Bash stands for ‘**B**ourne **A**gain **Sh**ell’ (pun).

A Bash script has a few key defining features:

It usually starts with `#!/usr/bash` read as ‘*shebang*’. So your interpreter knows it is a Bash script and to use Bash located in `/usr/bash`. This could be a different path if you installed Bash somewhere else such as `/bin/bash` (type `which bash` to check first).

if you change the shebang line to `#!/bin/python` the editor will recognize it as a python file.

The file extension is `.sh` but technically not needed if scripts first line has *shebang* and path to bash (`#!/usr/bash`).

To run the script from terminal use:

`$ bash script_name.sh`

Or if your script has `#!/usr/bash` as first line then you could run the script using:

`$ ./script_name.sh`

In normal txt file if you write

echo “some text", ls or other commands and then if you use `$ cat` on that file it will just show those written texts, but if you use `$ bash thatfile.txt` it will execute the commands without using shell within a shell or having `.sh` file.

Lines starting with `#` are comments.

For multiline comments there are two methods:

First you can use

```
>>name [comments]
[comments]
name
```

Where `name` is name of comment block, when you start comment after `>>name` you can type anything before again typing name of the comment block on a new line. You can start your comment from the same line of comment initiation or from the next line but remember to close comment from new line only.

Second method is similar,

```
:’[comment]
```

```
[comment]’
```

Here syntax is fixed for `:’  ’` you can’t change that just type anything between the two single quotation marks.

Bash scripts can take **arguments** to be used inside by adding a space after the script execution call.

Each argument can be accessed via `$` notation. The first as `$1`, the second as `$2` and so on. `$@` and `$*` give all the arguments. `$#` gives the length (number) of arguments.

Consider the `args.sh` script below.

```
#!/usr/bash

echo $1

echo $2

echo $@

echo ‘There are ‘ $# ‘arguments.’
```

Now executing the script as:

```
$ bash args.sh Sia Katy Kate Mark Rob
```

Will return:

```
Sia
```

```
Katy
```

```
Sia Katy Kate Mark Rob
```

```
There are 5 arguments.
```

Assigning variables

Similar to other languages, you can assign variables with the equals notation.

```
var1="Moon"
```

Then reference with `$` notation for getting value of that variable.

`echo $var1` will return `Moon`. If you don't use the `$` notation, then it value won't be called.

Notice there is *no space* between variable name and its value when assigning.

Single, double, back ticks

In Bash, using different quotation marks can mean different things. Both when creating variables and printing.

- Single quotes (`'some text'`) = Shell interprets what is between literally.
- Double quotes (`"some text"`) = Shell interprets literally **except** using `$` and back ticks.

The last way creates a 'shell-within-a-shell', outlined below. Useful for calling command-line programs. This is done with back ticks.

- Back ticks (`` `some text` ``) = Shell runs the command and captures STDOUT back into a variable.

Shell within a shell

Example of shell within a shell is below,

```
rightnow=`date`
```

```
echo $rightnow
```

will return current date as `date` is a command which gives current date and time. It can also be used like:

```
rightnow=”hello today’s date is `date`.”
```

```
echo $rightnow
```

will return `hello today’s date is <current date>.`.

There is another way for shell within a shell. We can also use `$(date)` instead of `` `date` ``. Both will do the same thing. Notice there can be any command other than date command like:

```
var=$(cat sample.txt)
```

```
echo $var
```

will return the contents of sample.txt.

Assigning numeric variables

```
age=6
```

without quotes will assign 6 to variable name age. Here is an example of using numeric variables.

```
model1=87.65
```

```
model2=89.20
```

```
echo "The total score is $(echo "$model1 + $model2" | bc)"
```

```
echo "The average score is $(echo "($model1 + $model2) / 2" | bc)"
```

notice how we used shell within a shell method here.

Arrays in Bash

An array in bash is a normal numerical-indexed structure, which is pretty similar to lists in python. Arrays in bash can have both numeric and string elements together.

In python: `my_list = [1, 2, 3, 4]`

Creation of arrays can be done in two ways in Bash.

1. Declare without adding elements

   `declare -a my_array`

2. Declare and add elements at the same time

   `my_array=(1 2 3)`

   notice no spaces around equals sign!

   Notice no commas are used to separate values only spaces are used to separate values.

Array properties

- All array elements can be returned using array[@]. Though do note, Bash requires curly brackets around the array name when you want to access these properties.

