A brief introduction to the bash shell

Presented for QLSC 612, Fundamentals of Neuro Data Science 2021 by Sebastian Urchs.

Based (very extensively) on the excellent lecture with the same name by Ross Markello from 2020 and the Software Carpentries "Introduction to the Shell" course.

Before we get started...

We're going to be working with a dataset from

https://github.com/neurodatascience/course-materials-2021/raw/master/lectures/26-July/03-intro-to-shell/shell-course.zip.

This link is on the course website as well: https://neurodatascience.github.io/QLS612-Overview/lectures-materials.html

Download that file and unzip it in your home directory:

- on a Mac: /Users/your-user-name
- on Linux or WSL: /home/your-user-name

What is the Shell

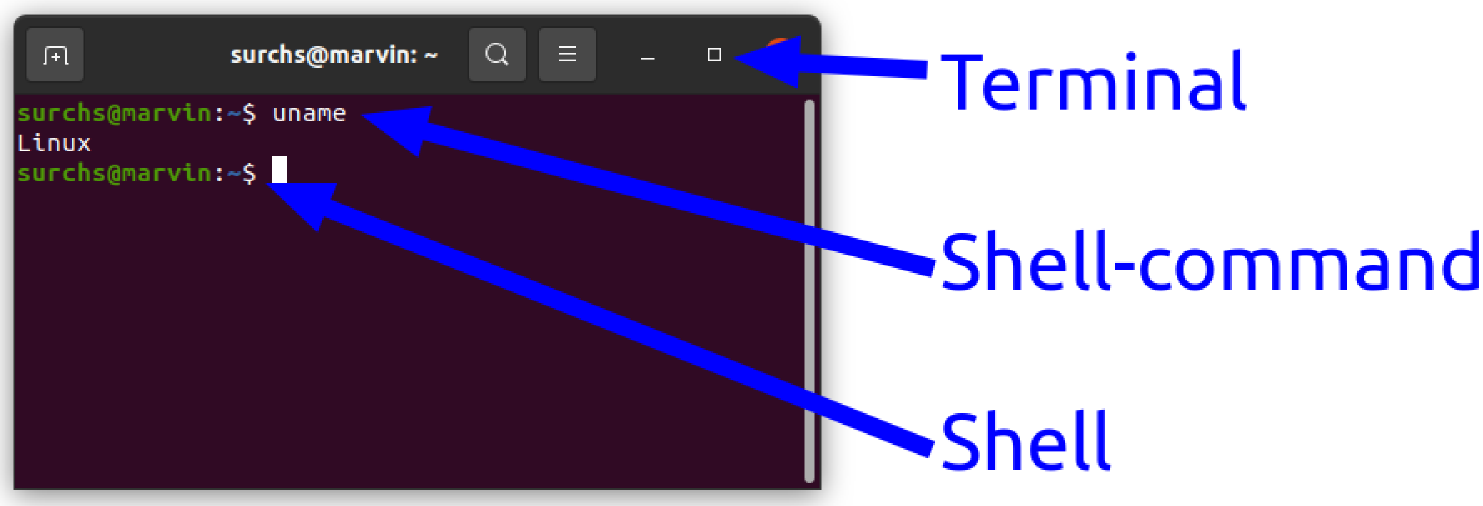
What is the Shell

- A shell is a program that **interpretes user input** into something the computer can understand

What is the Shell

- A shell is a program that **interpretes user input** into something the computer can understand

- A command-line shell runs inside a terminal that let's you type text
  - we call this a **command-line interface** (CLI), because you type commands in a line of text
  - this is in contrast to a **graphical user interface** (GUI) that you typically use
- command-line shell programs expect you to write **commands in a scripting language**



But what’s this ”bash shell”?

It's one of many available shells!

- `sh` - Bourne **SH**ell
- `ksh` - **K**orn **SH**ell
- `dash` - **D**ebian **A**lmquist **SH**ell
- `csh` - **C** **SH**ell
- `tcsh` - **T**ENEX **C** **SH**ell
- `zsh` - **Z** **SH**ell
- `bash` - **B**ourne **A**gain **SH**ell <-- We'll focus on this one!

WHY SO MANY?!

WHY SO MANY?!

- They all have different strengths / weaknesses
- You will see many of them throughout much of neuroimaging software, too!
    - `sh` is most frequently used in FSL
    - `csh`/`tcsh` is very common in FreeSurfer and AFNI

So we’re going to focus on the bash shell?

Yes! It's perhaps **the most common** shell, available on almost every OS:

- It's **the default** shell on most Linux systems
- It's the default shell in the Windows Subsytem for Linux (WSL)
- It's the default shell on Mac <=10.14
    - `zsh` is the new default on Mac Catalina (for licensing reasons 🙄)
    - But `bash` is still available!!

Alright, but why use the shell at all?

Isn't the GUI good enough?

- The GUI is great, but the shell is **very powerful**
- Some tasks take many "clicks" in a GUI, the shell is often extremely good at automating these
- You can write sequences of shell commands to connect the outputs of programs to other programs (pipelines)
- You can store the shell commands you used in a script file and execute them again later
    - this is a great way to document what you have done
    - it makes your work reproducible in a way that describing the "clicks" could not
- Also, you need to use the shell to accesss remote machine / high-performance computing environments

Alright, but why use the shell at all?

Isn't the GUI good enough?

- The GUI is great, but the shell is **very powerful**
- Some tasks take many "clicks" in a GUI, the shell is often extremely good at automating these
- You can write sequences of shell commands to connect the outputs of programs to other programs (pipelines)
- You can store the shell commands you used in a script file and execute them again later
    - this is a great way to document what you have done
    - it makes your work reproducible in a way that describing the "clicks" could not
- Also, you need to use the shell to accesss remote machine / high-performance computing environments

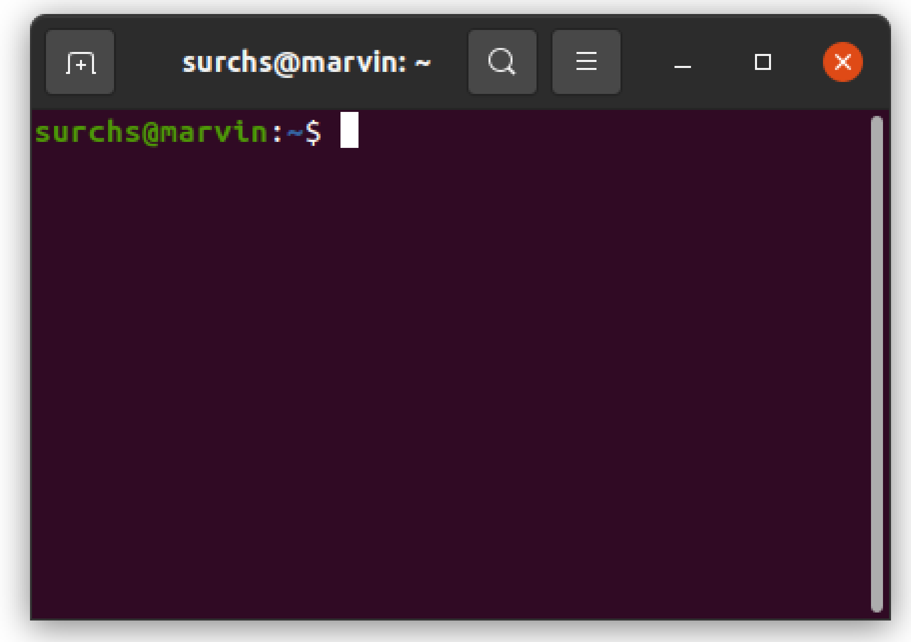
NOTE: We will not be able to cover all (or even most) aspects of the shell today.

But, we'll get through some *basics* that you can build on in the coming weeks.

The (bash) shell

Now, let's open up your terminal!

- **Windows**: Open the Ubuntu (WSL) application
- **Mac/Linux**: Open the Terminal application



When the shell is first opened, you are presented with a prompt, indicating that the shell is waiting for input:

```
$
```

The shell typically uses `$` as the prompt, but may use a different symbol.

When the shell is first opened, you are presented with a prompt, indicating that the shell is waiting for input:

```
$
```

The shell typically uses `$` as the prompt, but may use a different symbol.

IMPORTANT: When typing commands, either in this lesson or from other sources, **do not type the prompt**, only the commands that follow it!

Am I using the bash shell?

Let's check! You can use the following command to determine what shell you're using:

Am I using the bash shell?

Let's check! You can use the following command to determine what shell you're using:

In [ ]:
```
echo $SHELL
```

Am I using the bash shell?

Let's check! You can use the following command to determine what shell you're using:

In [ ]:
```
echo $SHELL
```

If that doesn't say something like `/bin/bash`, then simply type `bash`, press Enter, and try running the command again.

Voila! You're now in the bash shell.

Am I using the bash shell?

Let's check! You can use the following command to determine what shell you're using:

In [ ]:
```
echo $SHELL
```

If that doesn't say something like `/bin/bash`, then simply type `bash`, press Enter, and try running the command again.

