

# Introduction to Computing

## Concepts in this chapter

Concepts	
Computing	A brief history
Computer	Hardware and software
Programming languages	Levels of programming languages
Bits and bytes	Binary digit. All information is written down in bits

Computing is the process of using computers, hardware, software, and related technology to complete a given problem or task. We probably all perform some form of computing every day. Our cell phones are the most obvious use. Music downloads, gaming, email, GPS, programmable thermostats, credit and debit cards, and key cards all involve some form of computing. Computing can also be thought of as manipulation of information. Thanks to the rapid development of various technologies, we live in the age of excess information. Information about science, art, banking, business, entertainment, politics, weather, sports, people, societies, history, and so on ...

You will find that no matter what discipline you choose to study you will also need computing skills to maneuver through your field: from the liberal arts major gathering data from archeological digs to the data crunching needs of particle physicists working with data obtained from the large Hadron Collider to the genomics and bioinformatics research of today's biology related majors. Academicians across the world are realizing they must start students early in developing computational thinking.

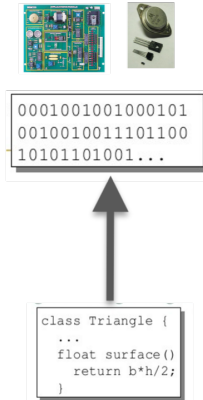
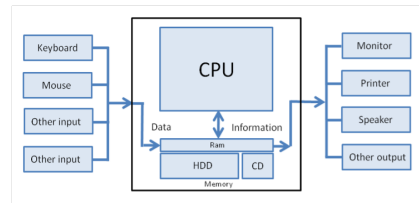
In this class you will begin to learn the process of computing through three different computing languages. Each section will build on the previous, but you will learn the basics first. First you will learn about the Unix shell, which allows you to give commands to the UNIX operating system. Then you will learn Python, which is an example of a very popular language that has proven to be quite useful in today's computing problems. Python is a very intuitive, and widely used in some scientific and engineering communities. You will build a project in Python that takes the idea of obtaining data, bringing it into a computer, manipulating the data, and finally presenting the data. Topics for this project are wide ranging but should focus on something you are interested in understanding further.

## A brief history

When did computers come to be? When did they first appear on the scene? Some would argue an abacus is a computer. Most would say the first invention of a programmable computer was the analytical engine, a mechanical device programmed with punch cards, designed by Charles Babbage in the 1800's. At the same time, a contemporary of Babbage, Ada Lovelace, was credited with conceiving of the first general purpose computer program.

A significant event in the history leading up to today's computers was the development from mechanical devices to vacuum tubes which controlled the flow of current through a circuit such that current in a tube could flow on or off to transistors which miniaturized the on-off nature of a signal. The contribution of van Neumann to the idea of a stored set of instructions was also key. Finally, around the same time, Alan Turing outlined the idea that given a finite set of steps or moves (also called a primitive set of instructions) any problem posed could be solved by the set of instructions given. This is referred to as the Turing Machine. These three events in the first half of the 20th century were foundational to the development of the computer as we know it today

## What does a computer consist of ?



**Hardware:** the term hardware refers to mechanical devices that make up computers. The CPU works on binary codes' 1's and 0's (machine language)

**Software:** It is a collection of programs to bring a computer hardware system into operation. A program is a set of instructions written in a computer language, translated into machine language, and executed by a computer.

The hardware, today, consists of a Central Processing Unit, memory, and inputs and outputs.

1. CPU – the central processing unit, performs the instructions
2. Memory – stores information (both data and instructions)
3. Inputs and outputs (keyboard, monitor, printer, mouse ...)

The circuits in a computer's processor are made up of billions of transistors. A transistor is a tiny switch that is activated by the electronic signals it receives. The digits 1 and 0 used in binary (a numerical system using 0 and 1) reflect the on and off states of a transistor. A binary digit, or bit, is the smallest unit of data in computing. It is represented by a 0 or a 1. Most of the main memory can be thought of as cells each storing one bit (0 or 1).

## Programming languages

There are many levels of instruction sets. Lowest level instructions executed by the CPU are in a language called machine code. This code consists of patterns of 0s and 1s, such that it can be understood directly by the computer. However, writing machine code is tedious and error prone. The human readable version of machine code is assembly code. Finally, there are higher level programs, such as Fortran, C, Java, Matlab, Python, etc. These higher-level languages need to be translated into machine code, i.e., the code that a computer can understand. Based on how this translation is achieved, the programming languages can be either compiled, interpreted or both.

## Various levels of languages and examples

**Lowest level** – machine code - patterns of ones and zeros are also called executable machine codes. Below is a machine code program for adding two numbers.

TABLE 4-3

A *machine code* program for adding 1234 and 4321. This is the lowest level of programming: direct manipulation of the digital electronics. (The right column is a continuation of the left column).

10111001	00000000
11010010	10100001
00000100	00000000
10001001	00000000
00001110	10001011
00000000	00011110
00000000	00000010
10111001	00000000
11100001	00000011
00010000	11000011
10001001	10100011
00001110	00000100
00000010	00000000

**Low level** – assembly code - Assembly programming requires an extensive understanding of the internal construction of the microprocessor you intend to use. Below is assembly program for adding two numbers.

TABLE 4-4

An *assembly* program for adding 1234 and 4321. An *assembler* is a program that converts an assembly program into machine code.

MOV CX,1234	;store 1234 in register CX, and then
MOV DS:[0],CX	;transfer it to memory location DS:[0]
MOV CX,4321	;store 4321 in register CX, and then
MOV DS:[2],CX	;transfer it to memory location DS:[2]
MOV AX,DS:[0]	;move variables stored in memory at
MOV BX,DS:[2]	;DS:[0] and DS:[2] into AX & BX
ADD AX,BX	;add AX and BX, store sum in AX
MOV DS:[4],AX	;move the sum into memory at DS:[4]

**Higher level languages** - High level programming languages are understandable by us humans. They contain words and phrases usually from the English language. A computer programmer does not have to understand the details of the computer architecture. Compilers and interpreters convert high-level code (source code) to machine code.

Higher level – Compiled: Fortran, C

Higher level – Interpreted: Bash, Matlab, Python

Programming languages like Python, Matlab, and bash (bash is the command language you will learn in Unix) use interpreters. Computer programs are text files containing the code of a programming language. An interpreter converts each high-level program statement, i.e. a single line of code, on by one, into the machine code, during program run. The interpreter displays errors of each line one by one.

Programming languages like C or Fortran use compilers. A compiler will scan the entire program and translate it as a whole directly into machine code, producing an executable file. You can then execute the machine code at a later time. The compiler displays all errors after compilation.

There are many languages in the world. Similarly, there are many coding languages. While different languages have a different syntax, some basic programming concepts are used by all languages. We will talk about some basic concepts in this course, and you will see how they appear in both languages that we will be working on.

## Encoding

Everything on a computer is represented as streams of binary numbers. Audio, images, and characters all look like binary numbers in machine code. These numbers are encoded in different data formats to give them meaning, e.g. the 8-bit pattern 01000001 could be the number 65, the character 'A', or a color in an image. Encoding formats have been standardized to help compatibility across different platforms.

For example, text is encoded in character sets, e.g. ASCII, Unicode.

*Numbers are (just) bits (Excerpted from ref. Leo Reyzin)*

How is 10100101 a number? Each position of the number is treated as a power of 2. From right to left:  $1 = 2^0$ ,  $2 = 2^1$ ,  $4 = 2^2$ ,  $8 = 2^3$ , . . . . Then you add the positions where the bit is equal to 1 and all others are multiplied by zero and are therefore not included. In the example  $10100101 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 1 \times 2^7 = 165$ . We can see that we use this concept every day; the number  $165 = 5 \times 1 + 6 \times 10 + 1 \times 100$  or  $5 \times 10^0 + 6 \times 10^1 + 1 \times 10^2$ . In the first example we are using the binary (or base 2) system and in the second case we are using the more familiar decimal (or base 10) system to describe the quantity 165.

*Text is (just) bits*

In fact, letters can be represented by a numerical code, and numbers, as we now know, are bits. There is a table of numerical codes for letters and other symbols, called ASCII (American Standard Code for Information Interchange). The table only goes up to 255 because it was decided at some point (apparently, at IBM in 1962) that a computer will work with chunks of 8 bits at a time. Eight bits are called a byte (the smallest amount the computer can byte off).

*Pictures are (just) bits*

We perceive color because our eye has sensors (called “cones”) for red, green, and blue light. Thus, to display a color to a human, we need to specify the intensity of its red, green, and blue (RGB) components.

We can use numbers for that. Typically, we use one byte (i.e., intensity between 0 and 255) per component, thus using three bytes, or 24 bits, to encode a color. See, e.g., <http://colormixers.com/mixers/cmr/>. Because there are 24 bits, this representation allows for  $2^{24} = 16,777,216$  possible colors. Another way to see the number of possible colors is to use the multiplication principle: observe that each of the three components has 256 possible values independent of the other components — thus, there are  $256 \cdot 256 \cdot 256 = 16,777,216$  possible combinations of the three components. The number of pixels in an image and the number of bits used for each pixel determine the file size of the image.

*Sound is (just) Bits.* Sound is just air pressure acting on our eardrum. We can view it as a graph of pressure as a function of time and then sample the graph at discrete time points to obtain a binary representation of sound.

# Unix

1. Getting started with Unix
2. The Unix filesystem: *navigate, manipulate, find and security*
3. Shell scripts, Variables and Metacharacters
4. Data files, text processing, formatting, and backing up
5. Data stream manipulation
6. Advanced text processing with the AWK programming language
7. Flow control: for loop
8. Flow control: if statements and while loop
9. Processes and Environment
10. Regex, Appendices and Tables

## Chapter 1. Getting started with Unix

Key commands and concepts in this chapter

Concepts	
UNIX operating system	Multi-user, multi-tasking operating system
Shell	User Interface for accessing operating system
Terminal	Command line interface
GUI	Graphical User Interface
home directory	Contains files for a given user
vi editor	Use to write and edit text files
General format Unix command	command (options) arguments
Shell error messages	When syntax is incorrect
Commands	
ls	List content of a directory
man	Help pages (manual)
[CTRL+d]	Exit a terminal session
clear	Clear the terminal screen of all text
date	Display current data
cal	Display a calendar

**Unix** is a family of [multitasking](#), [multiuser](#) computer [operating systems](#) that derive from the original [AT&T Unix](#), developed in the 1970s at the [Bell Labs](#) research center by [Ken Thompson](#), [Dennis Ritchie](#), and others ([Wikipedia](#)). There are multiple flavors of Unix and there are many systems which are UNIX-like in their architecture. Notable among these are the Mac OSX and the GNU/Linux distributions, including Ubuntu. The Unix operating system consists of many utilities along with the master control program, the [kernel](#). In [computing](#), the **kernel** is a [computer program](#) that manages [I/O](#) (input/output) requests from [software](#), and translates them into [data processing](#) instructions for the [central processing unit](#) and other [electronic](#)

**components** of a **computer** (*Wikipedia*). A UNIX kernel consists of many kernel subsystems like process management, memory management, file management, device management and network management.

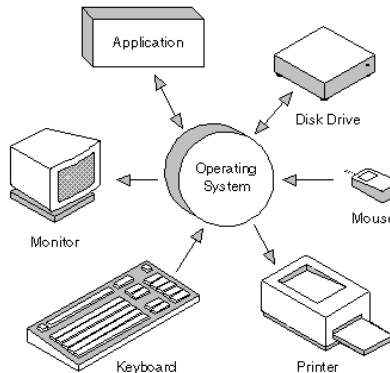
Some key features of the UNIX architecture concept are:

UNIX systems use a centralized operating system kernel which manages system and process activities.

UNIX systems are multiuser - more than 1 person can access the same computer.

UNIX systems are multitasking - multiple processes can run at the same time, or within small time slices and nearly at the same time, and any process can be interrupted and moved out of execution by the kernel. A process is the execution of instructions.

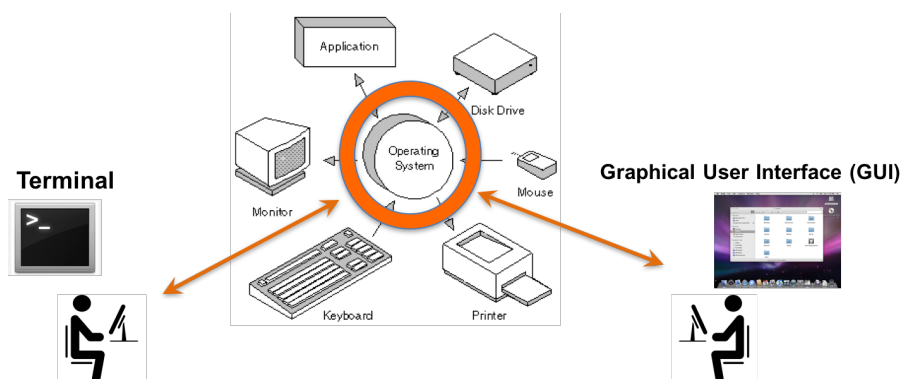
The Operating System (OS) is a system software which operates directly on the hardware devices of computer.




## 1.1 Shell Terminal – Command Line Interface

Your interface to the operating system is called a shell. An interface is a mechanism that is used to tell the computer what to do. Up to now you have been interacting with the computer using the graphical user interface, the GUI. In this mode a user visually decides what to do with files and apps. In a Mac, the Finder allows one to see the connections between different objects (files, directories, applications, etc.). GUI applications make use of the mouse to manipulate files.

Another way to tell the computer what to do is through the command line interface, CLI, provided by the terminal. This is a mechanism where the user types in specific commands in a shell command language.

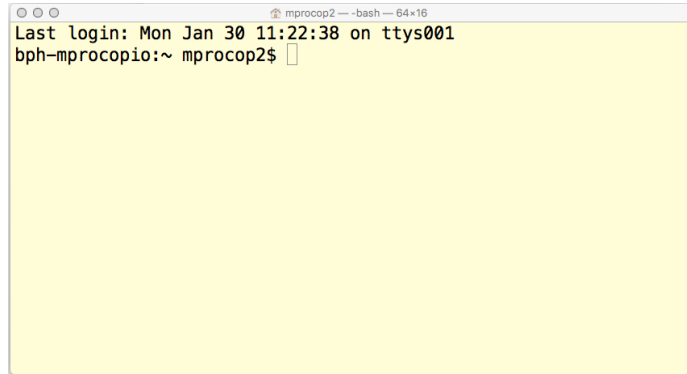


**Find the terminal in a Mac computer.** Go in Launchpad and search for  terminal. This is the terminal icon which you can “dock” .

**To find the WSL Ubuntu terminal in Windows or the terminal in Ubuntu VMware read the provided notes in Canvas.**

When you click on the terminal icon you will launch a shell. The terminal shell is the connection between what you, the user, want the computer to do and the bits that are processed at the hardware level.

On the right you see the terminal window. In the terminal window there is a symbol called a prompt. Immediately after the prompt is the cursor location. This is the point where text will be displayed as you start to type from the keyboard.



```
mprocop2 -- bash -- 64x16
Last login: Mon Jan 30 11:22:38 on ttys001
bph-mprocopio:~ mprocop2$
```

The user can type commands to perform

functions such as run programs, open and browse directories, and view processes that are currently running. Since the shell terminal is only one layer above the operating system, you can perform operations that are not always possible using the graphical user interface (GUI). Some examples include moving files within the system folder and deleting files that are typically locked.

The catch is, you need to know the correct syntax when typing the shell commands. You need to learn a shell "command language" which is "a high-level programming language" through which a user communicates with the operating system.

You can begin by typing a command at the cursor. The cursor is the small dark squarish box.

Try telling the computer to do something by typing a few commands at the prompt (in the example below the prompt is represented with a \$ sign):

Type the command `date` and press return.

```
date
Mon Jun  2 21:09:14 EDT 2014
```

Now type the command `cal` and press return

```
cal
      June 2014
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

Imagine what is happening. What is the stream of information that is being accessed? When you type `date`, the computer takes that information and copies the value of the system clock to memory, formats the data, and then prints the values to the screen. How do the above commands fit in the data stream concept of computing? The input in this case (the current year, month, day, hour, minutes, and second) is information stored on the computer but hidden from the user until accessed. The data are manipulated in format by the command `date`. Finally, the result is sent to the output, the screen.

The `clear` command will clear your terminal screen of all text.

