

Unix \$hell Genomics

Author&Instructor: Rhondene Wint

Teaching Asst: Fateme Hadi-Nezhad & Deepika Gunasekaran

What we will cover

- Overview of Linux command line
- Navigate and manipulate directories (folders) via the command line
- Manipulate files and data
- Writing bash scripts
- Installing and running programs via the command-line

Intro to Linux

Linux is general purpose open-source operating system.

Linux is inspired by UNIX, an older operating system.

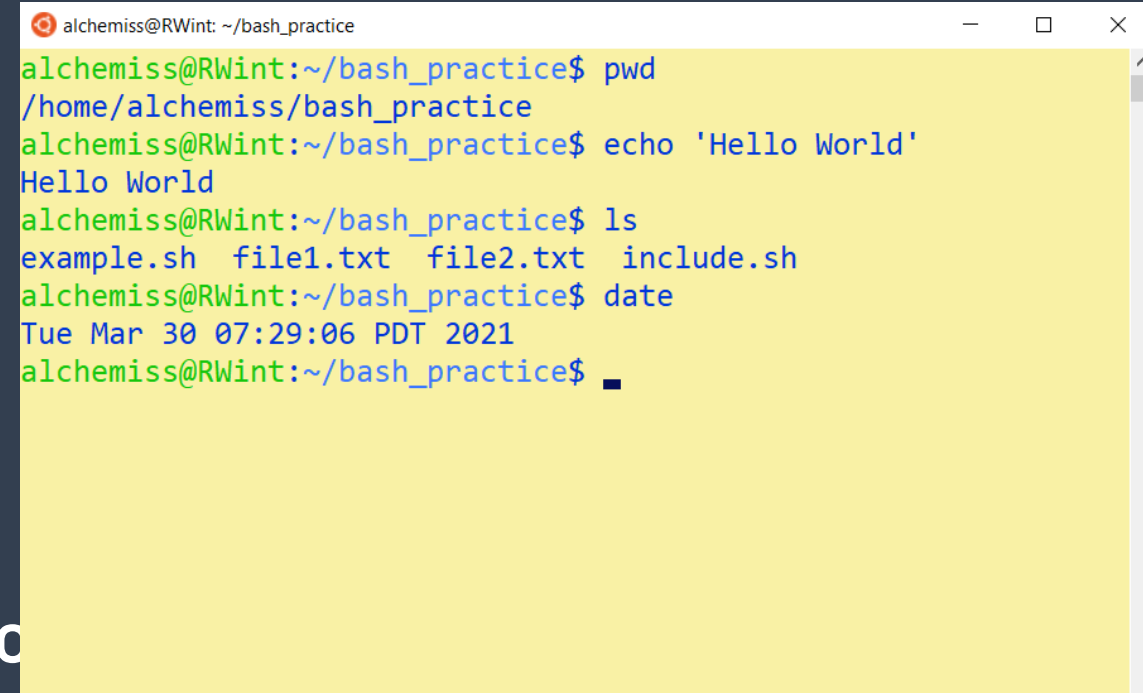
Linux is widely adopted in many industries.

Supercomputers run on linux environment.



Command-line interface

- Command-line interface (CLI) is a text-based environment where users **type commands and receive text-based output** to interact with the computer.
- A **'shell'** interprets commands for the command-line.
- Shell is the primary method for users to interact with High-performance computers (HPC) such as NERSC

A terminal window titled 'alchemiss@RWint: ~/bash_practice' with standard window controls. The terminal shows a series of commands and their outputs: 'pwd' returns '/home/alchemiss/bash_practice', 'echo 'Hello World'' outputs 'Hello World', 'ls' lists 'example.sh', 'file1.txt', 'file2.txt', and 'include.sh', and 'date' shows 'Tue Mar 30 07:29:06 PDT 2021'. The prompt returns to 'alchemiss@RWint:~/bash_practice\$' with a cursor.

```
alchemiss@RWint: ~/bash_practice
alchemiss@RWint:~/bash_practice$ pwd
/home/alchemiss/bash_practice
alchemiss@RWint:~/bash_practice$ echo 'Hello World'
Hello World
alchemiss@RWint:~/bash_practice$ ls
example.sh  file1.txt  file2.txt  include.sh
alchemiss@RWint:~/bash_practice$ date
Tue Mar 30 07:29:06 PDT 2021
alchemiss@RWint:~/bash_practice$
```

