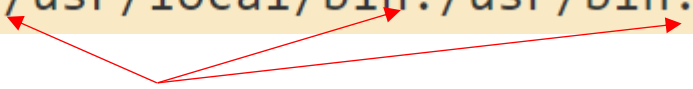
# What is a Bash Script?

I. Introduction:
- **Bash scripts** tell the **Bash shell** what it should do
- A plain **text** containing a **series of commands**
- Extension: **.sh**
- Should be given **execute permission** (by default, this permission is not set)
- Should always include the **Shebang** (#!/bin/bash) on the **first line**
- To execute the file: ./<filename>

II. Why "./" is needed?
- Normally, when we just type the **name** (like the name of the command) on the command line, Bash will try to find it in a **series of dirs** stored in a var called **$PATH**

```
user@bash: echo $PATH
/home/ryan/bin:/usr/local/bin:/usr/bin:/bin
```



- The **dirs** are **separated** by ":"
- Bash will **look through** those dirs **in order** and execute the **first instance** of the program or script that it finds
- As we see above, the **first dir** is the **bin dir** in the **home dir**

⇨ Scripts should be put here

- If a program or script is **not in one of the dir in $PATH**, we can tell Bash **where it should look** to find it by including either an **absolute** or **relative path** in front of the program or script name
- The dot "." in "./" is actually a **reference** to the **current dir**

III. The Shebang (#!):
- Following the Shebang is the path to the **interpreter** (or program) that should be used to **run or interpret** the rest of the lines in the file
- There are many types of scripts and they each have their own interpreters (Here, for Bash scripts it will be the path to Bash)

❗ ***Formatting is important***. Some involve spaces and the presence or absence of a space can make a difference

# Variables

I. Introduction: too familiar, not gonna mention

II. How do they work:

- 2 actions can be performed on vars:
  + **Set** value
  + **Read** value: place "**$**" before its name

III. Command line arguments (CLAs):

- A command can be supplied with **arguments** to **alter its behaviour**
  E.g.: ls -l /etc. **-l** and **/etc** are CLAs to the command ls
- Some special variables:

**$0** - The name of the Bash script.

**$1 - $9** - The first 9 arguments to the Bash script. (As mentioned above.)

**$#** - How many arguments were passed to the Bash script.

**$@** - All the arguments supplied to the Bash script.

**$?** - The exit status of the most recently run process.

**$$** - The process ID of the current script.

**$USER** - The username of the user running the script.

**$HOSTNAME** - The hostname of the machine the script is running on.

**$SECONDS** - The number of seconds since the script was started.

**$RANDOM** - Returns a different random number each time is it referred to.

**$LINENO** - Returns the current line number in the Bash script.

- To see more var: use the command env

IV. Set variables:

- Syntax: **var=val** (no space)
- When >= 2 words to be set in a var, use **quotes**
  E.g.: var="Hello world"
- There is a difference between single quotes and double quotes:
  + **Single** quotes: treat **every character**
  + **Double** quotes: allow **substitution**

E.g.:

```
user@bash: myvar='Hello World'
user@bash: echo $myvar
Hello World
user@bash: newvar="More $myvar"
user@bash: echo $newvar
More Hello World
user@bash: newvar='More $myvar'
user@bash: echo $newvar
More $myvar
user@bash:
```

V. Command substitution:

- Allow to take **output of a command** and save it as the **val** of a var
- Syntax: **var=$(command)**

E.g.:

```
user@bash: ls
bin Documents Desktop ...
Downloads public_html ...
user@bash: myvar=$( ls )
user@bash: echo $myvar
bin Documents Desktop Downloads public_html ...
user@bash:
```

VI. Exporting vars:

- Vars are limited to the process they were created in, in other word, to their scopes
- Refer to: https://ryanstutorials.net/bash-scripting-tutorial/bash-variables.php#setting (last section)

# Input

- To ask user for input: use command ***read <var1> <var2> ...***
- Use with option *-p* : specify a **prompt** (a short passage to be printed before taking user input)
- Use with option *-s* : make the input **silent** (make it disappear once finish typing)