```
clear
```

Command `+` will make what you are viewing bigger

Command `-` will make what you are viewing smaller

Command `n` will open a new terminal

### *Logout*

When you are finished with your terminal session you can type **logout** or **exit**. If you type **logout**, a sequence of “cleanup” commands will be invoked from your logout file. These might be commands that clear the history and clear the screen for security purposes. However, if you type **exit**, logout of the Apple computer, quit terminal, or close a terminal window, no major problems will occur. You just will not be running one aspect of the logout sequence.

When all else fails to close a terminal session, press **CTRL-d** (see table in appendices). This will return you to the previous shell if you were using one (yes, you can have more than one shell running from a single terminal session) or perform a logout if you are in your login shell. See the appendix for a table of ctrl commands.

## 1.2 Bash Shell – Bash command language

Common Unix terminal shells are:

- Bourne shell (sh)
- C shell (csh)
- TC shell (tcsh)
- Korn shell (ksh)
- **Bourne Again shell (bash)**

Most Unix systems have these terminal shells available to the user. The shells are command line interpreters (not compilers). Windows users may be more familiar with DOS, the shell that has long been included with the Windows operating system.

In our class, we will be using the bash shell.

To know which is the default shell in your computer, you can type at the terminal

```
echo $SHELL
```

If you're using bash you should get the following output:

```
/bin/bash
```

If the output is not /bin/bash then you need to change to bash by typing at the terminal

```
chsh -s /bin/bash
```

The default shell on Ubuntu and Mac should be bash.

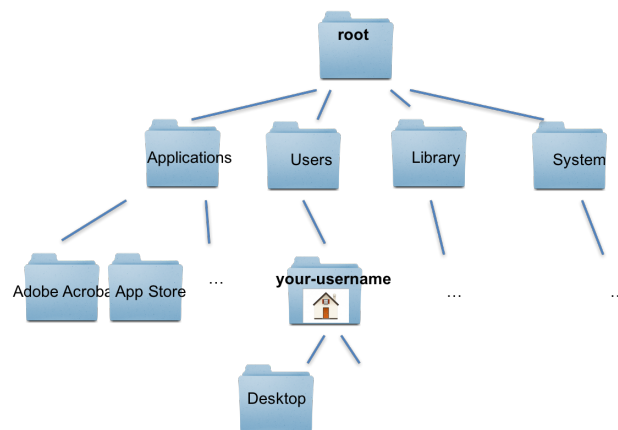
Outside of this course, you may find yourself having to (or wanting to) use a different shell. Remember they are basically the same idea with a few different commands and syntax requirements. New shells developed out of programmer's frustrations. It is not that one is necessarily better or worse, they meet different needs.



### 1.3 home directory (or log in directory)

The organization of files on a Unix system is in the style of a tree-like structure of directories (sometimes called folders) which may contain files and other directories. A file can hold information but cannot contain other files or directories. The initial point of the tree is called the root directory. The root directory may contain files and subdirectories which may contain other files and subdirectories. All other directories on the file system branch from this topmost layer, root.

In a Unix file system, a home directory, also called a login directory, is created automatically for every ordinary user. The home directory serves as the repository for a user's personal files, directories, and programs. In a Mac, the home directory is created automatically in the directory called Users. **The first thing you should know about the home folder is that it's not named home. In a Mac computer it features a home icon, but its title is the name you chose for your user account.** The name of your home directory is the username in all Unix-like systems, such as Ubuntu. Below a pictorial representation of a Unix file system.



When you open a terminal, you are in your home directory (or folder) within the file system. Open a terminal session, and now type the bash command `ls` (list the contents of a directory)

```
ls
```

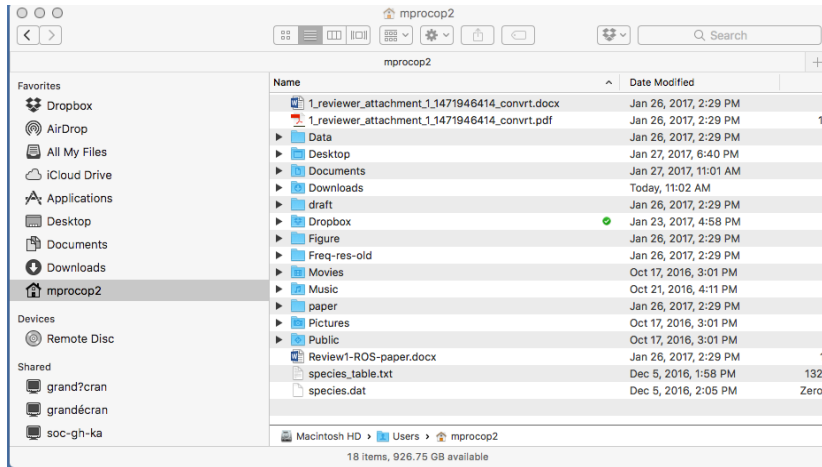
Since we are in your home directory, we list the contents of the home directory.

Use the Graphical User Interface (GUI) and check if you find the same contents in your home directory (home folder).

#### How to find your home directory in a Mac Computer using GUI?

- open Finder
- press the keyboard shortcut Command-Shift-H at the same time
- press the keyboard Command and up arrow key at the same time
- drag and drop the home icon to the Finder's sidebar for quick and easy access





Type **ls** in the terminal window. Now use the mouse to open your home folder on the desktop. What files are in the folder? Are they the same as in the terminal window? Can you identify which is a GUI command and which is a CLI command?

**To find your home or login directory (named with your username) in WSL Windows or VMware Ubuntu read the provided notes in Canvas.**

## 1.4 The vi editor

The vi editor is the most popular and classic text editor in the Unix family. Below are some reasons which make it a widely used editor.

- 1) It is available in almost all Linux Distributions and Mac
- 2) It works the same across different platforms and Distributions
- 3) It is user-friendly. Hence, millions of Unix users love it and use it for their editing needs

Nowadays, there are advanced versions of the vi editor available, and the most popular one is **vim** which is **vi improved**. Some of the other ones are **evis**, **nvi**, **nano**, and **vile**. It is wise to learn vi because it is feature-rich and offers endless possibilities to edit a file.

### Starting vi

Open a file with vi. Type:

```
vi myfile.txt
```

If **myfile.txt** does not exist, a screen will appear with just a cursor at the top followed by tildes (~) in the first column.

If **myfile.txt** does exist, the first few lines of the file will appear.

The status line at the bottom of your screen shows error messages and provides information and feedback, including the name of the file.

To work on vi editor, you need to understand **its operation modes**. They can be divided into two main parts.

## Command Mode

Command mode is the mode you are in when you start (default mode)

Letters of the keyboard are interpreted as commands that will alter the text of the file

Commands are case sensitive: j not the same as J

Most commands do not appear on the screen as you type them. Some commands will appear on the last line, like the colon followed by w and q :wq

## Insert (or Text) Mode

The mode in which text is created. (You must press <Return> at the end of each line unless you've set wrap margin.)

There is more than one way to get into insert mode but only one way to leave: return to command mode by pressing <Esc>

When in doubt about which mode you are in, press <Esc> to return to command mode.

## A Basic vi Session

- To enter vi, type: vi filename <Return>
- To enter insert mode, type: i
- Type in the text: This is easy.
- To exit insert mode and return to command mode, press: <Esc>
- In command mode, save changes and exit vi by typing: :wq <Return>  
You are back at the Unix prompt.

Make your first file by using vi

```
vi file_vi.txt
```

- To enter insert mode, type: i
- Type in this text

```
I LOVE COFFEE
I love coffee
I really love espresso
```

- To leave insert mode and return to command mode, press: <Esc>
- In command mode, save changes and exit vi by typing: :wq <Return>

## Basic Cursor Movement

*From Command Mode*

<b>k</b>	Up one line
<b>j</b>	Down one line
<b>h</b>	Left one character
<b>l</b>	Right one character (or use <Spacebar>)
<b>w</b>	Right one word
<b>b</b>	Left one word

## Entering, Deleting and Changing Text

*From Command Mode*

**i**     *Enter text entry mode*  
**x**     *Delete a character*  
**dd**   *Delete a line*  
**r**     *Replace a character*  
**R**     *Overwrite text, press <Esc> to end*

## Displaying Line Numbers

*From Command Mode*

**:set nu**     *Display line numbers*  
**:set nonu**   *Hide line numbers*

## Exiting vi

To exit you must be in command mode – press <Esc> if you are not in command mode  
You must press <Return> after commands that begin with a : (colon)

*From Command Mode*

**zz**     *Write (if there were changes), then quit*  
**:wq**     *Write, then quit*  
**:q**     *Quit (will only work if file has not been changed)*  
**:q!**     *Quit without saving changes to file*

## More On Cursor Movement

*From Command Mode*

**e**     *Move to end of current word*  
**\$**     *Move to end of current line*  
**^**     *Move to beginning of current line*  
**+**     *Move to beginning of next line*  
**-**     *Move to beginning of previous line*  
**G**     *Go to last line of the file*  
**:n**     *Go to line with this number (:10 goes to line 10)*

## More on Entering Text Mode

*From Command Mode*

**i**     *Insert text before current character*  
**a**     *Append text after current character*  
**I**     *Begin text insertion at the beginning of a line*  
**A**     *Append text at end of a line*

## More on Deleting Text

### From Command Mode

**dd** Delete a line (6dd deletes six lines)  
**d\$** Delete all characters to the end of the line.

## Other Useful Commands

### From Command Mode

<b>u</b>	<i>Undo last single change</i>
<b>U</b>	<i>Restore current line</i>

## Substitutions

The simplest way to do substitutions throughout the file is to use the `s` colon command. The basic form of this command is the following:

### From Command Mode

```
:%s/old/new/g    Substitutes old with new throughout the file
```

**s** means to substitute text matching the pattern (old)  
with text specified by (new)

**g** (global) is optional. It indicates you want to substitute all occurrences on the indicated lines.

If you do not use **g**, the editor substitutes only the first occurrence on the indicated lines.

Find more info here <https://staff.washington.edu/rells/R110/>

You should complete the tutorial on **vi** that is provided by the UNIX distribution, **vimtutor**.

Please remember it is assumed you will use `vi` for all editing of files. Therefore, it is very important that you become comfortable in `vi`. Most of the exercises will depend on `vi` and the first exercises are geared toward helping you understand basic file manipulations.

At the *command prompt* type **vimtutor**

vimtutor

Begin with lesson 1 in **vimtutor**

To exit, type

```
:q <enter>
```

**Keep in mind that vi is NOT a bash command.** It is a Unix editor you use from the terminal.

## Make the file long.txt

Despite the fact you are probably not entirely comfortable with **vi** yet, we will push forward with editing of files. You can work on the **vimtutor** on your own.

Make a file called `long.txt` by using **vi**

```
vi long.txt
```

Press *i* to insert text. On the first line enter number 1, second line 2, and so on, until you have reached 30. The beginning of the file should look like this:

```
1
2
3
```

To write the contents into the file and exit vi press *wq*

Once you have written the file, can you figure out where the file is located?

## 1.5 Viewing files: cat, head and tail, more, less

The command cat will show you on the terminal what the contents of the files are.

```
cat long.txt
```

Two other useful commands for viewing files are more and less.

Now try viewing this file with cat, more and less. Can you figure out the difference? Do you think the expression “less is more” applies in this case?

Can you guess what commands head and tail will do? Use the man command to find out how to invoke the head command (see section 1.7)

## 1.6 General format of UNIX commands

Most Unix commands are usually followed by one or more strings (i.e., sequences of characters) that comprise options and arguments. Each of these strings are separated by white space (which consists of one or more spaces or tabs). Most commands in Unix are lowercase. **Keep in mind that Unix is case-sensitive.** The general format of the Unix commands is:

***command (options) [arguments]***

The square brackets indicate that the enclosed items are optional. Most commands have at least a few options and can accept (or require) arguments. However, there are some commands that do not accept arguments, and very few with no options.

A command is an instruction given by a user telling a computer to do something. Commands are generally issued by typing them in at the command line (i.e., the all-text display mode) and then pressing the ENTER key, which passes them to the shell.

An option is a single-letter code, or sometimes a single word or set of words, that modifies the behavior of a command in some predetermined way. When multiple single-letter options are used, all the letters are placed adjacent to each other (i.e., not separated by spaces) and can be in any order. The set of options must usually be preceded by a single hyphen, again with no intervening space.

An argument, also called a command line argument, is a file name or a directory name or other information that is provided to a command.

In the example

```
date
```

the command is **date** and no options or arguments are given.

In the example

```
cal 2014
```

an argument (2014) was given to the command **cal**. Arguments modify the commands they support. In this example no options are given.

In the example

```
head -2 long.txt
```

the command is **head**, -2 is an option and long.txt is an argument.

## 1.7 the man command

The Unix man pages (Unix manual) are very useful for both the novice and the well-seasoned Unix guru. The manual pages are accessible through a utility (program or app) invoked (ran or executed) by the Unix command man. The man command in Unix displays documentation of commands. If you want to learn what the command head does and its options type:

```
man head
```

## 1.8 Shell Error messages

You can make different types of errors while typing Unix commands. Shell error messages will appear on the terminal after you type Unix commands followed by ENTER. The different error messages that the shell provides will help you to recognize and fix such errors. Here are some common Unix error messages and common source of errors.

### Error message: Command not found

When you get the error **Command not found** it means that UNIX shell searched for the command in the specified places where commands are found and could not find it by that name. Another cause might be that you misspelled the command name (typo).

This error message starts with **-bash**, which is the name of shell that interprets the command, followed by the command name that was not found, i.e.:

```
-bash: wrong command name: command not found
```

To fix this error explore the following suggestions:

- Make sure the command was not misspelled

- All UNIX commands are case sensitive, and you need to type correct spelling of a command.
- Make sure you followed the general format of a UNIX command

Below some common sources of **command not found** errors:

### Typos

If you type a command that does not exist, the shell warns you that you have made an error.

For example, if you want to view the file long.txt using the command cat, and make a typo like this:

```
can long.txt
-bash: can: command not found
```

You must correct the command from can to cat for the command to be recognized by the Unix shell, and then be executed. Making typos is common, so you should learn how to recognize such errors and correct the code.

### Type an option that does not exist

For example, if you want to use the command tail and its option to visualize the last 3 lines of the file long.txt, and you type:

```
tail -n3 long.txt
-bash: -n3: command not found
```

Here -n3 is the wrong option. The correct option is either -3 or -n 3. If you do not remember how to use an option, use the man page of the command.

### Not following the general format of a Unix command

Again, if you want to use the command tail and its option to visualize the last 3 lines of the file long.txt, and you type:

```
tail-3 long.txt
-bash: tail-3: command not found
```

In this case the option is correct (-3) but you didn't follow the Unix format. The general format requires a white space between a command, its options, and its arguments. If the option is omitted, there must be a white space between the command and its arguments. In the example above there must be a white space between the command tail and its option -3.

Also remember that you must respect the order of the format, i.e. command, followed by options and arguments.

### Error message: No such file or directory

When you get the error **No such file or directory** it means that UNIX shell searched for an argument name and could not find it. This error refers to a not existing argument of a Unix command.

This error message starts with the command name, followed by the input argument given to the command that the shell does not find.