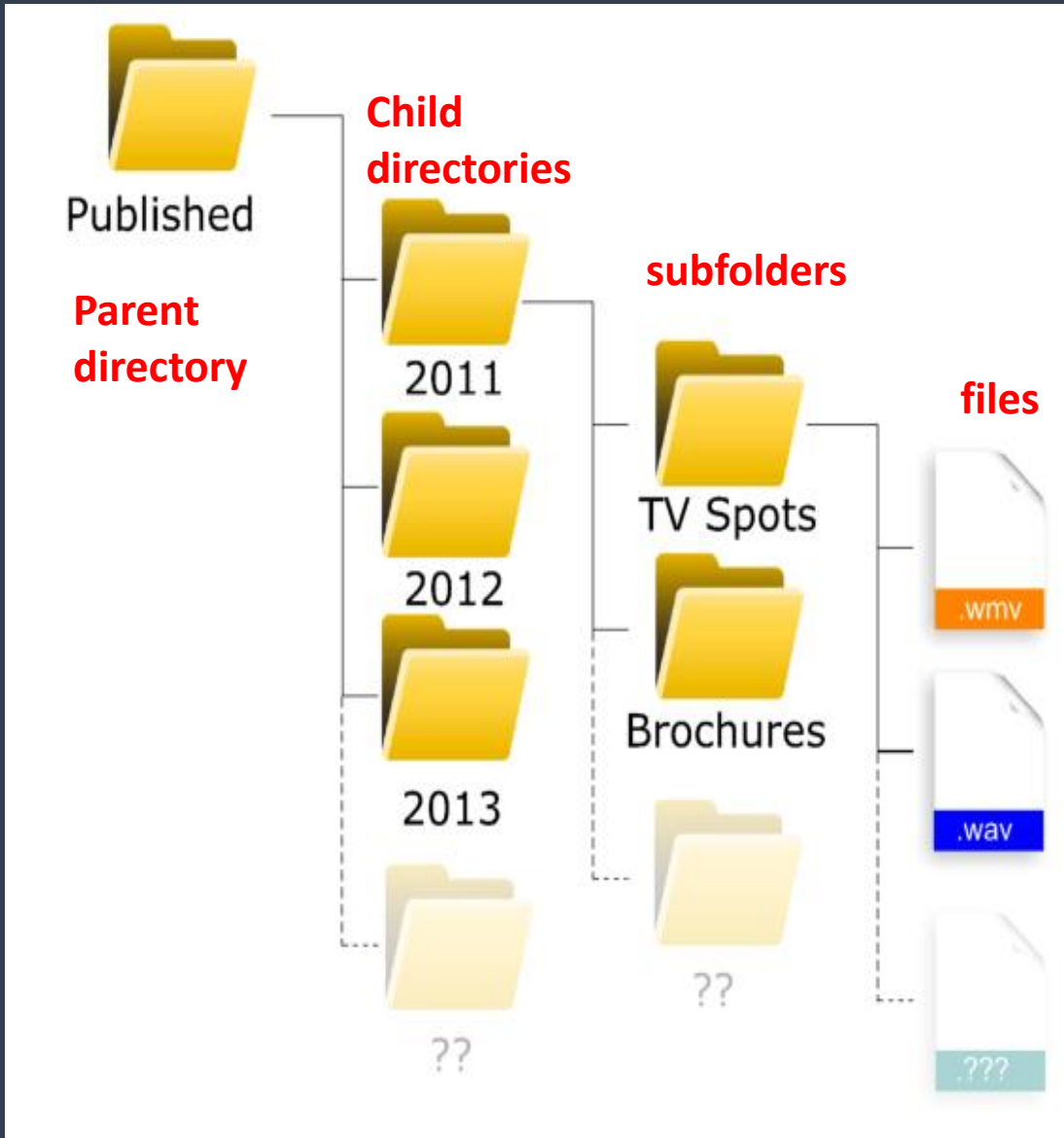
Bash shell for command-line

- Bash == 'Bourne Again Shell'
- Most popular shell for writing commands for the Unix command-line.
- Other types of Unix shells exist.
 - Zsh

yourusername@hostcomputer **Current folder**

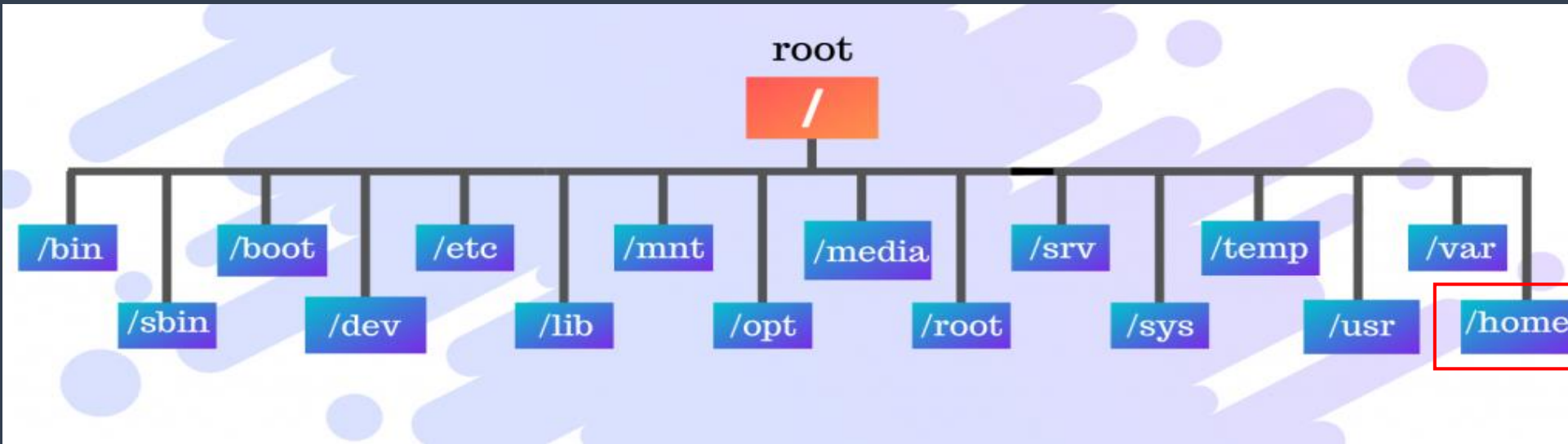
```
alchemiss@RWint: ~/bash_practice
alchemiss@RWint:~/bash_practice$ pwd
/home/alchemiss/bash_practice
alchemiss@RWint:~/bash_practice$ echo 'Hello World'
Hello World
alchemiss@RWint:~/bash_practice$ ls
example.sh  file1.txt  file2.txt  include.sh
alchemiss@RWint:~/bash_practice$ date
Tue Mar 30 07:29:06 PDT 2021
alchemiss@RWint:~/bash_practice$
```