  E.g.:

```
#!/bin/bash
# Ask the user for login details

read -p 'Username: ' uservar
read -sp 'Password: ' passvar
echo
echo Thankyou $uservar we now have your login details
```

```
user@bash: ./login.sh
Username: ryan
Password:
Thankyou ryan we now have your login details
user@bash:
```

- If there are more items than var names, the remaining items will be added to the last var; if there are less, the remaining var will be set to blank or null
- Reading from STDIN: refer to https://ryanstutorials.net/bash-scripting-tutorial/bash-input.php (last section)

# Arithmetic

I. Let:

- A built-in function allowing to do arithmetic
- Syntax: ***let <arithmetic-expression>***

  E.g.:

```
#!/bin/bash
# Basic arithmetic using let

let a=5+4
echo $a # 9

let "a = 5 + 4"
echo $a # 9

let a++
echo $a # 10

let "a = 4 * 5"
echo $a # 20

let "a = $1 + 30"
echo $a # 30 + first command line argument
```

  Note that:

  + Without quotes: no spaces
  + With quotes: spaces are optional and special characters don't need to escape

- Some operations:

| Operator | Operation |
|---|---|
| +, -, *, / | addition, subtraction, multiply, divide |
| var++ | Increase the variable var by 1 |
| var-- | Decrease the variable var by 1 |
| % | Modulus (Return the remainder after division) |

II. Expr:

- A built-in function allowing to print the answer
- Syntax: ***expr item1 operator item2***

  E.g.:

```
#!/bin/bash
# Basic arithmetic using expr

expr 5 + 4   #9

expr "5 + 4"    #5 + 4

expr 5+4  #5+4

expr 5 \* $1   #60 (if $1=12)

expr 11 % 2   1

a=$( expr 10 - 3 )
echo $a # 7
```

Note that:

+ No quotes needed and no spaces
+ When enclosed by quotes or spaces included, it just prints the line out

III. Double parentheses:

- Use to save the output of a command

  E.g.:

```
#!/bin/bash
# Basic arithmetic using double parentheses

a=$(( 4 + 5 ))
echo $a # 9

a=$((3+5))
echo $a # 8

b=$(( a + 3 ))
echo $b # 11

b=$(( $a + 4 ))
echo $b # 12

(( b++ ))
echo $b # 13

(( b += 3 ))
echo $b # 16

a=$(( 4 * 5 ))
echo $a # 20
```

IV. Length of a var:

- To get the length of a var: use *${#var}*

E.g.:

```
#!/bin/bash
# Show the length of a variable.

a='Hello World'
echo ${#a} # 11

b=4953
echo ${#b} # 4
```

# If statements

I. Basic:

- Syntax:

```
if [ <some test> ]
then
    <commands>
fi
```

E.g.:

```
#!/bin/bash
# Basic if statement

if [ $1 -gt 100 ]
then
   echo Hey that\'s a large number.
   pwd
fi

date
```

- Some common tests to be put inside if []:

| Operator | Description |
|---|---|
| ! EXPRESSION | The EXPRESSION is false. |
| -n STRING | The length of STRING is greater than zero. |
| -z STRING | The lengh of STRING is zero (ie it is empty). |
| STRING1 = STRING2 | STRING1 is equal to STRING2 |
| STRING1 != STRING2 | STRING1 is not equal to STRING2 |
| INTEGER1 -eq INTEGER2 | INTEGER1 is numerically equal to INTEGER2 |
| INTEGER1 -gt INTEGER2 | INTEGER1 is numerically greater than INTEGER2 |
| INTEGER1 -lt INTEGER2 | INTEGER1 is numerically less than INTEGER2 |
| -d FILE | FILE exists and is a directory. |
| -e FILE | FILE exists. |
| -r FILE | FILE exists and the read permission is granted. |
| -s FILE | FILE exists and it's size is greater than zero (ie. it is not empty). |
| -w FILE | FILE exists and the write permission is granted. |
| -x FILE | FILE exists and the execute permission is granted. |

Note that the [] is just a reference to the command *test*, man test for more info

- Indenting is optional

II. Nested: nah

III. If else:

- Syntax:

```
if [ <some test> ]
then
    <commands>
else
    <other commands>
fi
```

IV. Elif:

- Syntax:

```
if [ <some test> ]
then
    <commands>
elif [ <some test> ]
then
    <different commands>
else
    <other commands>
fi
```

