CDOT
SICT AR Meeting Area
People

get involved with CDOT
as a Student
as an Open Source Community Member
as a Company

courses
BTC640
BTH740
BTP300
DPI908
DPS901
DPS905
DPS909
DPS911
DPS914
DPS915
DPS924
DPS931
EAC234
ECL500
GAM531
GAM666
GAM670
GPU610
LUX Program
MAP524
OOP344
OPS235
OPS245
OPS335
OPS345
OPS435
OPS445
OPS535
OPS635
OSD600
OSD700
OSL640
OSL740
OSL840
Real World Mozilla
RHT524
SBR600

Tutorial11: Shell Scripting - Part 1

Contents [hide]

INTRODUCTION TO SHELL SCRIPTING

Main Objectives of this Practice Tutorial

- Plan and create a Shell Script
- Explain the purpose of the **she-bang line** contained at the top of a shell script.
- Set **permissions** and **execute** shell scripts.
- Use **environment** and **user-defined** variables in shell scripts.
- Use **Command Substitution** and **Math Operations** in shell scripts
- Explain the purpose of the **$?** exit status and the **test** command.
- Use **if** and **if-else** logic statements in shell scripts.
- Use a **for** loop statement with a list in shell scripts.

Tutorial Reference Material

Course Notes	Linux Command/Shortcut Reference		YouTube Videos
Slides: • Week 11 Lecture 1 Notes: PDF | PPTX	**Shell Scripting:** • Purpose **Variables:**	**Control Flow Statements:** • Purpose • test	**Instructional Videos:** • Bash Shell Scripting - Part 1 • Creating and Running

SEC520
SPO600
SRT210
ULI101

course projects

Course Project List
Potential Course Projects
Contrib Opportunities

links

CDOT
Planet CDOT
FSOSS

Tools

What links here
Related changes
Special pages
Printable version
Permanent link
Page information

Week 11 Lecture 2 Notes:

- PDF] | PPTX

Environment
- User Defined
- Positional Parameters

Commands / Techniques:

- read
- readonly
- Command Substitution

command
- if statement
- if-else statement
- for loop

a Shell Script

KEY CONCEPTS

A shell script is a **file** that contains **Unix/Linux commands** and **reserved words** to help **automatic** common tasks.

Creating & Executing Shell Scripts

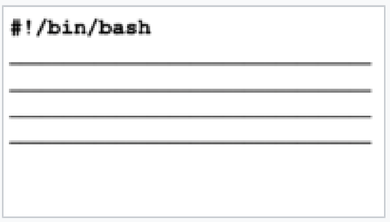
It is recommended to **plan** out on a piece of paper the purpose of the shell script.
Once you have planned your shell script by listing the **sequence of steps (commands)**,
you need to create a file (using a **text editor**) that will contain your Linux commands.

NOTE: Avoid using filenames of already existing Linux Commands to avoid confusion.
It is recommended to include a file extension that describes the type of shell for the shell script.

Using a Shebang Line

Since Linux shells have evolved over a period of time, using a she-bang line **forces** the shell script to run in a **specific shell**, which could prevent errors in case an older shell does not recognize newer features from more recent shells.

The **she-bang** line is a **special comment** at top of your shell script to run your shell script in a specific shell.



```
#!/bin/bash
```

The **shebang line** must appear on the **first line** and at the **beginning** of the shell script.

NOTE: The **shebang line** must appear on the **first line** and at the **beginning** of the shell script, otherwise, it will be treated as a regular comment and ignored.

Setting Permissions / Running Shell Scripts

To run your shell script by name, you need to assign **execute permissions** for the user.
To run the shell script, you can **execute** the shell script using a *relative*, *absolute*, or *relative-to-home* pathname

Examples:

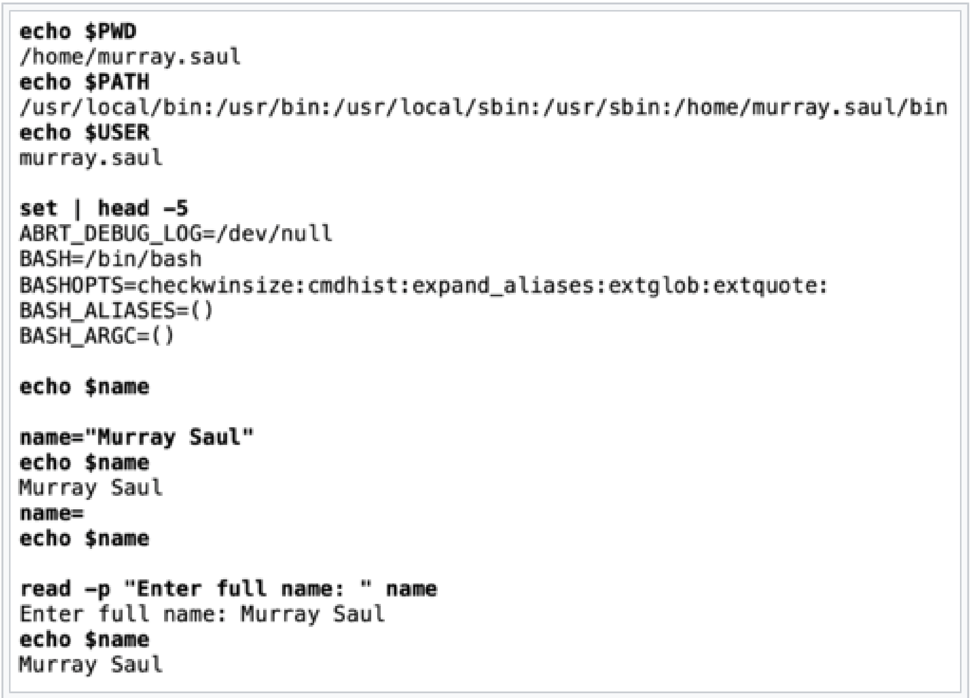
```
chmod u+x myscript.bash
./myscript.bash
```

```
/home/username/myscript.bash
~/myscript.bash
```

