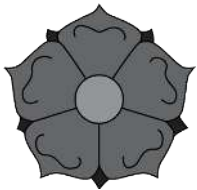


Workshop on Genomics 2025

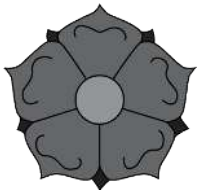


unix



Mercè Montoliu Nerín

January 7th, 2025



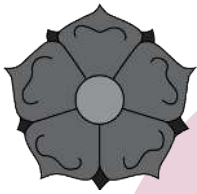
What is UNIX?

Operating system

powerful

multi-user

multitasking



Why is it important for bioinformatics?

Facilitates sharing and reproducing analyses

Access to powerful tools and applications

**Efficiency
and
speed**

Handling large datasets and running analyses efficiently

Using scripts to automate repetitive tasks



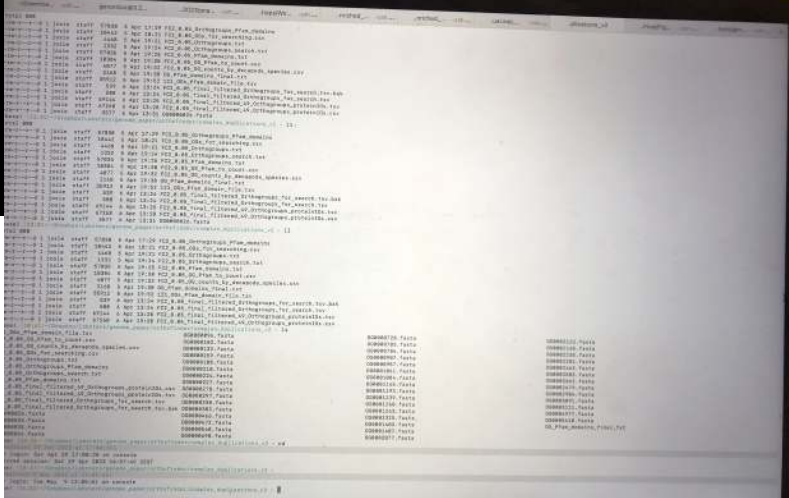
The terminal



Make it comfortable to work in

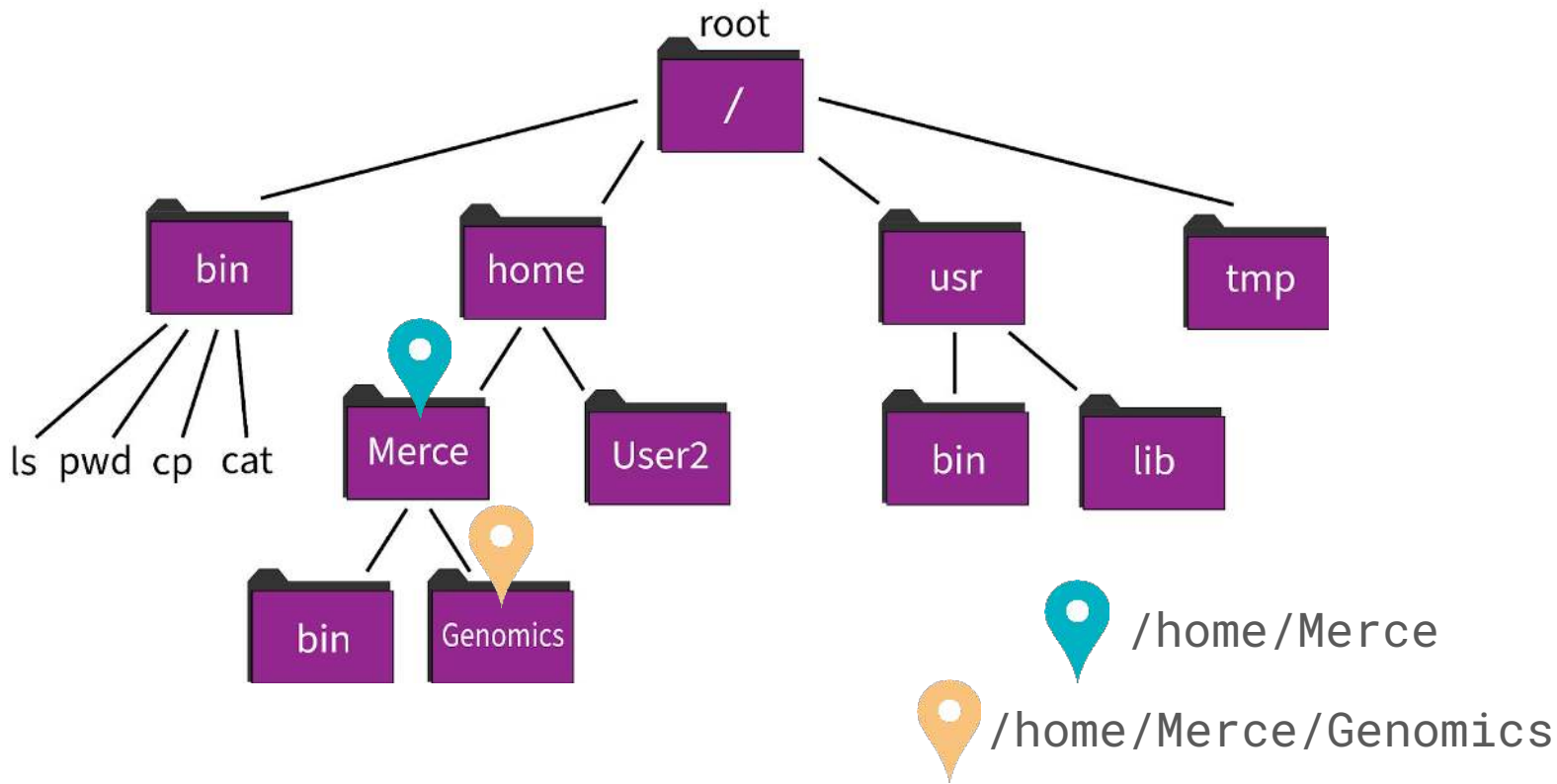
- Resize the window
- Change the font size
- Open multiple terminal windows (or tabs)
- Make sure you have the right combination of colours that work for **you**.

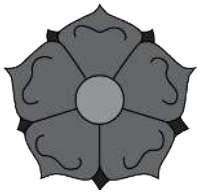


[illegible]