Introduction to File System Organization



- This is how files, folders (directory) and drives are organized on the hard disk.
- Tree structure with branches.
- E.g. 'Published/2011/TV Spots/file.wav'

Linux File System Organization



- “/” is known as the *root* folder. Root stores all other directories on the system. The subdirectories of root store specific kind of data.
- **“/home”** is where all users and their folders and files are stored.
 - “/home” commonly stores subdirectories like ‘Desktop’, ‘Documents’, ‘Downloads’, ‘Music’, etc.
 - The ‘~’ is shorthand for **‘/home/username/’**

Navigating folders and files



What we will do

1. Understand the difference between relative and absolute path

Learn commands for :

1. Listing the contents of a directory
2. Moving between directories
3. Making new directories and files
4. Copying and moving files
5. Renaming files and directories
6. Deleting files and directories

General syntax of shell commands

```
ls -lh /usr/bin
```

```
sort -u users.txt
```

```
grep -i "needle" haystack
```

Command

Option(s)

Argument(s)

Where am I? Obtaining the location of the current directory

- **'pwd'** command displays the full path of the folder that the shell is opened in.
- **P rint W orking D irectory**

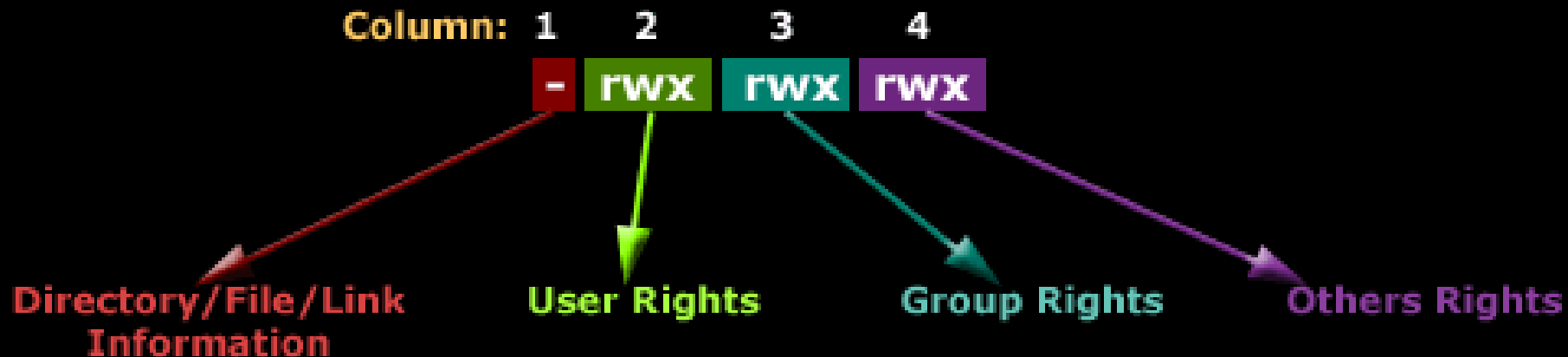
What's inside here? 'ls' command lists contents in a folder

- 'ls' lists the files and subfolders in the current working folder.
- ls has different options.
- Try these ls commands:
 - ls -l #displays folder contents as a long-formatted list
 - ls -lt #displays folder contents as a list in chronological order
 - ls -lF #displays long list format sorted by alphabetically
 - ls -lS # displays list of folder contents sorted by size (bytes)
 - ls -a #displays hidden files
 - ls -lh
- To view all the options of a bash command, type 'man name_of_bash_command '. E.g. 'man ls'