Voila! You're now in the bash shell.

Note: We just ran our first shell command!

The `echo` command does exactly what its name implies: it simply echoes whatever we provide it to the screen!

(It's like `print` in Python / R or `disp` in MATLAB or `printf` in C or ...)

What’s with the `$SHELL`?

- Things prefixed with `$` in bash are (mostly) **environmental variables**
    - All programming languages have variables!
- We can assign variables in bash but when we want to reference them we need to add the `$` prefix
- We'll dig into this a bit more later, but by default our shell comes with some preset variables
    - `$SHELL` is one of them and it stores the path to the shell program that currently interprets our commands

Navigating Files and Directories

- The **file system** is the part of our operating system for managing files and directories
- There are a lot of shell commands to create/inspect/rename/delete files + directories
    - Indeed these are perhaps the most common commands you'll be using in the shell!

So where are we right now?

- When we open our terminal we are placed *somewhere* in the file system!
    - At any time while using the shell we are in exactly one place
- Commands mostly read / write / operate on files wherever we are, so it's important to know that!
- We can find our **current working directory** with the following command:

So where are we right now?

- When we open our terminal we are placed *somewhere* in the file system!
    - At any time while using the shell we are in exactly one place
- Commands mostly read / write / operate on files wherever we are, so it's important to know that!
- We can find our **current working directory** with the following command:

In [ ]:
```
pwd
```

So where are we right now?

- When we open our terminal we are placed *somewhere* in the file system!
    - At any time while using the shell we are in exactly one place
- Commands mostly read / write / operate on files wherever we are, so it's important to know that!
- We can find our **current working directory** with the following command:

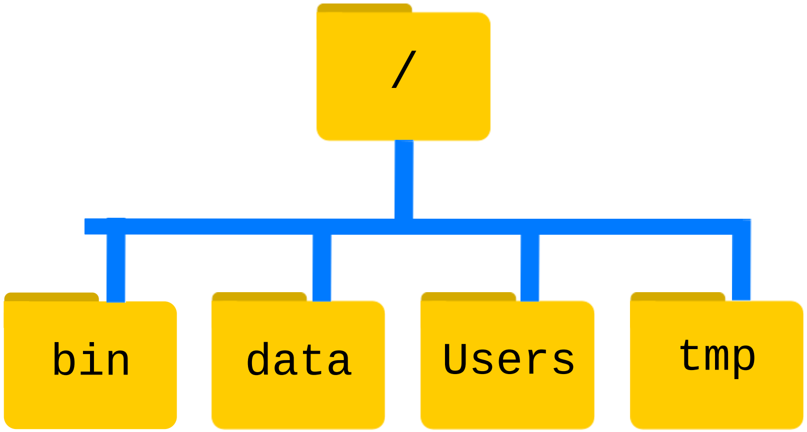
In [ ]:

```
pwd
```

- Many bash commands are acronyms or abbreviations (to try and help you remember them).
    - The above command, `pwd`, is an acronym for "**p**rint **w**orking **d**irectory">

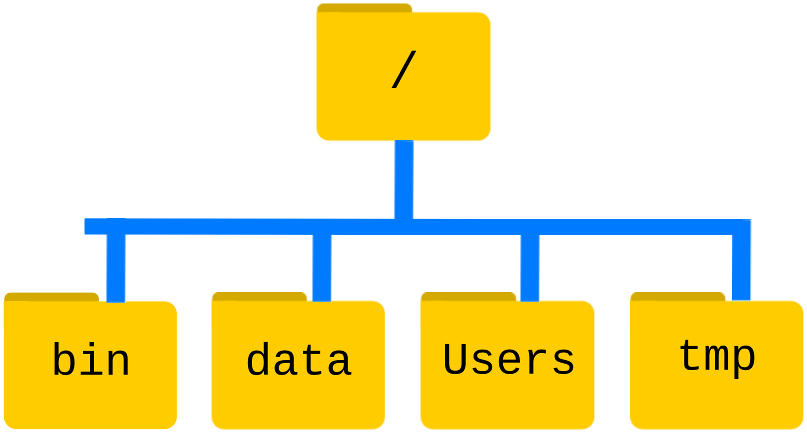
Let's look at the file system

Let's take a look at an example file-system (for a Macintosh):



Let's look at the file system

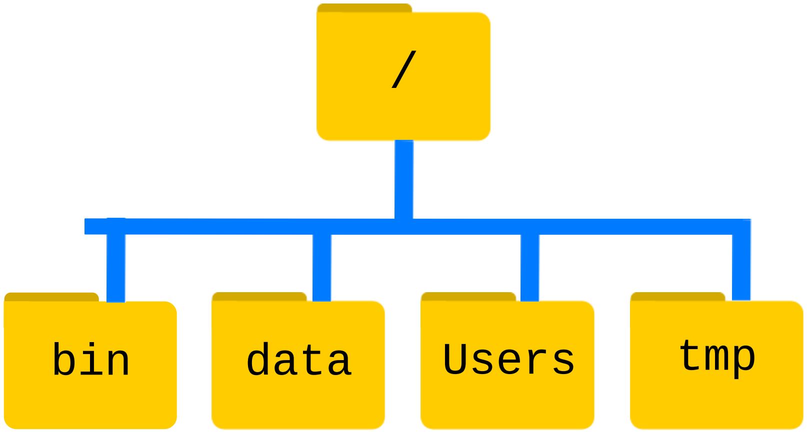
Let's take a look at an example file-system (for a Macintosh):



- The top (`/`) is the **root directory**, which holds the ENTIRE FILE SYSTEM.
- Inside are several other directories:
    - `bin` contains some built-in programs
    - `data` is where we store miscellaneous data files
    - `Users` is where personal user directories are
    - `tmp` is for temporary storage of files

Let's look at the file system

Let's take a look at an example file-system (for a Macintosh):



- The top (`/`) is the **root directory**, which holds the ENTIRE FILE SYSTEM.
- Inside are several other directories:
    - `bin` contains some built-in programs
    - `data` is where we store miscellaneous data files
    - `Users` is where personal user directories are
    - `tmp` is for temporary storage of files

Note: The filesystem on a Linux machine will have slightly different directory names (e.g. `/Users` is typically `/home`) but the same principles apply.

The `/` character has two meanings:

1. At the beginning of the path, it refers to the root directory
2. Inside a path, it is used as a separator between directories

The `/` character has two meanings:

1. At the beginning of the path, it refers to the root directory
2. Inside a path, it is used as a separator between directories

So let's remind ourselves how to see where we are and figure out what's in our directory:

The `/` character has two meanings:

1. At the beginning of the path, it refers to the root directory
2. Inside a path, it is used as a separator between directories

So let's remind ourselves how to see where we are and figure out what's in our directory:

In [ ]:

```
pwd
```

The `/` character has two meanings:

1. At the beginning of the path, it refers to the root directory
2. Inside a path, it is used as a separator between directories

So let's remind ourselves how to see where we are and figure out what's in our directory:

In [ ]:
```
pwd
```

We are inside the `home` directory (e.g. `User` on Mac) for the user `surchs` (me) and in a sub-directory called `shell-course`.

Let's see what is in here

The `ls` command will list the contents of the directory we are currently in (i.e. the **current working directory**):

The `ls` command will list the contents of the directory we are currently in (i.e. the **current working directory**):

In [ ]:
```
ls
```

The `ls` command will list the contents of the directory we are currently in (i.e. the **current working directory**):

In [ ]:
```
ls
```

`ls`, as we saw before, prints the contents of your **current working directory**.

We can make it tell us a bit more information about our directory by providing an **option** to the `ls` command

General syntax of a shell command

Consider this command (where we are looking inside the `interesting_files` directory) as a general example:

General syntax of a shell command

Consider this command (where we are looking inside the `interesting_files` directory) as a general example:

In [ ]:
```
ls -F interesting_files
```

General syntax of a shell command

Consider this command (where we are looking inside the `interesting_files` directory) as a general example:

In [ ]:
```
ls -F interesting_files
```

We have:

1. A **command** (`ls`),
2. An **option** (`-F`), also called a **flag** or a **switch**, and
3. An **argument** (`interesting_files`)

Options (a.k.a. flags, switches)

- Options change the behavior of a command
- They generally start with either a `-` or `--`
- They are case sensitive!

Options (a.k.a. flags, switches)

- Options change the behavior of a command
- They generally start with either a `-` or `--`
- They are case sensitive!

For example, `ls -l` will display the directory contents as a list.

Options (a.k.a. flags, switches)

- Options change the behavior of a command
- They generally start with either a `-` or `--`
- They are case sensitive!

For example, `ls -l` will display the directory contents as a list.

In [ ]:
```
ls -S interesting_files
```

Options (a.k.a. flags, switches)

- Options change the behavior of a command
- They generally start with either a `-` or `--`
- They are case sensitive!

For example, `ls -l` will display the directory contents as a list.