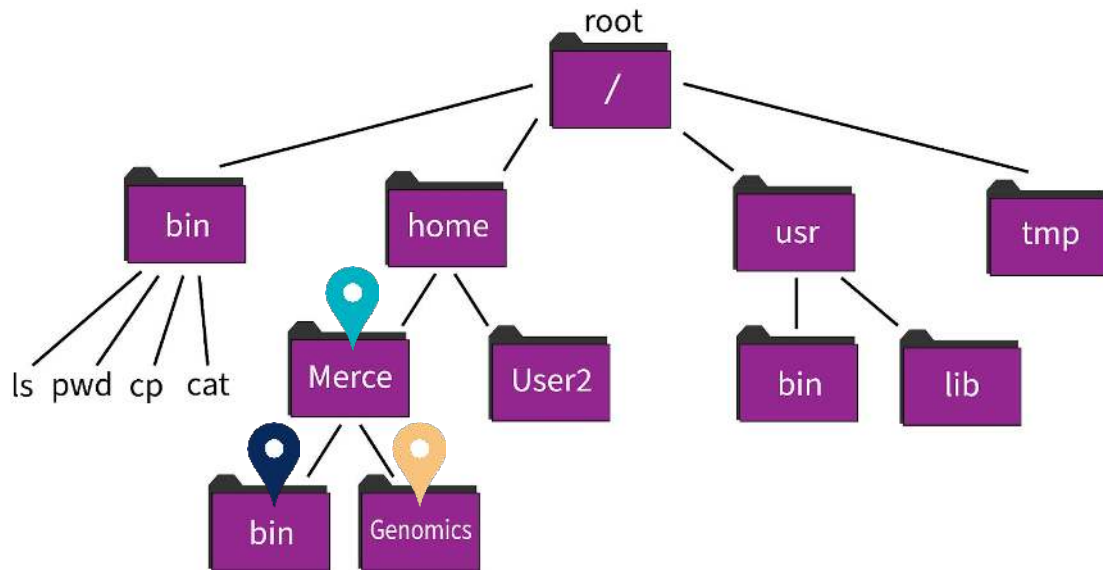


File system organization





Paths - Absolute vs Relative



Absolute paths



/home/Merce



/home/Merce/Genomics



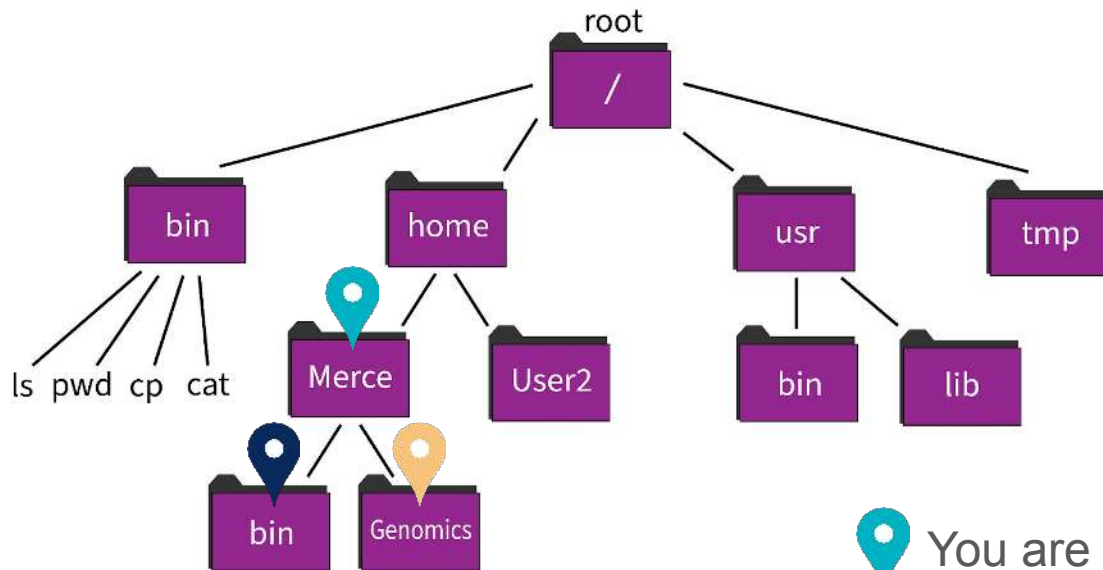
/home/Merce/bin

. refers to our current location

.. refers to the location above us



Paths - Absolute vs Relative



Absolute paths



/home/Merce



/home/Merce/Genomics



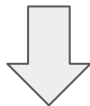
/home/Merce/bin



You are here!

Absolute: /home/Merce

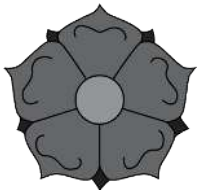
Relative: ./



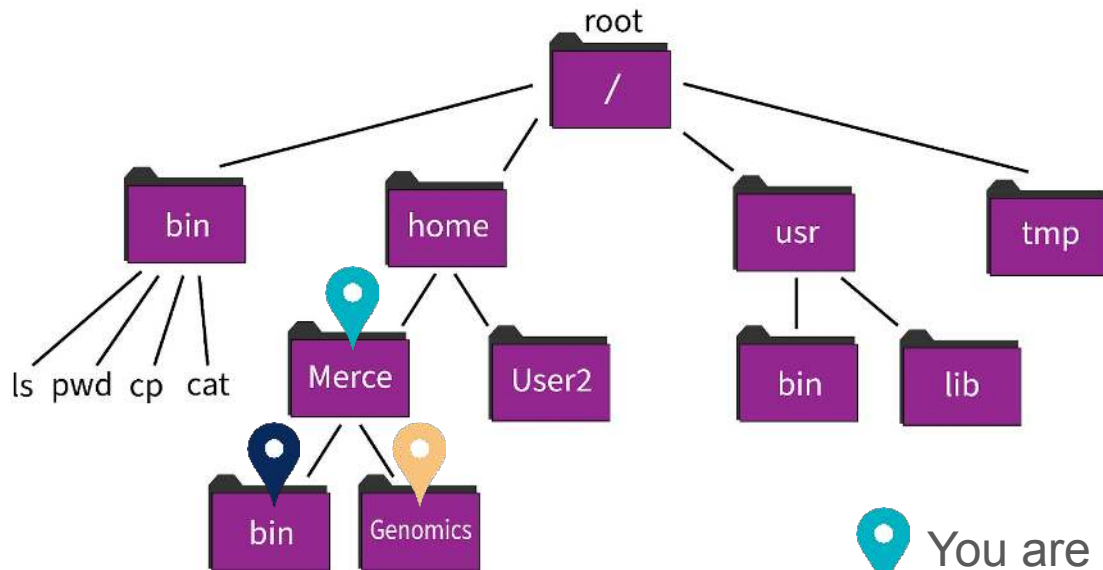
../Genomics

. refers to our current location

.. refers to the location above us



Paths - Absolute vs Relative



Absolute paths



/home/Merice



/home/Merice/Genomics



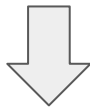
/home/Merice/bin



You are here!