  `my_array=(1 2 5 9)`

  `echo ${my_array[@]}`

  will return: `1 2 5 9`

  But if you do not pass anything to the array to return it will by default return the first element like so:

  `echo ${my_array}`

  will return: `1`

- The length of an array is accessed using `#array[@]`.

  `echo ${#my_array[@]}`

  will return: `4`

Manipulating array elements

Array elements are accessed using square brackets.

```
my_array=(15 20 300 42)
```

```
echo ${my_array[2]}
```

will return: `300` because it is like zero indexing in python.

Array elements can be set to different values using index notations.

```
my_array=(15 20 300 42 2 4 33 66)
```

```
my_array[0]=99 #just like in python
```

```
echo ${my_array[0]}
```

will return: `99`

Slicing can also be done on arrays like in python, using the notation `array[@]:N:M`. Here `N` is the starting index and `M` is how many elements to return.

```
my_array=(15 20 300 42 50)
```

```
echo ${my_array[@]:3:2}
```

will return: `42 50` (starting from 4th element up to two elements).

Appending to arrays

Elements can be appended to arrays using array+=(element).

For example:

```
my_array=(300 42 23 4 10)
```

```
my_array+=(99)
```

```
echo ${my_array[@]}
```

will return: `300 42 23 4 10 99`

Notice if you do not add parentheses around what you want to append it will be concatenated to the first element. Like so:

```
my_array=(300 42 23 2 50)
```

```
my_array+=99
```

`echo ${my_array[@]}`

will return: `30099 42 23 2 50`

Associative arrays

An associative array is similar to a normal array, but with key-value pairs, not numerical indexes. It is similar to python's dictionary. Note that this is only available in Bash 4 onwards, so check your version with `bass --version` in terminal.

In python: `my_dict = {‘city’: “Silvassa”, ‘population’: 140000}`

You can only create an associative array using the declare syntax (and uppercase `-A`).

You can either declare first, then add elements or do it all in one line.

To declare first, then add elements like this:

```
declare -A city_details # Declare first
```

```
city_details=([city_name]="New York" [population]=14000000) # Add elements
```

```
echo ${city_details[city_name]} # Index using key to return a value
```

will return: `New York`

and `echo ${city_details[@]}` will return: `New York 14000000`

for declaring and assigning all in one-line use:

```
declare -A city_details=([city_name]=”New York” [population]=1400000)
```

if you want to access only keys of an associative array use `!` like so:

```
echo ${!city_details[@]}
```

will return only all the keys.

Notice that if keys are strings and if you don't use quotes it will still work.

Do note that `@` and `*` can be used interchangeably, and both mean all.

IF Statement

A basic IF statement in Bash has the following syntax:

```
if [ condition ]; then

      # SOME CODE

else

      # SOME OTHER CODE

fi
```

Notice the space between the square brackets and conditional elements. Also notice the semi-colon after close-bracket `];`.

There is also `elif` feature in IF statements. The syntax is quite similar:

```
if [ condition ]; then

      #SOME CODE

elif [ condition ]; then
```

```
    #SOME CODE
else
    #SOME CODE
fi
```

There can be many `elif` statements. An IF statement may also contain `if` and `elif` without `else` line but `fi` line is mandatory. Notice `else` line has no ; ` then`.

IF statements with strings

```
x="Queen"
if [ $x == "King" ]; then
    echo "$x is a King!"
else
    echo "$x is not a King!"
fi
```

the result will be: `Queen is not a King!`

You could also use `!=` for 'not equal to', similar to that of python.

Arithmetic IF statement

Arithmetic IF statement can use the double-parenthesis syntax:

```
x=10
if (($x > 5)); then
    echo "$x is more than 5!"
fi
```

the output will be: `10 is more than 5!`

Arithmetic IF statements can also use square brackets and an arithmetic flag rather than (`>`, `<`, `=`, `!=` etc.)

The arithmetic flags are:

- `-eq` for 'equal to'
- `-ne` for 'not equal to'
- `-lt` for 'less than'
- `-le` for 'less than or equal to'
- `-gt` for 'greater than'
- `-ge` for 'greater than or equal to'

Using the arithmetic flags, we can create the above example like so:

```
x=10

if [ $x -gt 5 ]; then

      echo “$x is more than 5!”

fi
```

the output will be same: `10 is more than 5!`

Other Bash conditional flags

Bash also comes with a variety of file-related flags such as:

- `-e` or `-a` if the file exists
- `-s` if the file exists and has size greater than zero
- `-r` if the file exists and is readable
- `-w` if the file exists and is writable

This can be use like:

```
if [ -a testfile.txt ]; then

      echo “the file exists”

else

      echo “file does not exist”

fi
```

to take input from user use: read varialble name

IF with command-line programs.