Variables / Parameters

Environment Variables

Shell **environment variables** shape the working environment whenever you are logged in Common shell. Some of these variables are displayed via Linux commands in the diagram displayed on the right-side.



```
echo $PWD
/home/murray.saul
echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/murray.saul/bin
echo $USER
murray.saul

set | head -5
ABRT_DEBUG_LOG=/dev/null
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:
BASH_ALIASES=()
BASH_ARGC=()

echo $name

name="Murray Saul"
echo $name
Murray Saul
name=
echo $name

read -p "Enter full name: " name
Enter full name: Murray Saul
echo $name
Murray Saul
```

Examples of using **Environment** and **User Defined** variables.

You can issue the pipeline command `set | more` to view all variables.

Placing a dollar sign **"$"** prior to the variable name will cause the variable to expand to the value contained in the variable.

User Defined Variables

User-defined variables are variables that can be used in the shell script for **customized** purposes.

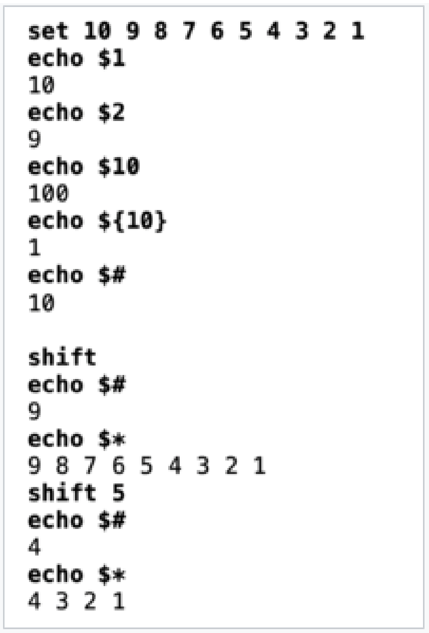
Data can be stored and removed within a variable using an **equal sign** (no spaces on either side of equal sign).

The **read** command can be used to prompt the user to enter data into a variable. The **readonly** command will prevent
the current value of the variable for the remainder of the execution of a shell script.

Positional Parameters and Special Parameters

A **positional parameter** is a variable within a shell program; its value is set from arguments contained in a shell script or using the **set** command.

Positional parameters are numbered and their values are



```
set 10 9 8 7 6 5 4 3 2 1
echo $1
10
echo $2
9
echo $10
100
echo ${10}
1
echo $#
10

shift
echo $#
9
echo $*
9 8 7 6 5 4 3 2 1
shift 5
echo $#
4
echo $*
4 3 2 1
```

Examples of using **positional** and **special** parameters.

accessed by using
a preceding "$" (eg. **$1**, **$2**, **$3**, etc.). The positional
parameter **$0** refers to
either the **name of shell** where command was issued, or
filename of shell script being executed.
If using **positional parameters** greater than **9**, then you
need to include number within braces.

Examples: **echo ${10}**, **ls ${23}**

The **shift** command can be used with positional
parameters to shift positional parameters
to the left by one or more positions.

There are a couple of ways to assign values as positional
parameters:

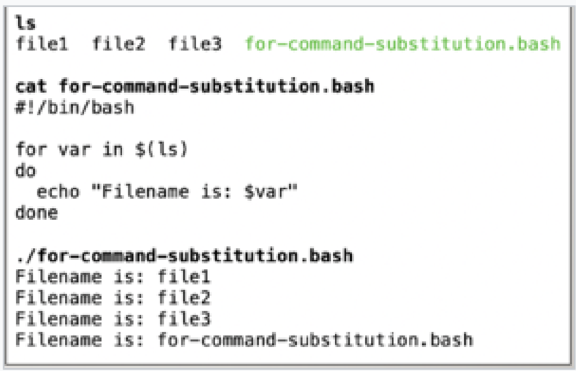
- Use the **set** command with the values as argument after the set command
- Run a shell script containing arguments

There are a group of **special parameters** that can be used for shell scripting.
A few of these special parameters and their purpose are displayed below:
$* , “$*” , "$@" , $# , $?