```
Command name: not existing file name or directory name: No such file or
directory
```



Common sources of this type of error message can be:

### Typos

If you want to view the content of the file `long.txt` using the `cat` command, and you type:

```
cat lon.txt
cat: lon.txt: No such file or directory
```

The command `cat` gets as argument `lon.txt` that the shell does not find. Indeed, the correct file name is `long.txt`. You will find more common sources of this type of error message in Chapter 3. **ALWAYS FIX ALL SHELL ERROR MESSAGES BY CORRECTING THE UNIX CODE.**

## 1.9 the class header

Headers are often used in files to describe what is in them. A header will consist of one or more lines of text and will be distinguishable from the rest of the file.

Add the class header to the file `long.txt` by following this format

```
#jhed:date:file name
```

For example:

```
#mprocop2:01/02/2020:long.txt
```

Add a header at the beginning of the file containing a `#` followed by username, date, filename, all separated by `:`colon or a comma. Notice the date. This is the date format to use in this class *mm/dd/yyyy*.

**The class header must be in the first line of each file you submit to Gradescope assignments.**

## Chapter 2. Unix File System: Navigate, Manipulate, find, Security

### Key commands and concepts in this chapter

Concepts	
Directory hierarchy	The system map
Traversing the directory tree	Moving around the system
Hidden files	Files that start with a dot. Need <code>ls -a</code> to view them
File System permission	Security in place
Commands	
<code>pwd</code> , <code>cd</code> , <code>ls</code>	Commands to view and maneuver through the directory tree
<code>touch</code> , <code>mkdir</code>	Creation of files and directories

cp, mv	Copy and move files and directories
rm, rmdir	Deletion of files and directories
find	Locate files and directory within the File System
chmod	Change permission settings

## 2.1 Unix file system

In a computer, a file system (sometimes written filesystem) is the way in which *files* are named and where they are placed logically for *storage* and retrieval. In Unix and Unix-like operating systems, the *file system* is considered a central component of the operating system. It was also one of the first parts of the system to be designed and implemented by *Ken Thompson* in the first experimental version of Unix, dated 1969.

The UNIX filesystem contains several different types of files:

- Ordinary Files
  - Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
  - Always located within/under a directory file
  - Do not contain other files
- Directories
  - Branching points in the hierarchical tree
  - Used to organize groups of files
  - May contain ordinary files, special files or other directories
  - Never contain "real" information which you would work with (such as text). Basically, just used for organizing files.
  - All files are descendants of the root directory, ( named / ) located at the top of the tree.
- Special Files
  - Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Output (I/O) operations
  - Unix considers any device attached to the system to be a file - including your terminal:
    - By default, a command treats your terminal as the standard input file (stdin) from which to read its input
    - Your terminal is also treated as the standard output file (stdout) to which a command's output is sent
  - Usually only found under directories named /dev

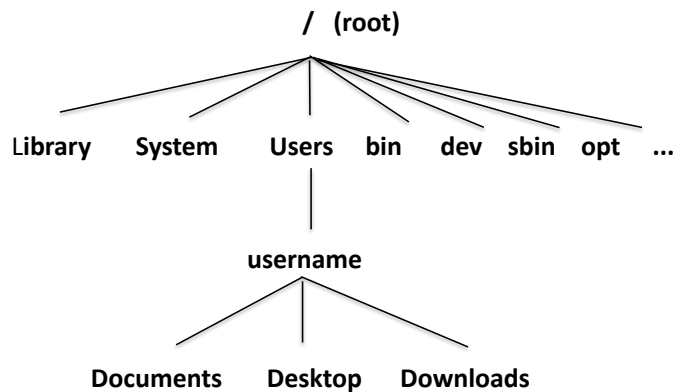
### *Directories beneath the root*

The Unix operating system comes with a lot of files that are necessary for the computer to run. Some of these files are executable. An executable file is a computer file that contains instructions in a form that a computer's operating system or application can understand and follow. Computers must use executable files to carry out the tasks that you give to them. Every application you run starts off with an executable program. Several directories exist beneath root. Some flavors of Unix will vary in the location and naming convention of these files. For example, user or home instead of Users. Most of these differences are merely related to system organization. Some typical directories beneath the root directory include:

/bin	User command files. Must be present for system to boot and run
/usr/bin	User commands files not required by the system
/sbin	Executable files, usually for system administration
/usr	Used for miscellaneous purposes, used by multiple users
/dev	Device files. Here computer contains a list of all devices it understands
/etc	System wide configuration files
/Users or /home	User home directories (or users)
/Volumes or /mnt	Mount point for a temporarily mounted filesystem
/opt	Add-on application software packages

## Directory hierarchy

The organization of files on a Unix system is in the style of a tree-like structure of directories (sometimes called folders) which may contain files and other directories. A file can hold information, but cannot contain other files, or directories. The initial point of the tree is called the *root directory*. The root directory may contain files and subdirectories which may contain other files and subdirectories. All other directories on the computer branch from this topmost layer, *root*. Its location is designated by a single forward slash `/`.



You can navigate the filesystem by typing specific commands in a terminal session, and you can decide where to go, i.e., where to be located within a filesystem. Although it may seem strange to "go" somewhere in a computer's filesystem, the concept is not so different from real life. After all, you can't just *be*, you have to be *somewhere*.

The place in the filesystem tree where you are located is called the ***current working directory***.

Every time you start a terminal session the shell considers you to be located in your login or home directory. Thus, when you start a new terminal session your home directory is your current working directory. In the directory structure above the home directory is *username*.

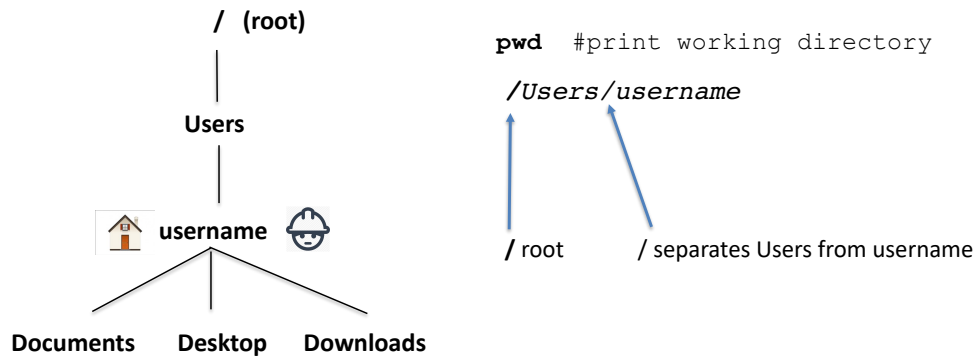
## 2.2 Print working directory: pwd

Command **pwd** will display the **absolute pathname** to your current working directory. In the example below, the `pwd` command shows that you are in your home directory.

```
pwd
```

If you are in your home directory, and you type *pwd*, you will get:

```
/Users/username
```



Absolute pathnames always start with `/` (root) and the subsequent slashes `/` in a pathname separate one directory from another.

## 2.3 Pathnames

To describe a specific file or directory in the filesystem hierarchy, you must specify a path. The path to a location can be defined as an absolute path, starting from the root anchor point, or as a relative path, starting from the current working directory. When specifying a path, you simply trace a route through the filesystem tree, listing the sequence of directories that the operating system must follow to go from one point to another. Each directory listed in the sequence is separated by a slash. The pathname always starts from your working directory or from the root directory.

It is initially confusing to some that Unix uses the slash character `/` to denote the filesystem root directory, and as a directory separator in paths. Just remember, when the slash is the first thing in the path to the file or directory, the path begins at the root directory. Otherwise, the slash is a separator. Each operating system has its own rules for specifying paths. Hereafter we will describe rules and commands to navigate a Unix Filesystem.

You must know how to use pathnames to navigate the UNIX file system.

**Absolute Pathname:** tells how to reach a file or directory beginning from the root; **absolute pathname always begins with `/` (slash)**. For example, for the directory shown in the figure above, the absolute path to the Desktop directory is:

```
/Users/username/Desktop/
```

However, the *username* will be **different for every user**. For a user named *mprocop2*, the absolute path is:

```
/Users/mprocop2/Desktop/
```

**Relative Pathname:** tells how to reach a file from the directory you are currently in (current or working directory); **never begins with `/` (slash)**, and never begins with your current working directory name.

For example, if you are in the /Users directory, the relative pathname to the Desktop directory of user mprocop2 is just:

```
mprocop2/Desktop
```

If you were already in the home directory of user mprocop2, the relative path to Desktop would have been just:

```
Desktop
```

**Remember that pathnames are arguments, and must be preceded by a Unix command, like**

```
cd /Users/username/Desktop/
```

There are some shortcuts for relative pathnames

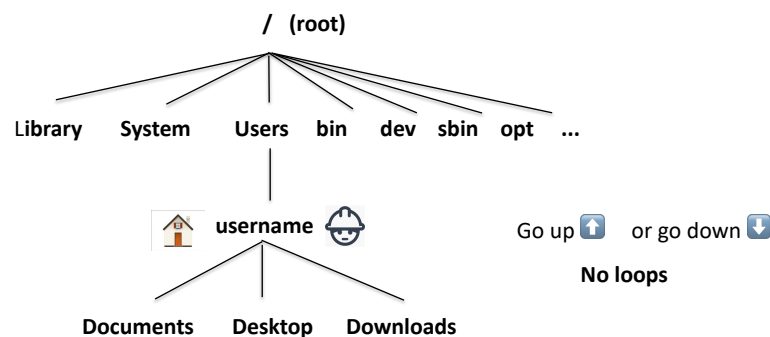
```
..      (double dots) to locate one directory back
../.../  to locate two directories back
~        locate home directory
/        locate root directory
```

## 2.4 Changing directory

When you first login you are in your personal home directory. In order to traverse the file system, you must specify a path, i.e. you have to list the sequence of directories (pathname) that the shell has to follow to go from one point to another:

```
cd pathname
```

You can go up or down the filesystem, but you cannot make loops.



## Navigating the directory tree with an absolute pathname

To go to Desktop directory with an absolute pathname

```
cd /Users/username/Desktop  
pwd
```

To go to Documents with an absolute pathname

```
cd /Users/username/Documents  
pwd
```

Absolute pathnames do not depend on where you are located in the file system.

To go to Users with absolute pathname

```
cd /Users  
pwd
```

## Navigating the directory tree up (back) with relative pathnames

To go back one directory relative to your current working directory:

```
cd ..  
pwd
```

To go back two directories relative to your current working directory:

```
cd ../../
```

To go to the root directory

```
cd /
```

Shortcuts to go to your home directory:

```
cd  
cd ~
```

## Navigating the directory tree down with a relative pathname

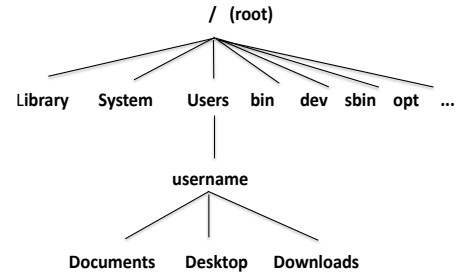
If your current directory is Users to go to Desktop

```
cd username/Desktop
```

If your current working directory is Desktop, to go to Downloads

```
cd ../Downloads
```

You should go back and then into Downloads because you cannot traverse directories that are in the same layer. You cannot make loops.



## 2.5 List the contents of a directory

To find out what files are in your current working directory type:

```
ls
```

**ls** provides a “listing” of the contents of the current directory. Try listing the contents of your home directory

```
cd
ls
```

Return to the root directory

```
cd /
ls
```

What files are in the root directory?

To list the contents of a directory far from your working directory without changing into that directory, you can use absolute or relative pathnames. For example, your current directory is Desktop directory:

```
pwd
/Users/username/Desktop
```

and you want to list the contents of your Users directory. You can use absolute pathnames:

```
ls /Users
```

or relative pathnames:

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

This pathname `../../` is a relative pathname indicating the shell to go back two directories. When we say go back, we refer to go toward the root directory.

If you now want to list the contents of directory Desktop, you can use again absolute pathname:

```
ls /Users/username/Desktop
```

In general, adding an “argument” to the **ls** command will list the directory contents of the given argument, which in this case is a pathname.

## 2.6 Hidden files

Hidden files start with a dot. They can be seen by adding an option `-a` to the `ls` command

```
ls -a
```

This command lists all files in the current directory including *hidden* files. Hidden files are necessary but can produce clutter for everyday computing tasks, therefore they are listed only when the option `a` is used. In fact, the Mac OSX system does not show these files to the user in the GUI application. Usually these are configuration files and are generally not to be altered. To make a file *hidden* preface the filename with a period.

## 2.7 File and directory creation: touch, mkdir

### Create a directory – mkdir command

Let's open a terminal. Our current working directory is home.

Directory creation can be accomplished in many ways. For directory creation you can use the command `mkdir`.

```
mkdir mydir
```

The directory will be created in your current working directory.

If you want to create a directory within a directory different from your current working directory, and you do not want to change directory, you can use absolute or relative pathnames.

If your working directory is your home directory, and you want to create a directory named `test` within the `mydir` directory, you can use absolute pathname:

```
mkdir /Users/username/mydir/test
```

or relative pathname

```
mkdir mydir/test
```

You can also create nested directories by using the `-p` option of the `mkdir` command. You can create both `mydir` and `test` by doing

```
mkdir -p mydir/test
```

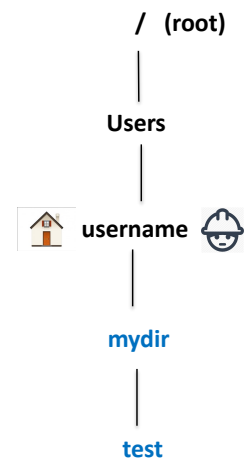
### Create a file: touch command

For file creation you can use the command `touch`.

To create a file called `file10.txt` within your current working directory:

```
touch file10.txt
```

`touch` creates an empty file





```
cat file10.txt
```

If you want to create a file within a directory different from your current working directory, and you do not want to change directory, you can use absolute or relative pathnames.

If your working directory is your home directory, and you want to create a file named file11.txt within the mydir directory, you can use

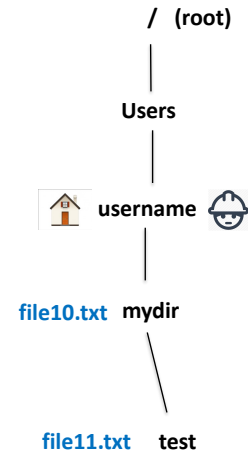
absolute pathname

```
touch /Users/username/mydir/file11.txt
```

or

relative pathname

```
touch mydir/file11.txt
```



## 2.8 Copy and move file and directory

Files can be copied, moved, and renamed throughout the system.

To copy a file, use the command cp.

```
cp file11.txt file12.txt
```

The file file12.txt will be created in your current working directory. Note that the command cp takes two arguments, the first, file11.txt is the original file, file12.txt is the copied file.

Suppose that your current working directory is your home directory, and you want to copy file11.txt (which is within mydir) into your current working directory, and rename it file11copy.txt, you can use absolute pathname:

```
cp /Users/username/mydir/file11.txt ./file11copy.txt
```

or relative pathname

```
cp mydir/file11.txt ./file11copy.txt
```

./ means current working directory.

To move a file, use the command mv.

Let's go to home directory

```
cd ~
```

Move file10.txt in directory test, and rename it file1.txt

```
mv file10.txt mydir/test/file1.txt
```

Is file10.txt still in your current directory?

```
ls
```

You can use the command `mv` to move a file and to rename it. You need two pathnames, same as the `cp` command.

## 2.9 Deleting files and directories: `rm` and `rmdir`

To remove unwanted files, use `rm`. To remove unwanted empty directories, use `rmdir`. Try this:

```
touch file15
ls
rm file15
ls
```

Again, you can use pathnames to delete files or directories located far from your current directory without the need to change directory. You can remove file1.txt from the test directory by using relative pathname:

```
rm mydir/test/file1.txt
```

Be careful with the commands `rm` and `rmdir` because after you delete files or directories from a filesystem, it is very hard and expensive to recover them. It is always a good habit to use the option `i`, which asks for every deletion to be confirmed (interactive). Try:

```
touch file15
rm -i file15
```

Try to remove mydir

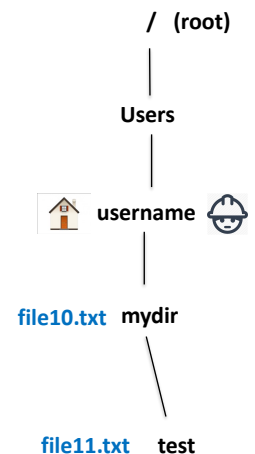
```
rmdir mydir
```

You will get an error message because mydir is not empty.

To remove non-empty directories, use option `-r` of the `rm` command.