Linux File Permissions

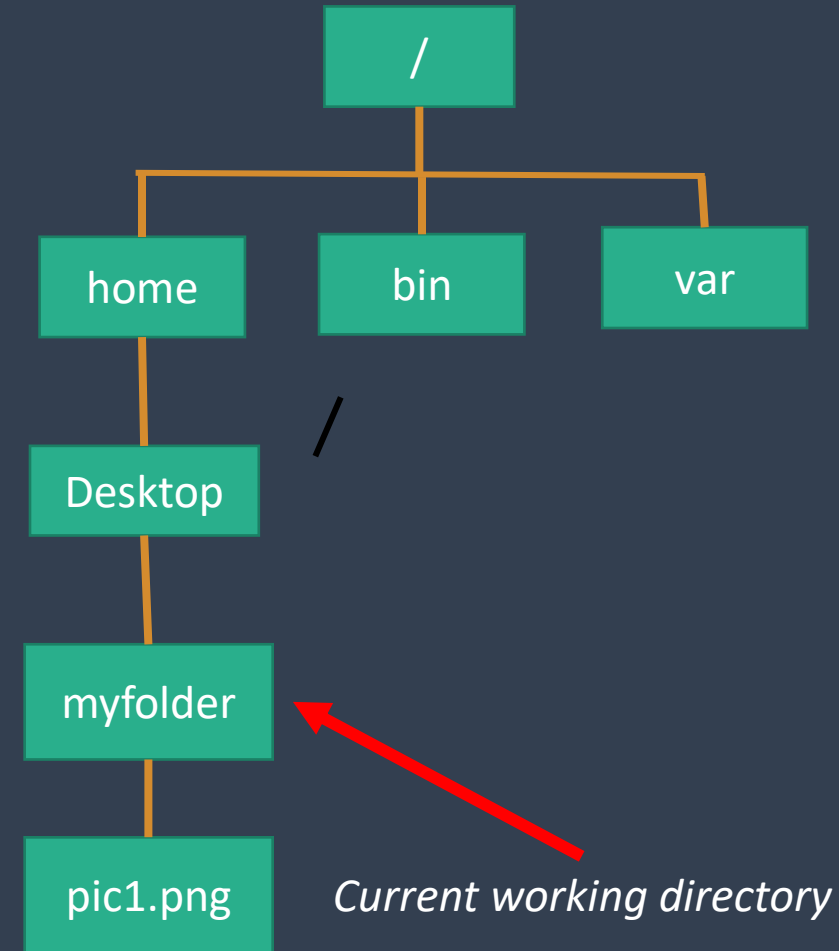
- The **ls -l** options reports 3 pieces of information for files/folders:
 - File type: file (-), folder (d) or hyperlink (l)
 - File permissions: read (r), write (w), execute (x)
 - Owner/user privileges: user and group