Command Substitution / Math Operations

Command Substitution:

***Command substitution** is a facility that
allows a command
to be run and its output to be pasted back on
the command line as arguments to another
command.* Reference:



```
ls
file1  file2  file3  for-command-substitution.bash

cat for-command-substitution.bash
#!/bin/bash

for var in $(ls)
do
  echo "Filename is: $var"
done

./for-command-substitution.bash
Filename is: file1
Filename is: file2
Filename is: file3
Filename is: for-command-substitution.bash
```

Example of how a **for loop with command substitution** works.

https://en.wikipedia.org/wiki/Command_substitution

Usage:

`command1 $(command2)`
`or`

```
command1 `command2`
```

Examples:

```
file $(ls)
mail -s "message" $(cat email-list.txt) < message.txt
echo "The current directory is $(pwd)"
echo "The current hostname is $(hostname)"
echo "The date is: $(date +'%A %B %d, %Y')"
```

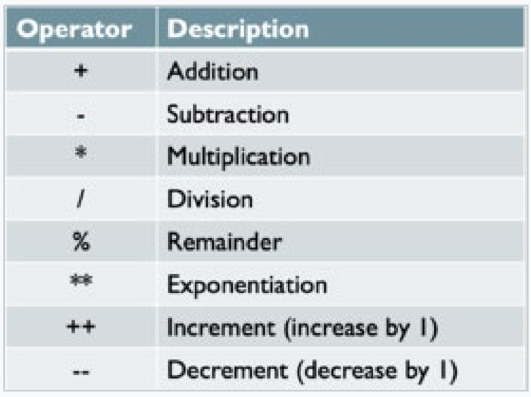
Math Operations:

In order to make math operations work, we need to convert numbers
stored as **text** into **binary numbers**.

We can do this by using 2 pairs of round brackets **(( ))**.

Examples:

```
num1=5;num2=10
echo "$(($num1 +
$num2))"
15
echo "$((num1-num2))"
-5
((product=num1*num2))
echo "$product"
50
```



Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
**	Exponentiation
++	Increment (increase by 1)
--	Decrement (decrease by 1)

Common Math Operator Symbols.

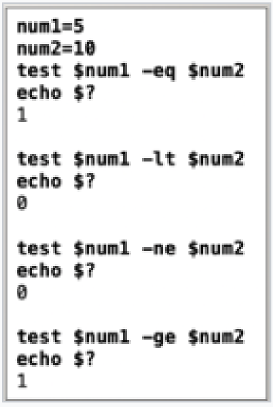
Control Flow Statements

Control Flow Statements are used to make your shell scripts
more **flexible** and can **adapt** to changing situations.

In order to use control flow statements, you need to test a condition to get **TRUE** (zero value) or **FALSE** (non zero value). This can be done two ways:

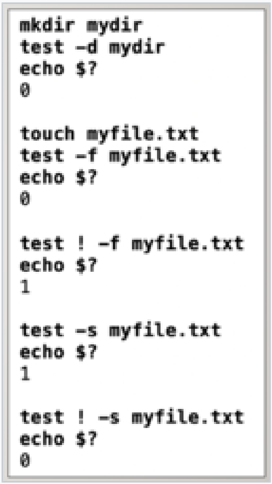
- Run a command to get the exit status **($?)**
- Use the **test command**

Refer to the diagrams on the right to see how to use the test command.



```
num1=5
num2=10
test $num1 -eq $num2
echo $?
1

test $num1 -lt $num2
echo $?
0

test $num1 -ne $num2
echo $?
0

test $num1 -ge $num2
echo $?
1
```

Examples of simple comparisons using the test command.



```
mkdir mydir
test -d mydir
echo $?
0

touch myfile.txt
test -f myfile.txt
echo $?
0

test ! -f myfile.txt
echo $?
1

test -s myfile.txt
echo $?
1

test ! -s myfile.txt
echo $?
0
```

Examples of using additional comparisons using the test command.

You CANNOT use the **<** or **>** symbols when using the test command since these are redirection symbols. Instead, you need to use **options** when performing numerical comparisons. Refer to the diagrams to the right **test options** and their purposes.

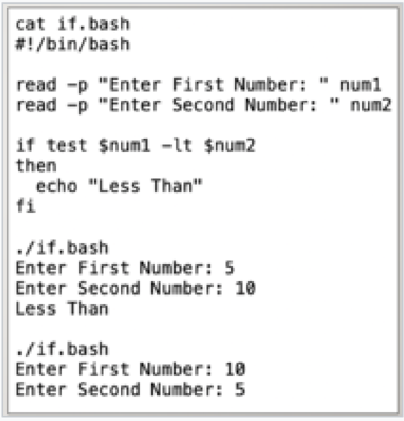
Logic Statements

A **logic statement** is used to determine which Linux commands are executed basedon the result of a condition:
TRUE (zero value) or **FALSE** (non-zero value).

There are several logic statements, but we will just concentrate on the if statement.

```
if test
condition
  then

command(s)
fi
```

Refer to the diagram to the right for using the **if logic statement** with the **test** command.



```
cat if.bash
#!/bin/bash

read -p "Enter First Number: " num1
read -p "Enter Second Number: " num2

if test $num1 -lt $num2
then
  echo "Less Than"
fi

./if.bash
Enter First Number: 5
Enter Second Number: 10
Less Than

./if.bash
Enter First Number: 10
Enter Second Number: 5
```

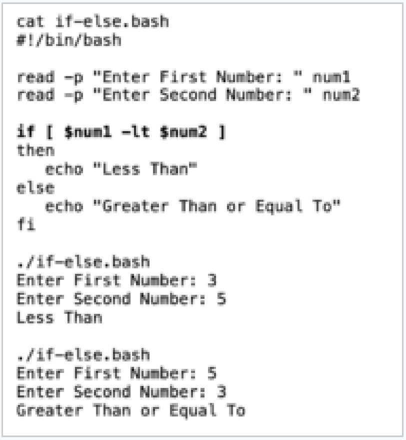
Example of using the **if** logic control-flow statement.

if-else statement:

Unlike using an *if* statement, an **if-else** statement take **two different sets of actions** based on the results of the test condition.

