BINF 2111 Midterm Study Guide (Fall 2023)

This is an extremely comprehensive compilation of everything you have learned so far. Although it is a lot of information, try to not be overwhelmed by everything. This was uploaded as a document so you can remove things you know and feel confident in. You likely know a lot more than you think!

Major Topics Covered (click on topic to go to that section)

- Basic bash commands
- Operators
- grep
- sed
- tr
- TSVs/CSVs
- Bioinformatics file types
- File manipulation
- Regular expressions
- File compression and extraction
- Arrays
- Variables
- Parameters
- Scripts
- Conditionals
- Loops
- Functions
- GitHub
- Terminology

Basic Bash Commands

COMMAND	MEANING	USAGE
`ls`	Lists everything in a directory	`ls [options] [folder]`
`echo`	Prints text to a location	`echo [phrase]`
`mkdir`	Create new directory	`mkdir [folder name]`
`cd`	Change directory	`cd [directory]`
`touch`	Make new file without any content	`touch [file name]`
`more`	View file one screen at a time	`more [file]`
`cat`	Print full contents of file	`cat [file]`

`mv`	Move file to a different location, rename file/folder	`mv [file] [new location]` `mv [old name] [new name]`
`pwd`	Print current location (working directory)	`pwd`
`wc`	Count the number of lines/words/bytes in a file	`wc [options] [file]`
`rm`	Remove a file or a directory/everything in it	`rm [file]` `rm -r [directory`
`cp`	Copy a file to a destination	`cp [file] [destination]`
`history`	Look at command history for past -num commands	`history` `history -100`
`man`	Look at manuals for individual commands	`man [command]`
`clear`	Clear terminal window	`clear`

`head`	Print the first 10 lines of a file
	
	Print the first num lines of a file
		`head file.txt`
		
		`head -20 file.txt`

`tail`	Print the last 10 lines of a file
	
	Print the last num lines of a file
		`tail file.txt`
		
		`tail -20 file.txt`

`whoami`	Prints the current user	`whoami`
`date`	Prints the current date	`date`
`diff`	Prints the difference between two files	`diff file1.txt file2.txt`

Command: ls

- Meaning: Lists everything in a directory

- Options:
  - `-a` show all files (including ones that start with .)
  - `-l` use long listing format (file sizes, dates, permissions, etc)
  - `-h` use human readable format (1G, 27K, 736M)

    - `-t` sort by time with newest first
    - `-o` similar to -l, but without group permissions
    - `-r` reverse the order while sorting

- Basic Usage:
    - `ls [options] [directory (optional)]`

- Examples:
    - `ls -thor`
      List everything in the current directory, sorting by time with the newest first, in human readable format, use a long listing format (without permissions), and in reverse order

    - `ls -alh Desktop/`
      List everything in the Desktop directory, showing all files in a long listing, human readable format

Command: echo

- Meaning: Print text to a location
    - Using > file after the command will put the text in the file
    - Using >> file after the command will append the text to the end of the file

- Basic Usage:
    - `echo [text]`

- Examples:
    - `echo "hello"`
      Print hello to the terminal

    - `echo "hello world" > file.txt`
      Print hello world to file.txt

    - `echo "hello student" >> file.txt`
      Append hello student to the end of file.txt

Command: mkdir

- Meaning: Create new directory

- Basic Usage:
    - `mkdir [folder name]`

- Examples:
    - `mkdir Midterm`

Makes a directory in the current directory called Midterm

- `mkdir Desktop/BINF2111`
  Makes a directory in Desktop called BINF2111

Command: cd

- Meaning: Change directory
- Basic Usage:
  - `cd [directory]`
- Examples:
  - `cd Desktop`
    Move into the Desktop directory
  - `cd ../`
    Move into one directory up from the current directory

Command: touch

- Meaning: Create new file with no contents
- Basic Usage:
  - `touch [file name]`
- Examples:
  - `touch file.txt`
    Create an empty file called file.txt

Command: more

- Meaning: View file one screen at a time
- Basic Usage:
  - `more [file name]`
- Examples:
  - `more file.txt`
    View file.txt one screen at a time

Command: cat

- Meaning: Print full contents of file
- Basic Usage:
  - `cat [file name]`

- Examples:
    - `cat file.txt`
      Prints the contents of file.txt

    - `cat > file1.txt`
      Create a new empty file called file1.txt

Command: mv

- Meaning: Move file to a different location OR rename file/folder

- Basic Usage:
    - `mv [file] [location]`
    - `mv [old file name] [new file name]`