In [ ]:

```
ls -S interesting_files
```

`ls -S` is another option that sorts the directory contents according to their size. We can combine several options in one command:

Options (a.k.a. flags, switches)

- Options change the behavior of a command
- They generally start with either a `-` or `--`
- They are case sensitive!

For example, `ls -l` will display the directory contents as a list.

In [ ]:

```
ls -S interesting_files
```

`ls -S` is another option that sorts the directory contents according to their size. We can combine several options in one command:

In [ ]:

```
ls -lS interesting_files
```

Arguments (a.k.a parameters)

- These tell the command what to operate on!
- They are only *sometimes* optional (as with `ls`)
    - In these cases, providing them will also change the behavior of the command!

compare:

Arguments (a.k.a parameters)

- These tell the command what to operate on!
- They are only *sometimes* optional (as with `ls`)
    - In these cases, providing them will also change the behavior of the command!

compare:

In [ ]:
```
ls
```

Arguments (a.k.a parameters)

- These tell the command what to operate on!
- They are only *sometimes* optional (as with `ls`)
    - In these cases, providing them will also change the behavior of the command!

compare:

In [ ]:
```
ls
```

In [ ]:
```
ls flying_circus
```

Arguments (a.k.a parameters)

- These tell the command what to operate on!
- They are only *sometimes* optional (as with `ls`)
    - In these cases, providing them will also change the behavior of the command!

compare:

In [ ]:
```
ls
```

In [ ]:
```
ls flying_circus
```

If we do not give `ls` an argument, it will list the contents of the current working directory.

So many options, where can I find help

Either `man ls` or `ls --help`!

This will vary depending on: (1) the command and (2) your operating system!

Generally try `man` first:

So many options, where can I find help

Either `man ls` or `ls --help`!

This will vary depending on: (1) the command and (2) your operating system!

Generally try `man` first:

In [ ]:

```
man ls
```

You can ask the shell what a command does

Sometimes you don't want to read the entire `man` page, but really just want to remember what a command does. A really useful helper tool is `whatis`.

You can ask the shell what a command does

Sometimes you don't want to read the entire `man` page, but really just want to remember what a command does. A really useful helper tool is `whatis`.

In [ ]:

```
whatis
```

You can ask the shell what a command does

Sometimes you don't want to read the entire `man` page, but really just want to remember what a command does. A really useful helper tool is `whatis`.

In [ ]:
```
whatis
```

It **does** expect an argument. "what do you want to know about". Let's see what `ls` does:

You can ask the shell what a command does

Sometimes you don't want to read the entire `man` page, but really just want to remember what a command does. A really useful helper tool is `whatis`.

In [ ]:
```
whatis
```

It **does** expect an argument. "what do you want to know about". Let's see what `ls` does:

In [ ]:
```
whatis ls
```

You can ask the shell what a command does

Sometimes you don't want to read the entire `man` page, but really just want to remember what a command does. A really useful helper tool is `whatis`.

In [ ]:
```
whatis
```

It **does** expect an argument. "what do you want to know about". Let's see what `ls` does:

In [ ]:
```
whatis ls
```

Not every command has a description:

You can ask the shell what a command does

Sometimes you don't want to read the entire `man` page, but really just want to remember what a command does. A really useful helper tool is `whatis`.

In [ ]:
```
whatis
```

It **does** expect an argument. "what do you want to know about". Let's see what `ls` does:

In [ ]:
```
whatis ls
```

Not every command has a description:

In [ ]:
```
whatis cd
```

We can do more than list directories

So many interesting things to see, let's change to a different working directory so we can do things there.

- The `cd` or "change directory" command will let us do that

Note: This is analogous to clicking and opening a directory in your graphical file explorer

We can do more than list directories

So many interesting things to see, let's change to a different working directory so we can do things there.

- The `cd` or "change directory" command will let us do that

Note: This is analogous to clicking and opening a directory in your graphical file explorer

In [ ]:

```
ls
```

We can do more than list directories

So many interesting things to see, let's change to a different working directory so we can do things there.

- The `cd` or "change directory" command will let us do that

Note: This is analogous to clicking and opening a directory in your graphical file explorer

In [ ]:

```
ls
```

In [ ]:

```
cd flying_circus
```

We can do more than list directories

So many interesting things to see, let's change to a different working directory so we can do things there.

- The `cd` or "change directory" command will let us do that

Note: This is analogous to clicking and opening a directory in your graphical file explorer

In [ ]:
```
ls
```

In [ ]:
```
cd flying_circus
```

Let's confirm that we have indeed changed directory by calling `pwd`:

We can do more than list directories

So many interesting things to see, let's change to a different working directory so we can do things there.

- The `cd` or "change directory" command will let us do that

Note: This is analogous to clicking and opening a directory in your graphical file explorer

In [ ]:
```
ls
```

In [ ]:
```
cd flying_circus
```

Let's confirm that we have indeed changed directory by calling `pwd`:

In [ ]:
```
pwd
```

What happens if we run cd without any arguments?

What happens if we run cd without any arguments?

In [ ]:
```
cd ~
```

What happens if we run cd without any arguments?

In [ ]:

```
cd ~
```

In [ ]:

```
pwd
```

What happens if we run cd without any arguments?

In [ ]:
```
cd ~
```

In [ ]:
```
pwd
```

We are back in our home directory! This is *incredibly* useful if you've gotten lost.

- `cd` without arguments brings you to your home directory
- the `~` (tilde) character is a shorthand for your home directory. So `cd ~` also brings you there
- the `-` (dash) character is a shorthand for the previous directory you were in. So `cd -` brings you back to where you just were

Let's get back to the `flying_circus` directory again.

Let's get back to the `flying_circus` directory again.

In [ ]:

```
cd shell-course/flying_circus
```

Let's get back to the `flying_circus` directory again.

In [ ]:
```
cd shell-course/flying_circus
```

In [ ]:
```
pwd
```

Let's get back to the `flying_circus` directory again.

In [ ]:
```
cd shell-course/flying_circus
```

In [ ]:
```
pwd
```

We can string together paths with the `/` separator instead of changing one directory at a time! Because the path we gave to `cd` did not start with the file system root directory (`/`), it was interpreted as a relative path, i.e. in reference to the home directory that we called `cd` from.

Relative versus absolute paths

So far, we have been using **relative** paths to change directories and list their contents.

- A **relative** path is **relative to the current working directory**. It does **not** begin with the file system root (`/`).
- An **absolute** path includes the entire path beginning with the file system root directory (`/`).

`pwd` prints the **absolute** path of the current working directory:

Relative versus absolute paths

So far, we have been using **relative** paths to change directories and list their contents.

- A **relative** path is **relative to the current working directory**. It does **not** begin with the file system root (`/`).
- An **absolute** path includes the entire path beginning with the file system root directory (`/`).

`pwd` prints the **absolute** path of the current working directory:

In [ ]:

```
pwd
```

Let's take a look around in this directory

Let's take a look around in this directory

In [ ]:
```
ls
```

Let's take a look around in this directory

In [ ]:
```
ls
```

On second thought, let's not go here, t'is a silly place.

How do we go back? There's a special notation to move one directory up:

Let's take a look around in this directory

In [ ]:
```
ls
```

On second thought, let's not go here, t'is a silly place.

How do we go back? There's a special notation to move one directory up:

In [ ]:
```
cd ..
```

Let's take a look around in this directory

In [ ]:
```
ls
```

On second thought, let's not go here, t'is a silly place.

How do we go back? There's a special notation to move one directory up:

In [ ]:
```
cd ..
```

Here, `..` refers to "the directory containing this one". This is also called the **parent** of the current directory.

Let's check that we are where we think we are:

Let's take a look around in this directory

In [ ]:
```
ls
```

On second thought, let's not go here, t'is a silly place.

How do we go back? There's a special notation to move one directory up:

In [ ]:
```
cd ..
```

Here, `..` refers to "the directory containing this one". This is also called the **parent** of the current directory.

Let's check that we are where we think we are:

In [ ]:
```
pwd
```

Seeing the unseen

`ls` is supposed to list the contents of our directory, but we didn't see `..` anywhere in the listings from before, right?

`..` is a special directory that is normally hidden. We can provide an additional argument to `ls` to make it appear:

Seeing the unseen

`ls` is supposed to list the contents of our directory, but we didn't see `..` anywhere in the listings from before, right?

`..` is a special directory that is normally hidden. We can provide an additional argument to `ls` to make it appear:

In [ ]:

```
ls -a
```

Seeing the unseen

`ls` is supposed to list the contents of our directory, but we didn't see `..` anywhere in the listings from before, right?

`..` is a special directory that is normally hidden. We can provide an additional argument to `ls` to make it appear:

In [ ]:
```
ls -a
```

The `-a` argument (show **a**ll contents) will list ALL the contents of our current directory, including special and hidden files/directories, like:

- `..`, which refers to the parent directory
- `.`, which refers to the current working directory

Hidden files

The last command also revealed a `.i_am_hidden` file:

Hidden files

The last command also revealed a `.i_am_hidden` file:

In [ ]:

```
ls -aFl
```

Hidden files

The last command also revealed a `.i_am_hidden` file:

In [ ]:
```
ls -aFl
```

The `.` prefix is usually reserved for configuration files, and prevents them from cluttering the terminal when you use `ls`.