Example:

```
if test
condition
  then

command(s)
  else

command(s)
fi
```

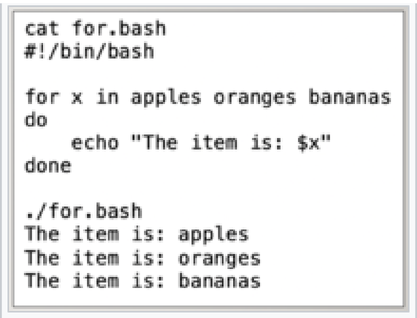


```
cat if-else.bash
#!/bin/bash

read -p "Enter First Number: " num1
read -p "Enter Second Number: " num2

if [ $num1 -lt $num2 ]
then
   echo "Less Than"
else
   echo "Greater Than or Equal To"
fi

./if-else.bash
Enter First Number: 3
Enter Second Number: 5
Less Than

./if-else.bash
Enter First Number: 5
Enter Second Number: 3
Greater Than or Equal To
```

Example of how an **if-else** control-flow statement.

Loop Statements

*A **loop statement** is a series of steps or sequence of statements executed repeatedly zero or more times satisfying the given condition is satisfied.*



```
cat for.bash
#!/bin/bash

for x in apples oranges bananas
do
    echo "The item is: $x"
done

./for.bash
The item is: apples
The item is: oranges
The item is: bananas
```

Example of using the **for** looping control-flow statement.

Reference: https://www.chegg.com/homework-help/definitions/loop-statement-3

There are several loops, but we will look at the **for loop** using a **list**.

```
for item in
list
do
     command(s)
done
```

Refer to the diagram above and to the extreme right side for an example using the **for loop** with a **list**.

INVESTIGATION 1: CREATING A SHELL SCRIPT

ATTENTION: This online tutorial will be required to be completed by **Friday in week 12 by midnight** to obtain a grade of **2%** towards this course

In this investigation, you will learn how to create and run a **Bash Shell script**.

Perform the Following Steps:

1. **Login** to your matrix account.

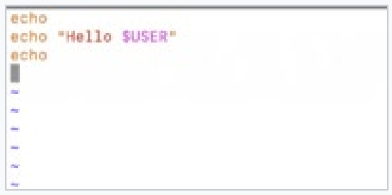
2. Issue a command to **confirm** you are located in your **home** directory.

   We want to create a Bash Shell script to welcome the user by their *username*. Let's first look at selecting an appropriate filename for your shell script.

3. Issue the following linux command to check if the filename called **hello** already exists as a command:
   which hello

   The output from this command should indicate that the shell did NOT find any directories that contained this filename that could represent a command; therefore, this shell script name CAN be used.

4. Use a **text editor** like **vi** or **nano** to create the text file called **hello**



```
echo
echo "Hello $USER"
echo
~
~
~
~
~
~
```

Using a **text editor** to add Linux commands in to the **hello** shell script.

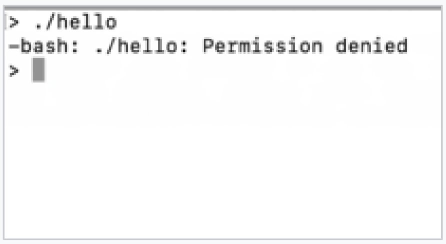
5. Enter the following two lines in your shell script:
   echo
   echo "Hello $USER"
   echo

NOTE: The variable called **USER** is an **environment variable** that contains the current user's login name. If you wanted to share your shell script with other users, when they run the shell script, they will greeted by their username. *Environment variables* make your shell script adaptable by ALL users.

6. **Save** your editing session and **exit** the text editor.

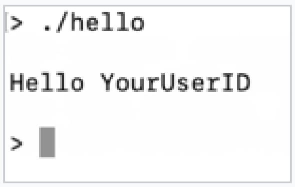
   Instead of issuing the **bash** command followed by your shell script pathname as an *argument*,
   let's simply run it by its filename. This is the most common method of running shell scripts.

7. Issue the following linux command to run your shell script in your current directory:
   ```
   ./hello
   ```
   You should notice an **ERROR message** indicating you don't have permissions to run the file. To fix this, you need to **add execute permissions** prior to running the shell script.



```
> ./hello
-bash: ./hello: Permission denied
> 
```

An **error message** will appear when trying to run a shell script by name that does NOT have **execute** permissions.



```
> ./hello

Hello YourUserID

> 
```

Output from running your **hello** shell script (YourUserID representing your username).

8. Issue the following linux command to **add** execute permissions for your shell script:
   ```
   chmod u+x hello
   ```

9. Issue the following to run your shell script:
   ```
   ./hello
   ```
   Hello twwong9

   Did your shell script run?

10. Issue the following Linux command to run a checking script:
    ```
    ~uli101/week11-check-1
    ```

11. If you encounter errors, make corrections and **re-run** the checking script until you receive a congratulations message, then you can proceed.

    In the next investigation, you will learn to create and run shell scripts that use **variables**, **positional** and **special parameters**. You will also learn how to add a **she-bang line** at the top of a shell script to force it to run in a specified shell.

    Proceed to the next investigation.