- Examples:
    - `mv file.txt ~/Desktop`
      Move file.txt to the Desktop

    - `mv file.txt file1.txt`
      Rename file.txt to file1.txt

Command: pwd

- Meaning: Print current location (working directory)

- Basic Usage:
    - pwd

- Examples:
    - pwd
      Print current location

Command: wc

- Meaning: Count the number of lines/words/bytes in a file
    - Default (no options) prints out:
      line count word count byte count file name

- Options:
    - `-c` print the byte count
    - `-l` print the line count
    - `-m` print the character count
    - `-w` print the word count

- Basic Usage:
    - `wc [options] [file name]`

- Examples:
    - `wc file.txt`
      Print out the line count, word count, and byte count of file.txt

    - `wc -l file.txt`
      Print out the line count of file.txt

Command: rm

- Meaning: Remove a file or a directory/everything in it
    - NOTE: Files/directories are deleted permanently when using this command. Be very careful to not delete anything important!

- Options:
    - `-r` Remove recursively, remove the directory and everything in it

- Basic Usage:
    - `rm [file name]`
    - `rm -r [directory name]`

- Examples:
    - `rm file.txt`
      Remove file.txt

    - `rm -r Lab1`
      Remove the Lab1 directory and everything in it

Command: cp

- Meaning: Copy a file to a destination

- Basic Usage:
    - `cp [file name] [location]`

- Examples:
    - `cp ~/Desktop/file.txt ./`
      Copy file.txt that is located on the Desktop to the current directory

    - `cp ~/Downloads/* ~/Desktop`
      Copy everything in the Downloads folder to the Desktop

Operators

Operator	Meaning	Usage

|	Pipe, used as “and". Use between two commands to do both commands.
	
	Used to use the output of the first command as the input of the second command.
	
	Used as "or". Use between two regex matches to match both.
		`echo “hello” | echo “world”`
		
		`ls | wc -l`
		
		`grep "hi|bye" file.txt`

>	Output, put the output of the command into a file	`echo “hello” > file.txt`
>>	Append, add the output of a command to the end of a file	`echo “hello” >> file.txt`
*	Wildcard, used as a placeholder for any character for zero or more times	`cat file*`

$	Find items at the end of a line
	
	Reference a variable
	
	Set a variable equal to a command
	
	Evaluate arithmetic
	
	Reference parameters/arguments
		`grep "TAG$" example.fasta`
		
		`$var`
		
		`$(echo "hello")`
		
		`$((a + b))`
		
		`$1, $2`

^	FInd items at the beginning of a line	`grep "^ATG" example.fasta`
.	Match any single character "wh." matches who, wha, why, whe	`grep "wh." file.txt`
++	Increment, add 1	`counter++`
--	Decrement, subtract 1	`counter--`
+=	Add the item on the left to itself and the item on the right	`((i+=1))`
&&	And. Use between two commands to do	`echo "hi" && echo "bye"`

	both commands Use between two conditionals to evaluate if both are true.	`if [[ condition1 && condition2 ]]`
`||`	Use between two conditionals to evaluate if either are true.	`if [[ condition1 || condition2 ]]`

`<`	Input a file into a command	`while read line`
		`do`
		`  echo $line`
		`done < file.txt`

`<<<`	Input a string into a command	`tr -d T <<< "This Is Lab2"`
`!`	Reverse/opposite. Usually meant to be "not"	`if [[ ! 1 -lt 2 ]]...`

grep

Meaning

- Finds text that matches a pattern and returns the lines containing that text

Options

- Important Options
  - `-c` This prints only a count of the lines that match a pattern
  - `-n` Display the matched lines and their line numbers.
  - `-v` This prints out all the lines that do not matches the pattern
  - `-E` Treats pattern as an extended regular expression (ERE)
  - `-o` Print only the matched parts of a matching line, with each such part on a separate output line.
- All Options
  - `-c` This prints only a count of the lines that match a pattern
  - `-h` Display the matched lines, but do not display the filenames.
  - `-i` Ignores, case for matching
  - `-l` Displays list of filenames only.
  - `-n` Display the matched lines and their line numbers.
  - `-v` This prints out all the lines that do not matches the pattern
  - `-e` Specifies expression with this option. Can use multiple times.
  - `-f` Takes patterns from file, one per line.
  - `-E` Treats pattern as an extended regular expression (ERE)
  - `-w` Match whole word

    - `-o` Print only the matched parts of a matching line, with each such part on a separate output line.
    - `-A n` Prints searched line and n lines after the result.
    - `-B n` Prints searched line and n line before the result.
    - `-C n` Prints searched line and n lines before and after the result.