Understanding The Linux File Permissions

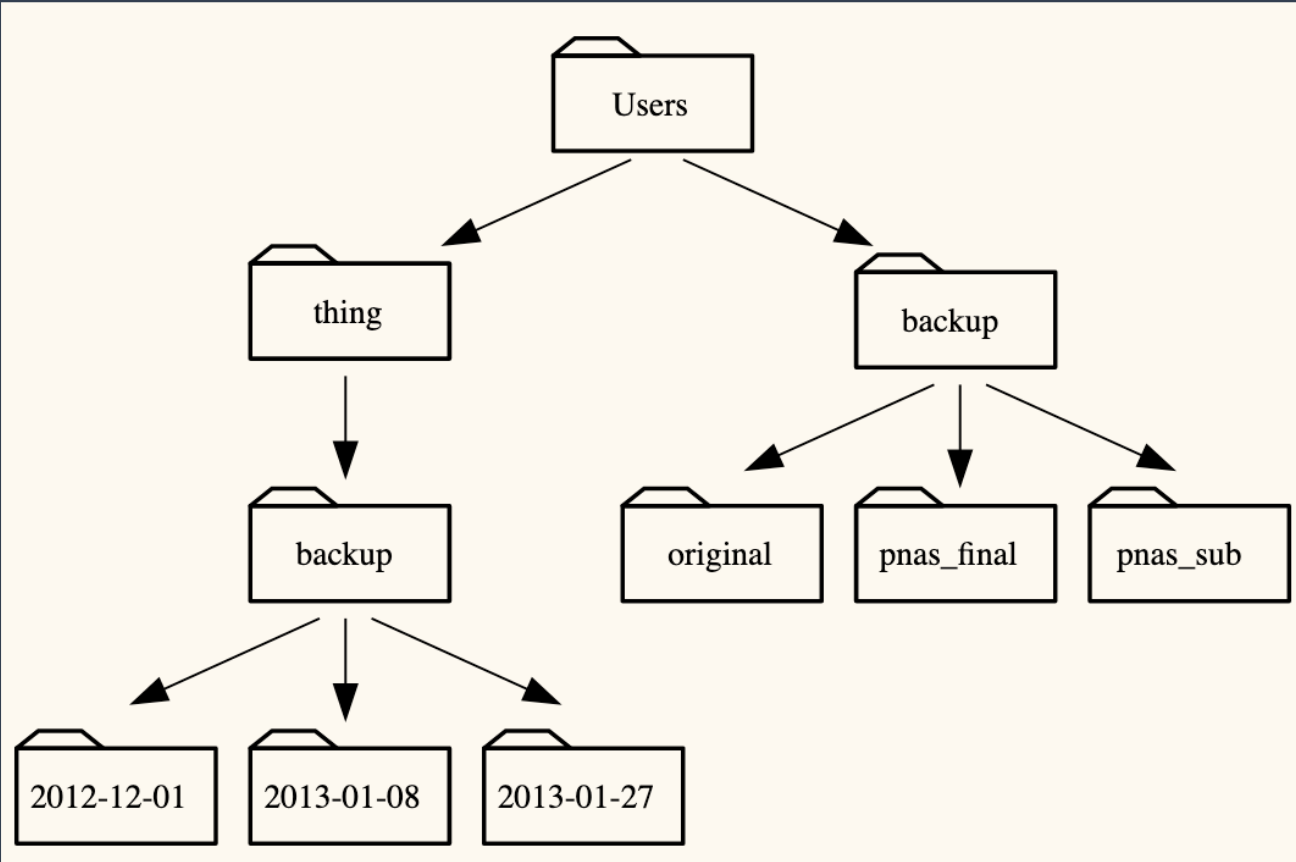


Path of a file or folder

- Path describes the location of a file or folder.
- Path is interchangeable with 'address'
- **Absolute path** is the full path that starts from root
 - `/home/Desktop/myfolder/pic1.png`
- **Relative path** starts from current location.
 - Begins with a single dot ' .' that is shorthand for current directory
 - `./pic1.png`
 - OR double dots ' .. ' that is shorthand for parent directory or one directory up
 - `ls ..` # lists contents in parent directory



Concept Check



1. What is the parent folder of 'thing'?
2. If you are in the 'pnas_sub' and you navigate 2 levels up, which folder would you arrive at?

Let's go somewhere else... Changing location to a different directory

- `cd /path/of/destination_folder/`
- `cd` = 'change directory'
- `cd ..` *#changes to parent folder, one folder up*
- `cd ./some_subfolder` *# changes to a subfolder of the current folder*
- If there is a space in the destination address, then enclose it in quotes
 - `cd '/home/Desktop/Project Folder'`

Creating and renaming files and folders

File commands

- To make a new empty file:

```
touch name.txt
```

- To rename a file

```
mv old_name new_name
```

Folder commands

- To make a new folder:

```
mkdir ./new_folder_name
```

```
mkdir -p ./new_folder/new_subfolder
```

- To rename a folder:

```
mv ./old_name ./new_name
```

Both commands accept absolute and relative paths

Copying and Moving files and folders

- To copy a file:

```
cp file.txt /destination_path/file.txt
```

- To copy a folder:

```
cp -r ./folder /destination_path /folder
```

- To move a file or folder :

```
mv source destination_path
```

- To move multiple files or folders:

```
mv file1 file2 /destination_path
```

```
mv folder1 folder2 /destination_path
```

- To move everything

```
mv * /destination_path
```

* is the *wildcard* symbol which means “any and everything”

Deleting files and folders

- To delete a file or *empty folder*: `rm file`
- To delete a non-empty folder
`rm -r ./folder`

Best Practices for Naming Files/Folders

- Rule #1: Avoid using special characters in a file name
 - \ / : * ? " < > | [] & \$, . are often used under the hood by the operating system for its own purposes, and may cause issues.
- Rule #2: Use under_scores and CamelCase instead of periods or spaces.
'Project folder' → 'Project_folder'
- Rule #3: Use short (<30 characters) but descriptive names .
 - 'notes.txt' → 'lab_meeting_notesMay15.txt'
 - 'Project_folder' → 'fungal_analysis_Sept'

Finding files and folders

- 'find' command searches for a file or folder in a target location

Syntax:

- **find** targetpath -search_criteria file
- E.g. `find ../Desktop/projectb -name mydoc.txt`
- We can also use wildcards to broaden our search
- `find ../Desktop/2020reports -name October*2020.txt`

Compression and Decompression of files

- To compress a file : `gzip file`
- To decompress a .gz file : `gunzip filename.gz`
- To decompress a tar.gz file: `tar -xzf file.tar.gz`
- To decompress a tar file: `tar -xf filename.tar`

Exercise 1

1. What is the name of the largest file in our working folder?
2. Create an empty folder named “new_folder” within the project folder.
 - i. Go to the empty folder and create an empty txt file there.
 - ii. Move back the parent directory
 - iii. Rename the “new_folder” to “FolderB”

File handling and Data Wrangling

What we will do

1. Display and format text with echo command
2. Create and manipulate bash variables
3. Read and display the contents of a file
4. Filtering specific information from files
 1. grep, cut, wc -l
5. Sorting data in a file
 1. sort
6. Learn about standard input and standard output streams
7. Write results to files
8. Build a data processing pipeline using 'pipes'

