Boosted Bespoke

Bare Bones Bash

Brought By Blissfully Baffled Bioinformaticians

Thiseas C. Lamnidis

Aida Andrades Valtueña

Aims of this session

- Aim:
    - Get more familiar with text modification with bash

Aims of this session

- Aim:
    - Get more familiar with text modification with bash
- Objectives:
    - How to find stuff?
    - Concept: for loop for and while loops?
    - Regular expressions (aka "Regex")
    - How to write a conditional statement (if/else)
    - *Simple* text modification with `sed` (i.e. witchcraft)
    - How to write a simple bash script

Aims of this session

- Aim:
    - Get more familiar with text modification with bash
- Objectives:
    - How to find stuff?
    - Concept: for loop for and while loops?
    - Regular expressions (aka "Regex")
    - How to write a conditional statement (if/else)
    - *Simple* text modification with `sed` (i.e. witchcraft)
    - How to write a simple bash script

Aims of this session

- Aim:
  - Get more familiar with text modification with bash
- Objectives:
  - How to find stuff?
  - Concept: for loop for and while loops?
  - Regular expressions (aka "Regex")
  - How to write a conditional statement (if/else)
  - *Simple* text modification with `sed` (i.e. witchcraft)
  - How to write a simple bash script

Preparation!

Preparation

You should have a folder and a metadata file from Session 1:

```
$ cd ~/BareBonesBash
```

- Boosted-BBB-meta: what does this file contain?

Preparation

You should have a folder and a metadata file from Session 1:

```
$ cd ~/BareBonesBash
```

- Boosted-BBB-meta: what does this file contain?

It contains metadata that will be use to sort the images in a nice structure in our messy folder

Preparation

You should have a folder and a metadata file from Session 1:

```
$ cd ~/BareBonesBash
```

- Boosted-BBB-meta: what does this file contain?

It contains metadata that will be use to sort the images in a nice structure in our messy folder

- Let's download the messy folder from our collaborator:

```
$ wget git.io/Boosted-BBB-images -O Boosted-BBB.zip
```

Preparation

You should have a folder and a metadata file from Session 1:

```
$ cd ~/BareBonesBash
```

- Boosted-BBB-meta: what does this file contain?

It contains metadata that will be use to sort the images in a nice structure in our messy folder

- Let's download the messy folder from our collaborator:

```
$ wget git.io/Boosted-BBB-images -O Boosted-BBB.zip
```

Since it is a zip file we need to unzip it, which command should we use?

Preparation

You should have a folder and a metadata file from Session 1:

```
$ cd ~/BareBonesBash
```

- Boosted-BBB-meta: what does this file contain?

It contains metadata that will be use to sort the images in a nice structure in our messy folder

- Let's download the messy folder from our collaborator:

```
$ wget git.io/Boosted-BBB-images -O Boosted-BBB.zip
```

Since it is a zip file we need to unzip it, which command should we use?

```
$ unzip Boosted-BBB.zip
```

Make sure to cd back to ~/BareBonesBash now!

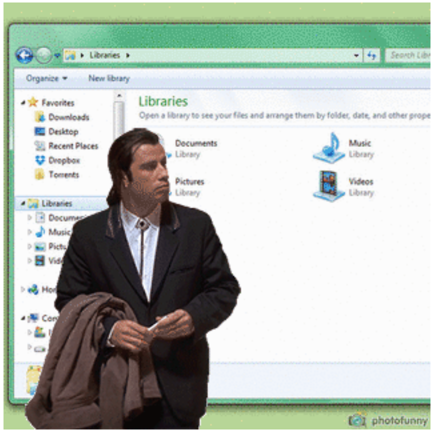
Outline

- Where is my stuff??
- Concept: for loops
- How to rename stuff
- Concept: Regular expressions
- While loop: to infinity and beyond!
- Conditionals: IF/ELSE
- Modifying files: SED, PASTE

Where is my stuff??

```
$ find # Don't run yet!
```

How can you search for files and directories hidden in layers and layers (of your very organised 😉) directories?



Where is my stuff??

```
$ find Boosted-BBB/  # Don't run yet!
```

- **First** part of the `find` command: *the place to look from*
  - e.g. `.` to indicate 'here'

Where is my stuff??

```
$ find Boosted-BBB/  # Don't run yet!
```

- **First** part of the `find` command: *the place to look from*
    - e.g. `.` to indicate 'here'
    - Could also use `~/`

Where is my stuff??

```
$ find Boosted-BBB/  # Don't run yet!
```

- **First** part of the `find` command: *the place to look from*
  - e.g. . to indicate 'here'
  - Could also use `~/`
  - Could use absolute path e.g. `/home/aida/`

> ***Question*** *What is the difference between `Boosted-BBB/` and `/home/aida/Boosted-BBB/`?*

Where is my stuff??

```
$ find Boosted-BBB/ -type f # Don't run yet!
```

- **First** part of the find command: *the place to look from*
    - e.g. . to indicate 'here'
    - Could also use ~/
    - Could use absolute path e.g. /home/james/
- **Second** part of the find command: *what type of things to look for?*
    - Use -type to define the filetype:
        - **f**ile
        - **d**irectory

Where is my stuff??

```
$ find Boosted-BBB/ -type f -name  # Don't run yet!
```

- **First** part of the `find` command: *the place to look from*
    - e.g. `.` to indicate 'here'
    - Could also use `~/`
    - Could use absolute path e.g. `/home/james/`
- **Second** part of the `find` command: *what type of things to look for?*
    - Use `-type` to define the filetype:
        - **f**ile
        - **d**irectory
- **Third** part of the `find` command: *what to look in*?
    - Use `-name` to say 'look in *names* of things'

Where is my stuff??

```
$ find Boosted-BBB/ -type f -name '*JPG*'  # Now GO!
```

- **First** part of the `find` command: *the place to look from*
  - e.g. `.` to indicate 'here'
  - Could also use `~/`
  - Could use absolute path e.g. `/home/james/`
- **Second** part of the `find` command: *what type of things to look for?*
  - Use `-type` to define the filetype:
    - **f**ile
    - **d**irectory
- **Third** part of the `find` command: *what to look in*?
  - Use `-name` to say 'look in *names* of things'
- **Finally** after `-name` we give the the 'strings' to search for
  - Use wildcards (`*`) for maximum laziness!

Where is my stuff??

We are looking for all files with the suffix JPG.

Let's first set the suffix we want to a variable, so we can easily change it in the future.

```
$ suffix="JPG"
```

Where is my stuff??

We are looking for all files with the suffix JPG.

Let's first set the suffix we want to a variable, so we can easily change it in the future.

```
$ suffix="JPG"
```

We can now call on this variable in our search. Try the following command:

```
$ find Boosted-BBB/ -type f -name '*$suffix*'
```

Where is my stuff??

We are looking for all files with the suffix JPG.

Let's first set the suffix we want to a variable, so we can easily change it in the future.

```
$ suffix="JPG"
```

We can now call on this variable in our search. Try the following command:

```
$ find Boosted-BBB/ -type f -name '*$suffix*'
```

That found no files!! Our original find command confirms that these files exist!

Where is my stuff??

We are looking for all files with the suffix `JPG`.

Let's first set the suffix we want to a variable, so we can easily change it in the future.

```
$ suffix="JPG"
```

We can now call on this variable in our search. Try the following command:

```
$ find Boosted-BBB/ -type f -name '*$suffix*'
```

That found no files!! Our original `find` command confirms that these files exist!

Now look at the command below:

```
$ find Boosted-BBB/ -type f -name "*$suffix*"
```

What has changed here? Run the second command.

Where is my stuff??

We are looking for all files with the suffix JPG.