Absolute: /home/Merice

Relative: ./



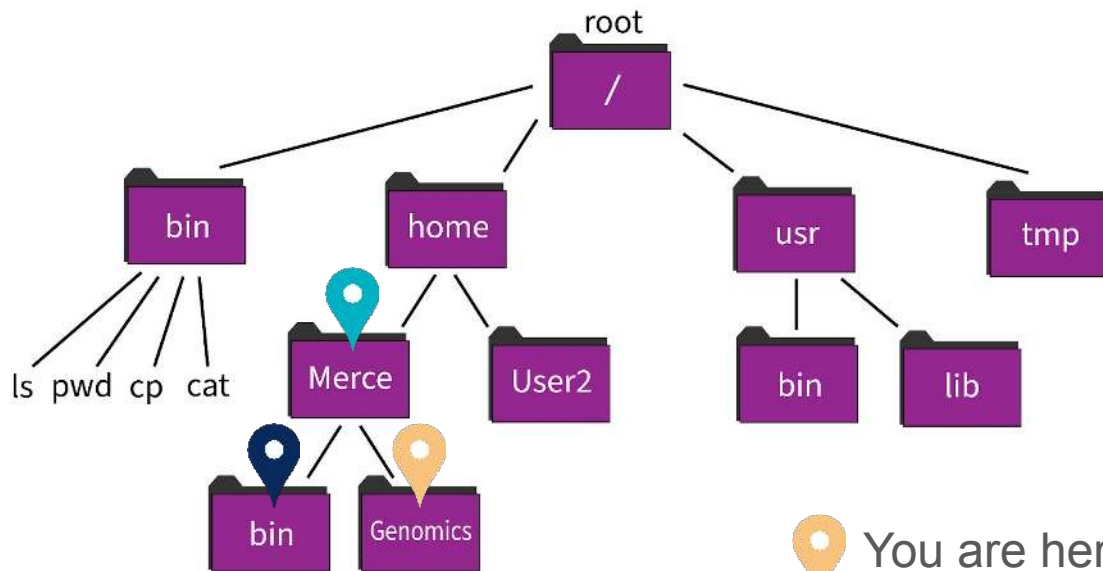
./bin

. refers to our current location

.. refers to the location above us



Paths - Absolute vs Relative



Absolute paths



/home/Merce



/home/Merce/Genomics

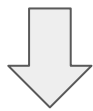


/home/Merce/bin



You are here! Absolute: /home/Merce/Genomics

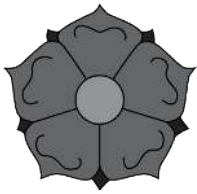
Relative: ./



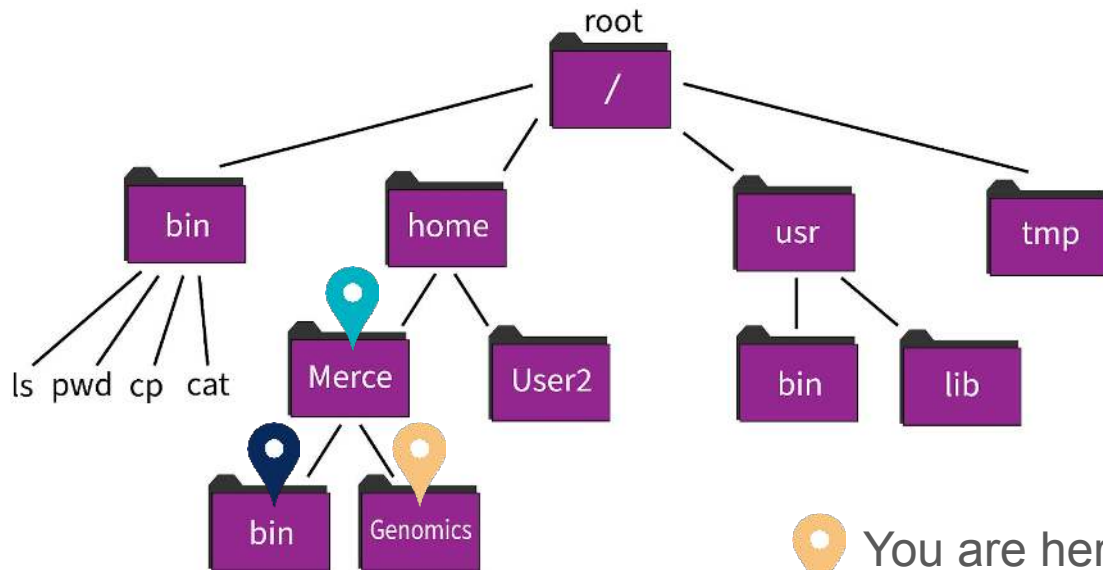
..

. refers to our current location

.. refers to the location above us



Paths - Absolute vs Relative



Absolute paths



/home/Merice



/home/Merice/Genomics

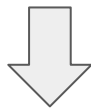


/home/Merice/bin



You are here! Absolute: /home/Merice/Genomics

Relative: ../



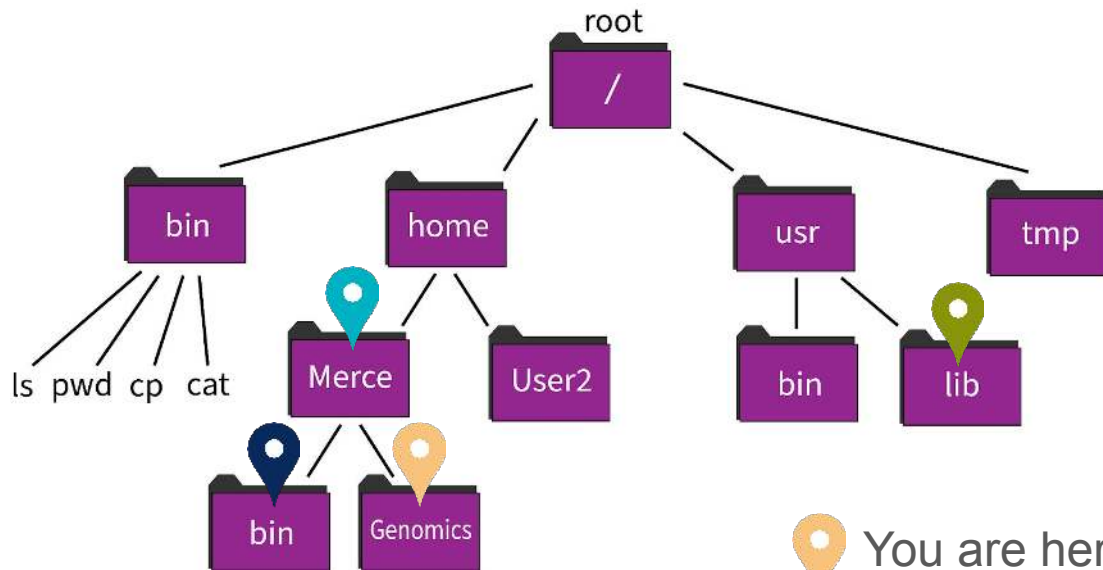
../bin

. refers to our current location

.. refers to the location above us



Paths - Absolute vs Relative



Absolute paths



/home/Merce



/home/Merce/Genomics



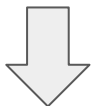
/home/Merce/bin

. refers to our current location

.. refers to the location above us



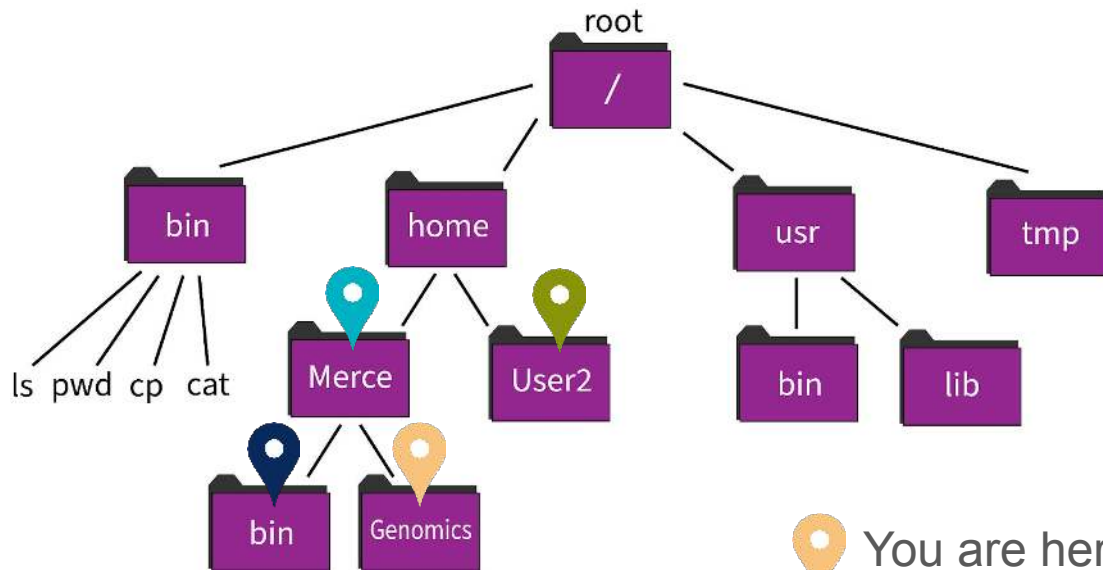
You are here!



How do we refer to this directory?



Paths - Absolute vs Relative



Absolute paths



/home/Merce



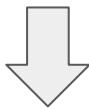
/home/Merce/Genomics



/home/Merce/bin



You are here!



../ ../User2

. refers to our current location

.. refers to the location above us



File system navigation

pwd - where am I?



cd - change directory





File system navigation

pwd - where am I?