Basic Usage

- `grep [options] "[regex]" [file]`

Examples

- `grep -cv "hello" file.txt file1.txt file2.txt`
  Count the number of lines that do not have "hello" in it

- `egrep -o "hi|bye" file.txt OR grep -Eo "hi|bye" file.txt`
  Find all lines that have "hi" or "bye" on them and only display "hi" or "bye", not the entire line. Egrep is necessary because we are searching for two different matches

- `grep -n "hello" file.txt`
  Display lines that have "hello" in it and the line numbers

sed

Meaning

- Stream editor. Diverse command that can manipulate text/files
    - Mac users need to use gsed

Options

- `-i` In-place, files are edited rather than printing output to the terminal
- `-E` or `-r` Use extended regular expressions rather than basic regular expressions (similar to using egrep)
- `-n` Used to find specific lines

Commands

- `s/[m]/[r]/[flags]` Substitute, match the regex ([m]) in a file and replace matched string with [r]
- `/[m]/d` Delete, delete the line containing the regex match [m]
- `[#]a [text]` Append, appending text after line [#] (use i for before)
- `#p` Print specific line

Flags

- `/g` Global, apply the replacement to all matches, not just the first
- `/I` Case Insensitive, match the regex case insensitively
- `/[#]` Nth, only replace the Nth match of the regex

Basic Usage

- `sed -i 's/hello/world/g' file.txt`
  Replace "hello" with "world" in file.txt

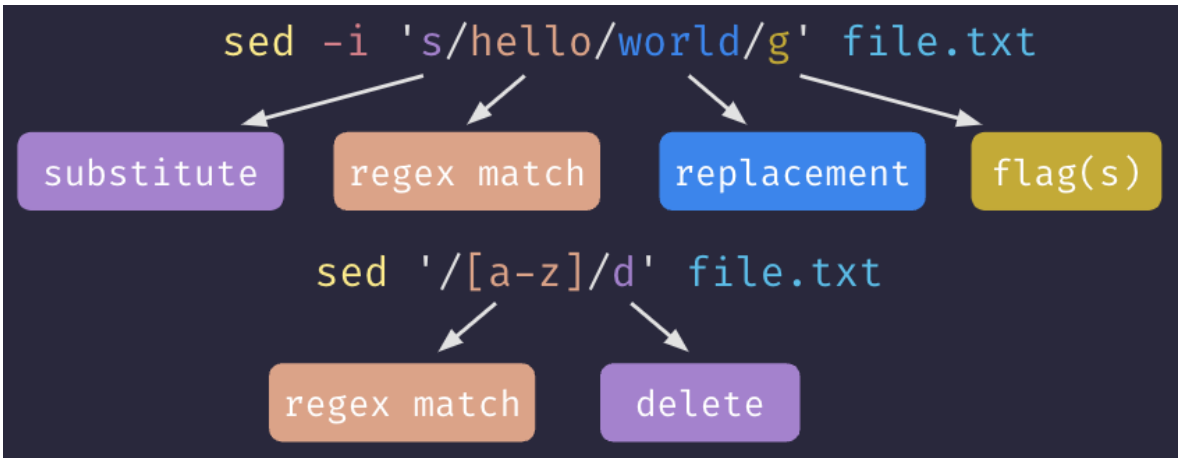
- `sed '/[a-z]/d' file.txt`
  Print out file.txt with no lowercase letters

- `sed -n '3p' file.txt`
  Print line three only

- `sed -n '4p;6p' file.txt`
  Print lines four and six

- `sed -n '2,5p' file.txt`
  Print lines 2 through 5

Breakdown



Examples

- `sed -i '/hello/d' file.txt`
  Edit file.txt so that lines with hello are deleted

- `sed -E 's/[Hh]ello/world/g' file.txt`
  Find all occurrences of Hello and hello and replace them with world

- `sed '2a hello' file.txt`
  Insert hello after the second line

- `sed '2i hello' file.txt`
  Insert hello before the second line

- `sed 's/hello/world/Ig' file.txt`
  Find all occurrences of hello, ignoring case, and replace with world

- `sed 's/hello/world/3' file.txt`
  Find the third occurrence of hello and replace with world

tr

Meaning

- Translate, change or delete characters in a file

Options

- `-c` Complement, apply other option to characters not in the given string
- `-d` Delete character(s)
- `-s` Squeeze, replace repeated characters with a single occurrence
- `-t` Truncate, if set 1 is larger than set 2, the size of set 1 will be matched to the size of set 2 (unavailable on Mac)