```
rm -r mydir
```

This will remove mydir and all nested directories and files within mydir.



## 2.10 More on Shell error messages

As explained in the previous section, you can make many errors while typing Unix commands. Below are some other examples of error messages.

**Error message: No such file or directory**

When you get the error **No such file or directory** it means that UNIX shell searched for an argument name and could not find it. For the commands learned in this Chapter, the arguments can be a file name or a directory name.

Below are some examples of errors.

### Typos in file names and directory names.

For example, you want to move file3.txt to the work directory but you make a typo:

```
mv fil3.txt work
```

and you get this error:

```
mv: rename fil3.txt to work/fil3.txt: No such file or directory
```

This error means that fil3.txt does not exist. Indeed, the correct file name is file3.txt.

### Files and directories do not exist or are in another pathname

For example, if you want to copy a directory named HW1 from the Download directory to your current working directory and type:

```
cp -r ~/Download/HW1
```

and you get this error (if your username is mprocop2)

```
-bash: cd: /Users/mprocop2/Downloads/HW1: No such file or directory
```

This error means that directory HW1 is not in the Downloads directory. It is possible that HW1 does not exist, or it is somewhere else in the file system.

### Error message: Not a directory

The error **Not a directory** occurs when a command requires as an argument that is a directory but instead it gets a file.

For example, the command cd requires as an input argument a directory name. If you apply cd to a file named file.txt:

```
cd file.txt
```

you get this error:

```
-bash: cd: file.txt: Not a directory
```

### Error message: Is a directory

The error message **Is a directory** occurs when a command requires a file as an argument but instead it gets a directory. For example, if you wrongly use the command cat with a directory:

```
cat dir1
```

you get this error:

```
cat: dir1: Is a directory
```

There are Unix commands designed to act on files, and so their arguments are files, while other commands are designed to act on directories, and so their arguments are directories. Generally, error messages will help you to understand the type of errors you made. You should learn to interpret error messages to correct errors.

**You can use other commands with** pathnames. For example, if your current working directory is your home directory and the file long.txt is in your Desktop directory, you can view the contents of the file by using

relative pathname:

```
cat Desktop/long.txt
```

or absolute pathname

```
cat /Users/username/Desktop/long.txt
```

The cat command will work on the last element of the pathname, which is a file.

## 2.11 find command

The **find** command is useful for locating specific files within the File System.

**The find** command searches the directory tree starting from the directory specified by a pathname to locate files and directories.

*Find pathname options argument*

The find command will search starting from the directory specified by pathname, which can be relative or absolute.

Options

**-name** pattern match - file name or directory name. Only the last component of the pathname being examined matches pattern.

**-iname** pattern match. Like **-name**, but the match is case insensitive.

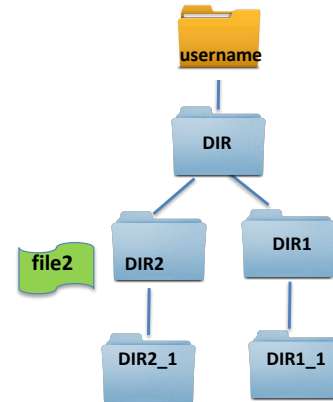
To practice some examples, make the directory structure depicted in the figure.

To find a regular file specify the option `-type f`. The dot means that the shell will start the search from the current directory.

```
find . -type f -name file2
```

To find a directory specify the option `-type d`. In this case the directory where to start the search is specified by an absolute path to home directory

```
find /Users/mprocop2 -type d -name DIR2
```



Now, let's find the location of the `pwd` command by starting the search from the root directory

```
find / -name pwd
find: /usr/sbin/authserver: Permission denied
/usr/share/zsh/5.8/help/pwd
/bin/pwd
find: /Library/Application Support/Apple/ParentalControls/Users: Permission denied
```

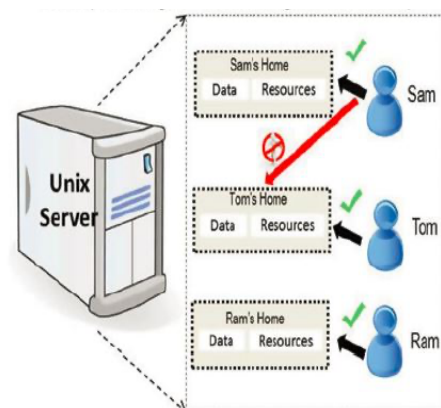
We will get lots of permission denied. Why? Because there are some security measures in place.

## 2.12 Security - File System Permission

UNIX based operating systems like OSX and Linux support multiple user accounts, i.e. multiple users can access the OS simultaneously. Users will log in to the filesystem via their own home directory. UNIX has a method called “File System Permission” which is used to protect users from accessing and modifying each other's files and directories.

UNIX supports multiple user accounts, and these accounts belong to some groups. The owner of a file or directory can decide which groups and users can read, write, and execute. Even the same read, write, and execute applies to the processes as the running processes belong to a particular user and group. Root users can read, execute, and write to any file or directory.

UNIX file system permissions are of two categories. Traditional File Permissions and Access Control Lists. By default, all files and directories will have traditional file permissions with them. We can apply ACLs to the files and directories, and the ACL will override the file permissions. ACLs are more advanced and have many more features than traditional file permissions. We would recommend you use file system permissions wherever you can to protect your files.



## file permissions, users, groups, and root

If you start to roam around the file system, you will soon find out you are unable to go just anywhere you please. This is because there are some security measures in place. These began not to prevent hackers from manipulating your system, but to prevent inadvertent destruction by the owner. When systems became multi-user and networked, additional security measures were needed. We will only touch the surface on how to secure your system.

The first level of security is contained in settings associated with files. The `-l` option (long format) for command `ls` lists these settings for a given file.

```
ls -l ~/Desktop/
total 696
drwxr-xr-x@ 10 maria  staff    320 Jan 12 11:54 Discovery-QB
drwxr-xr-x@  6 maria  staff    192 Jan 11 12:19 Faculty report
drwxr-xr-x@  9 maria  staff    288 Aug  9 16:36 NIH
-rw-r--r--@  1 maria  staff    475 Jan 20 12:06 file1.dat drwxr-xr-x@  7
maria  staff    224 Dec 14 22:44 trip
```

Displayed for the file1.dat file is:

```
-rw-r--r--      file mode
1              number of links
maria          owner name
staff          group name
475            number of bytes in the file
Jan 20 12:06   modification time
file1.dat      filename
```

File type

File size (in bytes)

File name

**-rw-r--r--@ 1 maria staff 475 Jan 20 12:06 file1.dat**

User Group Other

User Group

Permissions Owners

Last modification time

The *file mode* printed with the `-l` option consists of the file type (bit 1), permissions [owner (bits 2-4), group (bits 5-7), and other (bits 8-10)], and extended attributes if applicable (bit 11).

File type - We will only be concerned if the file is a regular file (-) or a directory (d).  
file1.dat is a regular file

Permissions - The next three fields are three characters each: owner permissions, group permissions, and other permissions.

Each field has three-character positions:

If **r**, the file is readable; if -, it is not readable.

If **w**, the file is writable; if -, it is not writable.

If **x** the file is executable; if -, it is not executable.

For file1.dat:

the owner can read and write the file

the group can only read the file

others can only read the file.

Extended attributes (+,@) are used in Mac OSX and often are a result of transfer of files or downloads.

## Changing permissions, chmod

Let's list only permissions for the file `file1.dat`:

```
ls -l file1.dat
-rw-r--r--@ 1 maria  staff  475 Jan 20 12:06 file1.dat
```

The file, *file1.dat*, gives 10 characters for the current mode. The first indicates the file type. The next 9 characters are 3 for owner, 3 for group, and 3 for world (or others). Here the owner has read and write privileges, the group and world have only read permissions.

To change permissions of a file, use the **chmod** command.

```
u user, g group, o others, a all
r read, w write (and delete), x execute (and access directory)
+ add permission, - take away permission
```

Permission	Files	Directories
r	can read the file	can ls the directory
w	can write the file	can modify the directory's contents
x	can execute the file	can cd to the directory

For example, to remove user permission to read the file

```
chmod u-r file1.dat
```

Now type

```
cat file1.dat
cat: file1: Permission denied
```

Give permission back

```
chmod u+r file1.dat
```

Other examples

```
chmod a+w file1.dat    # gives all permission to write
chmod g-r file1.dat    # removes group permission to read
chmod go-wr file1.dat  # removes w and r from g and o
chmod g-r,o-w file1.dat # removes r from g, and w from o
chmod u=rwx file1.dat  # give user permission to read write and execute
```

## Chapter 3. Shell scripts, Variables, and Metacharacters.

### Key commands and concepts in this chapter

Concepts	
Shell scripts	Text files where you write bash code
Metacharacters	Characters with special meaning in bash
Variables	Symbolic name for storing information
Arithmetic expansion	Evaluation of arithmetic operations in bash
Commands	
source or .	To run a script
echo	Print to screen

### 3.1 Make your first bash script

Instead of writing shell commands at the terminal prompt (i.e. use the shell interactively), you can write shell commands in a text file to make a shell script. A shell script is a text file containing shell commands. This saves time and prevents errors in typing when needing to repeat commands. In general, a program is a text file filled with a collection of commands. You will begin by writing your first program! In this case the file will contain shell commands that you have learned. For an interpreted language such programs are also called scripts. Thus, your first program will be a *shell script*.

Let's make our first shell script by using vi

```
vi script1.bash
```

Now add these two lines of code:

```
echo this is my first shell script
head -2 long.txt
```

You need to “**run**” the script at the terminal, so that the shell reads the file and executes the bash commands:

```
source script_1.bash
```

or

```
. script_1.bash
```

When a file is sourced (by typing either **source filename** or **. filename** at the command line), the lines of code in the file are executed as if they were printed at the command line. If the command line is bash shell, the script will be executed in bash.

You can also run your scripts with **bash**

```
bash script_1.bash
```

### Adding the class header to your script

Headers are often used in files to describe what is in them. A header will consist of one or more lines of text and will be distinguishable from the rest of the file.

Add the class header to your script as a comment, follow this format

```
#jhed:date:file name
```



For example:

```
#mprocop2:01/02/2020:script_1.bash
```

Add a header at the beginning of the file containing a # followed by username, date, filename, all separated by :colon or a comma. Notice the date. This is the date format to use in this class *mm/dd/yyyy*.

You can add additional comments to your script:

```
#mprocop2:01/02/2020: script_1.bash
# This is my first script in bash

echo this is my first shell script
head -2 long.txt
```

The class header must be in the first line of each file you submit to Gradescope for assignments.

### 3.2. Meta-characters and expansion

Meta-characters are characters with a special meaning instead of a literal meaning to the shell. You have already met the pound sign #. Some other meta-characters that we will talk about are: ~ \$ & \* ( ) [ ] { } ; ‘ “ < > \ \ ? !

The shell performs several different types of expansion depending on the type of meta-character. Those include filename and pathname expansion, brace expansion, variable expansion, and arithmetic expansion.

### 3.3 Variables and variable expansion

What is a variable? In general, in computing a variable is a symbolic name for information that is stored in memory locations. They are called variables because the stored information can change.

In UNIX, the information stored can be numbers and strings (a string is any finite sequence of characters (i.e., letters, numerals, symbols, and punctuation marks). You can recall from algebra, numbers can be referenced by a character or a string:

```
x=4
y=5
```

Once you have made a variable, the shell will use a metacharacter to recognize it as such. To see what the variable holds we use the echo command and a “\$”. The \$ sign merely tells the interpreter to expand the contents of the variable. It points (references) to the contents of the specified variable held in memory. This is another metacharacter because in the shell it has a special meaning.

```
x=4
echo $x
4
```

```
x=test
echo $x
test
```

### 3.4 Quotes

For extremely detailed information on how quotes should be used in bash, you may want to look at the "QUOTING" section in the bash man page. The existence of special character sequences that get "expanded" (replaced) with other values complicates how strings are handled in bash.

Enclosing characters in single quotes (') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash. Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of '\$', '\', and '!'. ' and '!'.

Another way to explore the use of quotes is to make a variable HW.

```
HW=Hello World
-bash: World: command not found
```

If you want to include two words separated by space you must use quotes otherwise you get an error message. Try this:

```
HW='Hello World'
echo $HW
Hello World
```

Type echo \$HW after each of these variable definition

```
HW="Hello World"
echo $HW
HW="Hello World!"
HW='Hello World!'
```

### 3.5 Escape character

Escape characters are used to remove the special meaning from a single metacharacter. A non-quoted backslash, \, is used as an escape character in bash.

```
echo $HW
echo \$HW
```

You can also use \ for removing the special meaning of whitespace characters in variable definition. Notice that you can echo a whitespace character:

```
echo Hello Word
Hello Word
```

You need to remove the special meaning of whitespaces in variable definitions, but not in echo statements.

```
HW=Hello Word
-bash: Word: command not found
```

```
HW=Hello\ Unix
echo $HW
Hello Word
```

### 3.6 Arithmetic expansion

The shell allows **integer arithmetic** to be performed by expansion. Arithmetic expansion takes the following syntax:

```
$((arithmetic expression))
```

You can perform simple integer math using shell syntax. Simply enclose the particular arithmetic expression between a `$((` and a `)` and bash will evaluate the expression.

```
Arithmetic operators:
+ addition
- subtraction
* multiplication
/ division (integer division rounded down)
% modulo (remainder)
** exponentiation
```

Here are some examples:

Open a terminal and type in this bash code:

```
echo $(( 100 / 4 ))
25
```

```
echo $(( 2+3 ))
5
```

```
a=23
echo $a
23
```

```
b=$(( a-2 ))
echo $b
21
```

```
c=$(( (a-2)/2 ))
echo $c
5
```

Now make a script using vi called `myscript.bash` and insert in it all the lines of bash code listed above. Run your script.

### 3.7 Brace expansion { .. }

You can run a command on more than one argument to create multiple files or directories, or to list contents

```
mkdir dir1 dir2 dir3 dir4 dir5
touch file1 file2 file3 file4
ls -l file1 file2 file3 file4
ls -l dir1 dir3
```

Instead of doing that above, you can use brace expansion to generate multiple command line arguments out of a single argument.