Summary

- The file system is responsible for managing information on the disk
- Information is stored in files, which are stored in directories (folders)
- Directories can also store other (sub-)directories, which forms a directory tree
- `cd path` changes the current working directory
- `ls path` prints a listing of a specific file or directory; `ls` on its own lists the current working directory.
- `pwd` prints the user’s current working directory
- `/` on its own is the root directory of the whole file system
- A relative path specifies a location starting from the current location
- An absolute path specifies a location from the root of the file system
- `..` means "the directory above the current one"; `.` on its own means "the current directory"

Modifying files and directories

So far we have mainly looked at contents of files and directories. But we can of course also make changes. Let's first see again where we are:

Modifying files and directories

So far we have mainly looked at contents of files and directories. But we can of course also make changes. Let's first see again where we are:

In [ ]:

```
pwd
```

Modifying files and directories

So far we have mainly looked at contents of files and directories. But we can of course also make changes. Let's first see again where we are:

In [ ]:
```
pwd
```

In [ ]:
```
ls
```

Creating a directory

Let us create a subdirectory called `notes`. We can use a program called `mkdir` for this.

Creating a directory

Let us create a subdirectory called `notes`. We can use a program called `mkdir` for this.

In [ ]:

```
mkdir notes
```

Creating a directory

Let us create a subdirectory called `notes`. We can use a program called `mkdir` for this.

In [ ]:

```
mkdir notes
```

Since we provided a relative path, we can expect that to have been creating in our current working directory:

Creating a directory

Let us create a subdirectory called `notes`. We can use a program called `mkdir` for this.

In [ ]:

```
mkdir notes
```

Since we provided a relative path, we can expect that to have been creating in our current working directory:

In [ ]:

```
ls -F
```

Creating a directory

Let us create a subdirectory called `notes`. We can use a program called `mkdir` for this.

In [ ]:

```
mkdir notes
```

Since we provided a relative path, we can expect that to have been creating in our current working directory:

In [ ]:

```
ls -F
```

(You could have also opened up the file explorer and made a new directory that way, too!)

Good naming conventions

1. Don't use spaces
2. Don't begin the name with `-`
3. Stick with letters, numbers, `.`, `-`, and `_`
    - That is, avoid other special characters like `~!@#$%^&*()`

Creating a text file

Let's

- navigate into our (empty) `notes` directory (with `cd`)
- confirm that it is in fact empty (with `ls`)
- and create a new file. For this we can use `nano`

Creating a text file

Let's

- navigate into our (empty) `notes` directory (with `cd`)
- confirm that it is in fact empty (with `ls`)
- and create a new file. For this we can use `nano`

In [ ]:

```
cd notes
```

Creating a text file

Let's

- navigate into our (empty) `notes` directory (with `cd`)
- confirm that it is in fact empty (with `ls`)
- and create a new file. For this we can use `nano`

In [ ]:
```
cd notes
```

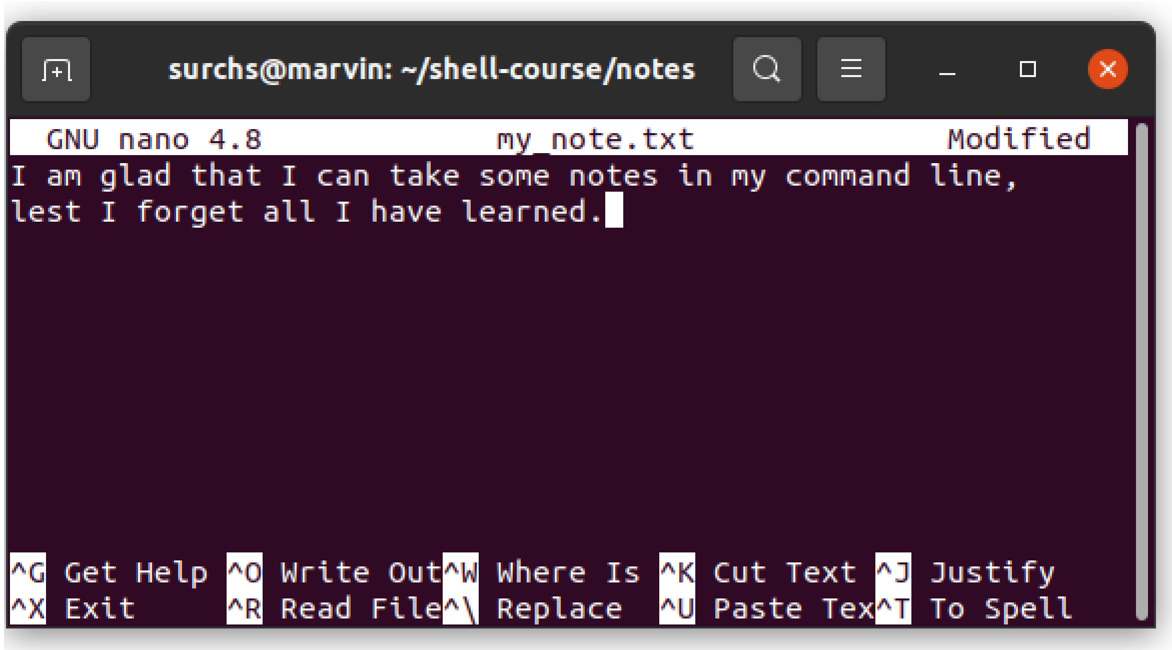
In [ ]:
```
# nano my_note.txt
```

`nano` is a useful command-line **text editor**. It only works with plain text (i.e., no graphs, figures, tables, or images!)

(You may be familiar with graphical editors like Gedit, Notepad, or TextEdit, or other command line editors like Emacs or Vim.)

`nano` is a useful command-line **text editor**. It only works with plain text (i.e., no graphs, figures, tables, or images!)

(You may be familiar with graphical editors like Gedit, Notepad, or TextEdit, or other command line editors like Emacs or Vim.)



`nano` uses the Control (CTRL) and ALT key to make changes. The command help along the bottom of the editor window refers to these keys with abbreviations:

- `^` for CTRL: `^G` means "press and hold CTRL together with the `G` key"
- `M` for ALT: `M-U` means "press and hold ALT together with the `U` key"

Let's save our note with `^O`, i.e. `CTRL+O` (the letter o)

`nano` doesn't print anything to screen, so let's make sure our file exists:

`nano` doesn't print anything to screen, so let's make sure our file exists:

In [ ]:
```
ls -F
```

`nano` doesn't print anything to screen, so let's make sure our file exists:

In [ ]:
```
ls -F
```

To check that we have indeed written to this file, let's display its contents. We can do this with `cat`

`nano` doesn't print anything to screen, so let's make sure our file exists:

In [ ]:
```
ls -F
```

To check that we have indeed written to this file, let's display its contents. We can do this with `cat`

In [ ]:
```
cat my_note.txt
```

Moving files and directories

Let's first go back up to our `shell-course` directory

Moving files and directories

Let's first go back up to our `shell-course` directory

In [ ]:
```
cd ~/shell-course
```

Moving files and directories

Let's first go back up to our `shell-course` directory

In [ ]:
```
cd ~/shell-course
```

In [ ]:
```
ls
```

Moving files and directories

Let's first go back up to our `shell-course` directory

In [ ]:
```
cd ~/shell-course
```

In [ ]:
```
ls
```

Let's look into this `dir_of_doom`.

Moving files and directories

Let's first go back up to our `shell-course` directory

In [ ]:
```
cd ~/shell-course
```

In [ ]:
```
ls
```

Let's look into this `dir_of_doom`.

In [ ]:
```
cd dir_of_doom
```

Moving files and directories

Let's first go back up to our `shell-course` directory

In [ ]:
```
cd ~/shell-course
```

In [ ]:
```
ls
```

Let's look into this `dir_of_doom`.

In [ ]:
```
cd dir_of_doom
```

In [ ]:
```
ls -F
```

Moving files and directories

Let's first go back up to our `shell-course` directory

In [ ]:
```
cd ~/shell-course
```

In [ ]:
```
ls
```

Let's look into this `dir_of_doom`.

In [ ]:
```
cd dir_of_doom
```

In [ ]:
```
ls -F
```

In [ ]:
```
ls -F the_wrong_dir
```

Moving files and directories

Let's first go back up to our `shell-course` directory

In [ ]:
```
cd ~/shell-course
```

In [ ]:
```
ls
```

Let's look into this `dir_of_doom`.

In [ ]:
```
cd dir_of_doom
```

In [ ]:
```
ls -F
```

In [ ]:
```
ls -F the_wrong_dir
```

All of these files are in the wrong directory.

Let's move the files in `the_wrong_dir` to `the_right_dir`. We can use the `mv` command for this!

Let's move the files in `the_wrong_dir` to `the_right_dir`. We can use the `mv` command for this!