Basic Usage

- `tr [options] [set 1] [set 2]`

Examples

- `cat file.txt | tr "[a-z]" "[A-Z]"`
  Turns all lowercase letters in file.txt into uppercase letters

- `tr "[:lower:]" "[:upper:]" < file.txt`
  Turns all lowercase characters in file.txt into uppercase characters

- `echo "This    Is    Lab2" | tr -s " "`
  Removes repeated spaces, output: This Is Lab2

- `tr -d T <<< "This    Is    Lab2"`
  Deletes T, output: his Is Lab2

- `echo "this is course number 199384" | tr -cd "[:digit:]"`
  Removes everything except digits, output: 199384

- `tr -t 'isef' '12' <<< "This is Lab2"`
  Replace is with 12, ignore replacing ef, output: Th12 12 Lab2

TSVs/CSVs

TSVs

- Definition
    - File that is set up like a table, separated by tabs
        - Stands for Tab Separated Value

- Examples
    - eukaryotes_NAs.tsv
    - eukaryotes_zero.tsv
    - eukaryotes.tsv
    - name_game.tsv

CSVs

- Definition
    - File that is set up like a table, separated by commas
        - Stands for Comma Separated Value

- Examples
    - empty_lines.csv
    - goldmedal.csv
    - name_game.csv
    - name_game_empty.csv

Commands To Transition Between the Two - **NEED TO KNOW**

- CSV to TSV
    - `sed 's/,/\t/g' file.csv > file.tsv`
    - `tr ',' '\t' < file.csv > file.tsv`
    - `awk 'BEGIN { FS=","; OFS="\t" } {$1=$1; print}' file.csv > file.tsv`

- TSV to CSV
    - `sed 's/\t/,/g' file.tsv > file.csv`
    - `tr '\t' ',' < file.tsv > file.csv`
    - `awk 'BEGIN { FS="\t"; OFS="," } {$1=$1; print}' file.tsv > file.csv`

Bioinformatics File Types

FASTA

- Definition
    - A text-based file format for representing either nucleotide sequences or amino acid sequences
        - Amino acids are represented by their single letter code
        - Nucleotides and amino acids cannot be in the same file

- File extensions
    - .fasta
    - .fna (nucleotide only)
    - .faa (amino acid only)
    - .fa

- Basic Format
  >Sequence_ID Sequence Information (organism, gene name, GC content, description of sequence)
  ATGTTAGCTAGTCTAAGTCGATCGAT…

- Examples
    - example.fasta
    - example2.fasta
    - CoV_Sprotein.faa
    - KO_nifH.faa
    - pUC19c.fasta
    - lab2_nucleotide.fasta
    - lab2_protein.fasta

<u>FASTQ</u>

- Definition
    - A text-based format for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores.
        - Both the sequence letter and quality score are each encoded with a single character
        - Usually how genomic sequencing data is stored.

- File extensions
    - .fastq
    - .fq
    - Sometimes none! (not very good practice)

- Basic Format
    - **Field 1** begins with a '@' character and is followed by a sequence identifier and an optional description
    - **Field 2** is the raw sequence letters.
    - **Field 3** begins with a '+' character
    - **Field 4** encodes the quality values for the sequence in Field 2, and must contain the same number of symbols as letters in the sequence.

  @SEQ_ID
  GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGT
  +
  !"*((((***+))%%%++)(%%%%).1***-+*"))**55CCF>>>>>>CCCCCCC

- Examples
    - corrupted.fastq
    - MultiN.fastq

File Manipulation

Command	**Meaning**	**Usage**
`cut`	Cut out sections of files	`cut [options] [file]`
`sort`	Sort a file line by line	`sort [options] [file]`
`uniq`	Prints or deletes the repeated lines in a file	`uniq [options] [file]`
`printf`	Format and print text	`printf [options] [input]`

Command: cut

- Meaning: Cut out sections of files

- Options:
    - `-c [#]` Character, cut by character [#]
    - `-f [#]` Field, cut by column [#]
    - `-d "[delim]"` Delimiter, comma (,) or tab (\t)
    - `--complement` Get the opposite/complement of what is requested. Used with `-f` or `-c`.