Let's first set the suffix we want to a variable, so we can easily change it in the future.

```
$ suffix="JPG"
```

We can now call on this variable in our search. Try the following command:

```
$ find Boosted-BBB/ -type f -name '*$suffix*'
```

That found no files!! Our original find command confirms that these files exist!

Now look at the command below:

```
$ find Boosted-BBB/ -type f -name "*$suffix*"
```

What has changed here? Run the second command.

Cleaning up the filenames

It seems that wherever the files are from have completely mangled the file names (.JPG.MP3.TXT... WHAT?!)

Lets clean up the filenames, and then we can sort the files into categories.

Cleaning up the filenames

It seems that wherever the files are from have completely mangled the file names (.JPG.MP3.TXT... WHAT?!)

Lets clean up the filenames, and then we can sort the files into categories.

We will need to repeat the clean up for each of the file names...

Cleaning up the filenames

It seems that wherever the files are from have completely mangled the file names (.JPG.MP3.TXT... WHAT?!)

Lets clean up the filenames, and then we can sort the files into categories.

We will need to repeat the clean up for each of the file names...

So, How do I **repeat** a command multiple times on a list of things?

Concept: for loops

- **for** loops allow us to go through a list of things and perform some actions. Let's see an example:

Concept: for loops

- **for** loops allow us to go through a list of things and perform some actions. Let's see an example:

```
$ Variable=Yes
$ for i in Greece Spain Britain; do
>  echo "Does $i have lovely food? $Variable"
> done
```

Concept: for loops

- **for** loops allow us to go through a list of things and perform some actions. Let's see an example:

```
$ Variable=Yes
$ for i in Greece Spain Britain; do
>  echo "Does $i have lovely food? $Variable"
> done
```

```
Does Greece have lovely food? Yes
Does Spain have lovely food? Yes
Does Britain have lovely food? Yes
```

Concept: for loops

- **for** loops allow us to go through a list of things and perform some actions. Let's see an example:

```
$ Variable=Yes
$ for i in Greece Spain Britain; do
>  echo "Does $i have lovely food? $Variable"
> done
```

```
Does Greece have lovely food? Yes
Does Spain have lovely food? Yes
Does Britain have lovely food? Yes
```

- The for loop went through the list `Greece Spain Britain` and printed a statement with each item in the list

Concept: for loops

- **for** loops allow us to go through a list of things and perform some actions. Let's see an example:

```
$ Variable=Yes
$ for i in Greece Spain Britain; do
>  echo "Does $i have lovely food? $Variable"
> done
```

```
Does Greece have lovely food? Yes
Does Spain have lovely food? Yes
Does Britain have lovely food? Yes
```

- The for loop went through the list `Greece Spain Britain` and printed a statement with each item in the list

- Let's clean up the file names with a for loop!

Cleaning up the filenames

But how should we remove the weird endings???

Cleaning up the filenames

But how should we remove the weird endings???

We first show an example that uses `cut` and `rev`.

Any guesses what these commands might do?

Cleaning up the filenames

But how should we remove the weird endings???

We first show an example that uses `cut` and `rev`.

Any guesses what these commands might do?

- `rev`: reverses a character string
- `cut`: cuts a string into multiple pieces

Cleaning up filenames

Let's try this out!

```
$ echo "aBcDeF 654321" | rev
```

```
123456 FeDcBa
```

Cleaning up filenames

Let's try this out!

```
$ echo "aBcDeF 654321" | rev
```

```
123456 FeDcBa
```

`cut` needs some arguments.

- `-d` specifies the field **d**elimiter we are using. Here it is space (" ").
- `-f` specifies which **f**ield we wish to cut out (the second one).

```
$ echo "aBcDeF 654321" | cut -d " " -f 2
```

```
654321
```

Cleaning up filenames

Using these tools, we can start cleaning up the desired filenames like this:

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   #mv $file $new_name
> done
```

Use echo to make a 'dry-run', and when you're happy with the proposed output uncomment the mv command and re-run the for loop.

Cleaning up filenames

Using these tools, we can start cleaning up the desired filenames like this:

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   #mv $file $new_name
> done
```

Use `echo` to make a 'dry-run', and when you're happy with the proposed output uncomment the `mv` command and re-run the `for` loop.

BUT WAIT! This code is cumbersome to write, read and understand.

Wait, what just happened?

$() tells bash to run the commands within parentheses and interpret the output as a string, which is then assigned to the variable new_name

Wait, what just happened?

$() tells bash to run the commands within parentheses and interpret the output as a string, which is then assigned to the variable new_name

We start out with a filepath:

Boosted-BBB/Friday/night/and/the/lights/are/low/fanta.JPG.MP3.TXT

Wait, what just happened?

`$()` tells bash to run the commands within parentheses and interpret the output as a string, which is then assigned to the variable `new_name`

We start out with a filepath:

```
Boosted-BBB/Friday/night/and/the/lights/are/low/fanta.JPG.MP3.TXT
```

We `rev`erse the filename:

```
TXT.3PM.GPJ.atnaf/wol/era/sthgil/eht/dna/thgin/yadirF/BBB-detsooB
```

Wait, what just happened?

$() tells bash to run the commands within parentheses and interpret the output as a string, which is then assigned to the variable new_name

We start out with a filepath:

```
Boosted-BBB/Friday/night/and/the/lights/are/low/fanta.JPG.MP3.TXT
```

We reverse the filename:

```
TXT.3PM.GPJ.atnaf/wol/era/sthgil/eht/dna/thgin/yadirF/BBB-detsooB
```

We cut the string at each -delimiter (.), and keep everything after the first delimiter (-fields 2-999):

```
    3PM.GPJ.atnaf/wol/era/sthgil/eht/dna/thgin/yadirF/BBB-detsooB
```

Wait, what just happened?

$() tells bash to run the commands within parentheses and interpret the output as a string, which is then assigned to the variable new_name

We start out with a filepath:

```
Boosted-BBB/Friday/night/and/the/lights/are/low/fanta.JPG.MP3.TXT
```

We reverse the filename:

```
TXT.3PM.GPJ.atnaf/wol/era/sthgil/eht/dna/thgin/yadirF/BBB-detsooB
```

We cut the string at each -delimiter (.), and keep everything after the first delimiter (-fields 2-999):

```
    3PM.GPJ.atnaf/wol/era/sthgil/eht/dna/thgin/yadirF/BBB-detsooB
```

We reverse what is left back to its original orientation:

```
Boosted-BBB/Friday/night/and/the/lights/are/low/fanta.JPG.MP3
```

Wait, what just happened?

`$()` tells bash to run the commands within parentheses and interpret the output as a string, which is then assigned to the variable `new_name`

We start out with a filepath:

```
Boosted-BBB/Friday/night/and/the/lights/are/low/fanta.JPG.MP3.TXT
```

We `rev`erse the filename:

```
TXT.3PM.GPJ.atnaf/wol/era/sthgil/eht/dna/thgin/yadirF/BBB-detsooB
```

We `cut` the string at each `-d`elimiter (`.`), and keep everything after the first delimiter (`-f`ields `2-999`):

```
    3PM.GPJ.atnaf/wol/era/sthgil/eht/dna/thgin/yadirF/BBB-detsooB
```

We `rev`erse what is left back to its original orientation:

```
Boosted-BBB/Friday/night/and/the/lights/are/low/fanta.JPG.MP3
```

We then rename the file to the new filename with `mv`.

Writing pretty code

It is a good idea to avoid clunky code like what you just saw.

How to make this code simpler? **Do not run this code!**

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   # mv $file $new_name
> done
```

Writing pretty code

It is a good idea to avoid clunky code like what you just saw.

How to make this code simpler? **Do not run this code!**

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   # mv $file $new_name
> done
```

We can make it shorter and better with **parameter expansion** (the magic).