V. Case:

```
case <variable> in
<pattern 1>)
    <commands>
    ;;
<pattern 2>)
    <other commands>
    ;;
esac
```

# Loops

I. While:

- Syntax:

```
while [ <some test> ]
do
    <commands>
done
```

E.g.:

```
#!/bin/bash
# Basic while loop

counter=1
while [ $counter -le 10 ]
do
   echo $counter
   ((counter++))
done

echo All done
```

II. Until:

- Syntax:

```
until [ <some test> ]
do
    <commands>
done
```

E.g.:

```
#!/bin/bash
# Basic until loop

counter=1
until [ $counter -gt 10 ]
do
   echo $counter
   ((counter++))
done

echo All done
```

III. For:

- Syntax:

```
for var in <list>
do
   <commands>
done
```

=> Take each item in the list, assign that item as the val of the var (Note: a list is a series of strings, seperated by spaces)

E.g:

```
#!/bin/bash
# Basic for loop

names='Stan Kyle Cartman'

for name in $names
do
   echo $name
done

echo All done
```

- To process a series of numbers, use **range**
  E.g.:

```
#!/bin/bash
# Basic range in for loop

for value in {1..5}
do
   echo $value
done

echo All done
```

  {starting..ending} => it can be increasing or decreasing
- It is possible to specify a value to increase/decrease by each time. E.g.: {10..0..2} means decreasing from 10 to 0 and each step decrease by 2
- Break and continue: nah

IV. Select:
- To create a simple menu system
- Syntax:

```
select var in <list>
do
   <commands>
done
```

- When invoked, it take all the items and present them on the screen with a number before each item. A prompt will be printed after this to allow user to select a number. After finishing selecting and hit enter, the corresponding item will be assigned to var and the commands are run. Once finished, a prompt will be displayed again to select another option
- To change the prompt, change the system var PS3

# Functions

I. Creating functions:

- Syntax:

```
function_name () {
    <commands>
}
```

and

```
function function_name {
    <commands>
}
```

Note that:

+ Unlike other programming languages, they have no argument passed to

E.g.:

```
#!/bin/bash
# Basic function

print_something () {
   echo Hello I am a function
}

print_something
print_something
```

II. Passing arguments:

- Use CLAs

```
#!/bin/bash
# Passing arguments to a function

print_something () {
   echo Hello $1
}

print_something Mars
print_something Jupiter
```

III. Return values:

- Bash functions don't allow us to do this
- However, they allow us to set a return status using keyword *return*
- Note this stupid: return status or exit status of 0 means success, other than 0 is error

```
#!/bin/bash
# Setting a return status for a function

print_something () {
   echo Hello $1
   return 5
}

print_something Mars
print_something Jupiter
echo The previous function has a return value of $?
```

- We can also use Command Substitution:

```
#!/bin/bash
# Setting a return value to a function

lines_in_file () {
   cat $1 | wc -l
}

num_lines=$( lines_in_file $1 )

echo The file $1 has $num_lines lines in it.
```

```
user@bash: cat myfile.txt
Tomato
Lettuce
Capsicum
user@bash: ./return_hack.sh myfile.txt
The file myfile.txt has 3 lines in it.
user@bash:
```

IV. Var scope:

- By default, a var is global
- To create a local var, use the syntax:

```
local var_name=<var_value>
```

```
#!/bin/bash
# Experimenting with variable scope

var_change () {
   local var1='local 1'
   echo Inside function: var1 is $var1 : var2 is $var2
   var1='changed again'
   var2='2 changed again'
}

var1='global 1'
var2='global 2'

echo Before function call: var1 is $var1 : var2 is $var2

var_change

echo After function call: var1 is $var1 : var2 is $var2
```

```
user@bash: ./local_variables.sh
Before function call: var1 is global 1 : var2 is global 2
Inside function: var1 is local 1 : var2 is global 2
After function call: var1 is global 1 : var2 is 2 changed again
```

Note: always use local var within functions

V. Overriding commands: refer to [https://ryanstutorials.net/bash-scripting-tutorial/bash-functions.php](https://ryanstutorials.net/bash-scripting-tutorial/bash-functions.php) (last section)