- Basic Usage:
    - `cut [options] [file]`

- Examples:
    - `cut -c 2,3 --complement file.txt`
      Print out everything except characters 2 and 3

    - `cut -d "," -f 1 file.txt`
      Print out column 1 in a CSV

Command: sort

- Meaning: Sort a file line by line

- Options:

    - `-t "[delim]"` Type, delimiter used in file
    - `-k [#]` Sort column [#]
    - `-n` Sort numerically
    - `-r` Sort in reverse order
    - `-u` Sort and remove duplicates

- Basic Usage:
    - `sort [options] [file]`

- Examples:
    - `sort -k 2n file.txt`
      Sort by column 2 numerically

    - `sort -ur file.txt`
      Sort (first column) in reverse order and remove duplicates

    - `sort -t "," -nr file.txt`
      Sort column 2 in a CSV numerically from largest to smallest

Command: uniq

- Meaning: Prints or deletes the repeated lines in a file
    - Duplicate lines must be adjacent to each other! Sort before using uniq!

- Options:
    - `-c` Count repeats
    - `-d` Only print repeated lines
    - `-u` Only print unique lines

- Basic Usage:
    - `uniq [options] [file]`

- Examples:
    - `uniq -c file.tsv`
      Count repeats

    - `uniq -cd file.tsv`
      Count and print repeated lines

    - `uniq -u file.tsv`
      Print unique lines

Command: printf

- Meaning: Format and print text

    - Not exclusively meant to manipulate files, but we learned it at the same time as the file manipulation commands, so it is in this section.

- Formats:
    - `%d` Signed decimal number
    - `%s` String
    - `\n` New line (like pressing Enter)
    - `\t` Tab (like pressing Tab)

- Basic Usage:
    - `printf [options] [input]`

- Examples:
    - `printf 'This is a line. \nThis is a new line'`
    - Print out This is a line.
      This is a new line

    - `printf "This is a number: %d\nThis is a string: %s" 72 "hello" > file.txt`
      Print out This is a number: 72
      This is a string: hello
      In file.txt

    - `printf "%s\n" "#!/bin/bash" "#This is a script" "echo "Hello World" > hello_world.sh`
      Print out #/bin/bash
      #This is a script
      echo "Hello World
      In hello_world.sh

Regular Expressions

Definition

- Regular expression, a pattern (or filter) that describes a set of strings that matches the pattern

Commonly Used Patterns

- `\d` or `[[:digit:]]` or `[0-9]` Any number from 0 to 9
- `\w` or `[[:alpha:]]` or `[A-Za-z]` Any letter, regardless of capitalization. `\w` includes numbers
- `\s` or `[[:space:]]` Any whitespace character (space, tab, newline, carriage return, form feed, and vertical tab)
- `[A-Z]` or `[[:upper:]]` Any uppercase character

- `[a-z]` or `[[:lower:]]` Any lowercase character
- `\n` New line
- `\t` Tab
- `[^match]` Anything but match, [^a-z] any non-lowercase character

Commands RegEx is Used in

- `grep`
- `sed`
- `tr`

File Compression and Extraction

Command	Meaning	Usage
`tar`	Tape archive, used to create Archive and extract the Archive files	`tar [options] [file]`
`gzip`	Compress a file to be gzipped	`gzip [file]`
`gunzip`	Uncompress a file that was gzipped	`gunzip [file.gz]`

Command: tar

- Meaning: Tape archive, used to create Archive and extract the Archive files

- Options:
  - `-x` Extracts files and directories from an existing archive
  - `-v` Displays verbose information
  - `-f` Specifies the filename of the archive to be created or extracted
  - `-z` Uses gzip compression when creating a tar file (gives .tar.gz)
  - `-c` Creates an archive by bundling files and directories together

- Basic Usage:
  - `tar [options] [file]`

- Examples:
  - `tar -xzvf file.txt.tar.gz`
    Extract and unzip file.txt.tar.gz to file.txt (make it the actual size)

  - `tar -czvf file.tar.gz file.txt`
    Zip and compress file.txt and call it file.tar.gz (make it smaller)

Command: gzip

- Meaning: Compress a file to be gzipped (make it smaller)

- Basic Usage:
  - `gzip [file]`

- Examples:
  - `gzip file.txt`
    Zip file.txt so that it becomes file.txt.gz

Command: gunzip

- Meaning: Uncompress a file that was gzipped (make it the actual size)

- Basic Usage:
  - `gunzip [file]`

- Examples:
  - `gunzip file.txt.gz`
    Unzip file.txt.gz so that it becomes file.txt

Arrays

Definition: A data structure that can store a fixed-size collection of elements of the same data type

Initializing:

- Contained in a set of parentheses, with a space between each element
  `array=("this" "is" "an" "item" "in" "an" "array")`

Finding Elements:

- First element is found at `array[0]`
  `echo ${array[0]}`

- Range of elements are found with colons
  - Remember that indices start at 0. Num2 is 1 index before the element you want.