You can also use many command-line programs directly in the conditional, removing the square brackets.

For example, if the file `words.txt` has ‘hello’ inside:

```
if grep -q hello words.txt; then # -q flag is used for suppressing
output if not used it will also return the line which contains the
text.

      echo “hello is inside”

fi
```

IF with shell-within-a-shell

The above example can also be used like:

```
if $(grep -q hello words.txt); then

      echo “hello is insede”
```

```
fi
```

AND and OR notation in Bash

The following symbols are used in Bash for AND and OR:

- `&&` for AND
- `||` for OR

For example:

```
x=10

if [ $x -gt 5 ] && [ $x -lt 11 ]; then

      echo “$x is more than 5 and less than 11”

fi
```

However this method is not working in Git Bash, but below method is working.

Or we can also use double-square-bracket notation:

```
x=10

if [[ $x -gt 5 && $x -lt 11 ]]; then

      echo “$x is more than 5 and less than 11”

fi
```

similarly `||` can be used for OR.

FOR Loops

The basic structure in Bash is similar to that of Python:

```
for x in 1 2 3
```

```
do
	echo $x
done
```

will give output:

```
1
2
3
```

While Python has ‘**range**’ Bash has something called ‘**brace expansion**’.

Syntax: `{START..STOP..INCREMENT}`

```
for x in {1..5..2}
do
	echo $x
done
```

will print:

```
1
3
5
```

It’s not like you can only print value of x with it, you could also use any text like * to print.

Another way to use range in Bash is the ‘**three expressions**’ syntax.

```
for ((x=2;x<=4;x+=2))
do
	echo $x
done
```

will print:

```
2
4
```

In the above example the first part (`x=2`) is START value, the second part (`x<=4`) is STOP value and the last part (`x+=2`) is INCREMENT value. Notice in the STOP value there must be `<`, if not it will result in infinite loop.

Glob expansions

Bash also allows pattern-matching expansions into a for loop using the `*` symbol such as files in a directory. For example:

```
for book in books/* #books is a directory with some files in it
do
    echo $book
done
```

will list all the files which are in books directory.

Shell-within-a-shell with FOR loop

You could loop through the result of a call to shell-within-a-shell:

```
for book in $(ls books/ | grep ‘air’)
do
    echo $book
done
```

will list all items in books directory which has ‘air’ in its file name.

WHILE loop

The WHILE loop has following syntax:

- The word while is used instead of for
- Surround the condition in square brackets

- Same flags can be used like in IF statements
- Multiple conditions can be chained similar to IF statements, also double brackets syntax of IF can be used

For example:

```
x=1
while [ $x -le 3 ];
do
	echo $x
	((x+=1))
done
```

CASE statements

Steps to build a CASE statements:

- Begin by selecting which variable or string to match against. You could also call shell-within-a-shell here.

- Add as many possible matches & actions as you like. You can use regex for the PATTERN. Such as Air* for start with Air, etc.
- Ensure to separate the pattern and code to run by a close-parenthesis and finish commands with double semi-colon.
- Include *) DEFAULT COMMAND;;, it is common but not required to finish with a default command that runs if none of the other patterns match.
- End with esac, which is backword spelled ‘case’.

Basic CASE statement format:

```
case ‘STRINGVAR’ in #STRINGVAR can be a string or a variable

      PATTERN1)

      COMMAND1 ;;

      PATTERN2)

      COMMAND2 ;;

      *)

      DEFAULT COMMAND;;

esac
```

imagine a scenario in which you have to create a single script to,

- Move the file into /sydney directory if the file contains sydney in it
- Delete the file if the file contains Melbourne or Brisbane in it
- Rename the file to IMPORTANT_filename if the file contains canaberra in it where filename is the original filename.

A complex IF statement for the above problem is:

```
if grep -q ‘sydney’ $1; then

      mv $1 syndey/

fi

if grep -q ‘melbourne|brisbane’ $1; then

      rm $1

fi

if grep -q ‘canberra’ $1; then

      mv $1 “IMPORTANT_$1”

fi
```

the case statement for the above problem:

```
case $(cat $1) in
      *sydney*)
      mv $1 sydney/ ;;
      *melbourne*|*brisbane*)
      rm $1 ;;
      *Canberra*)
      mv $1 “IMPORTANT_$1” ;;
      *)
      echo “No cities found” ;;
esac
```

Functions in Bash

Functions in Bash has the following syntax:

- Start by naming the function, pick a sensible name.
- Add open and close parenthesis after the function name with space between them.
- Add the actual code inside the curly brackets.