/home/Merce

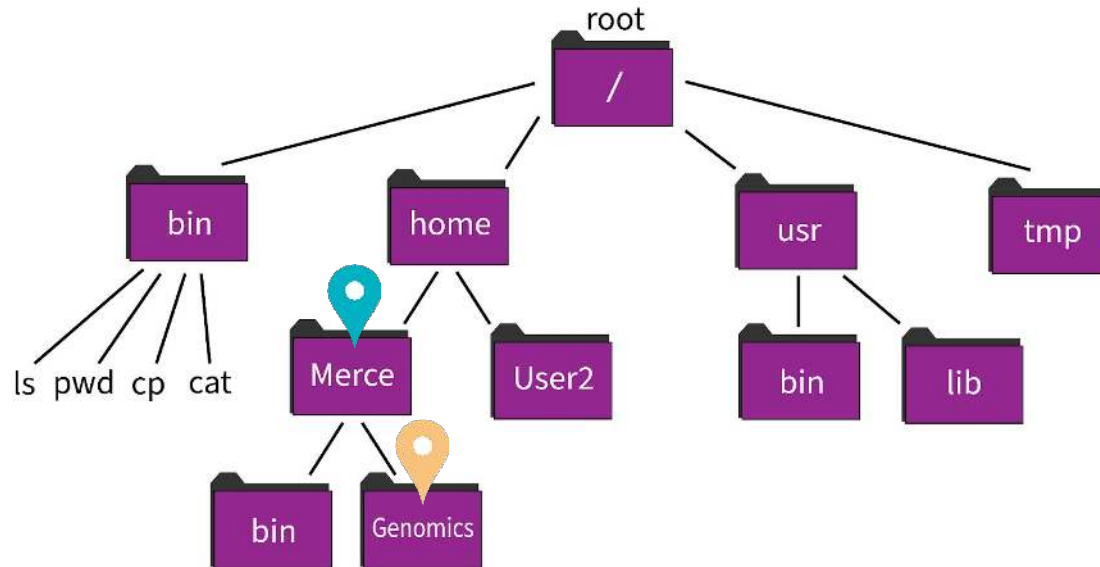


cd - change directory



```
> cd /home/Merce/Genomics
```

```
> cd ./Genomics
```





File system navigation

pwd - where am I?



/home/Merce/Genomics

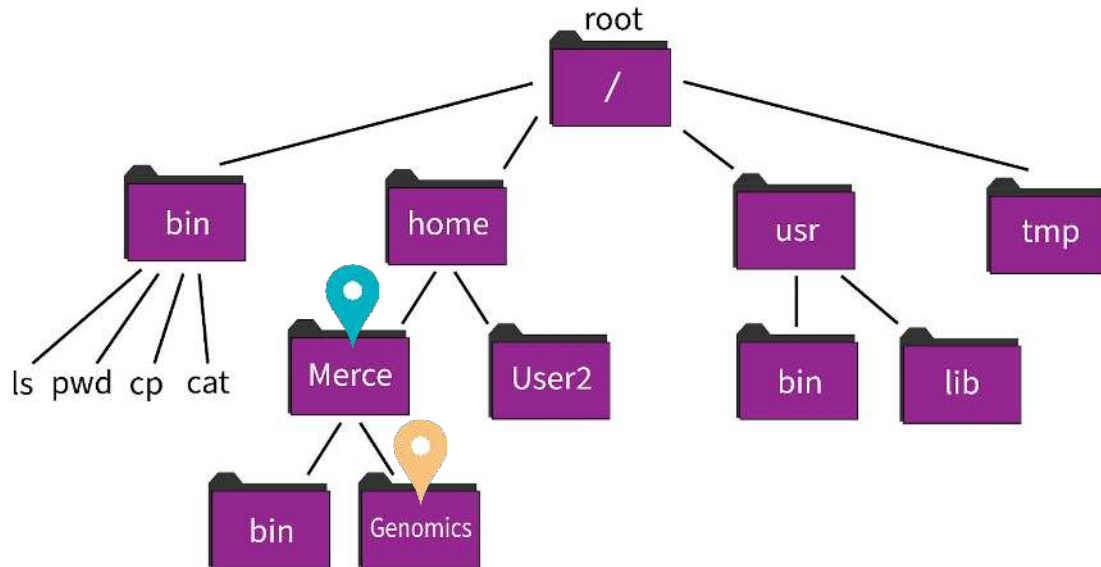


cd - change directory



```
> cd /home/Merce
```

```
> cd ../
```

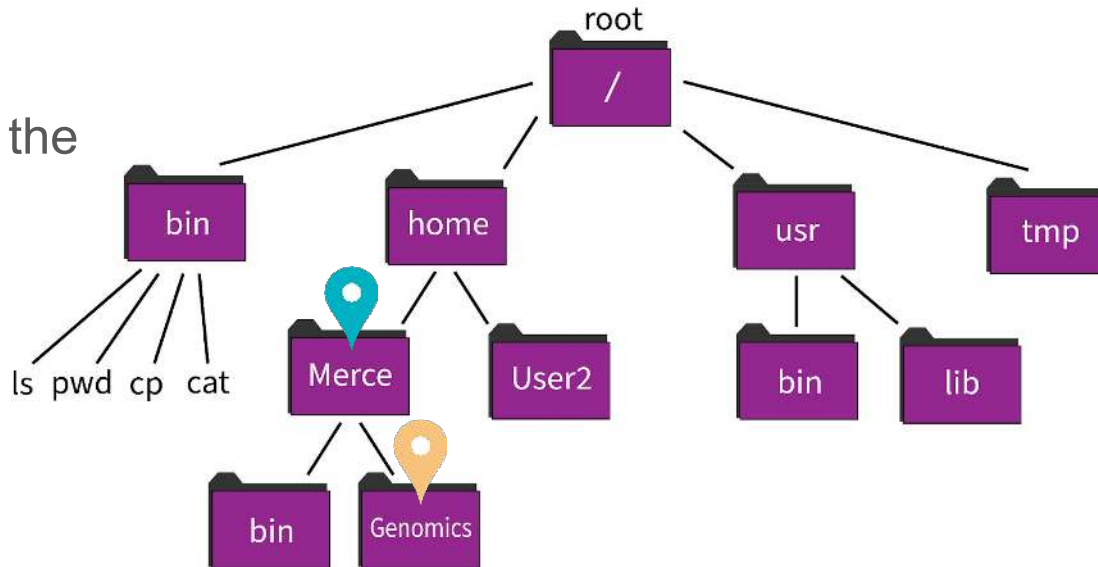




File system visualization

ls - shows you the contents the
directory you are in

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



```
> ls ../
```



Key shortcuts

Ctrl + C	halts current command
Ctrl + Shift + C	copy (linux) - Cmd + C (mac)
Ctrl + Shift + V	paste (linux) - Cmd + V (mac)
Ctrl + W	erases one word in current line
Ctrl + U	erases whole line
Ctrl + A	go to begining of line
Ctrl + E	go to end of line
Type <code>exit</code>	log out of current session



Create, copy, move, and remove files and folders

mkdir - create new directory

cp - copy file

mv - move file or directory

rm - remove file

"Unix was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things." - Doug Gwyn