  `array[@]:num1:num2`
  `echo ${array[@]:2:5} # elements 3 - 7 (indices 2 to 6)`

- Get all elements with @
  `array[@]`
  `echo ${array[@]}`

Deleting Elements

- Delete the element within the array
  `unset 'array[4]'`

- Delete "item" element
  `${array[@]/"item"}`
  `echo ${array[@]/"item"}` # print out array with no "item" element

- Delete any element that has "it" in it
  `${array[@]/it*/}`
  `echo ${array[@]/it*/}` # print out array with no element(s) that have "it"

Adding Elements

- Set the array equal to itself and the new item
  `array=("${array[@]}" "new_item")`

- Set the array equal to itself + new item
  `array+=('new_item')`

Variables

Definition

- A named container for a particular set of bits or type of data (e.g. integer, float, string, etc.)
  - Sort of like a shortcut to the data

Environmental Variables

- Variables in the computer's system that describe your environment (like the user and the location of the root directory)

- See all environmental variables with the command `env`
- Some examples:
  - $ROOT
  - $USER

Naming Variables

- Variable names should describe the variable itself
  - If I had an array variable of all the student's names in our class, I would call it students or names, rather than something like array or var1

- Names should not start with numbers
- Names should not contain the following:
  - Periods, colons, dashes

- Good Names
  - myvar
  - MYVAR
  - Myvar
  - mYVAR
  - _myvar
  - my_var
  - myvar_
  - my012

- Bad Names
  - 1myvar
  - my-var
  - my.var
  - my:var

Assigning Variables

- Variables are assigned with an equals sign with **no space** between the name, the equal sign, and the data

- Different Types of Variables
  - String
    ```
    string_var="This is a string"
    ```
  - Integer (whole number)
    ```
    int_var=27
    ```
  - Float (decimal number)
    ```
    float_var=19.51
    ```
  - Array (see Array section for more information)
    ```
    array_var=("this" "is" "an" "array")
    ```
  - Commands
    ```
    command_var=$(echo "hello")
    ```
  - Arithmetic (only whole numbers!)
    ```
    math_var=$(( 1 + 2 ))
    ```

Referencing Variables

- Refer to variables with a dollar sign
  ```
  $var
  ```

```
echo $var
```

Variable Commands

- Length of a String
  ```
  ${#string}
  ```

Scripts

Definition

- A list of programmatically-written instructions (commands) that can be carried out when ran

Creating and Editing

- Can make an empty file with `touch` or `nano` (or other text editor)
- Edit with an IDE (like VSCode) or text editor (like `nano`)

- Comments start with #
  ```
  # this is a comment
  ```

Necessities

- Must have your shebang at the beginning
  ```
  #!/bin/bash
  ```
- File has to end in `.sh`

Running

- Bash's Form of Compiling
  - Done with the `chmod` command (see `chmod` section below)

- Executing/Running
  - Done with the `bash` command or `./`
    ```
    bash script.sh
    ./script.sh
    ```

chmod

- Change the permissions of a file to make it executable
  - Use `-r` to change the permissions of a folder and everything in it

- Two modes: octal mode and symbolic mode
- Octal Mode
  - Three digit number:
    - First - Owner
    - Second - Group
    - Third - Others

    - Add the values to change permissions
        - 4 Read permission
        - 2 Write permission
        - 1 Execute permission
    - Usage
        - `chmod 777 file.txt`
            - Give read, write, and execute permissions (4+2+1=7) to the owner, group, and others
        - `chmod 643 file.txt`
            - Give read and write permissions (4+2=6) to the owner
            - Give read permissions (4) to the group
            - Give write and execute permissions (2+1=3) to others
- Symbolic Mode
    - Combination of letters and operators
    - Person
        - u Owner
        - g Group
        - o Others
        - a All
    - Add/Remove
        - + Add permissions
        - - Remove permissions
    - Permissions
        - r Read
        - w Write
        - x Execute
    - Usage
        - `chmod a+x file.txt`
            - Add execute permissions for all individuals
        - `chmod u+rw,go+r file.txt`
            - Add read and write permissions for the owner
            - Add read permissions for the group and others

Parameters

Definition

- A special kind of variable used in a function or program to refer to one of the pieces of data provided as input (sometimes called **input variables**)
- Can change every time you run the script

Referencing Parameters

- Referenced with $1, $2, $3, etc.