```
mkdir dir{1..5}
touch file{1..4}
ls -l file{1..4}
ls -l dir{1,3}
```

*To generate a sequence of numbers or letters: {start..end} (double dots separate start and end)*

```
{0..12} expanded is 0 1 2 3 4 5 6 7 8 9 10 11 12
{a..g}  expanded is a b c d e f g
```

*To generate a specific list of items, use comma {item1,item2,item3}*

```
{aa,1,cc,3} expanded is aa 1 cc 3
```

*If the brace expansion has a prefix or a suffix string, those strings are included in the expansion:*

```
a{0..3}b expanded is a0b a1b a2b a3b
```

*Brace expansions can be nested:*

```
{a,b{1..3},c} expanded is a b1 b2 b3 c
```

Try these:

```
echo {5..-2}
echo a{1,3,5}b
```

With brace expansion you can **create multiples files and directories**, and you can match existing files and directories

```
touch file_{1..11}.dat
ls -l file_{1,7}.dat
mkdir dir{Z..P}
```

### 3.8 Filename and pathname expansion – wildcards (\*, ?, [])

The shell allows the use of *metacharacters* for pattern recognition in filenames (filename expansion). It will supply the list of all filenames and pathnames matching the given pattern. Pathname and filename expansion work on existing files and directories.

#### Metacharacters called Wildcards:

*	- matches any string, including the null string
?	- matches any single character
???	- matches any three characters
[xyz]	- matches any one of the characters within the brackets.
[!xyz]	- matches any characters except the ones contained within the brackets.

When one of these metacharacters appears in the argument of a Unix command, the shell expands that argument by generating a list of names that match the names of **existing files, directories, and pathnames** and passes the list to a command. These special characters are also called wildcards because they act as the jokers do in a deck of cards.

By using them you can quickly refer to a group of e.g., files with similar names, saving the effort of typing the names individually.

#### The all-important *wildcard* \*

You might be used to using the mouse or find to select multiple files in a GUI interface. Within the UNIX environment, groups of files can be accessed by using special characters. The wildcard, \*, matches any number of characters.

Example

```
ls *.dat
```

The **ls** command above will list all files ending in *.dat* in the current working directory. The shell expands the wildcard \* into a list of filenames ending with *.dat*.

Example

```
rm *.txt
```

The **rm** command above will delete all files ending in *.txt* in the current working directory. The shell expands the wildcard \* into a list of filenames ending with *.txt*. **Be very careful when you use the wildcard \* with the commands rm and rmdir because you risk removing all your files and directories in one command line.**

Type the following lines of code:

```
touch feb86 jan12.89 jan19.89 jan26.89 jan5.89 jan85 jan86 jan87 jan88
touch file{1..10}.dat
touch file{5,10,45}.txt
```

This will create files for use in the examples below.  
Guess the output from the following commands:

```
ls *
ls jan*
```

```
ls ?????
ls feb??
ls *[*0-9]
ls [A-Z]*
ls *8*
ls [fjm][ae][bnr]*
rm feb??
rm *.89
ls ???8[56]
rm /Users/username/jan??
```

Try these:

```
ls *.txt
ls file??*.dat
ls file?.dat
ls file*
ls Desktop/*
ls /bin/?? #list all files in /bin directory that have only 2 characters in the
filename
ls /bin/???
ls /bin/[ab]??? #list all files in /bin directory that have only 4 characters
in the filename and the first character is either a or b
ls /bin/[!ab]?? #list all files in /bin directory that have only 4 characters
and the first character is neither a or b
```

Wildcards allow you to specify pathnames

Example

```
ls ../*
```

```
ls /Users/username/*
```

If you use absolute paths in your wildcard, bash will expand the wildcard to a list of absolute paths. Otherwise, bash will use relative paths in the subsequent word list.

### 3.9 Shell error messages: Syntax errors

Syntax errors occur when you do not follow the bash syntax. For example, to perform shell arithmetic you must enclose the particular arithmetic expression between `$((` and `)`.

Try to type:

```
myvar=$(( myvar - 1 ))
```

and you get this error

```
-bash: syntax error near unexpected token `)'
```

Can you figure out where the syntax error is?

## Chapter 4. Data files, text processing, formatting and back up

### Key commands and concepts in this chapter

Commands	
wc	Provide information about a file
grep	Find and extract a pattern from a file
sort	Sort content of a file
cut	Cut columns or fields
cat, paste	Join files
tr	Translate characters
sed	Translate characters and text
printf	Command for formatting output

### 4.1 Data files

Big data sets require storage and formats that permit easy data extraction. When a program extracts portions of data from a file to be printed or used for another purpose according to a ruleset, we call that *parsing* a file.

### Fields and separators

Generally, a Dataset is organized in fields. A **field is a unit of information** and can contain either numeric or non-numeric data arranged in rows or columns. A field can be a group of columns or rows. In the example below, the first field is the first name, and it is contained in 4 columns. One field can also be one column or one row. Usually, fields are arranged in columns. In this course we will refer to fields as a unit of information arranged in columns.

In the following example the data file has 5 fields separated by 2 spaces. In this example the fields are:

1. First name
2. Last name
3. birth year
4. career
5. area of focus

```
Jane Bolden 1932 author economics
John Talbot 1945 poet english
```

Here is another example:

```
Jane,Bolden,1932,author,economics
John,Talbot,1945,poet,english
```

In this example the **field separator** is a comma (,)

We will use field separators in several different ways this semester. The first is the use of a colon to separate fields in a header for files. This can be used to parse (extract particular information) from the files you create for various purposes such as searching and making tables of contents.

Now that you can work with files it is time to manipulate the data within the files. Just as in a word processing program you are used to using you can manipulate text in a file with UNIX commands. We will

explore commands that pull out words or phrases and commands that replace them with different text. You will find out how to pull out columns and rows and how to put various parts back together.

## File size

The size of a file or the amount of memory (in bytes) taken up by the file is determined by the number of characters in the file. To see this value, you can use the `ls` command. If a character takes 8 bits or 1 byte your total file size should reflect the number of characters.

Make an empty file and check its size using the `ls` command.

Now add 1 character to the file using `vi`. Did the size change?

when file was empty: file size = 0 byte.

when one character: file size = 2 bytes.

when two characters: file size = 3 bytes.

Why is there an extra character? It is due to the `vi` editor placing a character at the end of each line.

For each directory whose contents are displayed by the `ls` command, the total number of 512-byte blocks used by the files in the directory is displayed on a line by itself, immediately before the information for the files in the directory.

TRY IT:

How much space does your HOME directory use? (find the “total” using the “long” format)

How would you produce a listing of files by increasing size? Try `man` on `ls` command.

Another way to check file and directory size is the command `du`.

```
du -sh
```

Can you make sense of the output?

Listing the size of files in UNIX can be a bit confusing. There are several ways to go about doing this (`ls` and `du` for example). Depending on the command usage the output may be rounded up, it may refer to the actual file size or it may refer to the actual disk space reserved for the file. These considerations can lead to large discrepancies in the values reported for large files and/or many files.

## Quotas

As you use your computer you will need to check *quotas* or keep track of your total disk usage. The command to show disk usage is `df`

```
df -h
```

If you have limited disk space, you may need to start archiving data.



## 4.2 Text processing commands

One of the things that makes the shell an invaluable tool is the amount of available text processing commands, and the ability to easily pipe them (see next Chapter 5) into each other to build complex text processing workflows.

These commands can make it trivial to perform text and data analysis, convert data between different formats, filter lines, etc. and in general extract information from a data set.

When working with text data, the philosophy is to break any complex problem you have into a set of smaller ones, and to solve each of them with a specialized tool.

<b>wc</b>	get info on a data file
<b>grep</b>	extract lines matching a pattern
<b>cut</b>	extract columns or fields
<b>sort</b>	sort lines of text files
<b>cat</b>	join file vertically
<b>paste</b>	join file horizontally
<b>sed</b>	transform text

Before we start:

- Go on Canvas and download the zip file **data-temp.zip**, which is in the DATA folder
- Put the data-temp.zip into your home directory
- Unzip the file data-temp.zip and a directory called data-temp will be created.
- Open a terminal and cd to data-temp directory
- List the content of data directory. You will find the data files temp.dat, temp-clean.dat and temp-clean1.dat.

These files, taken from the EPA website, contain information on the temperature anomaly for the Southwestern United States.

They are organized as follows:

1<sup>st</sup> field is the station ID

2<sup>nd</sup> field is the state code

3<sup>rd</sup> field is the temperature anomaly

The first thing you might want to do before processing text of a data file is to get information on the size and number of lines of a data set, and then to use viewing commands (such as cat, more, less, head, tail) to look at the content of the file you are going to process.

### wc

**wc** will print information about the file

```
wc temp.dat
wc -l temp.dat
```

Which information does **wc** display to screen? What does the option -l do?

Use the man page to help you decipher the output.

## grep command

The **grep** command allows you to search one file or multiple files for lines that contain a pattern. If it finds that pattern on any line, the line will be sent to the output (the screen).

We can use **grep** to extract the lines of the file temp.dat that contain pattern AZ.

```
grep AZ temp.dat
201 AZ 1.602087638
202 AZ 1.480707645
203 AZ 1.78030303
204 AZ 1.806344697
205 AZ 1.889656508
206 AZ 1.90334022
207 AZ 1.503546832
```

### Options of the grep command

*Print the lines excluding the pattern using the -v option*

```
grep -v pattern filename
```

*Count the number of matching patterns using the -c option*

```
grep -c pattern filename
```

*Search for multiple patterns using the -e option (search for pattern1 OR pattern2)*

```
grep -e pattern1 -e pattern2 filename
```

Try these:

```
grep -e AZ -e CO temp.dat
grep -v CO temp.dat
grep -c CO temp.dat
```

## sort

The **sort** command will sort the contents of a file.

Options can include specific fields and a sequence of fields and sorts on numeric or alpha characters.

```
sort (options) filename
```

Useful options of the sort command:

```
-u          sort and remove duplicates
-knumber   sort lines based on a certain field number
-n         sort numerically (default it will sort alphabetically)
-tsep      specify field separator
```

The -t option is useful when fields are separated by something other than blank spaces.

What is the -k option for?

Use the command `sort` to sort based on the 1st field numerically

```
sort -k1 temp-clean.dat
```

Does this work? With the option `-n` `sort` will sort numerically

```
sort -nk1 temp-clean.dat
```

Use the command `sort` to sort alphabetically based on the second field

```
sort -k2 temp-clean.txt
```

### Field separator is different than whitespace characters

If fields in a file are separated, for example, with a comma, you would add the `-t` option, and specify the field separator:

```
sort -t, (other options) filename
```

This code below will sort `filename` numerically based on the 3<sup>rd</sup> field, with field separator being the colon

```
sort -t: -nk1 temp-clean1.dat
```

## cut

Whereas the **grep** command works on lines, you can extract specific columns or fields using **cut**. With the **-c option** you can specify character positions and specify the columns that you want to extract:

Try these

```
cut -c 1-3 temp-clean.dat
cut -c 1-5 temp-clean.dat
cut -c 6 temp-clean.dat
cut -c 6,8 file.txt
```

With the options **-f** and **-d** you can extract specific fields (**-f**). You should specify the field separator *sep* (**-dsep**) if it is different than tab.

The syntax for extracting a selection based on a field number and field separator is:

```
cut -d "sep" -f n filename
```

Extract the second field, and specify the field separator being one whitespace

```
cut -d" " -f2 temp-clean.dat
```

## paste

The `paste` command will horizontally concatenate files. Make a file called `fileA` and in it write `Hello`. Make another file called `fileB` and in it write `World`.

Try this:

```
paste fileA fileB
Hello World
```

Now use paste to obtain this output:

```
World Hello
```

Look at the man page of the paste command to see options. Try these:

```
paste -s fileA fileB
Hello
World
```

```
paste -d "," fileA fileB
Hello,World
```

## cat

The cat command is also used to vertically concatenate files

```
cat fileA fileB
Hello
Word
```

## sed

**sed** is a *stream editor* used to perform basic text transformations on an input stream (a file, or input from a pipeline). **sed** can perform many functions on files like searching, find and replace, insertion, or deletion. The most common use of sed command in UNIX is for substitution or for find and replace. By using sed you can edit files even without opening it, which is a much quicker way to find and replace something in a file than first opening that file in vi editor and then changing it.

**sed** takes the input text, does the specified operations on every line (unless otherwise specified) and prints the modified text to standard output (screen). The specified operations can be append, insert, delete, or substitute.

Make a file coffee.txt and write in it *I love tea*

Now you can use sed to replace the word *tea* with *coffee* and print the modified text to screen:

```
sed 's/tea/coffee/' coffee.txt
I love coffee
```

The **sed** command will perform substitution of text.

```
sed 's/text/replace/g' file
```

The substitute flag /g (global replacement) specifies the sed command to replace all occurrences of *text*. Without the g only the first instance on a given line would be replaced.

To delete a word use,

```
sed 's/tea//g' coffee.txt
```

By default, `sed` prints modified text to screen (stdout), so you may want to use the shell's redirect operator (see next Chapter) to save the modified text to a new file, like this:

```
sed 's/tea/coffee/' coffee.txt > tea.txt
```

Remember that `sed` does not modify the file, but it will output the modified text to screen

```
cat coffee.txt  
I love tea
```

### 4.3 `tr` and input redirection (<)

Before reading this paragraph, read Chapter 5.

The `tr` utility copies the standard input to the standard output with substitution of selected characters.

```
tr [options] charset1 charset2
```

The `tr` command does translation. It takes two sets of characters and replaces occurrences of the characters in the first set with the corresponding elements from the second set.

Use `tr` to translate all lowercase characters.

What does the following command do?

```
tr a-z A-Z
```

`tr` reads the standard input (the keyboard), and on receiving the "end of file" (^D), copies it to the standard output (the screen).

In UNIX, we can redirect both the input and the output of commands. Try this:

```
tr a-z A-Z > file  
type a word with lowercase letters  
type [ctrl-D]  
type cat file
```

`tr` takes input from a file by using input redirection. Use the file you just made as input to `tr`:

```
tr A-Z a-z < file
```

What is the result?

Redirect both standard input and output:

```
tr a-z A-Z < file > file1
```

You can also use `tr` in pipeline. Example:

```
echo gana  
echo gana | tr 'a' 'd'  
echo gana | tr 'an' 'di'
```

In this example we translate a blank character with a new line. A new line is represented by this character `\n`

```
echo hello hello | tr ' ' '\n'
```

What does the option `-s` of the command do? Try this

```
echo heeeeelloe | tr -s 'e'
```

## Difference between `tr` and `sed`

```
sed s/word1/word2/g filename
```

```
tr (options) charset1 charset2 < filename
```

If you do this example, you will think that `tr` and `sed` are quite similar:

```
echo Test+for+tr+and+sed | tr '+' ' '
Test for tr and sed
```

```
echo Test+for+tr+and+sed | sed 's/+/ /g'
Test for tr and sed
```

However, if you do this example, you will realize that they are different:

```
echo good | tr 'good' 'best'
bsst
```

`tr` has done character-based transformation and it is replacing `good` to `best` as `g=b`, `o=e`, `o=s`, `d=t`

```
echo good | sed 's/good/best/g'
best
```

## 4.4 `printf` for formatting text

`printf` stands for formatted print. With the `printf` command you have more control on the appearance of your output.

Example

```
var1=mass
var2=18.547
echo $var1 $var2 Kg
mass 18.547 Kg
```

```
printf "%-8s %.2f Kg\n" $var1 $var2
mass      18.55 Kg
```

General format of printf is:

```
printf "[%width].[precision]type" argument
```

printf formats and prints argument(s) under control of the format(s)

The **type** character specifies whether the corresponding argument is to be interpreted as a character, a string, an integer, or a floating-point number. width and precision are optional modifiers.

### Specify the type

```
%d decimal integer
%f floating point number
%s string
%e exponential- scientific notation
\n is for new line
```

Try these:

```
printf "%f\n" 1.6547
printf "%e\n" 1250000
printf "%d\n" 2
printf "%s\n" Two
```

### Specify precision (optional modifier)

The **precision** specification consists of a period (.) followed by a non-negative integer that, depending on the conversion type, specifies the number of string characters, the number of decimal places, or the number of significant digits to be output.

the **precision** modifier is a positive integer number which follows the dot  
**%.precisiontype**

**%.precisions** for string type specifies the number of string characters to output

**%.precisionf** for float type specifies the number of decimal places to output

**%.precisione** for scientific notation type specifies the number of decimal places to output

Try these:

```
printf "%.2f\n" 1.6547 #format float with 2 decimal places
printf "%.3f\n" 1.6547
printf "%.0f\n" 1.6547
printf "%.3e\n" 1250000 #format in scientific notation with 3 decimal
places
printf "%.1s\n" Two #output 1 character
printf "%.2d\n" 2
```

### Specify width (optional modifier)

The **width** specification is a non-negative integer that controls the number of characters that are output.

The **width** modifier specifies the number of characters to print.

**% width.precisiontype**

The **width** is an integer number, which precedes the dot.

If the width is larger than the number of characters of the output, it will add whitespace characters to the left.

With a Negative integer, whitespace characters are added to the right.

**%-width.precisiontype**

Try these:

```
printf "%.2f\n" 1.6547
printf "%5.2f\n" 1.6547
printf "%6.2f\n" 1.6547
printf "%3s\n" Two
printf "%4s\n" Two
printf "%5s\n" Two
printf "%-5s\n" Two
printf "%-6.2f Kg\n" 1.6547
```

### Define type and precision of multiple arguments and include text

```
printf "format1 format2" argument1 argument2
```

```
printf "%w.pctype1 %w.pctype2" argument1 argument2
```

```
printf "%-10.4s %.2f\n" Temperature 1.6547
Temp          1.65
```

The text you include within the " " will be printed to screen, including the space characters

```
printf "Mass %.2f .. in %s\n" 65.4747 Kg
Mass 65.47 .. in Kg
```

To see a detailed description and available formats, use the man pages. Several examples can be found at [https://www.gnu.org/software/gawk/manual/html\\_node/Printf.html#Printf](https://www.gnu.org/software/gawk/manual/html_node/Printf.html#Printf)

## 4.5 Back up

The simplest way to backup data is to use the copy command, **cp**. For example, make a *local* backup directory, BU:

```
mkdir $HOME/BU
cp filename $HOME/BU/filename
```



**FLASH drive backups:** When you insert your flash drive, you will be able to see its name and content in the /Volumes directory. First, to make a backup on a different local disk use `cp` as follows:

```
ls /Volumes/
```

In addition to Macintosh HD, you should see another directory, e.g., *my\_flash* or *USB DISK*, or a directory with your name. Let's assume that the name of your flash drive is *my\_flash*. If that's the case use the following command to back up another directory (e.g., your DATA directory) to your flash drive:

```
cp -r $HOME/DATA /Volumes/my_flash/
```

If the name of your flash drive is different, replace *my\_flash* in the above command with the name of your flash drive.

**Remote drive backups:** For copies to be run over the internet use “remote” copy, `rcp` instead of `cp`. And preferably “secure” copy, `scp` instead of `rcp`.