Symbolic links

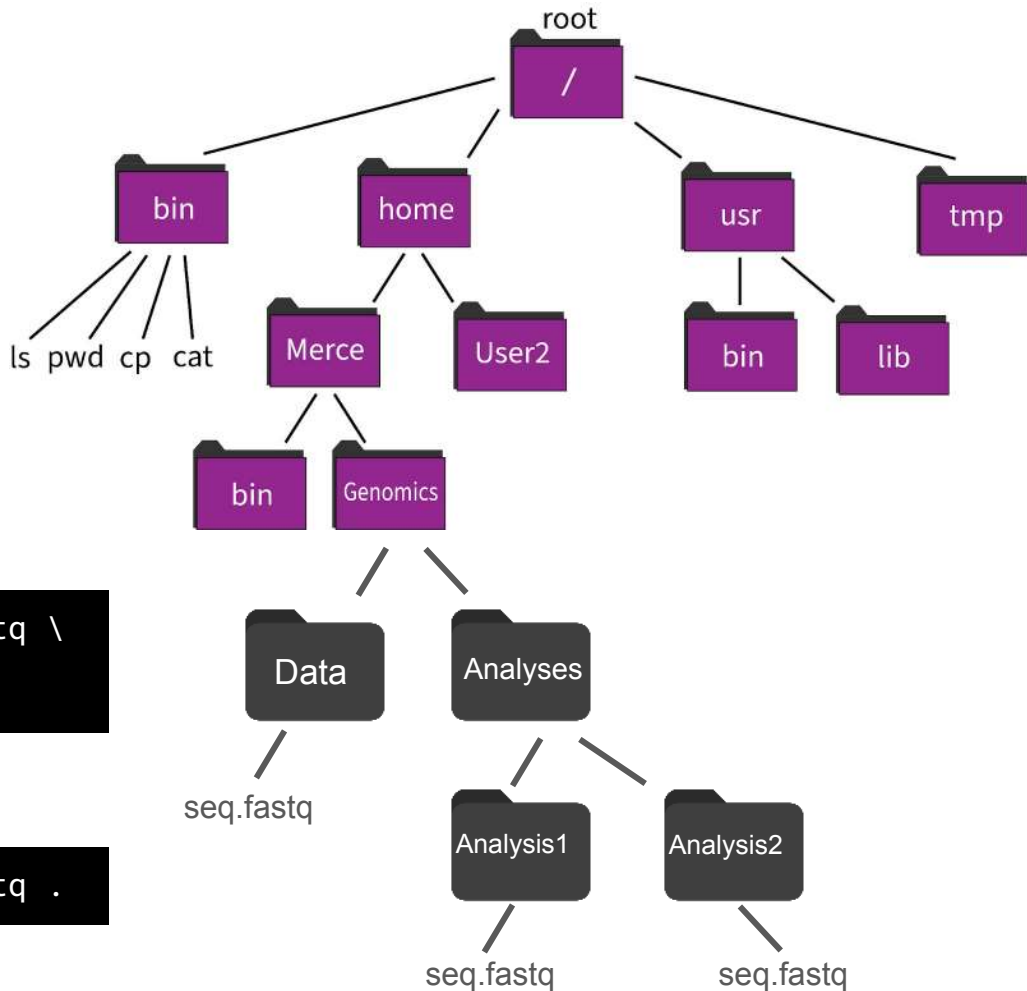
In **-s** /path/to/file link

create a symlink of file

```
> ln -s /home/Merce/Genomics/Data/seq.fastq \
/home/Merce/Genomics/Analyses/Analysis1/
```

If we are already inside the folder Analysis1:

```
> ln -s /home/Merce/Genomics/Data/seq.fastq .
```





Manual

man *command* - manual of the command

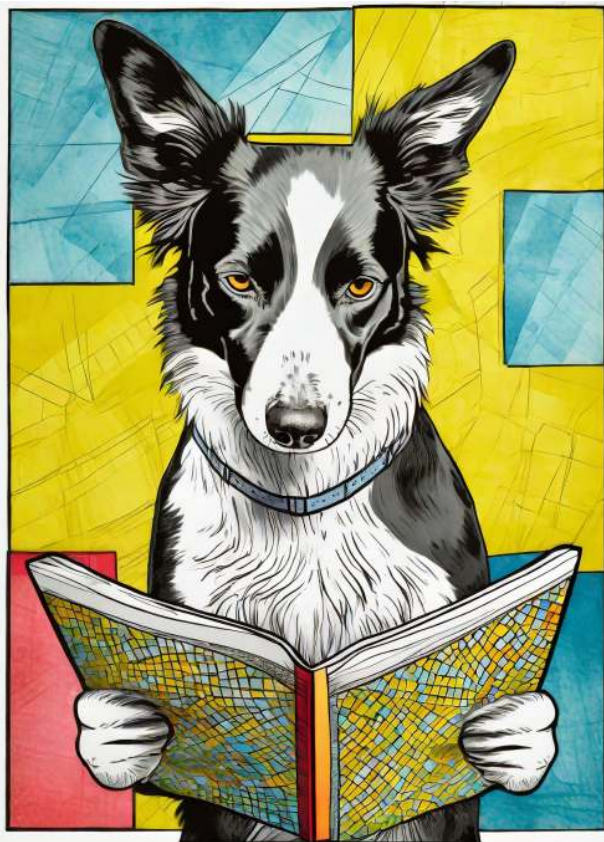
```
> man ls
```



ls -l formatted list

ls -h “human” formatted list

ls -lh combination of flags



Inputs and outputs

stdin It stands for standard input, and is used for taking text as an input.

stdout It stands for standard output, and is used to text output of any command you type in the terminal, and then that output is stored in the stdout stream.

stderr It stands for standard error. It is invoked whenever a command faces an error, then that error message gets stored in this data stream.



stdin, stdout, stderr

command *stdin*

if it works: prints in our terminal the *stdout*

if it fails: prints in our terminal the *stderr*

command *stdin* > *stdout*

if it works: *stdout* is redirected to a file

if it fails: prints in our terminal the *stderr*

command2 *stdin2* > *stdout*

stdout is redirected to a file and rewrites its contents

command2 *stdin2* >> *stdout*

stdout is redirected to a file and appended after its contents



stdin, stdout, stderr

command *stdin*

if it works: prints in our terminal the *stdout*

if it fails: prints in our terminal the *stderr*

command file1 > output.txt

stdin *stdout*

if it works: *stdout* is redirected to a file

if it fails: prints in our terminal the *stderr*

command2 file2 > output.txt

stdin *stdout*

stdout is redirected to a file and rewrites its contents

command2 file2 >> output.txt

stdin *stdout*

stdout is redirected to a file and appended after its contents



stdin, stdout, stderr

command file1 2> errors.txt
stdin *stderr*

if it works: prints in our terminal the *stdout*
if it fails: *stderr* is redirected to a file

command file1 &> output.txt
stdin *stdout&stderr*

redirects both *stdout* and *stderr* to a file

command file1 > output.txt 2> errors.txt
stdin *stdout* *stderr*

redirects both *stdout* and *stderr* to a
separate file each.



Explore file content

wc - word count (-l lines, -c characters, -w words)

less - visualize file contents in your terminal screen (press q to exit)

cat - prints contents of your file as *standard output* in your terminal

head - visualize the first 10 lines of a file

tail - visualize the last 10 lines of a file

The character **|** (**pipe**) is used to concatenate commands, so that we can run one command after the other, avoiding the creation of intermediate files.

```
command1 input | command2 > output
```

Instead of :

```
command1 input > output1  
command2 output1 > output2
```

Using pipe, the output of running *command1* on a given input gets directly piped into *command2*, and we obtain an output of these two consecutive commands, generating only one output.

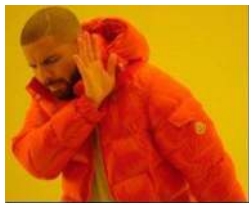


A bit more advanced file-handling commands

cat - prints contents of your file as *standard output* in your terminal

redirect to a command

```
cat fileA | command > output.txt
```



cat | grep



grep

concatenate files

```
cat fileA fileB >> fileC
```

```
cat fileA > fileC
```

```
cat fileB >> fileC
```



A bit more advanced file-handling commands

sort - puts in certain order a series of lines in our file

`sort -r fileA` sorts in reverse order

`sort -n fileA` sorts lines in fileA numerically

`sort -k 2 fileA` sort fileA by column 2

`sort -k 2nr fileA` sort fileA by column 2, numerically and in reverse order

`sort -V fileA` sort lines in fileA numerically natural.

`sort -u fileA` sort lines and removes duplicates -> `sort fileA | uniq`



A bit more advanced file-handling commands

Are these two files different?

diff - can tell us if there are differences between two files

```
diff -q fileA fileB
```

“Files fileA and fileC differ”

```
diff fileA fileB
```

prints differences





A bit more advanced file-handling commands

Splitting a file

split - split a given file into multiple files (default 1000)

```
split -l 20 fileA
```

produce x number of files from fileA, each containing 20 lines.

cut - extract specific parts of a file

```
cut -c 2 fileA
```

extract specific columns from a file



Text editors

Nano - The simpler option of text editor. All commands within the nano text editor are given by pressing the Control-key, usually represented as ^

- ^S save current file
- ^O save to (a different file)
- ^X exit from nano



Text editors

Vim - a highly configurable text editor built to make creating and changing any kind of text very efficient

- i start insert mode (you can start typing after where your cursor is)
- ESC exits insert mode (also Ctrl + C)
- :w save file without exiting
- :q exit file (if there are unsaved changes, it fails)
- :wq save and exit
- :q! exit without saving changes



Text editors

emacs - a text editor characterized by its extensibility and configurability. Some essential commands get activated by typing Control + X, then the command (while holding the control key), but there is a wide range of key combinations to be used to move and edit the text

Ctrl + x + s save file

Ctrl + x + c exit editor (if not saved, it ask if you want to save, then type "yes")

VIM

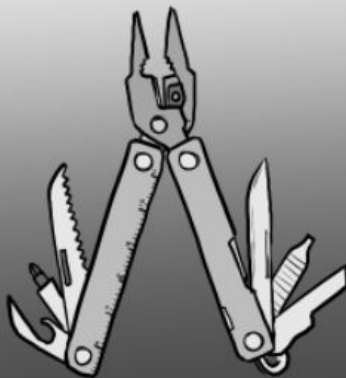
usable in just about
any environment.

does one thing, well.