- Can be set equal to another variable

```
param1=$1
param2=$2
```

- Can be used within commands as a variable
  `echo "This is the first parameter: $1"`

Running Scripts with Parameters

- Parameters are assigned when the script is ran
  `bash script.sh param1 param2`

- Example
  `bash script.sh hello this is 5 parameters`
  - `$1` - hello
  - `$2` - this
  - `$3` - is
  - `$4` - 5
  - `$5` - parameters

Keywords

- Questions that require you to use parameters usually contain the following words: **any**, **given**
- Examples
  - Write a bash script that prints a **given** range of lines from a **given** file.
  - Write a bash script that converts **any** TSV to a CSV.

Conditionals

If Statements

- Evaluates whether a statement is true or false

- Basic Setup

```
if [[ condition ]]; then
   commands
fi
```

- Conditions

Condition	**Meaning**
[[-z $string1]]	Is $string1 empty?
[[-n $string1]]	Is $string1 not empty?
[[$string1 == $string2]]	Is $string1 and $string2 equal?

[[$string1 != $string2]]	Is $string1 and $string2 not equal?
[[$num1 -eq $num2]]	Is $num1 and $num2 equal?
[[$num1 -ne $num2]]	Is $num1 and $num2 not equal?
[[$num1 -lt $num2]]	Is $num1 less than $num2?
[[$num1 -le $num2]]	Is $num1 less than or equal to $num2?
[[$num1 -gt $num2]]	Is $num1 greater than $num2?
[[$num1 -ge $num2]]	Is $num1 greater than or equal to $num2?
[[! CONDITION]]	Is the opposite of the condition true?
[[CON1 && CON2]]	Check if CON1 and CON2 are **both** true
[[CON1 || CON2]]	Check if CON1 **or** CON2 is true

- **IF:** If the statement is true, the commands are executed. If the statement is false, nothing happens

```
if [[ $num1 -gt $num2 ]]; then
    echo "Number 1 is greater than number 2"
fi
```

- **IF ELSE:** If the statement is true, the commands are executed. If the statement is false, a different set of commands are executed

```
if [[ $num1 -lt $num2 ]]; then
    echo "Number 1 is less than number 2"
else
    echo "Number 1 is NOT less than number 2"
fi
```

- **IF ELIF:** If the first statement is true, the commands are executed. If the first statement is false and the second statement is true, a different set of commands are executed. If both statements are false, nothing happens

```
if [[ $num1 -ge $num2 ]]; then
    echo "Number 1 is greater than or equal to number 2"
elif [[ $num1 -le $num2 ]]; then
    echo "Number 1 is less than or equal to number 2"
fi
```

- **IF ELIF ELSE:** If the first statement is true, the commands are executed. If the first statement is false and the second statement is true, a different set of commands are executed. If both statements are false, another different set of commands are executed.

```
if [[ $num1 -ge $num2 ]]; then
    echo "Number 1 is greater than or equal to number 2"
elif [[ $num1 -le $num2 ]]; then
    echo "Number 1 is less than or equal to number 2"
else
    echo "Neither of those statements are true"
fi
```

- Evaluating Multiple Conditions
  - Test if statement 1 and statement 2 are both true
    - Use &&

```
if [[ $num1 -lt $num2 && $num2 -ge $num3 ]]; then
    echo "Number 1 is less than number 2 AND number 2 is
greater than number 3"
fi
```

  - Test if statement 1 or statement 2 is true
    - Use ||

```
if [[ $num1 -le $num2 || $num2 -eq $num3 ]]; then
    echo "Number 1 is less than number 2 OR number 2 is
equal to number 3"
fi
```

Loops

For Loops

- Loop statement to look through a group of elements and run a command on each of those elements, one at a time

- Basic Setup

```
for i in group; do
    commands
done
```

- Examples
  - Loop through files that end in .fasta

```
for file in *.fasta; do
    wc -l $file
done
```

- Loop through a range of numbers

```
for i in {1..10}; do
    ((sum+=$i))
    echo "The sum of all the numbers thus far: $sum"
done
```

- Loop through an array

```
for item in "${array[@]}"; do
    echo $item
done
```

<u>While Loops</u>

- Do something **while** a condition is **true**
  - Can use the same conditions as if statements
  - Opposite of until loops

- Basic Setup

```
while [[ condition ]]; do
    commands
done
```

- Examples
  - Using a counter (increment with ++/decrement with --)

```
a=0
while [[ $a -lt 10 ]]
do
   echo a is currently $a
   ((a++))
done
```

  - Reading files line by line

```
while read line
do
    chars=$(echo $line | wc -c)
     sum=$((sum+chars))

     echo The sum of all the characters in the file is $sum
done < example2.fasta
```

  - Infinite loops

```
while :
```

```
do
  echo "An Infinite loop"
done
```

- Writing information into a file

```
while read line
do
    echo $line >> $filename
done
```

