed by curly braces.

**Comment (#)**
A comment is anything which starts with a number sign. It is not executed, but rather used to explain what the code is doing, i.e. # this statement prints from 1 to 100

**Conditional Constructs (if, else, elsif)**
Conditional constructs use code that only executes under certain conditions. The *if*, *else* and *elsif* statements are the most common form of branch control in Perl, allowing you to execute another statement or block depending on whether certain conditions have or have not been met, i.e. if ($x > 0) { printf ("x is a positive number.\n"); }

**Loops (for, foreach, while)**
A loop repeatedly executes a specific set of statements until a particular condition is reached, which allows for rapid iteration. The *for*, *foreach* and *while* statements are commonly used in Perl loops, i.e. for ($x = 0; $x <= 99; $x++) { print "$x\n"; } print "$x\n";

**Subroutines (sub)**
To simplify complex code, it is possible to refer to a set of statements as a subroutine, using the *sub* declaration. This allows them to be repeatedly referred to in a concise manner, instead of having to write out the complete set of statements each time they need to be used.
Variables in Perl

Variables are one of the most useful features of programming languages, allowing a name to be associated with a stored data value, such as a string of text or a number, which can change as the program executes. Perl does not require variables to be formally declared, as opposed to some other programming languages, and uses three kinds of variables: (1) scalars, (2) arrays and (3) hashes.

Scalar ($)
A scalar variable is indicated by a $, and can store a number or a string, i.e.
$sequence = "GCATTTGTGAGACCCCGTACGTAG";

Array (@)
An array variable is indicated by an @, and can store multiple values in an ordered list of data, i.e.
@rna = ("G", "C", "A", "U"); The first element in an array is numbered 0, the second 1, the third 2, etc, and a value can be retrieved by specifying its position, i.e.
$value = $rna[3];

Hash (%)
A hash variable is indicated by a %, and is also known as an associative array because it associates a key with each value stored in it, i.e. %stop = (amber => AUG, ochre => UAA, opal => UGA); The value can then be retrieved from the hash using the appropriate key, i.e. $value = $stop{"amber"};
Pattern Matching in Perl

It is possible to do sophisticated pattern matching and replacement in Perl using regular expressions and transliteration.

The following program uses a regular expression to search for opal stop codons in a case insensitive manner in DNA:

```perl
#!/usr/bin/perl
$sequence = "GCATTTGTGAGACCCGCGTACGTAG";
if ($sequence =~ m/TGA/i) { printf("Opal stop codon found.\n"); }
```

The following program uses transliteration to count the number of Gs and Cs in a DNA sequence:

```perl
#!/usr/bin/perl
$sequence = "GCATTTGTGAGACCCGCGTACGTAG";
$countGC = ( $sequence =~ tr/GC// );
printf("There are \$countGC Gs and Cs in the sequence.\n");
```

The following program uses transliteration to convert a DNA sequence to an RNA sequence:

```perl
#!/usr/bin/perl
$sequence = "GCATTTGTGAGACCCGCGTACGTAG";
$sequence =~ ( $sequence =~ tr/T/U/ );
printf("The sequence converted to RNA is \$sequence\n");
```
By combining conditional execution and loops with regular expressions, it is possible to build sophisticated parsers in Perl which will parse complex data and extract useful information from it. One of the most common uses of such parsing in bioinformatics is to reformat or extract useful information from raw BLAST output.

The Perl program that follows is a BLAST parser that cleans up and reformats raw BLAST output into a tab separated value text file which can then be imported and viewed as rows and columns of data in a spreadsheet or database.
#!/usr/bin/perl -w

#blastconv.pl
#Reads in BLAST align (normal or Microbial) files, returns tab seperated values in new file by deleting unwanted filler.
#Returns TSV for GI, description, length, identities, frame, subjectStart, subjectEnd
#Usage: perl blastconv.pl, then enter filename   (returns filename.converted)
#To prepare, trim top and bottom of BLAST align file, add ">gi|" to end (can add this to program if done often).
#Copyright (c) 2002 Oliver Jovanovic

print "Enter name of BLAST align file to convert: ";
$sinFileName = <STDIN>;
chomp $sinFileName;
open (INTEXTFILE, $sinFileName);
@INTEXT = <INTEXTFILE>;
$textFile = join ( '', @INTEXT);  #concatenate all lines
$textFile =~s/
//g;  #remove all newline characters
$textFile =~s/>gnl\||>gi\|/\n/g;  #replace all ">gi|" (GI) or ">gnlI" (Microbial) with newline
$textFile =~s/\n\s+/\s+/g;  #replace all "|... " with tab (GI)
$textFile =~s/\n\s+/\s+/g;  #replace all "|" with tab (for Microbial) (leaves GN)
$textFile =~s/Length\s=\s/\t/g;  #replace all "Length = " with tab (length)
$textFile =~s/Score..Identities\s=\s/\t/g;  #replace all "Score...Identities =" with tab (identities)
$textFile =~s/Positive\sFrame\s=\s/\t/g;  #replace all ", Postives...Frame = " with tab (frame)
$textFile =~s/Query\s+(?:Sbjct:\s\d+)/(?:Sbjct:\s\d+)/\t/g;  #replace all Query..(minimal)...(Sbj) with tab
$textFile =~s/\s+(?=$)/\t/g;  #replace space before subjectEnd with tab
while ($textFile =~/Sbjct:/g)  {
    $textFile = s/\(?:Sbjct:\s\d+\)/\t/g;  #while part of below pattern being removed exists anywhere in string
}
$textFile =~s/\s+/\s+/g;  #clean up multiple spaces in description (replace with one)

open (OUTTEXTFILE, ">".$sinFileName.".converted");
print OUTTEXTFILE $textFile;
close (INTEXTFILE);
close (OUTTEXTFILE);
Perl Resources

Learning Perl
A site for people learning Perl. Features an online library, including a free PDF version of Simon Cozens' Beginning Perl book (http://www.perl.org/books/beginning-perl/), answers to commonly asked questions, and many useful links.
http://learn.perl.org/

CPAN (Comprehensive Perl Archive Network)
A large searchable collection of Perl scripts, modules and documentation. A Perl module, CPAN.pm, can used to download and install Perl software from the CPAN archive.
http://www.cpan.org/

BioPerl
A toolkit of perl modules useful for building bioinformatics solutions in Perl. The toolkit is built in an object-oriented manner so that many modules depend on each other to achieve a task. BioPerl includes a number of Perl routines and modules for performing sequence analysis, parsing sequence files, and parsing the results of bioinformatics applications such as BLAST.
http://www.bioperl.org/
Programming References

Programming Books

*Developing Bioinformatics Computer Skills* by Cynthia Gibas & Per Jambeck
*Beginning Perl for Bioinformatics* by James Tisdall
*Mastering Perl for Bioinformatics* by James Tisdall
*Genomic Perl* by Rex A. Dwyer
*Learning Perl* by Randal L. Schwartz & Tom Christiansen
*Elements of Programming with Perl* by Andrew L. Johnson
*Learn to Program Using Python* by Alan Gauld
*The Quick Python Book* by Daryl Harms & Kenneth McDonald
*Programming Ruby* by Dave Thomas