Writing pretty code

It is a good idea to avoid clunky code like what you just saw.

How to make this code simpler? **Do not run this code!**

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   # mv $file $new_name
> done
```

We can make it shorter and better with **parameter expansion** (the magic).

Writing pretty code

We can now rewrite this code. **DO NOT RUN THIS CODE!**

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   # mv $file $new_name
> done
```

Writing pretty code

We can now rewrite this code. **DO NOT RUN THIS CODE!**

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   # mv $file $new_name
> done
```

To this:

```
$ for file in $(find Boosted-BBB/ -type f -name "*JPG*"); do
>   echo ${file} ${file%.*}
>   # mv ${file} ${file%.*}
> done
```

Try dry-running both and check the result is the same! Is there a difference in runtime?

Writing pretty code

We can now rewrite this code. **DO NOT RUN THIS CODE!**

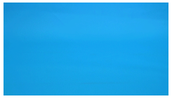
```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   # mv $file $new_name
> done
```

To this:

```
$ for file in $(find Boosted-BBB/ -type f -name "*JPG*"); do
>   echo ${file} ${file%.*}
>   # mv ${file} ${file%.*}
> done
```

Try dry-running both and check the result is the same! Is there a difference in runtime?

Result: **0.051s** versus **0.003s** when running echo!



Writing pretty code

We can now rewrite this code. **DO NOT RUN THIS CODE!**

```
$ for file in $(find Boosted-BBB/ -type f -name "*$suffix*"); do
>   new_name=$(echo $file | rev | cut -d "." -f 2-999 | rev)
>   echo $file $new_name
>   # mv $file $new_name
> done
```

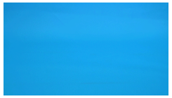
To this:

```
$ for file in $(find Boosted-BBB/ -type f -name "*JPG*"); do
>   echo ${file} ${file%.*}
>   # mv ${file} ${file%.*}
> done
```

Try dry-running both and check the result is the same! Is there a difference in runtime?

Result: **0.051s** versus **0.003s** when running echo!

When sure it works, remove the comment in the 2nd



Almost done!

We now have all the files named similarly, but some things are still a bit off. The file suffix `JPG` is conventionally written in lowercase characters (`jpg`).

Let's change all filename suffixes to be in lowercase letters!

Almost done!

We now have all the files named similarly, but some things are still a bit off. The file suffix `JPG` is conventionally written in lowercase characters (`jpg`).

Let's change all filename suffixes to be in lowercase letters!

Can be done with parameter expansion, but we can use **reg**ular **ex**pressions to do this without a for loop.

Almost done!

We now have all the files named similarly, but some things are still a bit off. The file suffix `JPG` is conventionally written in lowercase characters (`jpg`).

Let's change all filename suffixes to be in lowercase letters!

Can be done with parameter expansion, but we can use **reg**ular **ex**pressions to do this without a for loop.

- Regex is an important concept. You will find them in most programming languages.
- Syntax can vary from language to language, but here's how they work in bash.

Concept: Regular expressions

- Special strings and characters that define a 'search pattern'
- Used in 'Search' or 'Search/Replace' functions e.g. in excel!
- You have already used them!

Concept: Regular expressions

- Special strings and characters that define a 'search pattern'
- Used in 'Search' or 'Search/Replace' functions e.g. in excel!
- You have already used them!

To prepare, download the following file

```
$ wget git.io/Boosted-BBB-regex
$ mv Boosted-BBB-regex regex.txt
```

Concept: Regular expressions

- Special strings and characters that define a 'search pattern'
- Used in 'Search' or 'Search/Replace' functions e.g. in excel!
- You have already used them!

To prepare, download the following file

```
$ wget git.io/Boosted-BBB-regex
$ mv Boosted-BBB-regex regex.txt
```

Let's also look at the contents.

```
$ cat regex.txt
```

Regex Basics

Three regex special character 'categories'

Regex Basics

Three regex special character 'categories'

- ., *, ^, $ (etc.): special characters that are translated to regex function **first** ('escaped' with \ to find the literal symbol)

Regex Basics

Three regex special character 'categories'

- ., *, ^, $ (etc.): special characters that are translated to regex function **first** ('escaped' with \ to find the literal symbol)
- \t, \w, \D (etc.): **letter-based** special characters that must have \ to be 'translated'

Regex Basics

Three regex special character 'categories'

- ., *, ^, $ (etc.): special characters that are translated to regex function **first** ('escaped' with \ to find the literal symbol)

- \t, \w, \D (etc.): **letter-based** special characters that must have \ to be 'translated'

- [], (), (etc.): range, grouping, or 'capturing' matching regex within **brackets**

Regex Basics

Three regex special character 'categories'

- ., *, ^, $ (etc.): special characters that are translated to regex function **first** ('escaped' with \ to find the literal symbol)
- \t, \w, \D (etc.): **letter-based** special characters that must have \ to be 'translated'
- [], (), (etc.): range, grouping, or 'capturing' matching regex within **brackets**

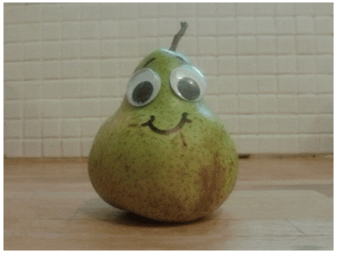
pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear

Regex Basics

```
$ grep '.ear' regex.txt
```

> *Finds strings containing: any character + ear*

- . : match any character



pear

pier

pir

per

par

pur

bear

beer

br

ber

be*r

rear

Regex Basics

```
$ grep 'p[iea]r' regex.txt
```

> *String starting with p+ one of i or e or a +r*

- . : match any character
- []: match range of characters within []

pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear

Regex Basics

```
$ grep 'p[^iea]r' regex.txt
```

> *String starting with p+ any character except i, e or a+r*

- . : match any character
- []: match range of characters within []
- [^]: match range of characters except the ones in the bracket

pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear

Regex Basics

```
$ grep 'be*r' regex.txt
```

> *String that starts with b+ zero or multiple 'e' +r*

- . : match any character
- []: match range of characters within []
- [^]: match range of characters except the ones in the bracket
- *: match 0 or more of the preceding items

pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear

Regex Basics

```
$ grep 'be\*r' regex.txt
```

> *String 'be*r'*

- . : match any character
- []: match range of characters within []
- [^]: match range of characters except the ones in the bracket
- *: match 0 or more of the preceding items
- \: do not interpret next character

pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear

Regex Basics

```
$ grep 'be\+r' regex.txt
```

> *String starting with b+ one or multiple 'e'+r*

- . : match any character
- []: match range of characters within []
- [^]: match range of characters except the ones in the bracket
- *: match 0 or more of the preceding items
- \: do not interpret next character
- \+: match 1 or more of the preceding items

pear

pier

pir

per

par

pur

bear

beer

br

ber

be*r

rear

Regex Basics

```
$ grep 'be\?r' regex.txt
```

> *String starting with b+ zero or one 'e'+r*

- . : match any character
- []: match range of characters within []
- [^]: match range of characters except the ones in the bracket
- *: match 0 or more of the preceding items
- \: do not interpret next character
- \+: match 1 or more of the preceding items
- \?: match 0 or 1 of the preceding items

pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear

Regex Basics

```
$ grep '^[rb]\+' regex.txt
```

> *Lines starting with one or multiple of: r or b*

- . : match any character
- []: match range of characters within []
- [^]: match range of characters except the ones in the bracket
- *: match 0 or more of the preceding items
- \: do not interpret next character
- \+: match 1 or more of the preceding items
- \?: match 0 or 1 of the preceding items
- ^: the beginning of the line

pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear

Regex Basics