```
scp -rp gbpc13@10.160.112.218:file1 ./ #use password xxxxxx
scp -rp ./file1 gbpc13@10.160.112.218: #use password xxxxxx
scp -rp gbpc13@10.160.112.218:nobel.tar.gz ./
```

Use the man pages to find out what the `r` and `p` options stand for.

More complete backups can be used with an incremental backup program. The UNIX **rsync** command is awesome at doing backups and even remote copies. It is like `scp` but has a ton more bells and whistles. If you make changes to the files, it will just replace the parts that were changed, speeding up the file transfer significantly. As with all UNIX commands, many options govern the use of the command. Read the man page to get an idea of the power in **rsync**.

Example 1. To back up your DATA directory to a “mounted” disk (this could be your **flash drive** or another mounted disk or another place in the current file system):

```
rsync -rp $HOME/.bash* /Volumes/my_flash/
```

To restore these files from the “mounted” disk:

```
rsync -rp /Volumes/my_flash/*.* $HOME/
```

Example 2. To back up your DATA directory to a “remote” disk:

```
rsync -rp $HOME/DATA username@remote_system:
```

And to restore these files from the “remote” disk:

```
rsync -rp username@remote_system:DATA/ $HOME/DATA/
```

## Compression and archiving

If your files are large you might want to compress them. To compress a file use **gzip**:

```
gzip filename
```

This will produce a file with a .gz extension, i.e., filename.gz  
To un-compress a file use gunzip:

```
gunzip filename.gz
```

An archive is a single file that contains any number of individual files plus information to allow them to be restored to their original form by one or more extraction programs. To archive a folder use **tar** (tape archive). Here (c for create):

```
tar -cvf filename.tar folder_name
```

To un-archive (untar) a file using the same command, tar, but different options (x for extract):

```
tar -xvf filename.tar
```

For archiving and compression, you can also use the zip command.

```
zip filename.zip file1 file2 file3
```

This will compress and archive file1, file2, and file3. To zip all of the files in a folder (e.g., dir1) do:

```
zip dir1.zip dir1/*
```

Note that the above command will NOT zip any hidden files, nor will it zip files in directories that are inside the dir1 directory. To zip hidden files and files in directories that are inside dir1, use the -r (recursive) option of the zip command.

```
zip -r dir1.zip dir1/
```

To recover original files, use unzip:

```
unzip filename.zip
```

Check the contents of the zip file:

```
unzip -l filename.zip
```

## Saving data

How will you backup your data? It will be your responsibility to be sure your data is backed up. You should use a flash drive. At various times we may also use JShare, a server, email, or some combination of these. One should always have backup mechanisms in place. Notice the use of plural. If you do not have a flash drive yet, please email your files to yourself just in case they are deleted. The following is an example of using the terminal to copy data to your flash drive. Where you see *filename*, insert the name of the file you want to save. Where you see FLASH, insert the name of your flash drive.

*NOTE: If the name of your flash drive has a space in it, click on its icon and rename.*

Insert your USB FLASH drive.

From your home directory type:

```
ls /Volumes  
Macintosh HD      FLASH
```

Using the output from the `ls` command, which gives your flash drive name, type:

```
cp filename /Volumes/FLASH/
```

To copy a directory type:

```
cp -r directory_name /Volumes/FLASH/
```

## 5. Data stream manipulation

### Key commands and concepts in this chapter

Concepts	
standard input, output, error	Streams of data
redirection	Specify where output goes from commands
pipe	Send output from one command to another as input
Commands	
cat	Writes files to standard output

In UNIX, a terminal by default contains three streams, one for input and two output-based streams.

The input stream is referred to as `stdin` and is generally mapped to the keyboard.

The standard output stream is referred to as `stdout`, and generally prints to the terminal.

The other output stream `stderr`, primarily used for status reporting (error messages), usually prints to the terminal like `stdout`.

### 5.1 Standard input, output, and error (`stdin`, `stdout`, `stderr`)

Many commands such as “`ls`”, “`echo`” or “`pwd`” produce some sort of output. This output is sent into a special file called “standard output” (`stdout`). By default, standard output is sent to screen. For example, if you use the command **echo**:

```
echo Hello
Hello
```

Output will be generated and shown on the screen. In this case, the command `echo` was executed successfully.

However, if you type **echos**, another output will be produced:

```
echos Hello
-bash: echos: command not found
```

In this case the output shows that the command was not executed successfully. Status messages (such as errors) are sent into another special file called “standard error” (*stderr*). Standard error is also sent to the screen by default.

Many programs take input from a device called standard input (*stdin*) which is attached to the keyboard.

DESCRIPTION	Name	default	Produced from
Standard input	stdin	keyboard	keyboard or program
Standard output	stdout	screen	program
Standard error	stderr	screen	program

## 5.2 Redirecting standard output (>)

The **redirect** sign > redirects the standard output of a command from the screen to a file. This is a way to save the output of a command into a file.

TRY IT:

```
echo Hello World! > data.txt
cat data.txt
Hello World!
```

Make this file.txt

```
1960 chemistry
1979 medicine
1988 physics
1998 economics
1998 peace
2005 medicine
2007 peace
1902 physics
1923 medicine
1945 literature
1960 chemistry
1979 medicine
```

```
grep physics file.txt > data.txt
cat data.txt
1988 physics
1902 physics
```

What does a double redirect, >>, do? This command will append a file and not overwrite it.

```
echo Hello World! >> data.txt
cat data.txt
1988 physics
1902 physics
Hello World!
```

### 5.3 Redirect standard error and standard out

Standard error can also be redirected to a file:

```
command 2> error.log
```

Both stderr and stdout to a file:

```
command &> file.out
```

Standard error to the so-called null device:

```
command 2> /dev/null
```

Both stderr and stdout to the null device:

```
command &> /dev/null
```

The command above is redirecting standard output into /dev/null, which is a place where you can dump anything you don't want to see or keep at the moment. The command also redirects standard error into standard output (you have to put an & in front of the destination when you do this). This command will not produce any output on the screen, it is very "quiet".

### 5.4 Redirecting standard input (<) and more about the cat command

The command **cat** will send its output by default to standard out, the screen. Try running **cat** by itself, i.e., type cat and hit enter:

```
cat
```

now type some text and press [return]

What does the **cat** command do?

The command takes standard input from the keyboard (the text you type) and concatenates it. The result is it repeats what you typed. This doesn't seem so useful; however, the next command will perhaps give you some ideas as to how to make it useful. The command cat will keep displaying what you typed until you hit Ctrl-D.

Try the following:

```
cat > file # using the cat command create a file for input
a[CTRL+D] [CTRL+D]
# input one character 'a' and exit the process [CTRL+D]
# first to end the file and the second is to stop the cat command.
```

```
cat > welcome.txt
Hello World!
[Cntl-D]
cat welcome.txt
Hello World!
```

More examples:

```
echo hello world > output
cat < output
```

The first line writes "hello world" to the file "output", the second reads it back and writes it to standard output (normally the terminal). In essence this redirect sends the input to the command cat.

Another example of standard input redirection is provided in Chapter 4 with the tr command.

## 5.5 Pipe |

A *pipe* | is a form of *redirection* that is used to send the output of a command to another command for further processing. Pipes are used to create what can be visualized as *a pipeline of commands*.

By using the pipe operator | the output text (*stdout*) of one command can be piped into the input (*stdin*) of another command

```
command (options) (arguments) | command (options)
```

**Commands after the first pipe do not have an argument.**

Use file.txt

```
1960 chemistry
1979 medicine
1988 physics
1998 economics
1998 peace
2005 medicine
2007 peace
1902 physics
1923 medicine
1945 literature
1960 chemistry
1979 medicine
```

```
grep medicine file.txt | head -1
```

In the example above the output of the grep command is piped (|) to the head command.

```
ls | wc
```

In the example above the output of the ls command is piped (|) to the wc command.

Try this - we use grep to extract all the lines of the file.txt containing the keyword physics, and then we sort numerically the output of grep:

```
grep physics file.txt | sort -n
```

The output of **grep physics** is **piped** (|) to the command **sort -n**. You can notice that several lines contain the keyword **physics**. Maybe now you only want to show the first line. You can use the pipe multiple times to extract exactly the information that you want:

```
grep physics file.txt | sort -n | head -1
```

You can also use redirection to save the output of a pipeline of commands into a file:

```
grep physics file.txt | sort -n > file-sorted.txt
cat file-sorted.txt
```

## 5.6 Command substitution

The standard output of execution of commands can be redirected into a variable.

```
mypwd=`pwd`
echo $mypwd
```

The command to be executed is enclosed in *the back quotes*. Those are not standard single quotes, but instead come from the keyboard key that normally sits above the Tab key.

As you can see, bash provides multiple ways to perform exactly the same thing. Using command substitution, we can place any command or pipeline of commands in between `` and assign it to a variable. Here's an example of how to use a pipeline with command substitution:

Example

```
var=`grep physics file.txt | sort -n | head -1`
echo $var
```

## 6. Advanced text processing with AWK

### Key commands and concepts in this chapter

Concepts	
awk	Programming language that acts on patterns, and is also a tool for processing rows and columns of a data file

### AWK

Another text processing tool often used for command line data extraction is AWK. AWK is a programming language which acts on patterns. A number of people also utilize the scripting features of AWK. The language was developed at Bell Labs in the 1970's (a powerhouse of computer architecture design). The name derives from its authors: Alfred Aho, Peter Weinberger, and Brian Kernighan. Like grep, AWK processes text but is not limited to lines of text.

Since AWK is a programming language, execution of AWK commands can be via the command line or an AWK program. To invoke **awk** within bash

```
awk 'condition {print action}' filename
```

To learn how **awk** works, make a file called file.dat with spaces as field separator.

```
205 AZ 1.889656508 AZ
206 AZ 1.90334022 P
207 AZ 1.503546832 AZ
401 CA 1.232179752 O
402 CA 1.138502927 A
501 CO 1.280255682 B
502 CO 1.808518423 V
503 CO 1.571410124 P
```

To search for pattern *AZ*, invoke **awk** like this:

```
awk '/AZ/ {print}' file.dat
205 AZ 1.889656508 AZ
206 AZ 1.90334022 P
207 AZ 1.503546832 AZ
```

Here *AZ* is the pattern, and *print* is the action. The search pattern is enclosed between two slash (/) characters. In this case, **awk** works like the **grep** command, i.e., one could have simply done:

```
grep AZ file.dat
```

However, **awk** can read a line and split it into fields. With **awk** one can extract lines based on **conditions** that you specify on fields, and you can print specific fields.

Fields are specified as following:

\$1 first field.

\$2 second field.

\$n nth field.

Whitespace character(s) or tab(s) is the default separator between fields in **awk**.

AWK is a programming language and the **awk** syntax is different than **bash** syntax. Here the \$ followed by a number identifies a field. Do not confuse the dollar in **awk** with the dollar in **bash**.

### Extract lines based on **conditions** you define on fields

Print only lines that contain keyword *AZ* in the 4<sup>th</sup> field:

```
awk '$4 == "AZ" {print}' file.dat
205 AZ 1.889656508 AZ
207 AZ 1.503546832 AZ
```

Print only lines that contain a number greater than 1.8 in 3<sup>rd</sup> field:

```
awk '$3 > 1.8 {print}' file.dat
205 AZ 1.889656508 AZ
206 AZ 1.90334022 P
502 CO 1.808518423 V
```

The syntax to apply conditions to numeric fields, or text fields is the following:



**Operators for numbers**

```

== is equal to
!= is not equal to
< less than
> greater than
<= less than or equal
>= greater than or equal

```

```

Syntax to define condition
$field == number
$field >= number

```

**Operators for strings**

```

== is equal to
!= is not equal to

```

```

Syntax to define condition
$field != "string"
$field == "string"

```

**Strings are enclosed within double quotes**

**Specify multiple conditions, use logical AND and OR**

You can use logical operator to extract lines satisfying multiple conditions.  
The syntax is the following:

AWK uses the following logical operators:

```

&& (AND)
|| (OR)

```

```

conditionA && conditionB
conditionA || conditionB

```

A	B	A    B	A && B
False	False	False	False
True	False	True	False
False	True	True	False
True	True	True	True

For example:

Print all the lines of the file.dat that contain pattern AZ or pattern CO in its 2<sup>nd</sup> field:

```

awk '$2 == "AZ" || $2 == "CO" {print}' file.dat
205 AZ 1.889656508 AZ
206 AZ 1.90334022 P
207 AZ 1.503546832 AZ
501 CO 1.280255682 B
502 CO 1.808518423 V
503 CO 1.571410124 P

```

Print all lines of file.dat that contain AZ in its 2<sup>nd</sup> field and temperature (3<sup>rd</sup> field) is greater than 1.9:

```

awk '$2 == "AZ" && $3 > 1.9 {print}' file.dat
206 AZ 1.90334022 P

```

Print all lines of file.dat that contain CA or CO in its 2<sup>nd</sup> field and in both cases temperature (3<sup>rd</sup> field) is greater than 1.2. Watch out for the order of operations:

```

awk '($2=="CA" || $2=="CO") && $3 > 1.2 {print}' file.dat
401 CA 1.232179752 O
501 CO 1.280255682 B
502 CO 1.808518423 V
503 CO 1.571410124 P

```

In case that both operators get specified, && gets performed first, unless you enclose || within ()

If you remove the ( ) and && gets performed first, you get a different output.  
This will select lines containing CO with temperature > 1.2, or lines containing in 2<sup>nd</sup> field CA.

```
awk '$2 == "CA" || $2 == "CO" && $3 > 1.2 {print}' file.dat
401 CA 1.232179752 O
402 CA 1.138502927 A
501 CO 1.280255682 B
502 CO 1.808518423 V
503 CO 1.571410124 P
```

## Print specific fields

```
awk 'condition {print action}' filename
```

If condition is not specified, awk will match *all* lines in the input file, and perform the print on each one.

Print all 2<sup>nd</sup> and 3<sup>rd</sup> field of all lines:

```
awk '{print $2, $3}' file.dat
AZ 1.889656508
AZ 1.90334022
AZ 1.503546832
CA 1.232179752
CA 1.138502927
CO 1.280255682
CO 1.808518423
CO 1.571410124
```

If a condition is specified, awk will extract lines matching that condition, and perform the print on those lines.

Print the 2<sup>nd</sup> and 3<sup>rd</sup> field of all lines containing AZ in 4<sup>th</sup> field:

```
awk '$4=="AZ" {print $2, $3}' file.dat
AZ 1.889656508
AZ 1.503546832
```

## Perform arithmetic operations on fields

You can perform arithmetic operations on fields within the {print action}.

```
+ addition
- subtraction
* multiplication
x**y (x^y) exponentiation
```

Print the sum of 1<sup>st</sup> and 3<sup>rd</sup> field of all lines

```
awk '{print $1+$3}' file.dat
206.89
207.903
208.504
402.232
```

```
403.139
502.28
503.809
504.571
```

Multiply 3<sup>rd</sup> field by 100 only for lines containing pattern AZ in 2<sup>nd</sup> field:

```
awk '$2=="AZ" {print $3*100}' file.dat
188.966
190.334
150.355
```

You can add text in the print action within double quotes. Separate text and fields by commas in the print action.