In [ ]:

```
mv the_wrong_dir/my_file1.txt the_right_dir
```

Let's move the files in `the_wrong_dir` to `the_right_dir`. We can use the `mv` command for this!

In [ ]:
```
mv the_wrong_dir/my_file1.txt the_right_dir
```

The first argument of `mv` is the file we're moving, and the last argument is where we want it to go!

Let's make sure that worked:

Let's move the files in `the_wrong_dir` to `the_right_dir`. We can use the `mv` command for this!

In [ ]:

```
mv the_wrong_dir/my_file1.txt the_right_dir
```

The first argument of `mv` is the file we're moving, and the last argument is where we want it to go!

Let's make sure that worked:

In [ ]:

```
ls the_wrong_dir
```

We can provide more than two arguments to `mv`, as long as the final argument is a directory! That would mean "move all these things into this directory".

We can provide more than two arguments to `mv`, as long as the final argument is a directory! That would mean "move all these things into this directory".

In [ ]:

```
mv the_wrong_dir/my_file2.txt the_wrong_dir/my_file3.txt the_right_dir
```

We can provide more than two arguments to `mv`, as long as the final argument is a directory! That would mean "move all these things into this directory".

In [ ]:

```
mv the_wrong_dir/my_file2.txt the_wrong_dir/my_file3.txt the_right_dir
```

We can make our life easier by using wildcards! Wildcards are simple patterns that can match any character in a file name:

- `*` (the asterisk) will match any character 0 or more times. i.e. `*.txt` will match both `a.txt` and `any.txt` (any file ending in `.txt`)
- `?` (the questionmark) will match any character exactly once. i.e. `?.txt` will match only `a.txt` but not `any.txt`

We can use wildcards to move any file that fits our pattern so we don't have to type each individual file name.

We can provide more than two arguments to `mv`, as long as the final argument is a directory! That would mean "move all these things into this directory".

In [ ]:
```
mv the_wrong_dir/my_file2.txt the_wrong_dir/my_file3.txt the_right_dir
```

We can make our life easier by using wildcards! Wildcards are simple patterns that can match any character in a file name:

- `*` (the asterisk) will match any character 0 or more times. i.e. `*.txt` will match both `a.txt` and `any.txt` (any file ending in `.txt`)
- `?` (the questionmark) will match any character exactly once. i.e. `?.txt` will match only `a.txt` but not `any.txt`

We can use wildcards to move any file that fits our pattern so we don't have to type each individual file name.

In [ ]:
```
ls the_right_dir/my_file?.txt
```

We can provide more than two arguments to `mv`, as long as the final argument is a directory! That would mean "move all these things into this directory".

In [ ]:
```
mv the_wrong_dir/my_file2.txt the_wrong_dir/my_file3.txt the_right_dir
```

We can make our life easier by using wildcards! Wildcards are simple patterns that can match any character in a file name:

- `*` (the asterisk) will match any character 0 or more times. i.e. `*.txt` will match both `a.txt` and `any.txt` (any file ending in `.txt`)
- `?` (the questionmark) will match any character exactly once. i.e. `?.txt` will match only `a.txt` but not `any.txt`

We can use wildcards to move any file that fits our pattern so we don't have to type each individual file name.

In [ ]:
```
ls the_right_dir/my_file?.txt
```

Note: `mv` is **quite dangerous**, because it will silently overwrite files if the destination already exists! Refer to the `-i` flag for "interactive" moving (with warnings!).

Copying files and directories

The `cp` (**c**o**p**y) command is like `mv`, but copies instead of moving! Let's use it to make a backup of the files in `the_right_dir`

Copying files and directories

The `cp` (**c**o**p**y) command is like `mv`, but copies instead of moving! Let's use it to make a backup of the files in `the_right_dir`

In [ ]:

```
mkdir backup
```

Copying files and directories

The `cp` (**c**o**p**y) command is like `mv`, but copies instead of moving! Let's use it to make a backup of the files in `the_right_dir`

In [ ]:
```
mkdir backup
```

In [ ]:
```
cp the_right_dir/my_file1.txt backup
```

Copying files and directories

The `cp` (**c**o**p**y) command is like `mv`, but copies instead of moving! Let's use it to make a backup of the files in `the_right_dir`

In [ ]:
```
mkdir backup
```

In [ ]:
```
cp the_right_dir/my_file1.txt backup
```

Let's confirm we have copied the file into `backup` and it is also still in `the_right_dir`. We could run two `ls` commands, but we can also just use a wildcard to look inside all directories!

Copying files and directories

The `cp` (**c**o**p**y) command is like `mv`, but copies instead of moving! Let's use it to make a backup of the files in `the_right_dir`

In [ ]:

```
mkdir backup
```

In [ ]:

```
cp the_right_dir/my_file1.txt backup
```

Let's confirm we have copied the file into `backup` and it is also still in `the_right_dir`. We could run two `ls` commands, but we can also just use a wildcard to look inside all directories!

In [ ]:

```
ls */my_file1.txt
```

Let's copy the complete `the_right_dir` to `backup`

Let's copy the complete `the_right_dir` to `backup`

In [ ]:

```
cp the_right_dir backup
```

Let's copy the complete `the_right_dir` to `backup`

In [ ]:
```
cp the_right_dir backup
```

To copy directories and all of its contents, we have to use the `-r` (**r**ecursive) flag:

Let's copy the complete `the_right_dir` to `backup`

In [ ]:
```
cp the_right_dir backup
```

To copy directories and all of its contents, we have to use the `-r` (**r**ecursive) flag:

In [ ]:
```
cp -r the_right_dir backup
```

Let's copy the complete `the_right_dir` to `backup`

In [ ]:
```
cp the_right_dir backup
```

To copy directories and all of its contents, we have to use the `-r` (**r**ecursive) flag:

In [ ]:
```
cp -r the_right_dir backup
```

In [ ]:
```
ls backup
```

Removing files

There is a large and useless file in our directory. Let's remove it. We can use `rm` to **r**e**m**ove it:

Removing files

There is a large and useless file in our directory. Let's remove it. We can use `rm` to **re**mo**ve** it:

In [ ]:

```
ls -lhS
```

Removing files

There is a large and useless file in our directory. Let's remove it. We can use `rm` to **re**mo**v**e it:

In [ ]:
```
ls -lhS
```

In [ ]:
```
rm big_file_with_no_purpose.txt
```

Removing files

There is a large and useless file in our directory. Let's remove it. We can use `rm` to **re**mo**v**e it:

In [ ]:
```
ls -lhS
```

In [ ]:
```
rm big_file_with_no_purpose.txt
```

The `rm` command deletes files. Let's check that the file is gone:

Removing files

There is a large and useless file in our directory. Let's remove it. We can use `rm` to **re**mo**v**e it:

In [ ]:
```
ls -lhS
```

In [ ]:
```
rm big_file_with_no_purpose.txt
```

The `rm` command deletes files. Let's check that the file is gone:

In [ ]:
```
ls
```

Deleting is **FOREVER** 💀💀

- The shell DOES NOT HAVE A TRASH BIN.
- You CANNOT recover files that have been deleted with `rm`
- But, you can use the `-i` flag to do things a bit more safely!
  - This will prompt you to type `Y` or `N` before every file that is going to be deleted.

Removing directories

Let's try and remove the `the_wrong_dir`:

Removing directories

Let's try and remove the `the_wrong_dir`:

In [ ]:
```
rm the_wrong_dir
```

Removing directories

Let's try and remove the `the_wrong_dir`:

In [ ]:

```
rm the_wrong_dir
```

`rm` only works on files, by default, but we can tell it to **r**ecursively delete a directory and all its contents with the `-r` flag:

Removing directories

Let's try and remove the `the_wrong_dir`:

In [ ]:
```
rm the_wrong_dir
```

`rm` only works on files, by default, but we can tell it to **r**ecursively delete a directory and all its contents with the `-r` flag:

In [ ]:
```
rm -r the_wrong_dir
```

Removing directories

Let's try and remove the `the_wrong_dir`:

In [ ]:

```
rm the_wrong_dir
```

`rm` only works on files, by default, but we can tell it to **r**ecursively delete a directory and all its contents with the `-r` flag:

In [ ]:

```
rm -r the_wrong_dir
```

Because **deleting is forever** 💀💀, the `rm -r` command should be used with GREAT CAUTION.

Summary

- `cp old new` copies a file
- `mkdir path` creates a new directory
- `mv old new` moves (renames) a file or directory
- `rm path` removes (deletes) a file
- `*` matches zero or more characters in a filename, so `*.txt` matches all files ending in `.txt`
- `?` matches any single character in a filename, so `?.txt` matches `a.txt` but not `any.txt`
- The shell does not have a trash bin: once something is deleted, it's really gone

Finding things with the shell

Oftentimes, our file system can be quite complex, with sub-directories inside sub-directories inside sub-directories.

What happens in we want to find one (or several) files, without having to type `ls` hundreds or thousands of times?

First, let's navigate back to `shell-course` directory:

Finding things with the shell

Oftentimes, our file system can be quite complex, with sub-directories inside sub-directories inside sub-directories.