INVESTIGATION 2: SHE-BANG LINE / VARIABLES / PARAMETERS

In this investigation, you will add a **she-bang** line at the top of your shell script to force the shell script to run in a
specified shell when executed. You will also learn how to use **variables**, **positional** and **special parameters**
to make your shell scripts more adaptable.

Perform the Following Steps:

1. Confirm that you are located in your **home** directory in your Matrix account.

   Let's run shell scripts with and without a **she-bang** line at the top of your shell script to demonstrate why using a *she-bang* line should be included in a shell script to force the shell script to be run in a *specific* shell.

2. Use a text editor to **edit** the **hello** shell script that you created in the previous investigation.

3. Add the following line to the bottom of the file (**copy** and **paste** to prevent *errors*):
   ```
   echo "The current shell you are using is: $(ps -o cmd= -p $$|cut -d' ' -f1)"
   ```

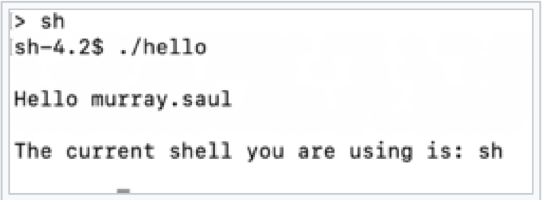
   FYI: This command displays the **name** of the *shell* that the shell script is running in. The command within `$( )` uses a technique known as **command substitution**.

4. Issue the following Linux command to change to an older shell called the **Bourne Shell**:
   ```
   sh
   ```

   You should notice your **shell prompt changed** which indicates that you are in a different shell.

5. Issue the following Linux command to run your shell script in the *Bourne Shell*:
   ```
   ./hello
   ```

   You should see that you are currently running the shell script "**sh**" which represents the **Bourne shell**.



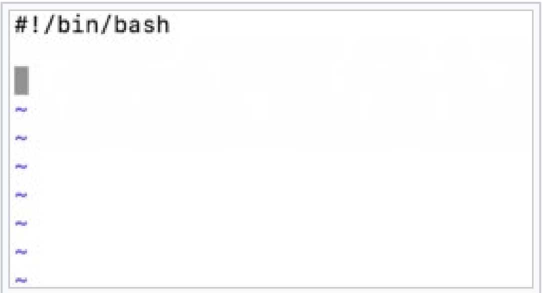
```
> sh
sh-4.2$ ./hello

Hello murray.saul

The current shell you are using is: sh
```

Changing the Bourne shell and running shell script **without** a **She-bang** line.

   NOTE: Due to the fact that shells (and their features) have **evolved** over a period of time, an error may occur if you include a **NEWER shell feature** (e.g. *Bash Shell*) but run it in an **OLDER shell** (For example: the *Bourne Shell*).

   You can add a **special comment** called a **she-bang line** at the BEGINNING of the FIRST line of your shell script to **force** it to run in the shell you want (for example: the Bash shell).

6. Edit your **hello** shell script using a text editor.



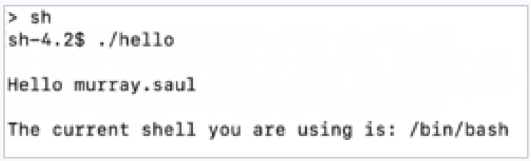
```
#!/bin/bash

~
~
~
~
~
~
~
```

Adding a **she-bang line** at the BEGINNING of the first line in you shell script forces the shell script to be run in that specific shell (in this case, the Bash shell).

7. **Insert** the following line at the **beginning** of the **first** line of your hello file:
   `#!/bin/bash`

   This is referred to as a **she-bang line**. It forces this script to be run in the **Bash Shell**. When your Bash Shell script finishes execution, you are returned to your current shell that you are using (which in our case in Matrix, is still the Bash shell).

8. **Save** your editing changes and **exit** your text editor.



```
> sh
sh-4.2$ ./hello

Hello murray.saul

The current shell you are using is: /bin/bash
```

Changing the Bourne shell and running shell script **with** a **She-bang** line (forcing script to run in the **Bash** shell).

9. While in the *Bourne shell*, issue the following Linux command:
   `./hello`

```
Hello twwong9

The current shell you are using is: /bin/bash
```

   You should notice that the shell name is running in the **Bash shell** (i.e. */bin/bash*).

   It is a good idea to rename your shell script to include an **extension** to indicate that it is a **Bash Shell** script.

10. Issue the following Linux command to rename your shell script file:
    `mv hello hello.bash`

11. Confirm that the renamed Bash shell script works by issuing:
    `./hello.bash`

12. Enter the following Linux command to **exit** the *Bourne shell* and return to your *Bash shell*:
    `exit`

    Environment variables are used to set the environment of the shell or shell scripts Let's include some **ENVIRONMENT variables** in our Bash Shell script.