Until Loops

- Do something **until** a condition is **false**
  - Can use the same conditions as if statements
  - Opposite of while loops

- Basic Setup

```
until [[ condition ]]; do
    commands
done
```

- Examples
  - Using a counter (increment with ++/decrement with --)

```
a=0
until [[ ! $a -lt 10 ]]
do
   echo a is $a
   ((a++))
done
```

  - Reading files line by line

```
until ! read line
do
    chars2=$(echo $line | wc -c)
    sum2=$((sum2+chars2))
    echo $sum2
done < example2.fasta
```

  - Infinite loops

```
itnum=0
until false
do
    echo "Iteration no: $itnum"
```

```
    ((itnum++))
done
```

- Writing information into a file

```
read filename
until ! read line
do
    echo $line >> $filename
done
```

Functions

Definition

- Self contained modules of code that accomplish a specific task, usually taking in data, processing it, and returning a result
  - Like a mini script within a script

Basic Setup

- Two different ways:
  - Parentheses after the function's name

```
function_name() {
     commands
}
```

  - No parentheses and the word "function" before the function's name

```
function function_name {
     commands
}
```

Functions with Arguments

- Very similar to parameters
- Each argument is specified by a dollar sign and the argument number
  - $1, $2, etc

- Example

```
function_name() {
     echo $1
     echo $2
}
```

Running Functions

- Functions are executed by "calling" them with their name

```
function_name
```

- Functions with arguments are called with the arguments after the name
  `function_name arg1 arg2`

GitHub

Definition

- A code hosting platform for version control and collaboration

Options For Use

- GitHub.com
- GitHub Desktop
- git (through command line)

Basics

- Projects are called "repositories" or "repos"

- Within a repo, there are directories and a README
  - A README is like a summary file in markdown format.
    - Look up markdown shortcuts to make your READMEs look better!

- Each directory should have it's own README
  - The README within the directory will explain what is in that directory.

- Multiple people can be on a single repo
  - For instance, all of you have read access to the RAW lab BINF 2111 repository
  - I have read/write access, so I can upload and change files

git

- Downloading
  - `git clone`
    Copy a repository into a new directory

  - `git pull`
    Regrab repository from GitHub, used after changes have been made to a repo

- Uploading
  - `git add`
    Add file contents to the index

  - `git commit -m "Description of changes"`
    Add a commit message that describes what was changed

  - `git push`

Put the changes on GitHub

- Other Basic Functions
    - `git status`
      Show the working tree status

    - `git fetch`
      checks server for updates without pulling them

Terminology

- **Terminal**: a command line interface (CLI), where you can type commands, manipulate files, execute programs, and open documents
- **Directory**: folder or path to a folder/file
- **UNIX/bash**: language used in terminal
- **Print**: display information
- **Command**: a specific word or phrase that tells the computer what to do
- **Run**: execute a command or program
- **Options/flags**: an addition to a command that slightly changes the command in a specified manner
- **String**: Sequence of characters and can contain letters, numbers, symbols and even spaces.

- **grep**: A command-line utility for searching plain-text data sets for lines that match a regular expression.
- **sed**: A stream editor that can perform lots of functions on file like searching, find and replace, insertion, or deletion.
- **RegEx**: Regular Expression, a pattern (or filter) that describes a set of strings that matches the pattern.

- **FASTA**: A text-based file format for representing either nucleotide sequences or amino acid sequences.
- **TSV**: Tab Separated Value - file where each column is separated by tabs
- **CSV**: Comma Separated Value - file where each column is separated by commas

---

- **Script**: A list of programmatically-written instructions (commands) that can be carried out when ran
- **Text editor**: A system or program that allows a user to edit text
- **Variable**: A named container for a particular set of bits or type of data (e.g. integer, float, string, etc.)
- **Array**: A data structure that can store a fixed-size collection of elements of the same data type

---

- **Conditional**: A programming element that tells the computer to execute certain actions, provided certain conditions are met
- **Loop**: A programming element that repeats a portion of code a set number of times until the desired process is complete
- **Parameter**: A special kind of variable used in a function or program to refer to one of the pieces of data provided as input (sometimes called **input variables**)
- **Iterate**: The repetition of a section of code within a computer program for a number of instances or until status is encountered

---

- **Increment**: The process of increasing a numeric value by another value, usually by 1 with num++
- **Function**: Self contained modules of code that accomplish a specific task, usually taking in data, processing it, and returning a result
- **Argument**: A special kind of variable used in a function to refer to one of the pieces of data provided as input to the function