What happens in we want to find one (or several) files, without having to type `ls` hundreds or thousands of times?

First, let's navigate back to `shell-course` directory:

In [ ]:

```
cd ~/shell-course
```

Finding things with the shell

Oftentimes, our file system can be quite complex, with sub-directories inside sub-directories inside sub-directories.

What happens in we want to find one (or several) files, without having to type `ls` hundreds or thousands of times?

First, let's navigate back to `shell-course` directory:

In [ ]:

```
cd ~/shell-course
```

Let's get our bearings with `ls`:

Finding things with the shell

Oftentimes, our file system can be quite complex, with sub-directories inside sub-directories inside sub-directories.

What happens in we want to find one (or several) files, without having to type `ls` hundreds or thousands of times?

First, let's navigate back to `shell-course` directory:

In [ ]:
```
cd ~/shell-course
```

Let's get our bearings with `ls`:

In [ ]:
```
ls
```

Unfortunately, this doesn't list any of the files in the directories. But we know from our previous exploration that there there are files and sub-directories. We can display the full sub-directory tree with the `tree` command:

Unfortunately, this doesn't list any of the files in the directories. But we know from our previous exploration that there there are files and sub-directories. We can display the full sub-directory tree with the `tree` command:

In [ ]:

```
tree
```

`tree` has options to display additional information, only show a certain depth of the tree and even filter certain file names. But if we are searching for a certain file name pattern, there is a better tool for us:

`find`

`tree` has options to display additional information, only show a certain depth of the tree and even filter certain file names. But if we are searching for a certain file name pattern, there is a better tool for us:

`find`

In [ ]:

```
find . -name 'my_*'
```

`tree` has options to display additional information, only show a certain depth of the tree and even filter certain file names. But if we are searching for a certain file name pattern, there is a better tool for us:

`find`

In [ ]:

```
find . -name 'my_*'
```

Remember, `.` means "the current working directory".

Here, `find` begins the search in the current working directory and then traverses the entire directory structure. With the `-name` option, we specify a pattern that includes a wildcard to specify the names we are looking for.

One of the results here is a directory. We can filter the results further by specifying that we only want to see file matches.

`tree` has options to display additional information, only show a certain depth of the tree and even filter certain file names. But if we are searching for a certain file name pattern, there is a better tool for us:

`find`

In [ ]:
```
find . -name 'my_*'
```

Remember, `.` means "the current working directory".

Here, `find` begins the search in the current working directory and then traverses the entire directory structure. With the `-name` option, we specify a pattern that includes a wildcard to specify the names we are looking for.

One of the results here is a directory. We can filter the results further by specifying that we only want to see file matches.

In [ ]:
```
find . -name 'my_*' -type f
```

Finding things inside of files

Searching for files and directories based on their names and meta-data is helpful, but often it is interesting to search inside a file as well.

For this, we can use `grep`. This is an abbreviation for "**g**lobally search for a **r**egular **e**xpression and **p**rint matching lines". If you can't remember this, just ask `whatis grep`.

Finding things inside of files

Searching for files and directories based on their names and meta-data is helpful, but often it is interesting to search inside a file as well.

For this, we can use `grep`. This is an abbreviation for "**g**lobally search for a **r**egular **e**xpression and **p**rint matching lines". If you can't remember this, just ask `whatis grep`.

In [ ]:

```
whatis grep
```

Finding things inside of files

Searching for files and directories based on their names and meta-data is helpful, but often it is interesting to search inside a file as well.

For this, we can use `grep`. This is an abbreviation for "**g**lobally search for a **r**egular **e**xpression and **p**rint matching lines". If you can't remember this, just ask `whatis grep`.

In [ ]:

```
whatis grep
```

Let's take a look in `hello_world.txt` and then use `grep` search for what we find inside.

Finding things inside of files

Searching for files and directories based on their names and meta-data is helpful, but often it is interesting to search inside a file as well.

For this, we can use `grep`. This is an abbreviation for "**g**lobally search for a **r**egular **e**xpression and **p**rint matching lines". If you can't remember this, just ask `whatis grep`.

In [ ]:

```
whatis grep
```

Let's take a look in `hello_world.txt` and then use `grep` search for what we find inside.

In [ ]:

```
cat helloworld.txt
```

Finding things inside of files

Searching for files and directories based on their names and meta-data is helpful, but often it is interesting to search inside a file as well.

For this, we can use `grep`. This is an abbreviation for "**g**lobally search for a **r**egular **e**xpression and **p**rint matching lines". If you can't remember this, just ask `whatis grep`.

In [ ]:

```
whatis grep
```

Let's take a look in `hello_world.txt` and then use `grep` search for what we find inside.

In [ ]:

```
cat helloworld.txt
```

In [ ]:

```
grep "Bash" helloworld.txt
```

The directory `flying_circus` contains the movie scripts for two Monty Python movies. Only one of them has a rabbit as an actor. Let's find out which one it is:

The directory `flying_circus` contains the movie scripts for two Monty Python movies. Only one of them has a rabbit as an actor. Let's find out which one it is:

In [ ]:

```
grep "rabbit" -i --count --no-messages flying_circus/*
```

The directory `flying_circus` contains the movie scripts for two Monty Python movies. Only one of them has a rabbit as an actor. Let's find out which one it is:

In [ ]:
```
grep "rabbit" -i --count --no-messages flying_circus/*
```

OK, only one of these files seems to have any mention of rabbits in it. We can use `man` to understand the options used here.

Note that the file `dangerous_rabbits.txt` was not a match, even though the file name contains "rabbit"

The directory `flying_circus` contains the movie scripts for two Monty Python movies. Only one of them has a rabbit as an actor. Let's find out which one it is:

In [ ]:
```
grep "rabbit" -i --count --no-messages flying_circus/*
```

OK, only one of these files seems to have any mention of rabbits in it. We can use `man` to understand the options used here.

Note that the file `dangerous_rabbits.txt` was not a match, even though the file name contains "rabbit"

In [ ]:
```
man grep
```

Context: passing information with pipes

A strength of using the shell is that you can connect the output of one command to the input of another command. To do so, you can use the `|` (pipe) character. When you connect commands together with the pipe (`|`) operator, we can the entire statement a **pipeline**.

Pipelines take the general form of:

`command1 -flags arguments | command2 -flags arguments`.

Let's say we want to use grep to search for the occurence of a word that we think could be quite common, like "Ni". We could just print all of the matches. But maybe we want to see the 10 last occurences. A pipe allows us to take the output of `grep`, and give it to another command, `tail`, that does just that.

Let's say we want to use grep to search for the occurence of a word that we think could be quite common, like "Ni". We could just print all of the matches. But maybe we want to see the 10 last occurences. A pipe allows us to take the output of `grep`, and give it to another command, `tail`, that does just that.

In [ ]:

```
whatis tail
```

Let's say we want to use grep to search for the occurence of a word that we think could be quite common, like "Ni". We could just print all of the matches. But maybe we want to see the 10 last occurences. A pipe allows us to take the output of `grep`, and give it to another command, `tail`, that does just that.

In [ ]:

```
whatis tail
```

In [ ]:

```
grep "Ni" --no-messages flying_circus/the_holy_grail.txt -nH | tail -n 10
```

Let's say we want to use grep to search for the occurence of a word that we think could be quite common, like "Ni". We could just print all of the matches. But maybe we want to see the 10 last occurences. A pipe allows us to take the output of `grep`, and give it to another command, `tail`, that does just that.

In [ ]:

```
whatis tail
```

In [ ]:

```
grep "Ni" --no-messages flying_circus/the_holy_grail.txt -nH | tail -n 10
```

`grep` and `tail` are two commands that each do a very specific thing. This is generally the case for shell commands on Unix systems, i.e. they follow the "Unix philosophy" of doing a single thing well. Pipes are a great way to combine the functionality of several commands to do what you want.

Note: in this example, we could have also used additional options for `grep` to achieve the same result without using a pipe.

By default, `grep` will show us the file name and the line in the text that contains the pattern match. Let's look for the word "swallow". We will limit our matches to 10 and also ask grep to print out the line following our match, so we can have more context.

By default, `grep` will show us the file name and the line in the text that contains the pattern match. Let's look for the word "swallow". We will limit our matches to 10 and also ask grep to print out the line following our match, so we can have more context.

In [ ]:

```
grep "swallow" -i -n --max-count 10 --after-context 1 flying_circus/*
```

By default, `grep` will show us the file name and the line in the text that contains the pattern match. Let's look for the word "swallow". We will limit our matches to 10 and also ask grep to print out the line following our match, so we can have more context.

In [ ]:
```
grep "swallow" -i -n --max-count 10 --after-context 1 flying_circus/*
```

Very interesting.

Summary

- we can print the structure of any given directory with `tree`
- `find` is a great tool to search for files and directories based on their name and other meta-data like size, age, and so on
- `grep` is a great tool to search within (text)files for occurences of a given string or even complex regular expressions
- pipes (`|`) allow us to combine the output of one command with the input of another command

Scripts and variables

One of the most powerful functions of using the shell is that you can write your commands into a text file called a shell script, and then ask the shell to execute each command in the script in sequence.