```
$ grep 'r$' regex.txt
```

> *Lines ending with r*

- . : match any character
- []: match range of characters within []
- [^]: match range of characters except the ones in the bracket
- *: match 0 or more of the preceding items
- \: do not interpret next character
- \+: match 1 or more of the preceding items
- \?: match 0 or 1 of the preceding items
- ^: the beginning of the line
- $: the end of the line

> *This can be intimidating, however there are lots of resources on the internet (reminder: Google everything!) to help, such as: https://regex101.com/. Note that regex's can be slightly different per shell and language!*

```
pear
pier
pir
per
par
pur
bear
beer
br
ber
be*r
rear
```

Regex example

```
fanta.JPG
BydgoszczForest.JPG
snore.JPG
Bubobubo.JPG
giacomo.JPG
netsukeJapan.JPG
nomnom.JPG
pompeii.JPG
AlopochenaegyptiacaArnhem.JPG
exhibitRoyal.JPG
stretch.JPG
weimanarer.JPG
excited.JPG
licorne.JPG
angry.JPG
```

Which Regex would you use to find all the files ending with .JPG??

Regex example

```
fanta.JPG
BydgoszczForest.JPG
snore.JPG
Bubobubo.JPG
giacomo.JPG
netsukeJapan.JPG
nomnom.JPG
pompeii.JPG
AlopochenaegyptiacaArnhem.JPG
exhibitRoyal.JPG
stretch.JPG
weimanarer.JPG
excited.JPG
licorne.JPG
angry.JPG
```

Which Regex would you use to find all the files ending with .JPG??

```
$ find Boosted-BBB/ -type f -name '*.JPG'
```

rename

rename lets you apply a regex to the name of files to rename them.

To convert all suffixes in the directory to lowercase characters:

```
$ find Boosted-BBB/ -type f -name '*.JPG' | rename 's/\.JPG$/.jpg/'
```

> *Check with find whether the names are now as you expect!*

rename

rename lets you apply a regex to the name of files to rename them.

To convert all suffixes in the directory to lowercase characters:

```
$ find Boosted-BBB/ -type f -name '*.JPG' | rename 's/\.JPG$/.jpg/'
```

> *Check with find whether the names are now as you expect!*

No for loop needed (yay for pipes!)!

The expression given to rename has three parts, separated by /

- First, we define we want to **s**ubstitute the regex matches for another string

rename

`rename` lets you apply a regex to the name of files to rename them.

To convert all suffixes in the directory to lowercase characters:

```
$ find Boosted-BBB/ -type f -name '*.JPG' | rename 's/\.JPG$/.jpg/'
```

> *Check with `find` whether the names are now as you expect!*

No `for` loop needed (yay for pipes!)!

The expression given to `rename` has three parts, separated by `/`

- First, we define we want to **s**ubstitute the regex matches for another string
- Second, we define the regex to query. `\.JPG$`

rename

rename lets you apply a regex to the name of files to rename them.

To convert all suffixes in the directory to lowercase characters:

```
$ find Boosted-BBB/ -type f -name '*.JPG' | rename 's/\.JPG$/.jpg/'
```

> *Check with find whether the names are now as you expect!*

No for loop needed (yay for pipes!)!

The expression given to rename has three parts, separated by /

- First, we define we want to **s**ubstitute the regex matches for another string
- Second, we define the regex to query. \.JPG$

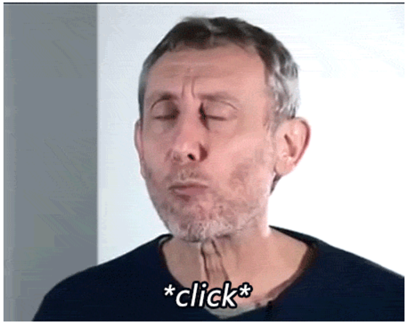
> *Remember: to escape a character (so read as an actual character, not as a regex), use \ before it*

- Finally, we specify what we want to substitute matches with, which is .jpg

Onwards!

Ok, so we can now use find to see all the new and pretty filepaths:

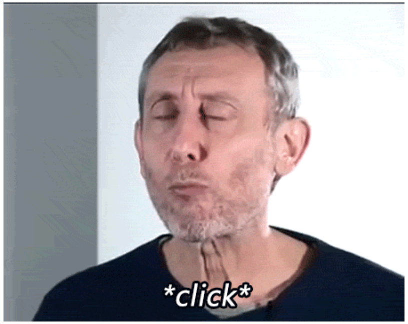
```
$ suffix="jpg"
$ find Boosted-BBB/ -type f -name "*${suffix}"
```



Onwards!

Ok, so we can now use find to see all the new and pretty filepaths:

```
$ suffix="jpg"
$ find Boosted-BBB/ -type f -name "*${suffix}"
```



We can finally start sorting the pictures into categories!

To do that, we need to keep track of all the file names. We can easily gather this information using a **redirect**!

Let's redirect!

We can get a list of all the file names by redirecting the stdout of the find command.

```
$ suffix="jpg"
$ find Boosted-BBB/ -type f -name "*${suffix}" > File_names.txt
```

Let's redirect!

We can get a list of all the file names by redirecting the stdout of the find command.

```
$ suffix="jpg"
$ find Boosted-BBB/ -type f -name "*${suffix}" > File_names.txt
```

This time, nothing was printed on your screen, because you redirected that output into a file.

You can cat the resulting file to see that everything worked.

Getting parts of a filepath

Before moving on, there are two useful commands you should know.

`basename` will tell you the file name, while stripping the path to the file.

```
$ basename Boosted-BBB//Having/the/time/of/your/life/bubobubo.JPG.MP3.TXT
```

```
bubobubo.JPG.MP3.TXT
```

Getting parts of a filepath

Before moving on, there are two useful commands you should know.

`basename` will tell you the file name, while stripping the path to the file.

```
$ basename Boosted-BBB//Having/the/time/of/your/life/bubobubo.JPG.MP3.TXT
```

```
bubobubo.JPG.MP3.TXT
```

`dirname` does the opposite. It will tell you the path to the directory that a file is in, while omitting the name of the file.

```
$ dirname Boosted-BBB//Having/the/time/of/your/life/bubobubo.JPG.MP3.TXT
```

```
Boosted-BBB//Having/the/time/of/your/life
```

Reading from a file

Ok, so you now have a file with all the paths to the images we need. But the folder structure is still a mess. It's time to read the contents of the file with a `while` loop!

Reading from a file

Ok, so you now have a file with all the paths to the images we need. But the folder structure is still a mess. It's time to read the contents of the file with a `while` loop!

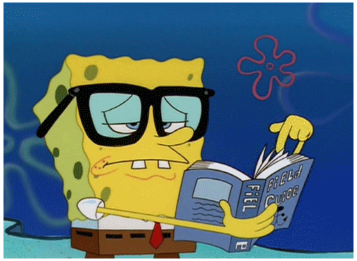
A `while` loop is a special type of repeating code that keeps going until it is interrupted.

Reading from a file

Ok, so you now have a file with all the paths to the images we need. But the folder structure is still a mess. It's time to read the contents of the file with a while loop!

A while loop is a special type of repeating code that keeps going until it is interrupted.

We will also use read. This command takes the contents of the file and loads them into a specified variable.



Reading from a file

Ok, so you now have a file with all the paths to the images we need. But the folder structure is still a mess. It's time to read the contents of the file with a while loop!

A while loop is a special type of repeating code that keeps going until it is interrupted.

We will also use read. This command takes the contents of the file and loads them into a specified variable.