Printing and formatting data with echo

- 'echo' command displays its arguments on the screen.
- `echo 'Hello World'`
- `echo` can also take options to modify how the data is displayed:
 - `echo -n` # removes trailing newline
 - `echo -e` # enable special backslash\escaped characters (\n, \t)
 - `echo -E` # disables backslash escaped characters, and displays them

Read and display information from files pt.1

- **'cat'** reads and displays the entire contents of a file
 - `cat DM_genes.txt`
- **head file.txt** displays the first 10 lines by default
 - `head -n x file.txt` #displays the first x lines
- **tail file.txt** displays the last 10 lines by default
 - `tail -n x file.txt` #displays the last x lines

Display information from files pt.2: less gives you more

- The '**less**' command allows you to view the contents of a file and navigate through that file.
 - **less** NB_CDS.fasta
 - Press '**q**' to exit back to the terminal display
- To display the file starting a **nth** line, add the **+n** option:
- E.g. **less +20 file.txt** #starts paging from the 20th line
- Pattern-matching: To open a file at the first occurrence of a pattern (word, motif, etc), use the following syntax:
 - **less +/pattern file.txt**

Defining and using bash variables

- To write programs we need variables.
- Like any programming language, we can create and use variables in Bash.
- To create a variable: **var='some information'**
- To reference or use a variable, type a dollar sign \$ at the beginning of the variable name:
- For example, this is how we print the contents of a variable

echo \$var

- Delete variables with **unset** command.

unset var

- We can also pass the output of a command into a variable:

var=\$(bash command)

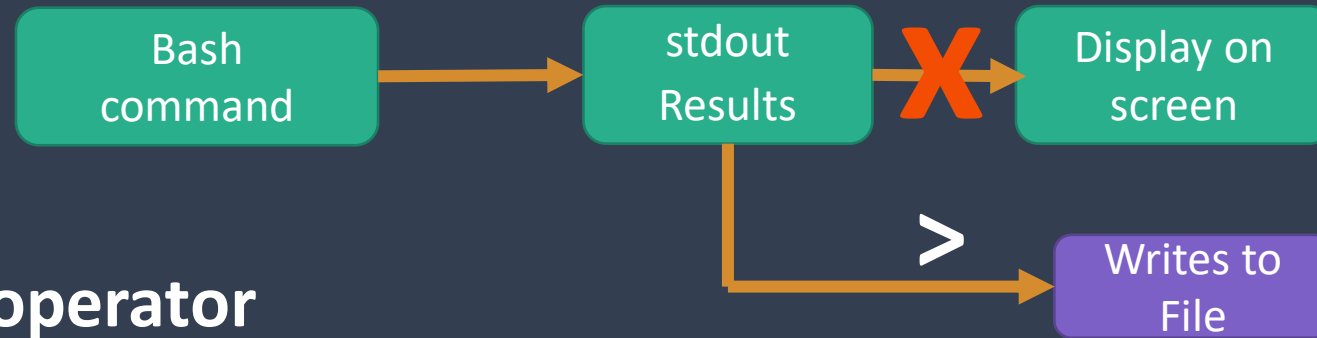
Standard Data Streams

- 'Stream': something that can transfer data.
- There are 3 separate data streams for the command-line:
 - standard input (stdin)
 - standard output (stdout)
 - standard error (stderr)



Output Redirection

- An essential command-line skill is to *redirect* or store the output of a bash command to a file.



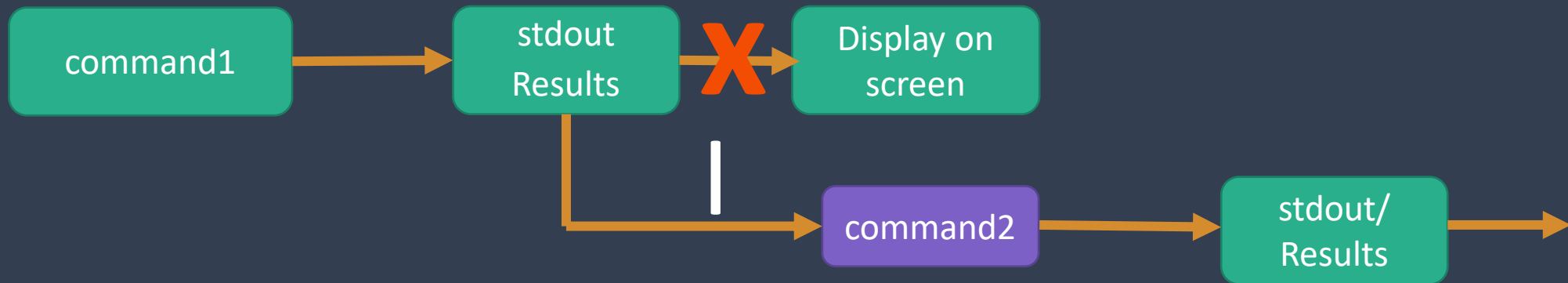
- **'>' is a redirection operator**
- **'>' writes to a new file. If the file already exists, its contents will be overwritten.**

E.g. `echo 'hello world \n this is a beautiful time.' > hello.txt`

- **'>>' appends to an existing file without overwriting it.**

Using *pipes* to connect bash commands

- We can also redirect the stdout from a command to be used as input for another command. This is referred to as 'piping'. Powerful stuff.