EMACS

flexible, customizable, and
packed with every feature
known to man.



NANO

mostly used by people
who do not know
what they are doing;
or psychopaths.





Bash scripts

Shell scripts often have the suffix `.sh`

Shell scripts must be executable `chmod 755` or `chmod +x`

Comments can be written in scripts with a `#`

Variables can be used to shorten long paths

Shell loops can be used to process lots of files

`\` can be used to wrap long commands across multiple lines

`#!/bin/bash` must be the first line, it specifies interpreter



Bash scripts - Variables

We can save variables under almost any name.

Variables can be string type:

```
evomics="Workhop_on_genomics_2024"
```

```
data="genome_assembly_file.fasta"
```

```
path="/home/genomics/workshop_materials/unix"
```

Integer type:

```
num=5
```

or float type:

```
pi=3.14
```

We can refer to the variables
using a dollar sign:

```
$evomics
```

```
${evomics}
```



Bash scripts - *for* loops

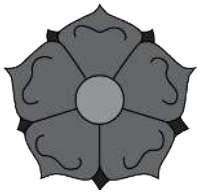
Loop over files inside a directory:

```
for file in ./unix/working_directory/*fastq
do
    commands $file
done
```

Loop over files that we stored inside a variable:

```
files="file1
file2
file3
file4"

for file in $files
do
    commands $file
done
```



Bash scripts - *while* loops

while loops

```
while read line
do
    command $line
done
```

We can pipe the command `ls -l` to this script to run the command on each of the files listed.



Bash scripts - Stay informed!

If we want to print messages to the standard output while the script is running we can do that using the echo command. This is specially useful when running a long pipeline of multiple commands, so that we can keep track of the stage that is currently running.

```
for file in ./unix/working_directory/*fastq
do
    echo "Command 1 is running on $file"
    command1 $file
    echo "Command 2 is running on $file"
    command2 $file
done
```




Bash scripts - Some tricks!

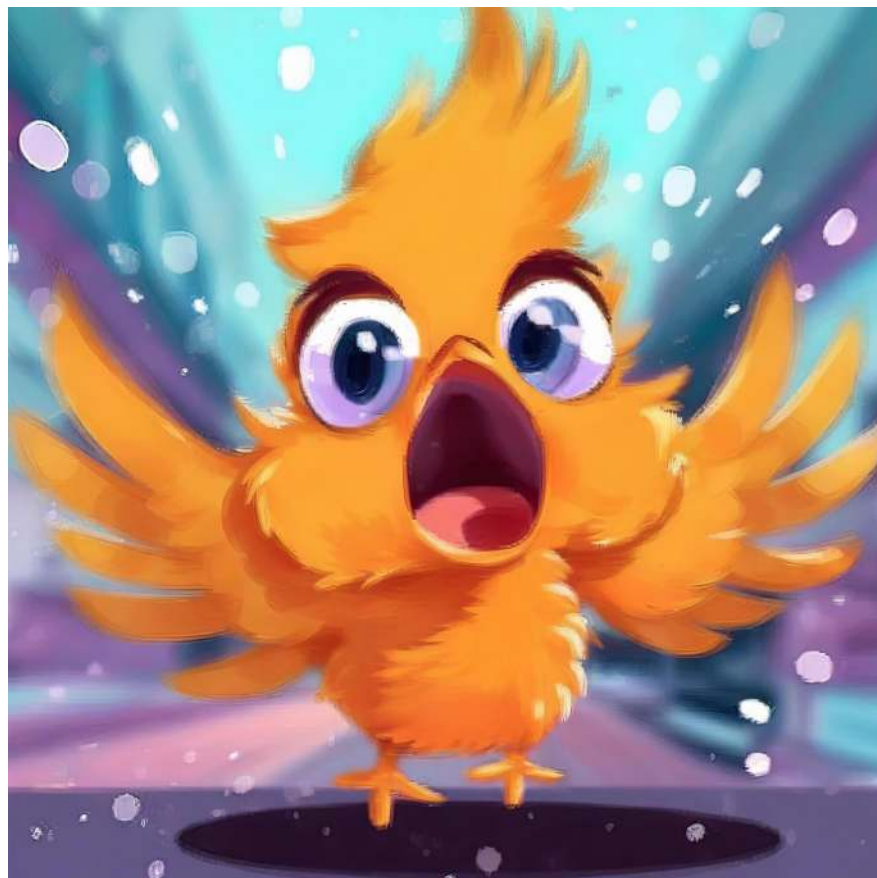
```
for file in ./unix/working_directory/*fastq
do
    file_name=$(basename $file ".fastq")

    command1 $file -out ${file_name}_command1.fastq
done
```

It is the same as:

```
for file in ./unix/working_directory/*fastq
do
    file_name=$(basename $file ".fastq")

    command1 $file -out "$file_name"_command1.fastq
done
```





What my family and friends
think I do



What my supervisor thinks I do



What I actually do

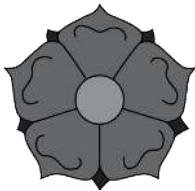
Google



ChatGPT



stackoverflow



Cheat-sheet

Workshop on Genomics 2024

unix cheatsheet

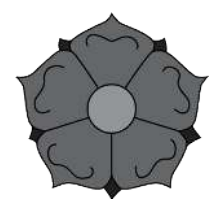
<code>pwd</code>	show current path / directory
<code>ls</code>	list directory
<code>cd <i>dir</i></code>	change to directory <i>dir</i>
<code>cd</code>	change to home
<code>cd ~</code>	change to home
<code>cd -</code>	change to previous working directory
<code>.</code>	current directory
<code>..</code>	parent directory
<code>mkdir <i>dir</i></code>	create directory <i>dir</i>
<code>cp <i>file1 file2</i></code>	copy <i>file1</i> to <i>file2</i>
<code>mv <i>file1 file2</i></code>	move <i>file1</i> to <i>file2</i> or rename <i>file1</i> to <i>file2</i>
<code>rm <i>file1</i></code>	delete <i>file1</i>
<code>ln -s <i>file link</i></code>	create symbolic link

<code>wc</code>	count (-l lines, -w words, -c characters)
<code>tail <i>file</i></code>	output last 10 lines of file
<code>head <i>file</i></code>	output first 10 lines of file
<code>less <i>file</i> / more <i>file</i></code>	visualize contents of file
<code>cat <i>file</i></code>	output file to standard output
<code>sort</code>	sort rows
<code>diff <i>fileA fileB</i></code>	differences?
<code>uniq</code>	keep unique rows
<code>cut -c 2</code>	cut column 2

<code>man command</code>	manual for command
<code>chmod +x <i>file</i></code>	makes <i>file</i> executable
<code>command1 command2</code>	runs <i>command2</i> on output of <i>command1</i>
<code>wget <i>web-address-to-file</i></code>	download file into current dir
<code>ssh <i>user@server</i></code>	connect to cluster
<code>tar -xzf <i>archive.tar.gz</i></code>	uncompress <i>tar.gz</i>
<code>tar -czf <i>archive.tar.gz archive</i></code>	compress <i>archive</i> to <i>.tar.gz</i>
<code>gzip</code>	compress <i>.gz</i>
<code>gunzip</code>	uncompress <i>.gz</i>

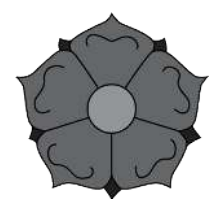
<code>Ctrl + C</code>	halts current command
<code>Tab</code>	autocomplete current line
<code>Arrow up</code>	previous commands
<code>Ctrl + Shift + C</code>	copy (linux)
<code>Cmd + C</code>	(mac)
<code>Ctrl + Shift + V</code>	paste (linux)
<code>Cmd + V</code>	(mac)
<code>Ctrl + W</code>	erases one word in current line
<code>Ctrl + U</code>	erases whole line
<code>Ctrl + A</code>	go to beginning of line
<code>Ctrl + E</code>	go to end of line
<code>exit</code>	log out of current session

#evomics2024



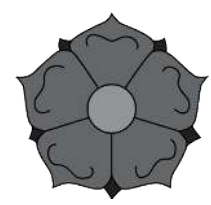
Cheat-sheet





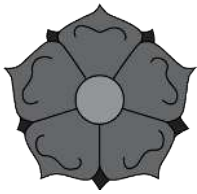
Rubber duck





Now it is your turn to practice!
And become each other's ducks!