```
$ mkdir images
$ while read filepath; do
>   echo "${filepath}" images/$(basename ${filepath})
>   # mv ${filepath} images/$(basename ${filepath})
> done < File_names.txt
```

When you're ready, uncomment the mv command to move each file from the original location into the new location!

> *Question: in this context, why do you have to use 'basename' for the target directory?*

Concept: While Loops

We have previously seen the concept of for loop:

```
$ for file in file1 file2 file3 file4; do
>   echo "${file}"
> done
```

For loops repeat a set of code for a set of items, by changing the value of a variable in each iteration.

Concept: While Loops

For loops are finite, they go through your list and stop when they run out of things.

Concept: While Loops

For loops are finite, they go through your list and stop when they run out of things.

Instead, a while loop keeps going until a statement is false.

```
$ while [statement]; do     #means while statement is true do
>   [whatever you want to do]
> done
```

Concept: While Loops

For loops are finite, they go through your list and stop when they run out of things.

Instead, a while loop keeps going until a statement is false.

```
$ while [statement]; do     #means while statement is true do
>   [whatever you want to do]
> done
```

An easy **pseudocode** example:

Concept: While Loops

For loops are finite, they go through your list and stop when they run out of things.

Instead, a while loop keeps going until a statement is false.

```
$ while [statement]; do     #means while statement is true do
>   [whatever you want to do]
> done
```

An easy **pseudocode** example:

```
$ n=3
$ while n > 0; do
>   echo $n
>   n=$n - 1
> done
```

```
3
2
1
```

Concept: While Loops

Didn't you say while loop are infinite? **Pseudocode**:

```
$ n=3
$ while n < 5; do
>   echo $n
>   n=$n - 1
> done
```

Concept: While Loops

Didn't you say while loop are infinite? **Pseudocode**:

```
$ n=3
$ while n < 5; do
>   echo $n
>   n=$n - 1
> done
```

- Always include a stop!
    - i.e. ensure your condition will eventually become 'false'!
- Emergencies: Ctrl + C (cancel the loop)

Concept: While Loops

Didn't you say while loop are infinite? **Pseudocode**:

```
$ n=3
$ while n < 5; do
>   echo $n
>   n=$n - 1
> done
```

- Always include a stop!
  - i.e. ensure your condition will eventually become 'false'!
- Emergencies: Ctrl + C (cancel the loop)

Following our example from the beginning...

```
$ while read filepath; do
>   echo "${filepath}" images/$(basename ${filepath})
>   # mv ${filepath} images/$(basename ${filepath})
> done < File_names.txt
```

...the condition read filepath becomes false when there are no more lines in the file File_names.txt (i.e. 'EOF')

Pasting things side by side!

As you remember from the beginning, we downloaded a metadata file, which includes different metadata categories for each file.

Lets look in the file!

```
$ cat Boosted-BBB-meta.tsv
```

Pasting things side by side!

As you remember from the beginning, we downloaded a metadata file, which includes different metadata categories for each file.

Lets look in the file!

```
$ cat Boosted-BBB-meta.tsv
```

Now we can put together a list that states which category each image is part of.

Pasting things side by side!

As you remember from the beginning, we downloaded a metadata file, which includes different metadata categories for each file.

Lets look in the file!

```
$ cat Boosted-BBB-meta.tsv
```

Now we can put together a list that states which category each image is part of.

You can use paste to paste the two lists together, and save the results!

```
$ ls -1 images/* | paste - Boosted-BBB-meta.tsv # > Annotations.txt
```

Pasting things side by side!

As you remember from the beginning, we downloaded a metadata file, which includes different metadata categories for each file.

Lets look in the file!

```
$ cat Boosted-BBB-meta.tsv
```

Now we can put together a list that states which category each image is part of.

You can use `paste` to paste the two lists together, and save the results!

```
$ ls -1 images/* | paste - Boosted-BBB-meta.tsv # > Annotations.txt
```

```
images/alopochenaegyptiacaArnhem.jpg    alopochenaegyptiacaArnhem    C    Funny
images/angry.jpg    angry    B    Artwork
images/bubobubo.jpg    bubobubo    C    Normal
...
images/snore.jpg    snore    B    Normal
images/stretch.jpg    stretch    B    Funny
images/weimanarer.jpg    weimanarer    A    Normal
```

> *Disclaimer: literally pastes columns, no matching done. Only works if both columns are in the corresponding order.*

Editing text with sed

To share these images with your internet friends, you need to properly specify the category names.

Let's add the actual category names to the Annotations.txt.

Editing text with sed

To share these images with your internet friends, you need to properly specify the category names.

Let's add the actual category names to the Annotations.txt.

You can use sed, short for **s**tream **ed**itor, with a regex to edit the contents of a datastream on-the-fly.

```
$ sed 's/A/dog/' Annotations.txt
```

Editing text with sed

To share these images with your internet friends, you need to properly specify the category names.

Let's add the actual category names to the Annotations.txt.

You can use sed, short for **s**tream **ed**itor, with a regex to edit the contents of a datastream on-the-fly.

```
$ sed 's/A/dog/' Annotations.txt
```

```
images/alopochenaegyptiacadogrnhem.jpg    alopochenaegyptiacaArnhem    C    Funny
images/angry.jpg    angry    B    dogrtwork
images/bubobubo.jpg    bubobubo    C    Normal
...
images/snore.jpg    snore    B    Normal
images/stretch.jpg    stretch    B    Funny
images/weimanarer.jpg    weimanarer    dog    Normal
```

Uh-oh!

Editing text with sed

To share these images with your internet friends, you need to properly specify the category names.

Let's add the actual category names to the Annotations.txt.

You can use sed, short for **s**tream **ed**itor, with a regex to edit the contents of a datastream on-the-fly.

```
$ sed 's/A/dog/' Annotations.txt
```

```
images/alopochenaegyptiacadogrnhem.jpg    alopochena                    nny
images/angry.jpg    angry    B    dogrtwork
images/bubobubo.jpg    bubobubo    C    Normal
...
images/snore.jpg    snore    B    Normal
images/stretch.jpg    stretch    B    Funny
images/weimanarer.jpg    weimanarer    dog    Normal
```

Uh-oh!

Editing text with sed

```
$ sed 's/\tA\t/\tdog\t/' Annotations.txt
```

```
images/alopochenaegyptiacaArnhem.jpg	alopochenaegyptiacaArnhem	C	Funny
images/angry.jpg	angry	B	Artwork
images/bubobubo.jpg	bubobubo	C	Normal
...
images/snore.jpg	snore	B	Normal
images/stretch.jpg	stretch	B	Funny
images/weimanarer.jpg	weimanarer	dog	Normal
```

> *On Macs, `sed` does not recognise `\t`.You will need to type in a tab character.*

Editing text with sed

```
$ sed 's/\tA\t/\tdog\t/' Annotations.txt
```

```
images/alopochenaegyptiacaArnhem.jpg	alopochenaegyptiacaArnhem	C	Funny
images/angry.jpg	angry	B	Artwork
images/bubobubo.jpg	bubobubo	C	Normal
...
images/snore.jpg	snore	B	Normal
images/stretch.jpg	stretch	B	Funny
images/weimanarer.jpg	weimanarer	dog	Normal
```

> *On Macs, `sed` does not recognise `\t`.You will need to type in a tab character.*

Use `-e` to provide multiple regular **e**xpressions to `sed`.

```
$ sed -e 's/\tA\t/\tdog\t/' -e 's/\tB\t/\tcat\t/' -e 's/\tC\t/\tbird\t/' Annotations.txt
```

Editing text with sed

```
$ sed 's/\tA\t/\tdog\t/' Annotations.txt
```

```
images/alopochenaegyptiacaArnhem.jpg    alopochenaegyptiacaArnhem    C    Funny
images/angry.jpg    angry    B    Artwork
images/bubobubo.jpg    bubobubo    C    Normal
...
images/snore.jpg    snore    B    Normal
images/stretch.jpg    stretch    B    Funny
images/weimanarer.jpg    weimanarer    dog    Normal
```

> *On Macs, sed does not recognise \t.You will need to type in a tab character.*

Use -e to provide multiple regular **e**xpressions to sed.