- ' | ' is known as the pipe operator
- Pipes enable us to build data processing pipelines on the command-line.

`head -n 20 file.txt | tail -n 5`

- We can combine any number of commands in a pipeline

More commands for filtering and extracting data

wc (word count)

- **wc -l** #outputs the number of lines in the file

'cut'

- The **cut** command extracts parts of a lines of text. Especially useful for extracting data from columns (fields) of a table (default tab-delimited).
 - **cut -f1** #extracts data from the first column
 - **cut -f1,5** #extracts data from the 1st and 5th columns
 - **cut -f3-5** #extracts data from the 3rd to 5th column (inclusive)
 - **cut -f2-4,6** #extracts data from the 2nd to the 4th columns, and the 6th col
- We can specify the type of delimiter using the **cut -d** option
- **cut** is not limited to tabular data. It can operate on text file or input.

```
echo 'coding in bash is fun' | cut -d ' ' -f3
```

Filtering cont'd: Pattern matching with 'grep'

- 'grep' is adept at searching for patterns in a large amount of text
- syntax: `grep -options pattern filename`
- If a match is found, `grep` returns all the lines containing the target pattern
- To search multiple files:
`grep pattern file1 file2 file3`
- To return only the matched pattern instead of entire line: `grep -o`
- To search for exact match only: `grep -w`
- To count the number of matches: `grep -c`
- *grep is case sensitive.* To ignore case during search: `grep -i`
- To show the *n* number of lines after a match use: `grep -A`
- Inverse grep search: to return all instances that DO NOT match the pattern
`grep -v pattern`

Filtering cont'd: Sorting data

- **'sort'** : returns sorted list by ascending (default) order (alphabetically, smallest to largest)
 - `sort DM_genes.txt`
 - `sort -r DM_genes.txt` # reverse sort
 - `sort nums.txt`
- To do a numeric-based sort: **sort -n**
 - `sort -n nums.txt`
- Sorting within a column (k) with start position (s) and end position (e), in table delimited by a character *d* e.g. comma (,), tabs (\t), space ' '
 - `sort -nks,e -t 'd'`
- When sorting by a single column, these numbers will be the same.
 - E.g. `sort -nk1,1` returns sorted list based on values in column 1 only.
 - **Task** : what if we want to save values of the top 20 expressed genes in *Pirfi3* dataset?

Sorting Cont'd

- To find the 10 most frequent occurrences in that column

```
sort k1,1 | tail -n10
```

- **sort | uniq** : uniq takes a list of sorted items and returns all unique instances
 - **uniq -c** : counts the number of times each unique item occurs

Concept Check #2: Data Wrangling with Bash

Practical

- A. How many genes are in the *Acia1_CDS.fasta* file?
- B. **Motif finding:** how many times does the motif 'GATCCA' occur in the in the *Acia1_cds.fasta*
- B.ii Retrieve and store only the DNA sequences from *Acia1_CDS.fasta* into a an output file called 'Acia1_mRNA_seqs_only.txt'
- C. How many genomic features are in the *S.cerevisiae* gtf file?
- D. Extract first 4 columns from *S.cerevisiae* gtf file, and store into a new file called "S_cer.bed"
- E. Extract and save the Start, End and Attributes columns of only 'CDS' features from the *S.cerevisiae* gtf file

Filtering cont'd: Text processing with sed

- 'sed' (Stream Editor) is a versatile and powerful command-line tool for line-by-line text processing.
- sed can process text that in input from a file or piped stream. However, by default sed does not edit the file in place.

Syntax: `sed -options 'pattern' input`

- Here are a few (of many) useful sed commands:

1. Replace a single instance of a pattern: `sed 's/old/new/'`
2. Replace all instances of a pattern: `sed 's/old/new/g'`
3. Extract the i^{th} line from a file/stream: `sed -n 'ip'`
4. Extract line i up to line j : `sed -n 'ip;jp'`
5. Delete a line that contains the pattern: `sed '/pattern/d'`
6. Delete all line that do NOTVmatch pattern: `sed '/pattern/!d'`

Bash scripting Pt2. Outline

1. Data Structures: String and Lists/Arrays
2. Implement control structures
 1. For loops
 2. While loops
 3. If-then-else
3. Write functions
4. Save and execute bash code as scripts/programs
 1. Read input into a script