This is very helpful if you want to

- run the same set of commands repeatedly (e.g. every time you log into your computer)
- keep a detailed record of what commands you used to create an output
- share a set of commands with someone, or run their commands

This is all very useful. So what do we need to do to turn a text file into a shell script?

Let's take a look into `interesting_files` where we will find some shell scripts.

Scripts and variables

One of the most powerful functions of using the shell is that you can write your commands into a text file called a shell script, and then ask the shell to execute each command in the script in sequence.

This is very helpful if you want to

- run the same set of commands repeatedly (e.g. every time you log into your computer)
- keep a detailed record of what commands you used to create an output
- share a set of commands with someone, or run their commands

This is all very useful. So what do we need to do to turn a text file into a shell script?

Let's take a look into `interesting_files` where we will find some shell scripts.

In [ ]:

```
cd interesting_files
```

Scripts and variables

One of the most powerful functions of using the shell is that you can write your commands into a text file called a shell script, and then ask the shell to execute each command in the script in sequence.

This is very helpful if you want to

- run the same set of commands repeatedly (e.g. every time you log into your computer)
- keep a detailed record of what commands you used to create an output
- share a set of commands with someone, or run their commands

This is all very useful. So what do we need to do to turn a text file into a shell script?

Let's take a look into `interesting_files` where we will find some shell scripts.

In [ ]:

```
cd interesting_files
```

In [ ]:

```
ls -F
```

Anatomy of a shell script

Let's display the contents of the file `run_me.sh`

Anatomy of a shell script

Let's display the contents of the file `run_me.sh`

In [ ]:

```
cat run_me.sh -n
```

Anatomy of a shell script

Let's display the contents of the file `run_me.sh`

In [ ]:
```
cat run_me.sh -n
```

A shell script needs to contain two things:

- the `#!/bin/bash` statement in the line 1 is called a hash-bang (shebang) and declares what shell program shall be used to execute this script. Here we use the bash shell
- the `echo "Thank you, very kind!"` statement in line 4 is the shell command - this is what gets executed.

Lastly there is

- The statement `#` in line 3 is a comment. The `#` (hash) will prevent the remaining text in this line from being executed. This is a good way to explain in human readable form what your script does
- our text file also uses the file ending `.sh` to show that it is a shell script

Anatomy of a shell script (contd.)

However, in order to run (i.e. "execute") the script, the right content is not enough. Our script file must also have the right permission.

Remember that `ls -F` displays executable files by appending a `*` to the file name.

Anatomy of a shell script (contd.)

However, in order to run (i.e. "execute") the script, the right content is not enough. Our script file must also have the right permission.

Remember that `ls -F` displays executable files by appending a `*` to the file name.

In [ ]:

```
ls -F
```

Anatomy of a shell script (contd.)

However, in order to run (i.e. "execute") the script, the right content is not enough. Our script file must also have the right permission.

Remember that `ls -F` displays executable files by appending a `*` to the file name.

In [ ]:
```
ls -F
```

This looks good. Let's execute our script:

Anatomy of a shell script (contd.)

However, in order to run (i.e. "execute") the script, the right content is not enough. Our script file must also have the right permission.

Remember that `ls -F` displays executable files by appending a `*` to the file name.

In [ ]:
```
ls -F
```

This looks good. Let's execute our script:

In [ ]:
```
./run_me.sh
```

What would happen if we try to run `run_me_too.sh`?

What would happen if we try to run `run_me_too.sh`?

In [ ]:

```
ls -lF
```

What would happen if we try to run `run_me_too.sh`?

In [ ]:
```
ls -lF
```

In [ ]:
```
./run_me_too.sh
```

What would happen if we try to run `run_me_too.sh`?

In [ ]:

```
ls -lF
```

In [ ]:

```
./run_me_too.sh
```

We can change the permission of this script with the `chmod` command.

What would happen if we try to run `run_me_too.sh`?

In [ ]:
```
ls -lF
```

In [ ]:
```
./run_me_too.sh
```

We can change the permission of this script with the `chmod` command.

In [ ]:
```
chmod +x run_me_too.sh
```

What would happen if we try to run `run_me_too.sh`?

In [ ]:
```
ls -lF
```

In [ ]:
```
./run_me_too.sh
```

We can change the permission of this script with the `chmod` command.

In [ ]:
```
chmod +x run_me_too.sh
```

In [ ]:
```
ls -lF
```

What would happen if we try to run `run_me_too.sh`?

In [ ]:
```
ls -lF
```

In [ ]:
```
./run_me_too.sh
```

We can change the permission of this script with the `chmod` command.

In [ ]:
```
chmod +x run_me_too.sh
```

In [ ]:
```
ls -lF
```

In [ ]:
```
./run_me_too.sh
```

How does the shell know where to find commands

Note that when we execute our own shell scripts, we have to specify their path. Just typing the file name does not work:

How does the shell know where to find commands

Note that when we execute our own shell scripts, we have to specify their path. Just typing the file name does not work:

In [ ]:
```
run_me.sh
```

How does the shell know where to find commands

Note that when we execute our own shell scripts, we have to specify their path. Just typing the file name does not work:

In [ ]:
```
run_me.sh
```

In [ ]:
```
./run_me.sh
```

How does the shell know where to find commands

Note that when we execute our own shell scripts, we have to specify their path. Just typing the file name does not work:

In [ ]:
```
run_me.sh
```

In [ ]:
```
./run_me.sh
```

But we can just type `cd` or `ls` and the shell knows what command we mean. How does this work?

How does the shell know where to find commands

Note that when we execute our own shell scripts, we have to specify their path. Just typing the file name does not work:

In [ ]:
```
run_me.sh
```

In [ ]:
```
./run_me.sh
```

But we can just type `cd` or `ls` and the shell knows what command we mean. How does this work?

When you execute a command like `cd` or `ls`, the shell will go and look if it is aware of any programs with that name. `bash` is smart enough to give you suggestions for typos or programs you could install but haven't:

How does the shell know where to find commands

Note that when we execute our own shell scripts, we have to specify their path. Just typing the file name does not work:

In [ ]:
```
run_me.sh
```

In [ ]:
```
./run_me.sh
```

But we can just type `cd` or `ls` and the shell knows what command we mean. How does this work?

When you execute a command like `cd` or `ls`, the shell will go and look if it is aware of any programs with that name. `bash` is smart enough to give you suggestions for typos or programs you could install but haven't:

In [ ]:
```
pwwd
```

How does the shell know where to find commands

Note that when we execute our own shell scripts, we have to specify their path. Just typing the file name does not work:

In [ ]:
```
run_me.sh
```

In [ ]:
```
./run_me.sh
```

But we can just type `cd` or `ls` and the shell knows what command we mean. How does this work?

When you execute a command like `cd` or `ls`, the shell will go and look if it is aware of any programs with that name. `bash` is smart enough to give you suggestions for typos or programs you could install but haven't:

In [ ]:
```
pwwd
```

Where does the shell it look for these programs?

The `$PATH` variable

Your shell has a variable called `$PATH` that contains all of the places where it will look for programs to run. We can use `echo` to take a look inside:

The `$PATH` variable

Your shell has a variable called `$PATH` that contains all of the places where it will look for programs to run. We can use `echo` to take a look inside:

In [ ]:

```
echo $PATH
```

The `$PATH` variable

Your shell has a variable called `$PATH` that contains all of the places where it will look for programs to run. We can use `echo` to take a look inside:

In [ ]:
```
echo $PATH
```

These are the directories our current shell is looking inside.

Note how several values (here directories) are delineated by the character `:`

So which of these directories is e.g. `ls` inside of? Lucky for us, there is a shell command to tell us. It is called `which`:

The `$PATH` variable

Your shell has a variable called `$PATH` that contains all of the places where it will look for programs to run. We can use `echo` to take a look inside:

In [ ]:
```
echo $PATH
```

These are the directories our current shell is looking inside.

Note how several values (here directories) are delineated by the character `:`

So which of these directories is e.g. `ls` inside of? Lucky for us, there is a shell command to tell us. It is called `which`:

In [ ]:
```
/usr/bin/ls
```

The `$PATH` variable

Your shell has a variable called `$PATH` that contains all of the places where it will look for programs to run. We can use `echo` to take a look inside:

In [ ]:

```
echo $PATH
```

These are the directories our current shell is looking inside.

Note how several values (here directories) are delineated by the character `:`

So which of these directories is e.g. `ls` inside of? Lucky for us, there is a shell command to tell us. It is called `which`:

In [ ]:

```
/usr/bin/ls
```

`which` is a great helper tool to see which command you are currently calling. This can be immensely helpful when you have multiple versions of a tool with the same name in different locations (e.g. different python versions).

We can also call `ls` by using its absolute path. The shell just normally resolves this for us.