Like so:

```
function_name () {
      #function_code
      return #something
}
```

Alternate Bash function structure

```
function function_name {
      #function_code
      return #something
}
```

Calling a Bash function

You can call the function in script just by writing its name.

Passing arguments in Bash functions

Passing arguments in functions is pretty similar to passing arguments in a script.

$1, $2, $n means the nth argument.

$@ and $* means all the arguments.

$# gives the length (number) of arguments.

Here in this example we pass some arguments to the function call in the script:

```
function print_filename {
      echo “The first file was $1”
      for file in $@
      do
            echo “this file has name $file”
      done
}
print_filename “lort.txt” ”mod.txt” “a.py”
```

you can also pass arguments to bash function through the script, like so:

```
function return_percentage {
      percent=$(echo “scale=2; 100 * $1 / $2” | bc)
      echo $percent # this echo is used as return statement
}
# below line is a variable which takes the value of above return.
return_test=$(return_percentage 456 632)
# the above line is calling the function with arguments
echo “percent is $return_test”
```

the above code is using shell-within-a-shell to call the function with arguments.

Scope

Scope refers to how accessible a variable is.

- **Global** means something is accessible anywhere in the program, including inside loops, IF statements, functions, etc.
- **Local** means something is only accessible in a certain part of the program.

All variables in Bash are global by default.

But if you want to restrict scope use local keyword. Like so:

```
function print_filename {
      local first_filename=$1
}
print_filename “LOTR.txt” “model.txt”
echo $first_filename
```

Scheduling scripts

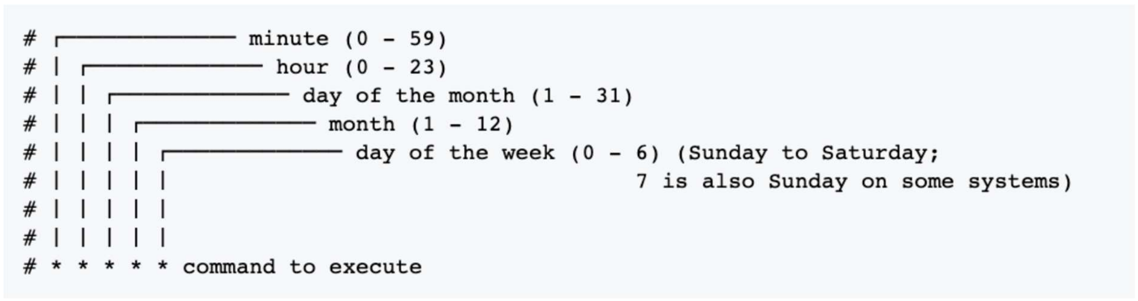
We schedule scripts with cron. The name comes from the Greek word for time, *chronos*.

It is driven by something called a crontab, which is a file that contains cronjobs, which each tell crontab what code to run and when.

To see ehat schedules (cronjobs) are currently programmed using command:

```
crontab -l
```

the below image demonstrates how you construct a cronjob inside the crontab file.



```
# ┌───────────── minute (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of the month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                   7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * command to execute
```

- There are 5 stars to set, one for each time unit.

- The default, `*` means ‘every’.

For example:

- `5 1 * * * bash myscript.sh`

  Minutes star is 5 (5 minutes past the hour). Hour star is 1 (after 1 AM). The last three are *, so every day and month.

  Overall: run every day at 1:05 AM.

- `15 14 * * 7 bash myscript.sh`

  Minutes star is 15 (15 minutes past the hour). Hours star is 14 (after 2 PM). Next two are * (every day of month, every month of year). Last star is 7 (on Sundays).

  Overall: run at 2:15 PM every Sunday.

You can also do some advance things like running a script multiple times of a day or every ‘X’ time increments. For example:

`15,30,45 * * * *` will run at the 15, 30 and 45 minutes’ mark for whatever hours are specified by the second star.

`*/15 * * * *` runs every 15 minutes and every hour, day, etc.

To create a `cronjob`, in terminal type `crontab -e` to edit your list of `cronjobs`. Maybe it will open a editor in which you will write your `cronjob` like: `30 1 * * * myscript.sh`

Now to check if the `crontab` is created use `crontab -l` to list your `cronjobs`.