String Manipulation Pt.1

- Let's define a string variable as str: `str='this is text data.'`
 - Get length of a string: `echo ${#str}`
 - To get the substring from specific positions: `${str:start:end}`
 - `echo ${str:0:4}` #prints the first 4 characters
 - `echo ${str:5}` #prints all characters starting at position 5
 - To remove characters from beginning (prefix) of string: `${str#prefix}`
 - `echo ${str#this}` #displays string without 'this'
- To remove characters at the end (suffix) of string: `${str%suffix}`
- `echo ${str%data}` #displays string without 'data'
- To replace a word with another: `${str/old/new}`
- `echo ${str/text/string}` #replaces 'text' with 'string'

Arrays/Lists in Bash

- An **array** is a variable containing multiple values separated by white space. *Similar to python lists*
- Declare an array by enclosing values between ():
 - `nums=(1 2 3 4 5)`
- To index an array and access its elements, use curly braces {}:
 - `echo ${nums[0]}` #prints 1st element
 - `echo ${nums[4]}` #prints 5th element
 - `echo ${nums[@]}` #prints all elements in array
- To extract sequence of elements from an array:
 - `echo ${array[@]:start:end}`

Arrays/Lists in Bash

- Arrays are useful for capturing multiple values from the output of a bash command.

Example: create a list of all files in the current folder

```
files=$(ls *.*)
```

- We can also create an empty array:

```
declare -a myarr
```

- Then add values to the existing array:

```
myarr+=(1)
```

```
myarr+=(shoe)
```

```
myarr+=($(ls . * | grep -c txt))
```

```
echo ${myarr[*]}
```

For loops

Syntax:

```
for <item> in <list of items>  
do  
set of commands <item>  
done
```

- For loops can iterate over an array or list of space separated items.
- Combining for loops with arrays is an efficient way to automate the processing of multiple files.

Example #1: iterate over space separated values

```
for name in John Reza Carlos Mike  
do  
echo $name  
done
```

Example #2: Iterate over an array of file names to rename all files in the folder

```
files=(ls * .) #create filename array  
for file in ${files[@]} #notation to loop over arr  
do  
mv $file ./ $file_v2  
done
```

Activity: Combine arrays and for loops to batch process multiple files

If-then-*else-fi

Syntax:

```
if
  some condition
then
  command
else
  some command
fi
```

- *else portion is optional

- Report whether a file contains a certain gene

```
if
grep -q not3 genes.fasta
then
  echo 'not3 found!'
else
  echo 'not3 not found!'
fi
```

While loops

Syntax:

```
while (condition)
do
    commands
done
```

- Combining 'while' and 'read -r' are great for reading and processing a file line-by-line
- Example: convert each gene name to upper case

```
while read -r line
do
    echo $line | tr [a-z] [A-Z]
done < genes_list.txt #input redirection
```

Bash functions

- A function is a named block of statements that will execute within the shell.
- Functions allow us to easily re-use code making the code easier to manage and read.

Syntax for defining a function:

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

- write greeting function:

```
print_msg() {  
    echo 'Welcome!'  
}
```


Handling function arguments

- Often, we need functions to process input data for us.
- We use the `$n` to distinguish the inputs in the function definition , where n is the *position* in which the argument is passed to the function.
- Example: write a function that processes two files. It will display the contents of the first file and prints the number of lines in the second file.

#define the function

```
get_info() {  
    file1=$(cat $1)      #reads the contents of the 1st file into a variables  
    file2=$(cat $2 | wc -l ) #reads and counts number of lines of 2nd file  
    echo -e " The first file contains: \n $file1 "  
    echo -e "The second file contains $file2 number of lines"  
}
```

#call the function on the two input files

```
get_info file1.txt file2.txt
```

Handling function outputs

- We can pass the output of a function into to a variable, a file or even another bash command.
- Example #1: count the number of times “the” occurs in the output of the the get_info() function.

```
get_info file1 file2 | grep -c 'the'
```

- Example #2: write the output of the function to a new file.

```
get_info file1 file2 > get_info.output.txt
```

Running code as executable bash scripts

- Writing code as bash scripts is essential for setting up reproducible data processing pipelines

Steps to writing a bash script

1. Open a new text file using a text editor (notepad, notepad++, etc)

`msg_script.sh`

2. Always write the 'shebang' header on the first line

`#!/bin/bash`

3. Write the desired the bash code in the script

`echo "my first bash script"`

4. Save and close the script.

5. Make the script an executable program

`chmod +x msg_script.sh`

5. Run the script:

`./msg_script`



Congrats!
You have written your
first command-line
tool!

Handling input in bash scripts

We can write bash scripts to accept user input.

Method #1: as positional arguments

- Same as referencing arguments in functions.

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

```
name=$1
age=$2
echo "Your name is $name. Your age is $age."
```

#run the script

```
input_example1.sh Rhondene 28
```

Method #2: "read" command

- More interactive
- Syntax : `read <variable_name>`

Example:

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

```
echo "Enter name"
read name
echo "Enter age"
read age
echo "Your name is $name. Your age is $age."
```

#run the script

```
input_example2.sh
```

Example of Bash Scripts

```
#!/bin/bash

echo "Enter the gene ids"
read genelist

for gene in $genelist
do
    echo $gene
done
```