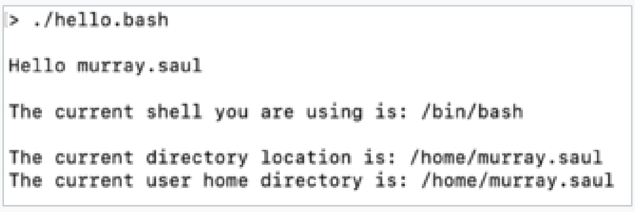
13. Use a text editor to edit the shell script called **hello.bash**

14. Add the following lines to the bottom of the *hello.bash* file:

```
echo
echo "The current directory location is: $PWD"
echo "The current user home directory is: $HOME"
echo
```

15. Save your editing changes and exit your text editor.

16. Run your modified Bash shell script by issuing:
    `./hello.bash`



```
> ./hello.bash

Hello murray.saul

The current shell you are using is: /bin/bash

The current directory location is: /home/murray.saul
The current user home directory is: /home/murray.saul
```

Running modified *hello.bash* Bash shell script by using

    Take time to view the output and the values of the environment variables.

relative pathname: `./hello.bash`

You can modify the PATH variable to include the current directory (i.e. ".") so you can run the command by just script filename (eg. **`hello.bash`** as opposed to **`./hello.bash`**)

Hello twwong9

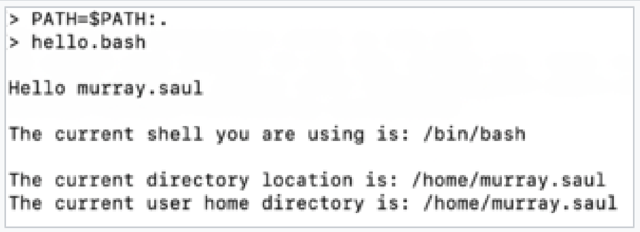
The current shell you are using is: /bin/bash

The current directory location is: /home/twwong9
The current user home directory is: /home/twwong9

17. Issue the following Linux command to add your current directory to the **PATH** environment variable:
```
PATH=$PATH:.
```

18. Issue the following Linux command to confirm that the current directory **"."** has been **added** to the end of the **PATH** environment variable:
```
echo $PATH
```

/home/uli101/bin:/usr/local/rvm/gems/ruby-3.0.2/bin:/usr/local/rvm/gems/ruby-3.0.2@global/bin:/usr/local/rvm/rubies/ruby-3.0.2/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/oracle/instantclient_12_2:/usr/local/rvm/bin::/home/twwong9/.local/bin:/home/twwong9/bin:.


```
> PATH=$PATH:.
> hello.bash

Hello murray.saul

The current shell you are using is: /bin/bash

The current directory location is: /home/murray.saul
The current user home directory is: /home/murray.saul
```
Running modified *hello.bash* Bash shell script by entering just **filename** (i.e. `hello.bash` and NOT `./hello.bash` shown in previous diagram).

19. Issue the following to run your Bash shell script just by name:
```
hello.bash
```

Did your Bash shell script run? Yes

20. Exit your Matrix session, and log back into your Matrix session.

21. Re-run the **hello.bash** shell script by just using the name.

What did you notice? Issue hello.bash can still run the script

The setting of the **PATH** environment variable only worked in the current session only.
If you exit the current Matrix session, then the recently changed settings for environment variables will be lost.
You will in a future tutorial how to set environment variables in **start-up** files.

22. Issue the following Linux command to run a checking script:
```
~uli101/week11-check-2 | more
```

If you encounter errors, make corrections and **re-run** the checking script until you receive a congratulations message, then you can proceed.

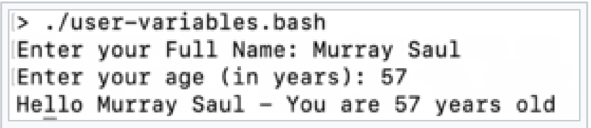
Unlike **Environment variables** that are used to set the environment of the shell or shell scripts,
User-created variables are "customized" that the user can set or allow a user to set the variables' values.
Let's create a Bash shell script that contain **user-created variables**.

23. Use a text editor to create a Bash shell script called **user-variables.bash**

24. Add the following lines to the beginning of the *user-variables.bash* file:
```
#!/bin/bash
read -p "Enter your Full Name: " name
read -p "Enter your age (in years): " age
echo "Hello $name - You are $age years old"
```

25. Save your editing changes and exit your text editor.

26. Issue the **chmod** command to add **execute permissions** for the **user-variables.bash** file.


```
> ./user-variables.bash
Enter your Full Name: Murray Saul
Enter your age (in years): 57
Hello Murray Saul - You are 57 years old
```
Prompting user to enter data via the **read -p** command storing into **user-created variable**.

27. Issue the following to run the user-variables.bash Bash shell script (enter **your Full name** and **your age** when prompted):
```
./user-variables.bash
```

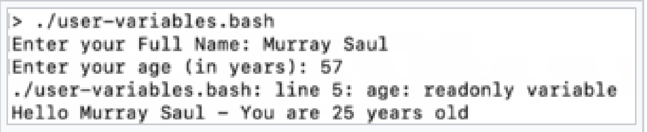
What did you notice?   **Hello TSZ WA WONG - You are 27 years old**

28. Use a text editor to **modify** your Bash shell script called **user-variables.bash**

29. **Insert** the following lines immediately **below** the **she-bang** line:
```
age=25
readonly age
```

30. Save your editing changes and exit your text editor.

31. Issue the following to run the user-variables.bash Bash shell script:
```
./user-variables.bash
```


```
> ./user-variables.bash
Enter your Full Name: Murray Saul
Enter your age (in years): 57
./user-variables.bash: line 5: age: readonly variable
Hello Murray Saul - You are 25 years old
```
Trying to change the value of a **read-only** variable.

What do you notice when you try to change the age variable? Why? **./user-variables.bash: line 5: age: readonly variable**
Hello TSZ WA WONG - You are 25 years old

A **positional parameter** is a special variable within a shell program; its value is set from **arguments** contained in a shell script or using the set command.
Let's use **positional parameters** and **special parameters** in a Bash shell script.