```
$ sed -e 's/\tA\t/\tdog\t/' -e 's/\tB\t/\tcat\t/' -e 's/\tC\t/\tbird\t/' Annotations.txt
```

When you are happy with the results, it is time to save the edits.

sed can edit a file **i**n place, with the -i option.

```
$ sed -i -e 's/\tA\t/\tdog\t/' -e 's/\tB\t/\tcat\t/' -e 's/\tC\t/\tbird\t/' Annotations.txt
```

Cleanin' up my closet

Lets actually organise our into descriptive folders based our metadata file!

For this, we need to use conditionals. This is a comparison of two things, and if they are the same something happens, if different, something else happens.

The most basic conditional is an if else statement.

Cleanin' up my closet

Lets actually organise our into descriptive folders based our metadata file!

For this, we need to use conditionals. This is a comparison of two things, and if they are the same something happens, if different, something else happens.

The most basic conditional is an if else statement.

The basic syntax is like this

```
$ if [[ ${my_variable} == "banana" ]]; then
>   echo "Monkey takes a banana and runs away happy."
> else
>   echo "Monkey doesn't want that."
> fi
```

You can have sequential conditions too with elif, short for **el**se **if**.

```
$ if [[ ${my_variable} == "banana" ]]; then
>   echo "Monkey takes a banana and runs away happy."
> elif [[ ${my_variable} == "mango" ]]; then
>   echo "Monkey takes a mango and eats it while staring at you."
> else
>   echo "Monkey doesn't want that."
> fi
```

Conditions of conditionals

- [[behaves different to [. Usually, [[is what you want. [*Long story, trust us.*]

Conditions of conditionals

- [[behaves different to [. Usually, [[is what you want. [*Long story, trust us.*]
- You can evaluate mathematical equations with ((

Conditions of conditionals

- [[behaves different to [. Usually, [[is what you want. [*Long story, trust us.*]
- You can evaluate mathematical equations with ((

```
$ if (( 5 - 2 == 3)); then
>   echo "YES"
> fi
```

```
YES
```

Conditions of conditionals

- `[[` behaves different to `[`. Usually, `[[` is what you want. [*Long story, trust us.*]
- You can evaluate mathematical equations with `((`
- `!` can be used as a "not".

Conditions of conditionals

- [[behaves different to [. Usually, [[is what you want. [*Long story, trust us.*]
- You can evaluate mathematical equations with ((
- ! can be used as a "not".

```
$ if (( 5 - 2 == 3)); then
>   echo "YES"
> fi
```

```
YES
```

```
$ if ! (( 5 - 2 == 3)); then
>   echo "YES"
> else
>   echo "NO"
> fi
```

```
NO
```

Conditions of conditionals

- Some options can be used to check if files exist, or is a variable has non-zero length.

Conditions of conditionals

- Some options can be used to check if files exist, or is a variable has non-zero length.

```
$ if [[ -f Annotations.txt ]]; then
>   echo "File exists."
> fi

$ if [[ -n ${banana} ]]; then
>   echo "Variable is set."
> else
>   echo "Variable is NOT set."
> fi
```

```
File exists.
Variable is NOT set.
```

Conditions of conditionals

- You can even combine multiple conditionals
  - `&&`: 'AND' - both must evaluate true
  - `||`: 'OR' - at least one must evaluate true

Conditions of conditionals

- You can even combine multiple conditionals
  - &&: 'AND' - both must evaluate true
  - ||: 'OR' - at least one must evaluate true

```
$ LifeUniverseEverything=42
$ hitchhikers="awesome"
## AND
$ if [[ ${LifeUniverseEverything} == 42  && ${hitchhikers} == "awesome" ]]; then
>   echo "Don't panic!"
> fi
```

Conditions of conditionals

- You can even combine multiple conditionals
  - &&: 'AND' - both must evaluate true
  - ||: 'OR' - at least one must evaluate true

```
$ LifeUniverseEverything=42
$ hitchhikers="awesome"
## AND
$ if [[ ${LifeUniverseEverything} == 42  && ${hitchhikers} == "awesome" ]]; then
>   echo "Don't panic!"
> fi
```

```
$ LifeUniverseEverything=41
$ hitchhikers="awesome"
## OR
$ if [[ ${LifeUniverseEverything} == 42  || ${hitchhikers} == "awesome" ]]; then
>   echo "Still don't panic!"
> fi
```

> *Play around with the variables to get a feel!*

Sorting the images by category

We now want to create a directory for each category, and move images into each.

Sorting the images by category

We now want to create a directory for each category, and move images into each.

Let's add some conditionals! Before running, remember to try a *dry run* with `echo`!

Sorting the images by category

We now want to create a directory for each category, and move images into each.

Let's add some conditionals! Before running, remember to try a *dry run* with echo!

```
$ cd ~/Boosted-BBB/

## Parse the annotations file into variables
$ while read line; do
>   image_name=$(echo "${line}" | cut -f1)
>   animal=$(echo "${line}" | cut -f3)
>
>   echo "${image_name}	${animal}"
>
> done < Annotations.txt
```

```
images/alopochenaegyptiacaArnhem.jpg	bird
images/angry.jpg	cat
images/bubobubo.jpg	bird
images/bydgoszczForest.jpg	bird
[...]
images/pompeii.jpg	dog
images/snore.jpg	cat
images/stretch.jpg	cat
images/weimanarer.jpg	dog
```

Sorting the images by category

We now want to create a directory for each category, and move images into each.

Let's add some conditionals! Before running, remember to try a *dry run* with echo!

```
$ cd ~/Boosted-BBB/

## Parse the annotations file into variables
$ while read line; do
>   image_name=$(echo "${line}" | cut -f1)
>   animal=$(echo "${line}" | cut -f3)
>
>   # echo "${image_name}    ${animal}"
>
>   ## Make a new directory for each animal, if one doesn't exist.
>   mkdir -p images/${animal}
>
>
>
>
>
>
>
>
>
>
```

Sorting the images by category

We now want to create a directory for each category, and move images into each.

Let's add some conditionals! Before running, remember to try a *dry run* with echo!

```
$ cd ~/Boosted-BBB/

## Parse the annotations file into variables
$ while read line; do
>   image_name=$(echo "${line}" | cut -f1)
>   animal=$(echo "${line}" | cut -f3)
>
>   # echo "${image_name}    ${animal}"
>
>   ## Make a new directory for each animal, if one doesn't exist.
>   mkdir -p images/${animal}
>
>   ## If animal matches one of the three, move the image.
>   if [[ ${animal} == "cat" ]]; then
>     mv ${image_name} images/cat/
>   elif [[ ${animal} == "dog" ]]; then
>     mv ${image_name} images/dog/
>   elif [[ ${animal} == "bird" ]]; then
>     mv ${image_name} images/bird/
>   fi
>
```

Housekeeping

Let's see if everything moved where we wanted.

```
$ find ~/BareBonesBash/Boosted-BBB/images/ -type f -name "*jpg"
```

Housekeeping

Let's see if everything moved where we wanted.