```
awk '$2=="AZ" {print $2, "Temp", $3*100, "C"}' file.dat
AZ Temp 188.966 C
AZ Temp 190.334 C
AZ Temp 150.355 C
```

### awk with field separators different than blank space:

When the field separator is not the blank space you have to specify the field separator by using the option `-F`:

**awk** `-F separator 'condition { print actions }'`

Example: data file file1.dat with comma as field separator

```
205,AZ,1.889656508
206,AZ,1.90334022
207,AZ,1.503546832
401,CA,1.232179752
402,CA,1.138502927
501,CO,1.280255682
502,CO,1.808518423
503,CO,1.571410124
```

Extract all lines that contain CO in the second field and print all fields:

```
awk -F, '$2=="CO" {print}' file1.dat
501,CO,1.280255682
502,CO,1.808518423
503,CO,1.571410124
```

Extract all lines that contain CO in the second field, and print only the 3<sup>rd</sup> field:

```
awk -F, '$2=="CO" {print $3}' file1.dat
1.280255682
1.808518423
1.571410124
```

## awk with printf

You can use `printf` instead of `print` to format the output of `awk`.

**awk** `'condition {printf "formats", arguments}' filename`

The format specification is the same as `printf` in `bash`. In `awk`, arguments are separated by commas, and an argument in `awk` could be a field, text, or calculations. In `bash`, arguments are separated by spaces.

format is given by `"%[width].[precision]type"`

width and precision are optional modifiers

```
%type
%s string
%i or %d integer
%f float or real number
%e scientific notation or exponential
\n new line
```

Example.

```
awk -F, '$2=="CA" {printf "%.2f\n", $3}' file1.dat
```

This will extract all lines of data file `file1.dat` that contain `CA` in the second field and print only the temperature (3<sup>rd</sup> field). Format the output to obtain (two decimal digit format):

```
1.23
1.14
```

Try this:

```
awk -F, '$2=="CA" {printf "Station ID=%-10d T=%.3f\n", $1, $3}' file1.dat
```

```
Station ID=401      T=1.232
Station ID=402      T=1.139
```

Overview:

- Often `awk` is used for single command line editing of a file.
- Only two types of data are in `awk`: numbers and strings of characters.
- `Awk` reads one line at a time and splits each line into fields.
- A field is a sequence of characters that doesn't contain any blanks or tabs.
- First field in current input line is called `$1`, the second `$2`, ... `$n`.
- The entire line is referenced as `$0`.

## Chapter 7. Flow control: for loop

### Key commands and concepts in this chapter

Commands	
for; do; done	Basic elements of a for loop
Concepts	
loops	programming tool that enables repeated execution of commands
for loop	operates on lists of items and repeats a set of commands for every item in a list

Flow control is specification of the order in which certain commands are executed. The main use of loops is to automate repetitive tasks. We will talk about the so called **for** loops in this Chapter, and **while** loops in next Chapter.

### 7.1 The for loop

For loops will go sequentially through a list and repeat execution of commands for each item in the list. The list can be a set of numbers, words, files, or pathnames. The general format of the for loop in bash is:

```
for var in item1 item2 item3 ... itemN
do
    command
    command
    command
done
```

The for loop will take each item in the list (in order, one after the other), assign that item as the value of the variable **var**, execute the commands between do and done, then go back to the top, grab the next item in the list and repeat over. For loops iterate through a set of values (items) until the list is exhausted. The list of values is defined as a series of items, separated by spaces.

### 7.2 For loop examples

Example 1. Make a script for\_1.bash, and in it make a for loop over that iterates over a list of words:

```
#mprocop2:09182020:for_1.bash:
for i in Moon Stars Sun
do
    echo $i
done
```

Run the script:

```
. for_1.bash
```

What exactly happened? The `for i` part of the `for` loop defines a variable, called a loop control variable, in this case `i`, which is successively set to the values "Moon", "Stars", and "Sun". After each assignment, the body of the loop (the code between the `do ... done`) was executed once. In the body of the loop, we refer to the loop control variable `i` using standard variable expansion syntax `$i` like any other variable. In this example the repeated task is simply echoing the loop control variable `i`.

Example 2: Make a script `for_2.bash`, and in it make a `for` loop over a list of numbers.

```
#mprocop2:09/18/2020:for_2.bash

for i in 1 20 3 44
do
    echo number $i
done
```

Run the script:

```
. for_2.bash
```

Try changing the name of variable `i` into `x` and run the script again. Is there any difference?

Example 3: Make a script `for_3.bash`, and in it make a `for` loop over a list of files. We will also include two different commands within the loop.

```
#mprocop2:09/18/2020:for_3.bash
#echo a list of numbers

for i in for_1.bash for_2.bash
do
    ls -l $i
    cat $i
done
```

Run the script:

```
. for_3.bash
```

### 7.3 Using shell expansion in for loops

The **for** conditional always takes a list of values after the "in" statement. In the first example we specified three English words, in the second example we specified four numbers, in third example we specified two files. You can use shell expansion to specify the list of values when needed.

Example 4: This example uses the `*` wildcard to specify a list of files.

```
#mprocop2:09/18/2020:for_4.bash

for i in *.bash
do
    ls -l $i
    cat $i
done
```

In this example we specify a list of values by using wildcards. The `*.bash` will be expanded in a list of values being all files ending with `.bash` present in current working directory.

Run the script:

```
. for_4.bash
```

Example5: Change into **data-temp** directory (data-temp.zip is provided on Canvas), and from there make this script for\_5.bash

```
#mprocop2:09/18/2020:for_5.bash

for i in *.dat
do
    head -5 $i
done
```

In this case, the shell will expand \*.dat with a list of all files ending with .dat. The loop variable `i` will take each file ending with ending with `.dat`, and for each file it will display the first 5 lines. Think about which metacharacters you should use to loop only over files temp-clean.dat and temp-clean1.dat.

Try to modify the script and test it out.

```
. for_5.bash
```

Example 6: In the case below we are using a brace expansion to generate a list of numbers 1 – 10.

```
#mprocop2:09/18/2020:for_6.bash

for i in {1..10}
do
    echo number $i
    echo $((i*4))
done
```

Run the script:

```
. for_6.bash
```

Change the list to be from 1 to 1000 and test your script.

If you change the braces to {a..z}, does this loop still work?

Example 7: In the case below we are using quotes to limit the list from three elements, to just one

```
#mprocop2:09/18/2020:for_7.bash

for i in "Moon Stars Sun"
do
    echo $i
done
```

Compare this example with Example 1. How many loop iterations, i.e. number of items (values) in the list, are there in example 1 and how many iterations in example 6?

## 7.4 Passing arguments from command line: @

A script can have arguments passed to it from the command line when running a script. When you run a script and want to pass to it arguments from the command line, you need to write the arguments right after the code to run that script.

```
. script.bash argument1 argument2 argument3
```

Within the script these arguments are treated as variables and referenced as follows:

\$1 will be expanded to the 1st argument, as called from the command line.

\$2 will be expanded to the 2nd argument, as called from the command line.

\$3 will be expanded to the 3rd argument, as called from the command line.

and so on...

\$@ refers to *all* command-line arguments separated by spaces.

Example 8: In this example, we pass all command line arguments to the script for\_8.bash by writing in the script \$@.

```
#mprocop2:09/18/2020:for_8.bash
for i in $@
do
    echo $i
done
```

Then we run the script by specifying the arguments, for example, Sun Moon and Stars:

```
. for_8.bash Sun Moon Stars
```

We can easily change the arguments (and in this case the list of values of the for loop), by specifying different arguments:

```
. for_8.bash 1 33 Stars
```

Try running this script with different arguments, including different words, numbers, files, and a mixture of words, numbers, and files.

Example 9: Make a script called for\_9.bash and try this out.

```
#mprocop2:09/18/2020:for_9.bash
# this script takes one command line argument

echo multiply numbers from 1 to 10 by a value entered from the command line
var=$1

for i in {1..10}
do
    echo $((i*var))
done
```

Then we run the script by specifying the first argument. The number 3 is passed to variable var by running the script in this way and having the \$1 within the script.

```
. for_9.bash 3
```



We can change the value of variable `var` by entering a different first argument from the command line.

```
. for_9.bash 10
```

Passing arguments from the command line is very useful because you can change values of variables, including values listed in a for loop, from the command line, without the need to edit the script. This would be useful for you (the developer of scripts) and users (people using your scripts). It is a good habit to write a few comment lines within a script to remind yourself and tell users how to run a script.

## 7.5 Counting loop

A common type of program loop is one that is controlled by an integer that counts up from an initial value to an upper limit. Such a loop is called a **counting loop**.

A counting loop has two parts that must be correct:  
 The counter must be initialized before the for loop.  
 The counter must be increased- this is the repeated task.

```
count=0    #define variable count and set it to 0

for i in a b c
do
    count=$(( count + 1 )) # increment variable count by 1
done

echo The final count is $count
```

Here is a modification of the counting loop.

```
count=0    #define variable count and set it to 0

for i in a b c
do
    count=$(( count + 1 )) # increment variable count by 1
    echo variable i is $i   # show value of i
    echo variable count is $count # show value of count
done
```

## 7.6 Summing loop

Script to calculate a sum

```
sum=0

for i in 2 5 8
do
    sum=$(( sum + $i )) # increment variable sum by the value of i
done

echo the final sum is $sum
```

And a modification of this script

```
sum=0
```

```
for i in 2 5 8
do
    sum=$(( sum + $i ))
    echo variable i is $i
    echo variable sum is $sum
done
```

## Chapter 8. Flow control: if statements and while loop

### Key commands and concepts in this chapter

Concepts	
if conditionals	Used to determine course of action depending on if a condition is true or false
while loop	Continue doing an action as long as the condition is true.
Commands	
if, then, else, elif, fi	Basic commands of if statements
[ expression ]	Test command
while, do, done	Basic commands of while loop

Decision making is one of the most fundamental concepts of computer programming.

Like in any other programming language, decision making is implemented by the following conditional statements:

```
if statement
if.. else statement
if.. elif.. else statement
```

in bash they can be used to decide whether to execute a block of commands or not based on a test

### 8.1 If statement

The most basic if statement is the following:

```
if TEST-COMMAND
then
    statements
fi
```

*If the TEST-COMMAND is TRUE, the STATEMENTS get executed.  
If TEST-COMMAND is FALSE, nothing happens, the STATEMENTS get ignored.*

**The TEST-COMMAND is given by [ expression ]**

Example: Write this in a script called if-num.bash:

```
age=20
if [ $age -gt 18 ]
then
    echo I can vote
fi
```

Run the script

```
. if-num.bash
```

Now change the value of the variable age

```
age=10
if [ $age -gt 18 ]
then
    echo I can vote
fi
```

Run the script

```
. if-num.bash
```

The test command is False, and the statement is ignored.

## 8.2 Test command [ expression ]

Look up the man page for test to see all the possible operators you can use to test expressions:

```
man [ ] # leave one space between [ ]
```

Below are the ones we use in this course:

### Operators for existence of files and directories

```
[ -d dirname ]    True if dirname exists and is a directory.
[ -f filename ]   True if filename exists and is a regular file.
```

### String operators

```
[ s1 = s2 ]       True if the strings s1 and s2 are identical.
[ s1 != s2 ]       True if the strings s1 and s2 are not identical.
```

### Arithmetic operators with integer numbers n1 and n2

```
[ n1 -eq n2 ]     True if the integers n1 and n2 are equal.
[ n1 -ne n2 ]     True if the integers n1 and n2 are not equal.
[ n1 -gt n2 ]     True if the integer n1 is greater than the integer n2.
[ n1 -ge n2 ]     True if n1 is greater than or equal to n2.
[ n1 -lt n2 ]     True if n1 is less than n2.
[ n1 -le n2 ]     True if n1 is less than or equal to n2.
```

### Example of if statement

In a script called if-string.bash

```
s1=abc
s2=def

if [ $s1 != $s2 ] #Test if the strings s1 and s2 are not identical.
then
    echo strings $s1 and $s2 are different
fi
```

Run the script

```
. if-string.bash
```

### More Examples of if statement

In a script called if-file.bash

```
script=script1.bash

if [ -f $script ] #Test the existence of file script1.bash within current
directory
then
    echo file $script exists in my current directory
fi
```

In a script called if-file1.bash

```
script=script1.bash

if [ -f ~/Desktop/$script ] #Test the existence of file script1.bash within
Desktop directory

the
    echo file $script exists in Desktop director
fi
```

In a script called if-dir.bash

```
if [ -d ~/data-temp ] #Test existence of directory data-temp within home
then

    echo directory data-temp exists in my home directory
fi
```

You should run each script to see the output.

### 8.3 If else statement

Sometimes we want to execute a certain set of instructions if a statement is true, and another set of instructions if it is false. We can accommodate this with the else mechanism.

If the TEST-COMMAND is TRUE, the STATEMENTS1 gets executed.

If TEST-COMMAND is FALSE, the STATEMENTS2 gets executed.

```
if TEST-COMMAND
then
    statements1
else
    statements2
fi
```

Example: modify if-num.bash to:

```
age=10
if [ $age -gt 18 ]
then
    echo I can vote
else
    echo I cannot vote
fi
```

### 8.4 If – elif – else statement

Sometimes we may want to test more than one condition (or expression) and this can be done by adding elif. You can have one or more elif clauses in the statement. The else clause is optional.

```
if TEST-COMMAND1
then
    statements1
elif TEST-COMMAND2
then
    statements2
else
    statements3
fi
```

**The conditions are evaluated sequentially.** Once a test is True the remaining conditions or tests are not performed, and the bash shell moves to the end (fi). Let's add an elif to the previous script:

```
age=18
if [ $age -gt 18 ]
then
    echo I can vote
elif [ $age -eq 18 ]
then
    echo I just turned 18 so I can vote
else
    echo I cannot vote
fi
```

## 8.5 Nested conditional

We looked at for and if conditionals; now let's nest an if statement inside a for loop:

```
for i in {1..10}
do
    if [ $i -le 3 ]
    then
        echo $i
    else
        echo $i greater than 3
    fi
done
```

For each number a test condition is performed.

## 8.6 The while loop

A *while* statement will execute as long as a particular test condition is true, and has the following format:

```
while [ expression ]
do
    statements
done
```

In the example below the variable *i* is first set to 0, so as the test starts True.

Then inside the **while** loop, variable *i* is incremented by 1. Eventually the condition *\$i -ne 10* becomes false, causing the loop to terminate.

```
i=0 #set i to 0
while [ $i -ne 10 ]
do
    echo $i
    i=$(( $i + 1 )) # this statement increments the variable by one
done
```

What would have happened if the increment statement was omitted?

## 9. Processes and The Unix Environment

### 9.1 Processes

The basic operation of the Unix operating system (and most others today) revolves around processes and files. A program that is currently running (i.e., being executed) is termed a process.

A process, in simple terms, is an instance of a running program. Linux and Unix are multitasking operating systems, i.e. a system that can run multiple tasks (process) during the same period of time. When a system starts up, the operating system initiates a few of its own activities as processes and launches a program called `init`. A program can launch other programs. This is expressed in the scheme of parent process and child process. The operating system maintains information about each process to help keep things organized - each process is assigned a process ID number or PID. Processes have owners and permissions.

The commands `ps` and `top` are used to look at running processes.

The initial process started at “boot time” is `launchd` (or `initd`). At the terminal type

```
ps -l
```

This command tells us that the 1<sup>st</sup> process (PID=1) was launched, and its name is **launchd**. At the terminal type

```
ps aux
```

You can now see a list of all the currently running processes. All PIDs are greater than 1. Some processes have been launched by PID 1 and some by other PIDs.

You can see all of your currently running processes by typing:

```
ps -fu username
```

The output will be something like:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
502	788	787	0	Mon12PM	ttys000	0:00.02	-bash
502	75485	75484	0	Fri11AM	ttys001	0:00.04	-bash

where the columns indicate:

UID	user ID
PID	process ID
PPID	parent process ID
C	cpu usage time
STIME	start time
TTY	terminal type (shell identification) to which the PIDs are spawned from
TIME	running time
CMD	command in execution (a dash indicates a login shell)

### Parent and child processes

More than one process can run inside the *same terminal* (job control). Open a terminal (login shell), and type `bash`. Typing `bash` will launch a child-shell or subshell. A *subshell* is a child process launched by the shell.