32. Use a text editor to create a file called **parameters.bash**

33. Add the following lines to the beginning of this file:
```
#!/bin/bash
echo \$0: $0
echo \$2: $2
echo \$3: $3

echo \$#: $#
echo \$*: $*
```

```
shift 2
echo \$#: $#
echo \$*: $*
```

34. Save your editing changes and exit your text editor.

    Notice how the quoting character "\" is used to display positional parameters like "**$2**" as opposed to the value stored in the second positional parameter.

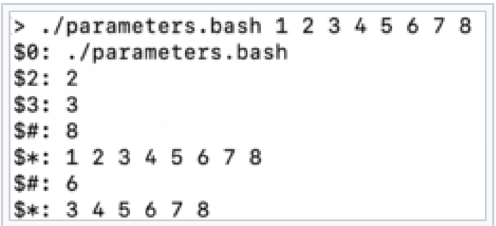
35. Issue the **chmod** command to add **execute permissions** for the user for the **parameters.bash** file.

36. Issue the following to run the **user-variables.bash** Bash shell script:
    `./parameters.bash`

```
$0: ./parameters.bash
$2:
$3:
$#: 0
$*:
$#: 0
$*:
```

    What happened?

    The values for some of the *positional parameters* and *special parameters* may NOT be
    displayed properly since you did NOT provide any **arguments** when running your Bash shell script.



```
> ./parameters.bash 1 2 3 4 5 6 7 8
$0: ./parameters.bash
$2: 2
$3: 3
$#: 8
$*: 1 2 3 4 5 6 7 8
$#: 6
$*: 3 4 5 6 7 8
```

Results from running shell script (with arguments) that use **positional parameters** and **special parameters**.

37. Issue the following to run the user-variables.bash Bash shell script with arguments:
    `./parameters.bash 1 2 3 4 5 6 7 8`

$0: ./parameters.bash
$2: 2
$3: 3
$#: 8
$*: 1 2 3 4 5 6 7 8
$#: 6
$*: 3 4 5 6 7 8

    What do you notice?

    Take some time to view the results and how the *parameters* have changed when using the **shift** command. **shift 2 make 3 to be the first argument**

In the next investigation, you will learn to use **command substitution** and **math operations** in your shell scripts.

INVESTIGATION 3: COMMAND SUBSTITUTION / MATH OPERATIONS

In this investigation, you will learn how to use **command substitution** and **math operations** in your shell scripts.

Command Substitution

Command Substitution is a method of running a Linux command that provides **stdout** that is used as **argument(s)** for another Linux command.

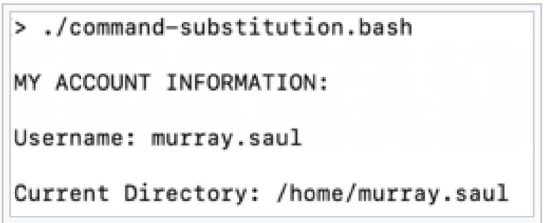
For example:

```
echo "The current date and time is: $(date)"
```

Let's create a Bash shell script that uses command substitution that displays **text** and values of **environment variables** in a series of `echo` statements.

Perform the Following Steps:

1. Confirm that you are located in your **home** directory in your Matrix account.
2. Use a text editor to create a Bash shell script called **command-substitution.bash**
3. Add the following lines to the beginning of this file:
```
#!/bin/bash
echo
echo "MY ACCOUNT INFORMATION:"
echo
echo "Username: $(whoami)"
echo
echo "Current Directory: $(pwd)"
echo
```
4. Save your editing changes and exit your text editor.
5. Issue the **chmod** command to add execute permissions for the **command-substitution.bash** file.
6. Issue the following to run the user-variables.bash Bash shell script:
```
./command-substitution.bash
```
Confirm that your shell script displays the correct information for your Matrix account.



```
> ./command-substitution.bash

MY ACCOUNT INFORMATION:

Username: murray.saul

Current Directory: /home/murray.saul
```

Output of a shell script using command substitution.

MY ACCOUNT INFORMATION:

Username: twwong9

Current Directory: /home/twwong9

Math Operations

Since you do NOT have to declare the **data-type** of a variable (as opposed to compiled program such as the C-programming language), numbers would be stored as **text** in variables. Therefore, it is important to use the construct `(( ))` to convert numbers (stored as *text*) into **numbers**.

We will now learn how to use this construct in order to perform math operations for shell scripts.

Perform the Following Steps:

1. Confirm that you are located in your **home** directory in your Matrix account.

   Let's demonstrate that the Unix/Linux shell stores numbers as ascii text which can cause problems when performing math operations.

2. Issue the following Linux command from the shell:

```
echo "1 + 2"
```

What did you notice? `1 + 2`

3. To demonstrate the need for the **(( ))** construct, issue the following Linux commands (using the *math construct*):

   ```
   echo "$((1 + 2))"
   ```

   ```
   echo "((1 + 2))"
   ((1 + 2))
   ```

   What did you notice? `3`

   The `(( ))` construct converted values **1** and **2** from *text* to **binary numbers**.
   The **$** in front of the construct **expands** the result of the calculation.