The `$PATH` variable

Your shell has a variable called `$PATH` that contains all of the places where it will look for programs to run. We can use `echo` to take a look inside:

In [ ]:
```
echo $PATH
```

These are the directories our current shell is looking inside.

Note how several values (here directories) are delineated by the character `:`

So which of these directories is e.g. `ls` inside of? Lucky for us, there is a shell command to tell us. It is called `which`:

In [ ]:
```
/usr/bin/ls
```

`which` is a great helper tool to see which command you are currently calling. This can be immensely helpful when you have multiple versions of a tool with the same name in different locations (e.g. different python versions).

We can also call `ls` by using its absolute path. The shell just normally resolves this for us.

In [ ]:
```
/usr/bin/ls
```

You can change the `$PATH` variable

When you start a new shell, the `$PATH` variable gets set by a number of startup files on your system. The system wide startup files are protected and you should (in most cases) not try to change them as this will affect the way your system behaves. There are also user-level startup files in your home directory where you can make changes to the `$PATH` variable (and other variables) that will just affect your shells.

For example, `/home/surchs/.bashrc` is a config file where I can make changes to my `$PATH` variable to have my shell search additional directories for programs.

To take a look, we can use the tool `cat`. Again, let's check what it does.

You can change the `$PATH` variable

When you start a new shell, the `$PATH` variable gets set by a number of startup files on your system. The system wide startup files are protected and you should (in most cases) not try to change them as this will affect the way your system behaves. There are also user-level startup files in your home directory where you can make changes to the `$PATH` variable (and other variables) that will just affect your shells.

For example, `/home/surchs/.bashrc` is a config file where I can make changes to my `$PATH` variable to have my shell search additional directories for programs.

To take a look, we can use the tool `cat`. Again, let's check what it does.

In [ ]:

```
whatis cat
```

Let's now take a look inside the `.bashrc` file

Let's now take a look inside the `.bashrc` file

In [ ]:

```
cat /home/surchs/.bashrc -n | tail
```

Let's now take a look inside the `.bashrc` file

In [ ]:
```
cat /home/surchs/.bashrc -n | tail
```

The statement `export PATH="$PATH:$HOME/bin"` in line 120 adds a directory `bin` in my home directory to the shell $PATH. Notice again the `:` character to separate the new from the old value.

You can create your own variables

The `$PATH` variable is important, but it is just a normal variable. You can create your own variables.

You can create your own variables

The `$PATH` variable is important, but it is just a normal variable. You can create your own variables.

In [ ]:

```
MY_VAR=10
```

You can create your own variables

The `$PATH` variable is important, but it is just a normal variable. You can create your own variables.

In [ ]:
```
MY_VAR=10
```

In [ ]:
```
echo ${MY_VAR}
```

You can create your own variables

The `$PATH` variable is important, but it is just a normal variable. You can create your own variables.

In [ ]:
```
MY_VAR=10
```

In [ ]:
```
echo ${MY_VAR}
```

- variable names are case sensitive
- to access the value of a set variable we prepend the `$` character to the variable name
- we use `{` and `}` to delineate the variable name

You can create your own variables

The `$PATH` variable is important, but it is just a normal variable. You can create your own variables.

In [ ]:
```
MY_VAR=10
```

In [ ]:
```
echo ${MY_VAR}
```

- variable names are case sensitive
- to access the value of a set variable we prepend the `$` character to the variable name
- we use `{` and `}` to delineate the variable name

In [ ]:
```
echo MY_VAR
```

You can create your own variables

The `$PATH` variable is important, but it is just a normal variable. You can create your own variables.

In [ ]:
```
MY_VAR=10
```

In [ ]:
```
echo ${MY_VAR}
```

- variable names are case sensitive
- to access the value of a set variable we prepend the `$` character to the variable name
- we use `{` and `}` to delineate the variable name

In [ ]:
```
echo MY_VAR
```

In [ ]:
```
echo ${my_var}
```

You can create your own variables

The `$PATH` variable is important, but it is just a normal variable. You can create your own variables.

In [ ]:
```
MY_VAR=10
```

In [ ]:
```
echo ${MY_VAR}
```

- variable names are case sensitive
- to access the value of a set variable we prepend the `$` character to the variable name
- we use `{` and `}` to delineate the variable name

In [ ]:
```
echo MY_VAR
```

In [ ]:
```
echo ${my_var}
```

In [ ]:
```
echo ${MY_VAR}iscool
```

Two kinds of shell variables

There are two different kinds of variables in a shell:

- `shell variables` only exist inside your current shell instance. They are not shared with any programs you execute from this shell. By convention we use small caps for shell variables.
- `environment variables` by contrast are shared with programs you execute in the shell. By convention we use ALL CAPS for environment variables (like `$PATH`).

Any new variable you declare (or set) starts out as a `shell variable`. To "promote" it to an environment variable, you have to `export` it. You can also "demote" an environment variable with `export -n`. You can see all of the environment variables in your shell with `printenv`.

Two kinds of shell variables

There are two different kinds of variables in a shell:

- `shell variables` only exist inside your current shell instance. They are not shared with any programs you execute from this shell. By convention we use small caps for shell variables.
- `environment variables` by contrast are shared with programs you execute in the shell. By convention we use ALL CAPS for environment variables (like `$PATH`).

Any new variable you declare (or set) starts out as a `shell variable`. To "promote" it to an environment variable, you have to `export` it. You can also "demote" an environment variable with `export -n`. You can see all of the environment variables in your shell with `printenv`.

In [ ]:

```
printenv | tail
```

Environment variables get passed to programs

The script `i_can_see_variables.sh` is printing the value of two variables:

`ENV_VAR` and `shell_var`. Let's create them and see how they function differently inside the script

Environment variables get passed to programs

The script `i_can_see_variables.sh` is printing the value of two variables:

`ENV_VAR` and `shell_var`. Let's create them and see how they function differently inside the script

In [ ]:

```
ENV_VAR="important setting"
export ENV_VAR
shell_var="local value"
```

Environment variables get passed to programs

The script `i_can_see_variables.sh` is printing the value of two variables:

`ENV_VAR` and `shell_var`. Let's create them and see how they function differently inside the script

In [ ]:

```
ENV_VAR="important setting"
export ENV_VAR
shell_var="local value"
```

Note how we export the variable itself and not its value (which would require the `$`)

Let's call the script.

Environment variables get passed to programs

The script `i_can_see_variables.sh` is printing the value of two variables:

`ENV_VAR` and `shell_var`. Let's create them and see how they function differently inside the script

In [ ]:

```
ENV_VAR="important setting"
export ENV_VAR
shell_var="local value"
```

Note how we export the variable itself and not its value (which would require the `$`)

Let's call the script.

In [ ]:

```
./i_can_see_variables.sh
```

Environment variables get passed to programs

The script `i_can_see_variables.sh` is printing the value of two variables:

`ENV_VAR` and `shell_var`. Let's create them and see how they function differently inside the script

In [ ]:

```
ENV_VAR="important setting"
export ENV_VAR
shell_var="local value"
```

Note how we export the variable itself and not its value (which would require the `$`)

Let's call the script.

In [ ]:

```
./i_can_see_variables.sh
```

Because environment variables are passed to child processes (e.g. programs) they can change the behaviour of your system. Some tools and installation procedures will ask you to modify environment variables, e.g. by editing the `.bashrc` file in your home directory.

Summary

- the shell will look for programs in your command in directories defined in the `$PATH` variable
- `$PATH` and other environment variables are set by startup files at the sytem and user level
- you can edit the startup files for your user in your home directory (e.g. `~/.bashrc`)
- more generally, you edit any variable and also create new variables
- to retrieve the value of a variable, we need the `$` character (e.g. `$VAR` vs VAR)
- there are two types of variables: "shell variables" and "environment variables"
    - only environment variables get passed to programs you call from the shell
    - you can turn a "shell variable" into an "environment variable" with `export`
- shell scripts are text files that contain shell commands to be executed in sequence
    - the first line of your script typically declares what shell should run it
    - this statement (e.g. `#!/bin/bash`) is called the shebang
- shell scripts need to have execution permission to be run. You change file permission with `chmod`
- to run a shell script or any command not in the `$PATH`, we specify the path to the command

Overall Summary

- The bash shell is very powerful!
- It offers a command-line interface to your computer and file system
- It makes it easy to operate on files quickly and efficiently (copying, renaming, etc.)
- Sequences of shell commands can be strung together to quickly and reproducibly make powerful pipelines

Also consider:

- bash and other shells are great for many tasks, particularly when they involve changes to your files and directories
- But bash is not the right tool to create complex pipelines and programs like the ones needed for research analyses
- For these tasks, modern programming languages like Python offer better error handling, control flow, debugging and other features

References

There are lots of excellent resources online for learning more about bash:

- The GNU Manual is *the* reference for all bash commands: http://www.gnu.org/manual/manual.html
- "Learning the Bash Shell" book: http://shop.oreilly.com/product/9780596009656.do
- An interactive on-line bash shell course: https://www.learnshell.org/
- The reference page of the software carpentry course: https://swcarpentry.github.io/shell-novice/reference.html