```
bash
```

Now let's see the processes by typing:

```
ps -f
UID      PID  PPID  C  STIME  TTY     TIME    CMD
2137144251 52103 52102   0   6:51PM ttys000   0:00.01 -bash
2137144251 52110 52103   0   6:51PM ttys000   0:00.00 bash
```

PID stands for process ID, while PPID is the parent process ID. In this case, you can tell that the second bash process is the child process of the first bash process.

To kill (or stop) a shell process, look for the processes with a dash and choose the process number that you want to end. Be sure that the PID is not for your current tty. Then issue the following command:

```
kill processid
```

Replace *processid* with the process ID number you identified.

Launch a new terminal session.

Determine the terminal name. Use the command **tty**.

```
tty
/dev/ttys007
```

The terminal name is ttys007.

Determine its PID. Use the command **ps**.

```
ps -t ttys007
  PID TTY          TIME CMD
 36979 ttys007    0:00.02 login -pf maria
 36980 ttys007    0:00.19 -bash
 89132 ttys007    0:00.00 ps -t ttys007
```

Using the command **kill**, terminate the newly made terminal session.

```
kill 36980
```

OR

```
kill -KILL 36980
```

The option -KILL will force exit.

## 9.2 The Unix Environment

When working on a UNIX system, an *environment* is created. The environment consists of variables that are set through several different means. Most are set by the system and within your own startup files. Others may be set within user programs. When a shell process starts it inherits the environment from its parent process.

You can see all variables that have been set in the current shell environment by typing:

```
env
```



One of the variables listed is `HOME`. Where does this variable point? The variable `HOME` provides yet another way to return to your “home” directory. Thus,

```
ls $HOME
```

lists the contents of your home directory.

```
cd $HOME
pwd
/Users/mprocop2
```

Another variable is `SHELL`. This tells the computer what shell you currently use. If you are not in the *bash* shell, you need to change shells.

```
echo $SHELL
/bin/bash
```

## Creating environment variables

Environment variables are a great way to save time typing lengthy paths. There are many other uses for environment variables as we shall see. Here is an example:

```
cd $HOME
mkdir -p CODE/PYTHON
```

Make an environment variable, to this new directory:

If bash shell:

```
PYTHON=$HOME/CODE/PYTHON
export PYTHON
```

Repeat the above but for a directory in the `CODE` directory named `UNIX`.

```
mkdir $HOME/CODE/UNIX
UNIX=$HOME/CODE/UNIX
export UNIX
```

How do these environment variables work?

```
echo $UNIX
ls $UNIX
cd $UNIX
cd ~
ls -latr
```

The **export** command is used to export a variable to the environment of all child processes running in the current shell.

Open a new shell (terminal). Will the same sequence of commands work in the new shell, or do you first have to define the variable again and export it? You will find the variable is not available. The way to automatically make it available is to add the commands to your `.bashrc` file. An example of a `.bashrc` file is on Canvas, named `bashrc.orig`. The commands in this “configuration” file are executed when you open a new shell.

## Shell vs environmental variables

**Environmental variables** are automatically transferred/available to a child-shell (or sub-shell) when it is created. By convention, environmental variable names are given in upper case.

**Shell variables** are local to the shell in which they are defined; they are not available to child-shells. By convention, shell variable names are generally given in lower case.

**Every UNIX process runs in a specific *environment*.** The concept of parent and child processes is important to understand environmental variables. As short reminder, a parent process, e.g. login shell, can launch a child process (subshell).

Try this:

Open a terminal and define shell variable hw:

```
hw="Hello World" #define shell variable
echo $hw
Hello World
```

launch a subshell

```
bash
echo $hw
```

Nothing will be printed because shell variables are available only in the local shell where they are defined.

Now export hw into the environment. The export command is used to export a variable into the environment of all the child processes running in the current shell:

```
export hw
bash # launch a subshell
echo $hw
Hello World
```

Now variable hw is available to all child processes. To make an environmental variable available to all login shells, i.e. every time to open a new terminal session, you have to define/export the new environmental variable in the .bashrc configuration file.

## PATH

If you type a command the shell must be able to find the command. For example, if you type ls, the shell must know where to find it. It does this by looking up a list of directories that are defined sequentially in an environment variable. This variable is called PATH. The PATH variable contains a colon separated list of files to be searched for executable programs.

```
echo $PATH
```

You will notice that the output contains files with binary executable files. You can add your own paths to this list. For example, you can add your scripts directory to this list.

```
export PATH=$PATH:$HOME/CODE/UNIX
```

This command will add this new directory to the current list of paths:

```
echo $PATH
```

### Optional Adding `#!/bin/bash` as the 1st line to your script

The combination of characters `#!` is called a shebang. When `!` is following the `#` character, the line that follows is not a comment anymore. Usually the `#!` characters are followed by the location of the program to be used to run the script. For the bash shell, we will use:

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

First line of your scripts:

Note that **`#!/bin/bash`** should be the **first line** of your script. This 1<sup>st</sup> line is not necessary if “sourcing” a file from a terminal bash shell, because all commands are interpreted by the bash shell.

If you want your program to be executed by the Bourne shell the shebang line would be:

`#!/bin/sh` – Execute the file using the Bourne shell, or a compatible shell, with path `/bin/sh`

While for the majority of you bash executable will be located in the `/bin` directory, for some of you (Windows users) it may be in some other directories. It is thus a good idea to, instead of `#!/bin/bash`, use the following as it is more portable on different systems:

**`#!/usr/bin/env bash`**

To run scripts in Python from the terminal you would use:

`#!/usr/bin/env python` – Execute using Python by looking up the path to the Python interpreter automatically via [env](#). You will learn what environment (`env`) is in Chapter 6.

Now update your first script: add the first line starting with `#!/bin/bash` or `#!/usr/bin/env bash`

Add a few comments:

```
#!/usr/bin/env bash
#mprocop2:01/25/2022:script_1.bash
# hash tags are comments in bash

echo Hello World!           # This is a comment
```

Try making your `script_1.bash` file executable by using the `chmod` command. Now try running this file like this:

```
./script_1.bash
```

## 9.3 Alias

Alias is used to rename a command. For example, instead of typing your favorite options for the `ls` command, such as:

```
ls -lrt
```

You can make an alias for it

```
alias lrt='ls -lrt'
```

Try running the lrt command.

Unintended removal of files can occur. Recovering these files is often possible but certainly not convenient. Instead of using the default options for **rm** command you can make an **alias** to the command with desired options. A useful option for the rm command is **-i** which requires confirmation before a file is deleted.

```
alias rm='rm -i'
```

## 9.4 CUSTOMIZATION – runtime configuration files

To customize your shell environment you must edit some configuration scripts called startup files. In bash we will be editing these configuration scripts: `.bash_profile` and `.bashrc`, which usually are in home directory. While `.bash_profile` gets executed when you login, `.bashrc` is executed when you open a new window or type `bash` in your terminal window. In both files, users can specify and export various environmental variables, such as the `PATH` variable. Aliases can also be defined in these scripts. Placing environment variables in these scripts avoids the arduous task of entering the command every time you log in. To avoid duplicating the setup of variables and aliases, we will only be placing them in the `.bashrc` file. In `.bash_profile` we will simply source the `bashrc` file.

If you do not find `.bash_profile` and `.bashrc` in your home directory (try it with `ls -a`), you can email the Prof, and she will send the files to you.

Make a copy of the files inside your `$HOME` directory. Make these files hidden (use the copy command to add the dot before the filename, e.g., `cp bashrc .bashrc`). You should now see the hidden startup files `.bashrc` and `.bash_profile` in your home directory. Note, if these two files are not in your home directory but somewhere else, you won't have your new environment variables and aliases available.

```
export UNIX=$HOME/CODE/UNIX
export CODE=$HOME/CODE
export DATA=$HOME/DATA
export PYTHON=$HOME/CODE/PYTHON
```

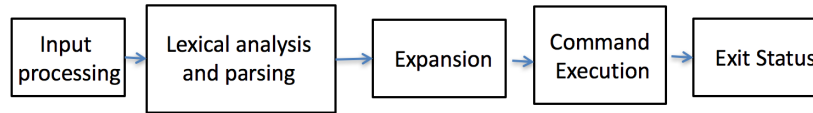
## 9.5 Bash command processing sequence

When you type a command in your terminal or run a bash script, the bash shell performs a series of operations. For example, when you type: (or have this in a script, and you run the script)

```
echo Hello
Hello
```

The shell needs to identify what is a command and what is an argument. Then it needs to find where the command is located, execute the command, and in some cases produce some sort of output. In practice, things are even more complicated since there are some special characters called *meta*-characters that have a special meaning, instead of a literal meaning, to the shell. For example, we learned about the pound sign `#`, which tells the shell to ignore the characters that follow that sign and treat them as a comment. The shell also performs something called *expansion*, i.e., it replaces some meta-characters with some other characters.

We will talk about *meta-characters* and *expansion* in this chapter. A general recipe that the bash shell follows when processing a file or your commands from the terminal is presented below.



- **Input processing:** take characters from the terminal or a file and break them into lines. The lines are sequences of characters terminated by newline characters.
- **Lexical analysis and parsing:** identify *meta-characters* and separate the stream of characters into words. Words are sequences of characters separated by *meta-characters*.
- **Expansion:** some characters/words can be expanded into some other characters.
- **Command execution:** the set of expanded words is decomposed into a command name and a set of arguments. The shell begins to search each directory location specified in the PATH variable for the bash commands the users entered. When the shell searches for a program or commands, it searches each directory specification in order; if it finds the program in a directory, the shell quits looking; otherwise, it continues to the next directory specification.
- **Exit status:** the shell will report if an error occurred in command execution. If the command/program is not found in any directory specification in the PATH variable, the shell reports an error, and returns to step 1. If the program is found, the shell creates a copy of itself, almost identical except for a new PID, this becomes the CHILD of the initial shell the parent sleeps at this point (wait system call) as the child process runs. When the child process completes, the child dies and sends a termination signal to the parent, which causes the parent process (shell) to wake up.

## Chapter 10. Regex, Appendices and Tables

### Regex

Regular Expressions (RegEx) is a *metalanguage* used for pattern matching. If you have a particular criterion you are trying to select for or write (*xyz*) you will want to make use of regular expressions. The notation allows a user to match and select data strings. RegEx are not used by the shell directly but find their use in many UNIX utilities: vi, less, sed, egrep, grep, awk, python, and others. In fact, the name grep stands for *global regular expressions print*. Some annoying differences persist between the different flavors. We will try to keep it simple here and you can augment as you need.

**Note: The metacharacters used to match filenames (ie, for filename expansion) in the shell (\$, \*, ?, []) do not have the same meaning as they do in Regular Expressions which are used to match text.**

There are two key ideas at work in regular expressions (RegEx):

ordinary characters i.e., just the actual character

special characters or metacharacters

Both were introduced earlier in the course but in the context of filename expansion. Here we will use the same concepts to extract information from files. An ordinary character is literally just that, the character.

It is a *literal*. If a character performs an action or has a meaning beyond its “literal” nature it is a *metacharacter*.

**TABLE: Regular Expression Characters**

Operator	Function	Syntax	Result
.	Any single character	x.z hop.ins a..	xyz any character in place of k a followed by any two characters
^	Beginning of string (line)	^Hopkins	Hopkins at beginning of line
\$	End of string (line)	N\$ Hopkins\$ x\$ ^abcd\$ ^\$	All nitrogen atoms Hopkins at end of line x only if it is the last character on the line a line containing just the characters abcd a line that contains no characters
[ ]	Single character within bracket	[O] [Tt] [a-z] [a-zA-Z] [Hh]opkins	Every line (any line with O) lower or uppercase t lowercase letter any alphabetic character
[ ^ ]	Negation character. Single character <b>not</b> contained in the brackets	[^A-Z] [^0-9] [^a-zA-Z]	No uppercase letters any nonnumeric character any nonalphabetic character

There are additional RegEx metacharacters. To familiarize yourself with them you can look up this webpage:

<http://regexone.com/lesson/>

Make a file dirlist.txt in the following way:

```
ls /bin/ > dirlist.txt
ls /usr/bin/ >> dirlist.txt
ls /sbin/ >> dirlist.txt
ls /usr/sbin/ >> dirlist.txt
```

Now look at that file with one of the file viewing commands.

What is the difference between matching zip and .zip:

```
grep zip dirlist.txt
grep .zip dirlist.txt
```

Now try the metacharacters that will match zip at the beginning and ending of a line:

```
grep ^zip dirlist.txt
grep zip$ dirlist.txt
grep ^zip$ dirlist.txt
```

Now try matching or NOT matching a single character

```
grep [bg]zip dirlist.txt
grep [^bg]zip$ dirlist.txt
```

Finally, try figuring out what happens now:

```
grep [A-Z] dirlist.txt
```

## Appendices and Tables

Key binding	Function
<i>Tab</i>	Complete filename up to next non-unique character
<i>Ctrl-c</i>	Kill foreground process, i.e., cancel command or interrupt program
<i>Ctrl-z</i>	Suspend foreground process, type <b>fg</b> to resume
<i>Ctrl-d</i>	Terminate input, or exit shell (function can depend on context)
<i>Ctrl-s</i>	Suspend output
<i>Ctrl-u</i>	Clear the command line
<i>Ctrl-q</i>	Resume output
<i>Ctrl-o</i>	Discard output
<i>Ctrl-l</i>	Clear screen

## Redirection

Operator	Function	Usage
<b>&gt;</b>	Redirect standard output to file	<i>command &gt; output</i>
<b>&gt;&gt;</b>	Redirect standard output and append to file	
<b>&lt;</b>	Redirect standard input from file	<i>command &lt; input</i> <i>grep abc &lt; file.dat</i>
<b>&lt;&lt;</b>	Redirect standard input from command source. Used for here documents	
<b>&gt;!</b>	Redirect standard output and overwrite file	

>>!	Redirect standard output to file or append to file	
>&	Redirect standard output/error to file	
>>&	Redirect standard output/error and append to file	
<<<	Redirect a word to standard input to a command	<i>cat &lt;&lt;&lt;'Hello World!'</i>
>>&!	Redirect standard output/error to file or append to file and overwrite	
	<b>Pipe standard output to standard input</b>	
&	Pipe standard output/error to standard input	

## Test conditions for Strings, Files, and Integers

### String Condition Tests

[ "a" \< "b" ]      since a comes before b is "less than b"

```
[ "a" \< "d" ];echo $?
0
```

Operator	True if
string1 = string 2	string1 matches string2
string1 != string 2	string1 does not match string2
string1 == string2	string1 is equal to string2
string1 != string2	string1 is not equal to string2
string1 < string2	string1 is less than string2
string1 > string2	string1 is greater than string2
-n string1	string1 is not null
-z string1	string1 is null
&&	Logical AND
	Logical OR

### File Condition Tests

Example: [ condition ]

Operator	True If
-a file	file exists
-d file	file exists and is a directory
-f file	file exists and is a regular file (e.g. is not a directory)



<code>-r file</code>	You have read permission on file. Can also be used with <code>-w</code> , <code>-x</code> , <code>f</code> write, and execute permissions respectively.
<code>-s file</code>	file exists and is not empty
<code>file1 -nt file2</code>	file1 is newer than file2
<code>file1 -ot file2</code>	file1 is older than file2

## Integers

Integer variables can take the following conditionals in addition to those for strings.  
For example: `[ 3 -gt 2 ]` is equivalent to `[ 3 > 2 ]`

Operator	Meaning
<code>-lt</code>	Less than
<code>-gt</code>	Greater than
<code>-le</code>	Less than or equal to
<code>-ge</code>	Greater than or equal to
<code>-eq</code>	Equal to
<code>-ne</code>	Not equal to

## Useful miscellaneous UNIX commands: diff, date, who, finger, history

There are many useful commands (programs) that are packaged with UNIX. We have looked at several during class sessions. Spending time roaming the man pages or the GNU website manuals might trigger new inspiration about how to tackle a problem. Often you might be trying to reinvent a wheel that has been optimized. Additionally, some commands have stripped down versions that are optimized for speed. So, if things are getting sluggish you should do some investigating for a more optimal code.

Here are a few notable commands:

Use **diff** to compare 2 files

Use **date** to echo the date

Use **whoami** to see how you are logged in at this terminal session

Use **who** to see who is logged in

Use **finger** to get information about a user

Use **history** to view and access your command history stored in `~/.bash_history`:

**history** Displays the history stored in `~/.bash_history` file.

**!!** repeats the last command

**!3** repeats command 3

**!command** repeats most recent command starting with the given *command*