```
$ find ~/BareBonesBash/Boosted-BBB/images/ -type f -name "*jpg"
```

```
~/BareBonesBash/Boosted-BBB/images/cat/snore.jpg
~/BareBonesBash/Boosted-BBB/images/cat/giacomo.jpg
~/BareBonesBash/Boosted-BBB/images/cat/excited.jpg
~/BareBonesBash/Boosted-BBB/images/cat/angry.jpg
~/BareBonesBash/Boosted-BBB/images/cat/stretch.jpg
~/BareBonesBash/Boosted-BBB/images/dog/licorne.jpg
~/BareBonesBash/Boosted-BBB/images/dog/fanta.jpg
~/BareBonesBash/Boosted-BBB/images/dog/weimanarer.jpg
~/BareBonesBash/Boosted-BBB/images/dog/pompeii.jpg
~/BareBonesBash/Boosted-BBB/images/dog/nomnom.jpg
~/BareBonesBash/Boosted-BBB/images/bird/alopochenaegyptiacaArnhem.jpg
~/BareBonesBash/Boosted-BBB/images/bird/bubobubo.jpg
~/BareBonesBash/Boosted-BBB/images/bird/netsukeJapan.jpg
~/BareBonesBash/Boosted-BBB/images/bird/bydgoszczForest.jpg
~/BareBonesBash/Boosted-BBB/images/bird/exhibitRoyal.jpg
```

Good! Everything moved into the correct subfolder!

I have to do this every day!

We are already being lazy by getting the computer to loop through each file.

But what do you do if you have to do the same thing EVERYDAY?

Do you really wanna write all the commands every time?!

I have to do this every day!

We are already being lazy by getting the computer to loop through each file.

But what do you do if you have to do the same thing EVERYDAY?

Do you really wanna write all the commands every time?!

The ultimate goal of anyone working on the command line is to make a program which you can run with a single command and it does all the work for you.

That program is called a ***script***.

What's a script?

Similar to a play/movie script that tells actors what to do and the sequence in which they should do it, a computer script is a file containing all the commands that you want the computer to perform in a given order.

So let's start writing your first script `first_script.sh`! Open a text editor, we will use `nano`

```
$ nano first_script.sh
```

What's a script?

Similar to a play/movie script that tells actors what to do and the sequence in which they should do it, a computer script is a file containing all the commands that you want the computer to perform in a given order.

So let's start writing your first script first_script.sh! Open a text editor, we will use nano

```
$ nano first_script.sh
```

element.clientHeight + 0.02 *
window.scroll(0, scrollHeight);
}

Your first script!

The first thing you almost always need to do with any script is to specify which language the script is using. This is done with a 'shebang'

It consists of a #! to indicate it's a shebang, then a path to a list that *unix stores locations of all programs in.

On the first line of your text editor window, type:

```
#! /usr/bin/env bash
```

Your first script!

For your first script we want the program to print "Hello world!"

How did we told bash to print something in screen?

Your first script!

For your first script we want the program to print "Hello world!"

How did we told bash to print something in screen?

```
#! /usr/bin/env bash

echo "Hello world"
```

save the file by presing **Ctrl+X**, press **"Y"** to confimr you want to save and press enter to save it as `first_script.sh`.

Your first script!

For your first script we want the program to print "Hello world!"

How did we told bash to print something in screen?

```
#! /usr/bin/env bash

echo "Hello world"
```

save the file by presing **Ctrl+X**, press **"Y"** to confimr you want to save and press enter to save it as `first_script.sh`.

That's it! You've made your first script!

How do you run a script?

Now to run the script, we do:

```
$ bash ./first_script.sh
```

How do you run a script?

Now to run the script, we do:

```
$ bash ./first_script.sh
```

```
Hello world
```

Input Variables

So now we want to change our script to instead of saying `Hello world`, it say `Hello <your_name>`

So our script looked like:

```
#! /usr/bin/env bash

echo "Hello world"
```

We can use variables for the arguments passed to a script.

Input Variables

So now we want to change our script to instead of saying `Hello world`, it say `Hello <your_name>`

So our script looked like:

```
#! /usr/bin/env bash

echo "Hello world"
```

We can use variables for the arguments passed to a script.

Wait... what are arguments??

Input Variables

So now we want to change our script to instead of saying Hello world, it say Hello <your_name>

So our script looked like:

```
#! /usr/bin/env bash

echo "Hello world"
```

We can use variables for the arguments passed to a script.

Wait... what are arguments??

It is a user supplied value that the script will use to perform the tasks

Input Variables

In Bash, the arguments passed on the command line can be called ${1},${2} ...

${1} is the first argument

Input Variables

In Bash, the arguments passed on the command line can be called ${1},${2} ...

${1} is the first argument

${2} is the second argument

Input Variables

In Bash, the arguments passed on the command line can be called ${1},${2} ...

${1} is the first argument

${2} is the second argument

${3} is the third argument

Input Variables

In Bash, the arguments passed on the command line can be called `${1}`,`${2}` ...

`${1}` is the first argument

`${2}` is the second argument

`${3}` is the third argument

and so on.

Input Variables

In Bash, the arguments passed on the command line can be called ${1},${2} ...

${1} is the first argument

${2} is the second argument

${3} is the third argument

and so on.

Let's go back to our script and change the printing message

Input Variables

This is our script from before:

```
#! /usr/bin/env bash


echo "Hello world"
```

Input Variables

This is our script from before:

```
#! /usr/bin/env bash

echo "Hello world"
```

First, we want to pass our name to the script as an argument.

Input Variables

This is our script from before:

```
#! /usr/bin/env bash

name=${1}

echo "Hello world"
```

First, we want to pass our name to the script as an argument.

Input Variables

This is our script from before:

```
#! /usr/bin/env bash
name=${1}
echo "Hello world"
```

First, we want to pass our name to the script as an argument.

Then, we want to print out "Hello < name >" instead of "Hello World".

Input Variables

This is our script from before:

```
#! /usr/bin/env bash

name=${1}

echo "Hello ${name}"
```

First, we want to pass our name to the script as an argument.

Then, we want to print out "Hello < name >" instead of "Hello World".

Input Variables

This is our script from before:

```
#! /usr/bin/env bash

name=${1}

echo "Hello ${name}"
```

First, we want to pass our name to the script as an argument.

Then, we want to print out "Hello < name >" instead of "Hello World".

The script now needs an argument to run, so we will run:

```
$ bash ./first_script.sh Aida
```

Input Variables

This is our script from before:

```
#! /usr/bin/env bash

name=${1}

echo "Hello ${name}"
```

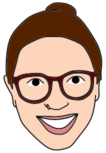
First, we want to pass our name to the script as an argument.

Then, we want to print out "Hello < name >" instead of "Hello World".

The script now needs an argument to run, so we will run:

```
$ bash ./first_script.sh Aida
```

```
Hello Aida
```



Best practices when coding

There are a few best practices that you should follow when writing code, to ensure that anyone can understand your code.

Best practices when coding

There are a few best practices that you should follow when writing code, to ensure that anyone can understand your code.

- Comment your code: add a short description of the steps. So in our first_script.sh, we should include:

Best practices when coding

There are a few best practices that you should follow when writing code, to ensure that anyone can understand your code.

- Comment your code: add a short description of the steps. So in our first_script.sh, we should include:

```
#! /usr/bin/env bash

## Read name from positional arguments
name=${1}

## Printing Hello and the specified variable into screen
echo "Hello ${name}"
```

Best practices when coding

There are a few best practices that you should follow when writing code, to ensure that anyone can understand your code.

- Comment your code: add a short description of the steps. So in our first_script.sh, we should include:

```
#! /usr/bin/env bash

## Read name from positional arguments
name=${1}

## Printing Hello and the specified variable into screen
echo "Hello ${name}"
```

- Give variables informative names.

Best practices when coding

There are a few best practices that you should follow when writing code, to ensure that anyone can understand your code.

- Comment your code: add a short description of the steps. So in our first_script.sh, we should include:

```
#! /usr/bin/env bash

## Read name from positional arguments
name=${1}

## Printing Hello and the specified variable into screen
echo "Hello ${name}"
```

- Give variables informative names.
- Try to have all bash variables in ${}. This helps distinguish them visually and ensures all variables are interpreted correctly.

Best practices when coding

There are a few best practices that you should follow when writing code, to ensure that anyone can understand your code.

- Comment your code: add a short description of the steps. So in our first_script.sh, we should include:

```
#! /usr/bin/env bash

## Read name from positional arguments
name=${1}

## Printing Hello and the specified variable into screen
echo "Hello ${name}"
```

- Give variables informative names.
- Try to have all bash variables in `${}`. This helps distinguish them visually and ensures all variables are interpreted correctly.
- Keep the code simple: try to simplify your code instead of having 1,000 lines

Best practices when coding

There are a few best practices that you should follow when writing code, to ensure that anyone can understand your code.

- Comment your code: add a short description of the steps. So in our first_script.sh, we should include:

```
#! /usr/bin/env bash

## Read name from positional arguments
name=${1}

## Printing Hello and the specified variable into screen
echo "Hello ${name}"
```

- Give variables informative names.
- Try to have all bash variables in ${}. This helps distinguish them visually and ensures all variables are interpreted correctly.
- Keep the code simple: try to simplify your code instead of having 1,000 lines
    - Avoid duplicating code.

Good coding practices

Add a help message. In our basic script, we could add the following:

```
#! /usr/bin/env bash

name=${1}

if [[ ${name} == "--help" || ${name} == "-h" ]]; then
  ## Print help message
  echo "This script prints Hello <your_name> into screen."
  echo "To run it type: bash ./first_script.sh <your_name>"
else
  ## Printing Hello and the specified variable into screen
  echo "Hello ${name}"
 fi
```

Good coding practices

Add a help message. In our basic script, we could add the following:

```
#! /usr/bin/env bash

name=${1}

if [[ ${name} == "--help" || ${name} == "-h" ]]; then
  ## Print help message
  echo "This script prints Hello <your_name> into screen."
  echo "To run it type: bash ./first_script.sh <your_name>"
else
  ## Printing Hello and the specified variable into screen
  echo "Hello ${name}"
 fi
```

You will often go back to old scripts and not remember the options and arguments they need.

Having help text will make it easier to remember.

Debugging your code

- Try your code outside the script

Debugging your code

- Try your code outside the script
- Add print statements to check the variables/commands render properly

Debugging your code

- Try your code outside the script
- Add print statements to check the variables/commands render properly
- Write the script by its functional parts.
  - Think what you want your script to do
  - Write/Draw the steps to do
  - Write code for the first step -> try it -> write code for the next step -> try it -> repeat until the end
  - Simplify your code

Debugging your code

- Try your code outside the script
- Add print statements to check the variables/commands render properly
- Write the script by its functional parts.
  - Think what you want your script to do
  - Write/Draw the steps to do
  - Write code for the first step -> try it -> write code for the next step -> try it -> repeat until the end
  - Simplify your code
- Explain your code to someone else! Talking through the logic of it will often make the problem obvious.
  - This is called the Rubber ducky approach, as many programmers have a rubber duck on their desk to explain their code to.

Things to keep in mind

- Code for the same task can be written in multiple ways
    - Some code is more **efficient** -> a.k.a runs faster.
    - Some code is more **readable**.

Things to keep in mind

- Code for the same task can be written in multiple ways
    - Some code is more **efficient** -> a.k.a runs faster.
    - Some code is more **readable**.
    - Some code is *both!*

Things to keep in mind

- Code for the same task can be written in multiple ways
  - Some code is more **efficient** -> a.k.a runs faster.
  - Some code is more **readable**.
  - Some code is *both!*
  - Some code is neither...

Things to keep in mind

- Code for the same task can be written in multiple ways
  - Some code is more **efficient** -> a.k.a runs faster.
  - Some code is more **readable**.
  - Some code is *both!*
  - Some code is neither...

- Practice makes perfect: the more you do it, the more you learn.

Quiz time!

Time to practice!

Your task now will be to generate a script to perform the image sorting that we have shown you in this presentation **and email it to us**.

BUT: This time you will need to make **an extra subdirectory within each of the categories** with the secondary description of the images!

> *That is column 3 of the metadata file. (Artwork, Baby, Funny, Historical, Normal)*

Time to practice!

Your task now will be to generate a script to perform the image sorting that we have shown you in this presentation **and email it to us**.

BUT: This time you will need to make **an extra subdirectory within each of the categories** with the secondary description of the images!

> *That is column 3 of the metadata file. (Artwork, Baby, Funny, Historical, Normal)*

For this, please make a new directory and download the data again:

```
$ mkdir ~/Boosted-BBB_scripting
$ cd ~/Boosted-BBB_scripting

## Get images zip and metadata file
$ wget git.io/Boosted-BBB-images # On Mac: `curl -LO`
$ wget git.io/Boosted-BBB-meta # On Mac: `curl -LO`

## Unzip image folders and rename metadata file
$ unzip Boosted-BBB-images
$ mv Boosted-BBB-meta  \
~/Boosted-BBB_scripting/Boosted-BBB-meta.tsv
```

Time to practice!

Your task now will be to generate a script to perform the image sorting that we have shown you in this presentation **and email it to us**.

BUT: This time you will need to make **an extra subdirectory within each of the categories** with the secondary description of the images!

> *That is column 3 of the metadata file. (Artwork, Baby, Funny, Historical, Normal)*

For this, please make a new directory and download the data again:

```
$ mkdir ~/Boosted-BBB_scripting
$ cd ~/Boosted-BBB_scripting

## Get images zip and metadata file
$ wget git.io/Boosted-BBB-images # On Mac: `curl -LO`
$ wget git.io/Boosted-BBB-meta # On Mac: `curl -LO`

## Unzip image folders and rename metadata file
$ unzip Boosted-BBB-images
$ mv Boosted-BBB-meta  \
~/Boosted-BBB_scripting/Boosted-BBB-meta.tsv
```

You are set up to start now!

Today I learned...

- find to locate files or directories
- What is a for loop
- Regular expressions for weird and wonderful pattern matching
- rename for renaming files
- While loops (reading contents of files)
- sed for on-the-fly string manipulation *within* files
- If statements and conditionals (if this, then do that, else do this)
- Scripts and arguments (now you're a programmer! Yes, you!)

Rerun: Enter the janitor!

Despite being lazy - you should ALWAYS keep your room tidy.

- This stops losing files
- Prevents getting lost in a maze of directories
- Accidentally permanently deleting a days worth of work

[*don't ask how many times this has happened.*]

Rerun: Enter the janitor!

Despite being lazy - you should ALWAYS keep your room tidy.

- This stops losing files
- Prevents getting lost in a maze of directories
- Accidentally permanently deleting a days worth of work

  [don't ask how many times this has happened.]

Lets remove:

- the Boosted-BBB directory
- **all of its contents**.

```
$ cd ~     # Don't delete a directory
         # while we are still in it!
$ rm -r Boosted-BBB*
```

Rerun: Enter the janitor!

Despite being lazy - you should ALWAYS keep your room tidy.

- This stops losing files
- Prevents getting lost in a maze of directories
- Accidentally permanently deleting a days worth of work

  [don't ask how many times this has happened.]

Lets remove:

- the Boosted-BBB directory
- **all of its contents**.

```
$ cd ~     # Don't delete a directory
         # while we are still in it!
$ rm -r Boosted-BBB*
```

There is more!

- This was a reduced version of previous BBB series
- You can find all the slides and walkthroughs here: https://barebonesbash.github.io/#/

Thanks to...

- Stephan Schiffels
    - for giving support and advice on cluster setup in our initial runs of BBB
- James Fellows Yates
    - Creator of many of those slides for the BBB courses we've given
- Zandra Färgenas
    - for the wonderful images of ourselves <3
- Google
    - Pretty much teaching all of this
- giphy, tenor
    - For procrastination
- fontawesome.com
    - for icons for making the logo