Bioinformatic Files

Parsing and Editing



Regular Expressions

The dot (.) represents a single character, any single character. It can be a digit, a letter, a symbol, and even a space.

If we want to match our sequence "ATG", but this time we want to include the next nucleotide, we can do that with "ATG.", in this case, the character after 'ATG' was a 'T', so we find 'ATGT':

```
ATAGCATCAAATGTAGCATTTACGTAGTAGCTATAGCTATTACGTAGGGCTACTTTATAGCATCAAATCTAGCATCTACG
TAGCATCAAATCTAGCACGTACGTAGTAGCTCATGCTATTACGTAGCGCAACAGCTCAACCTCAGGCTACTTTATAGCAT
CAAATCTAGCATTA
```



Regular Expressions

Regular expressions (or RegEx) are like special codes used to search for patterns in text.

Special characters in RegEx

We can match any character using regular expressions, except those that have a special meaning in RegEx.

The below listed characters are special characters in RegEx:

.	+	*	?	{	^
\$	()	[]		\	

HOW TO REGEX

STEP 1: OPEN YOUR FAVORITE EDITOR



STEP 2: LET YOUR CAT PLAY ON YOUR KEYBOARD





Regular Expressions

The star (*) means "zero or more occurrences of the previous character."

We can combine it with the dot in the following way ".*". So we match zero or more occurrences of any character.

If you wanted to find all sequences that start with "ATG" and end with "TAA", you could use a regular expression like "ATG.*TAA", which means "find 'ATG', followed by zero or more of any character, followed by 'TAA'":

ATAGCATCAAATGTAGCATTTACGTAGTAGCTATAGCTATTACGTAGGGCTACTTTATAGCATCAAATCTAGCATCTACGTAGCATC
AAATCTAGCACGTACGTAGTAGCTCATGCTATTACGTAGCGCAACAGCTCAACCTCAGGCTACTTTATATCTAGCATTAACGTAGTA



Regular Expressions

The plus (+) means "one or more occurrences of the previous character".

We can combine it with the dot in the following way ".+". So we match one or more occurrences of any character.

If you wanted to find all sequences that start with "ATG" and end with "TAA", you could use a regular expression like "ATG.+TAA", which means "find 'ATG', followed by one or more of any character, followed by 'TAA'":

ATAGCATCAAATGTAGCATTTACGTAGTAGCTATAGCTATTACGTAGGGCTACTTTATAGCATCAATCTAGCATCTACGTAGCATC
AAATCTAGCACGTACGTAGTAGCTCATGCTATTACGTAGCGCAACAGCTCAACCTCAGGCTACTTTATATCTAGCATTAACGTAGTA



Regular Expressions

The plus (+) means "one or more occurrences of the previous character".

But, what would happen if we have this sequence? Can we use `ATG.+TAA` to match it?

ATAGCATCAAATGTAACATTTACGTAGTAGCTATAGCTATTACGTAGGGCTACTTTATAGCATCAAATCTAGCATCTACGTAGCATC
AAATCTAGCACGTACGTAGTAGCTCATGCTATTACGTAGCGCAACAGCTCAACCTCAGGCTACTTTATAGCATCAAATCTAGCATTC

"`ATG.*TAA`" would be able to match it, but not "`ATG.+TAA`", as it requires that there is at least one character in between 'ATG' and 'TAA'



Regular Expressions

The question mark (?) matches zero or one time the previous character

If we want to match our sequence starting with "ATG" and ending with TAA, and we know sometimes there is a T after ATG, but sometimes not, we can do that with "ATGT?TAA", in this case, the character after 'ATG' can be a T, or can be nothing, and both the following sequences would be matched:

ATAGCATCAAATGTAACATTTACGTAGTAGCTATAGCTATTACGTAGGGCTACTTTATAGCATCAAATCTAGCATCTACGTAGCATC
AAATCTAGCACGTACGTAGTAGCTCATGCTATTACGTAGCGCAACAGCTCAACCTCAGGCTACTTTATAGCATCAAATCTAGCATT

ATAGCATCAAATGTTAACATTTACGTAGTAGCTATAGCTATTACGTAGGGCTACTTTATAGCATCAAATCTAGCATCTACGTAGCAT
CAAATCTAGCACGTACGTAGTAGCTCATGCTATTACGTAGCGCAACAGCTCAACCTCAGGCTACTTTATAGCATCAAATCTAGCATT



Regular Expressions

The curly brackets (`{ }`) can reference the amount of times we expect the previous character to occur. It has three main configurations:

`{m}` - previous character exactly m number of times

`{m,n}` - previous character m to n number of times

`{m,}` - previous character m or more number of times

If we want to find 3 'A' in a row, we could use "`A{3}`"

ATAGCATCATAATGTAGCATTTACGTAGTAGCTATAGCTATTACGTAGGGCTAAAAATAGCATCATATCTAG

We can also specify that we want to find 'A' a minimum of 3 times and a maximum of 5 with "`A{3, 5}`", then we would match:

ATAGCATCATAATGTAGCATTTACGTAGTAGCTATAGCTATTACGTAGGGCTAAAAATAGCATCATATCTAG

(We could get the same result in this specific sequence, by using "`A{3, }`")



Regular Expressions

The caret (^) is used to match the beginning of the line.

So that if we want to match sequences that start with "ATG", we can use ^ATG.

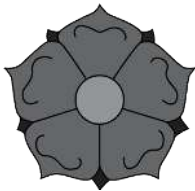
For example, in the following sequences:

ATGATAGCTTAACATTTACGTAGTAGCTATAGCTATT

GTCATGAGCTATTAGCATCACATCTAGCACGTTTCATG

ATGCTATGAAGTCTACTTTATAGCATCAAATCTAGTA

The regular expression ^ATG matches ATG only in the first and third lines because they begin with "ATG"



Regular Expressions

The dollar sign (\$) is used to match the end of a line or string.

If we want to match sequences that end with "TAA", we can use "TAA\$".

For example, in the following sequences:

TATAGCTAAAGTCTACTTTATAATCAATGATAGCTTAA

ATGAGCTATTAGCATCACATCTAGCAGTCATGAGCTAT

GTAGCATTTACGTAGTAGCTATAGCTATGCTAAGAAGT

The regular expression TAA\$ matches TAA only in the first line because it ends with "TAA"



Regular Expressions

Square brackets (`[ ]`) are used to define a set of characters to match.

For example, `[ACGT]` matches any single character that is A, C, G, or T.

If we want to find sequences where "ATG" is followed by a C or G, we can use `ATG[CG]`.

In the sequence:

ATAGCATCAAATGCTAACATTTACGTAGTAGCTATAGCTATTACGTATGGCTACTTTATAGCATCAAATCT

The regular expression `ATG[CG]` matches both ATGC and ATGG.



Regular Expressions

The pipe (|) means "or" and allows you to specify alternative patterns.

For instance, `ATG|TAA` matches either "ATG" or "TAA"

In the sequences:

ATGACGACGTAGCGCAACAGCTCAACCTCAGGCTACTTTATAGCATCAAATCTAGCATTTAAATAG
TCTAGCATGACGACGTAGCGCAACAGCTCAACCTCAATAGCTATTACGTAGTGCAATGTACTATTA
ACCTCAGGCTACTTTATATAGCTATTACGTAGAGCATCAAATCTAGCATTTAAATAGCCCGTATCC

The regular expression `ATG|TAA` matches ATG or TAA when found.



Regular Expressions

Parentheses `(( ))` are used for grouping and capturing.

If we want to capture sequences that follow the pattern "ATG" followed by any two characters and "TAA," we can use `"(ATG..TAA)"` to match and capture the sequence "ATGCTTAA":

ATGCTTAAATGCCAGTAA

Captured groups can be referenced later in the same regular expression or used in programming languages. To find repeated sequences like "ATGTACTAA", you can use:

`(ATGTACTAA).*\1`

It would match in the following sequence:

GTAAATGTACTAACAGTAACGTAGCGATGTACTAAACCTCAATAG



Regular Expressions

The **backslash (\)** is used as an escape character to treat special characters literally.

For example, if you want to match a literal dot, use `"\."` instead of `"."`

If we want to find `"A.T"` as it appears (with the dot), we can use `"A\\.T"`

A.TGACTTAAG.A.T

The regular expression `"A\\.T"` matches A.T twice in the sequence.



grep

The command `grep` will print the lines matching a given pattern.

`grep PATTERN file`

`grep -e PATTERN file` (Pattern uses regex)

Understanding `grep` with a simple fasta file:

```
>contig1
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```





grep

Understanding grep with a simple fasta file:

```
>contig1
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

Find a specific sequence within our sequences:

```
grep "AGGGG" file.fasta
```

-> will print only the first sequence:

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```



grep

Find sequence headers only:

```
grep ">" file.fasta
```

-> will print all fasta headers:

```
>contig1
```

```
>contig2
```

Understanding grep with a simple fasta file:

```
>contig1
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```




grep

Understanding grep with a simple fasta file:

```
>contig1
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

Count number of sequences:

```
grep ">" file.fasta | wc -l
```

-> will count how many lines contain ">", which will match with the number of sequences: 2

we can also use the flag -c in grep to do the same:

```
grep -c ">" file.fasta
```



grep

Understanding grep with a simple fasta file:

```
>contig1
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

Print the DNA sequences with no headers:

```
grep -v ">" file.fasta
```

-> will print all lines that do not contain ">":

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

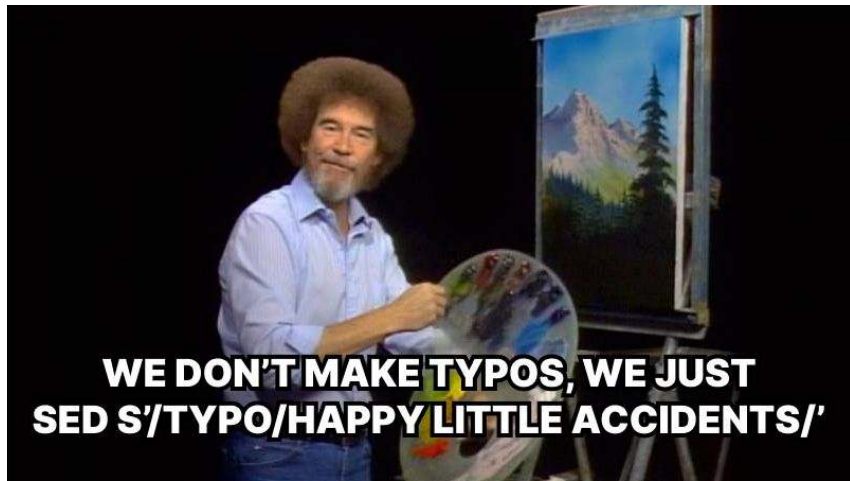




sed

sed ("stream editor") is a tool that can parse a file line by line, and transform text, using a compact programming language that can fit in one line. Sed is a powerful tool with a big array of possible commands, but the most common one is the substitution, in which we find a pattern and substitute it for another string.

```
sed 's/patternA/patternB/' file.txt
```





sed

Understanding sed with a simple fasta file:

```
>contig1 assembled a
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2 assembled b
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

Modify the fasta header to contain "sequence" instead of "contig":

```
sed 's/contig/sequence/' file.fasta
```

We will obtain the entire file with the replacement:

```
>sequence1 assembled 2025 a
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>sequence2 assembled 2025 b
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```



sed

Understanding sed with a simple fasta file:

```
>contig1 assembled a
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2 assembled b
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

Substitute spaces for underscores, in order to avoid problems with other programs:

In this case we add the flag "g" at the end, to make sure it replaces each occurrence even if there is multiple within the same line:

```
sed 's/ /_/g' file.fasta
```

We will obtain the entire file with the replacements:

```
>sequence1_assembled_2025_a
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>sequence2_assembled_2025_b
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```



sed

Understanding sed with a simple fasta file:

```
>contig1 assembled a
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2 assembled b
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

Simplify a fasta header:

```
sed 's/ .*//' file.fasta
```

We use regular expressions to match the first space we find in a line, followed by any character (.), any number of times (*)

We will obtain the entire file with the replacement:

```
>contig1
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```



sed

Understanding sed with a simple fasta file:

```
>contig1 assembled a
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2 assembled b
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```

Simplify a fasta header:

```
sed 's/ .*//' file.fasta
```

We use regular expressions to match the first space we find in a line, followed by any character (.), any number of times (*)

We will obtain the entire file with the replacement:

```
>contig1
```

```
AATCTAGCATTTACGTAGTAGCTAAAGCTAAACCTCAGGGGCTACTTTAT
```

```
>contig2
```

```
ATTTACGTAGCATCAAATCTAGCATTTACGTAGTAGCTAAAGCTATTACG
```



AWK

AWK is a language designed for text processing, like sed and grep. AWK is a standard feature of most Unix-like operating systems. AWK reads one line at a time, searching for a specific pattern to execute the desired action. It requires a condition, and an action:

```
awk condition {action} file.txt
```

AWK is a language field aware (column aware):

\$0 refers to the whole line

\$1, \$2, \$3 ... refers to columns 1, 2, 3 ...





AWK

Print only the lines containing genes in contig1:

```
awk '$1="contig1" {print}' file.bed
```

We would get the following printed out:

```
contig1 20 1305 gene1 . +  
contig1 4674 8563 gene4 . -
```

Understanding awk with a simple BED file:

```
contig1 20 1305 gene1 . +  
contig1 4674 8563 gene4 . -  
contig2 1239 5387 gene6 . -  
contig3 546 3524 gene9 . +
```



AWK

Understanding awk with a simple BED file:

contig1	20	1305	gene1	.	+
contig1	4674	8563	gene4	.	-
contig2	1239	5387	gene6	.	-
contig3	546	3524	gene9	.	+

Count how many genes we have in our file:

```
awk '$1="contig"' file.bed | wc -l
```

or we can create a counter after each condition is met, in this case finding the word contig in column 1. And we use the function END to mark that an extra action is done when all lines are finished being parsed:

```
awk '$1="contig" {count++} END {print count}' file.bed
```

both these commands will print: 4



AWK

Understanding awk with a simple BED file:

contig1	20	1305	gene1	.	+
contig1	4674	8563	gene4	.	-
contig2	1239	5387	gene6	.	-
contig3	546	3524	gene9	.	+

We can also use the function BEGIN to add an action before we start parsing the lines in our file:

```
awk 'BEGIN {print "We have these many genes:"} $1="contig" {count++} END {print count}' file.bed
```

This command will print:

We have these many genes:

4



AWK

Understanding awk with a simple BED file:

contig1	20	1305	gene1	.	+
contig1	4674	8563	gene4	.	-
contig2	1239	5387	gene6	.	-
contig3	546	3524	gene9	.	+

Finally, we can combine information in multiple columns to create our conditions.

Print out the gene names of all genes that are larger than 2000 bp:

We need can use the information in column 2 and 3, which marks the start and end of each gene, and we will print the information in column 4 (gene name) if column 3 - column 2 is larger than 2000:

```
awk '($3 - $2 > 2000) {print $4}' file.bed
```

It will print:

gene4

gene6



Exercises

Time to put into practice everything you learned!

Inside the folder `/home/genomics/workshop_materials/unix_tutorial`, you will find a folder called `data`. It contains a dataset that includes:

- `genome_assembly.fasta`
- `sequencing_reads.fastq`
- `variant_analysis.vcf`
- `annotation.gff`