4. Issue the following Linux commands demonstrating other types of math calculations:

   ```
   echo "$((2 - 3))"     -1
   echo "$((2 * 3))"     6
   echo "$((2 / 3))"     0
   echo "$((2 ** 3))"    8
   ```

   NOTE: You may notice that **dividing 2** by **3** shows a **zero** result. To perform decimal calculations would require
   the use the **awk** or **bc** Linux commands (we will **NOT** cover that method to work with *decimal numbers* in this course).

   You can use the *math construct* with variables as well.

5. Issue the following Linux commands demonstrating using the *math construct* with **variables**:

   ```
   num1=34
   num2=12
   echo "$((num1 * num2))"
   ```

   What did you notice? **408**

   You can create variables and assign them values in the *math construct* as well.

6. Issue the following Linux commands demonstrating using the math construct with **variables**:

   ```
   num1=5
   num2=3
   ((result = num1 ** num2))
   echo "The result is: $result"      The result is: 125
   ```

7. Use a text editor to create a Bash shell script called **dog-years.bash**

8. Add the following lines to the beginning of this file:

   ```
   #!/bin/bash
   echo  echo: newline
   dogFactor=7
   read -p "Please enter your age (in years): " humanYears
   ```

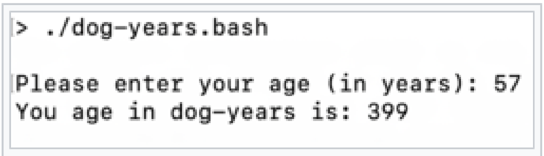
```
((dogYears = humanYears * dogFactor))
echo "You age in dog-years is: $dogYears"
echo echo: newline
```

9. Save your editing changes and exit your text editor.

10. Issue the **chmod** command to add execute permissions for the user for the **dog-years.bash** file.

11. Issue the following to run the **dog-years.bash** Bash shell script:
    `./dog-years.bash`

    Enter your age to see what happens. **You age in dog-years is: 189**



```
> ./dog-years.bash

Please enter your age (in years): 57
You age in dog-years is: 399
```

Output of a shell script with math operations using the **math construct**.

12. Issue the following to run a checking script:
    `~uli101/week11-check-3 | more`

    If you encounter errors, make corrections and **re-run** the checking script until you receive a congratulations message, then you can proceed.

In the next investigation, you will use **control-flow statements** to allow your shell scripts to perform differently under different situations.

INVESTIGATION 4: CONTROL FLOW STATEMENTS

In this investigation, you will learn how to use **control-flow statements** to make your shell script *behave differently* under *different situations or conditions*.

Perform the Following Steps:

1. Confirm that you are located in your **home** directory in your Matrix account.

2. Issue the following Linux commands at the Bash shell prompt to assign values to several variables:
    ```
    course="ULI101"
    number1=5
    number2=10
    ```

    You can test conditions by issuing **Linux commands / pipeline commands** or by using the **test** command. We will demonstrate using the **test** command in this tutorial, and then we will demonstrate how to test by issuing a *Linux command / pipeline command* in a later tutorial.

3. Issue the following Linux command to test a condition:
    `test $course = "ULI101"`

The **$?** variable is used to store an **exit status** of the previously-issued command (including the test command).
If the exit status is **zero**, then it indicates a *TRUE* value and if the status is **non-zero**, then it indicates a *FALSE* value.

4. Issue the following Linux command to view the **exit status** of the previously-issued **test** command:
   ```
   echo $?        0
   ```
   Based on the *exit status* value, is the result *TRUE* or *FALSE*? **TRUE**

5. Issue the following Linux command to test another condition:
   ```
   test $course = "uli101"
   ```

6. Issue the following Linux command to view the *exit status* of the previously-issued **test** command:
   ```
   echo $?    1
   ```
   Based on the *exit status* value, is the result TRUE or FALSE? **FALSE**
   The value is non-zero (FALSE) since UPPERCASE characters are different than lowercase characters.

7. Issue the following Linux command to test another condition:
   ```
   test $course != "uli101"
   ```

8. Issue a linux command to display the value of **$?** **0**

   What is the result? Why? **True, Because it is true that ULI101 is not equal to uli101**

9. Issue the following Linux command to test a condition involving earlier assigned variables:
   ```
   test $number1 > $number2
   ```
   accidentally create a file called 10

10. Issue a Linux command to display the value of **$?** **0**

    The result is supposed to be 1

    NOTE: You will notice that something is **wrong**.
    The exit status **$?** shows a zero (TRUE) value, but the number 5 is definitely NOT greater than 10.
    The problem is that the symbols **<** and **>** are interpreted as REDIRECTION symbols!

11. To prove this, issue the following Linux command :
    ```
    ls -l 10
    ```

    You should notice a file called **"10"**. The incorrectly issued **test** command **used redirection** to create an **empty** file and assigning the exit status variable a *TRUE* value!

    To prevent problems when issuing the **test** command when comparing numbers, you can use the following **test options**:
    -lt (<), **-le** (<=), **-gt** (>), **-ge** (>=;), **-eq** (=), **-ne** (!=)

12. Issue the correct Linux command to **properly** test both values:
    ```
    test $number1 -gt $number2
    ```

13. Issue a Linux command to display the value of **$?**.   1

    You should notice that the exit status value is now *FALSE* which is the correct result.

14. The **test** command can be substituted by **square brackets [ ]** which contains the **test** condition
    within the square brackets. You need to have spaces between the brackets and the test condition;
    otherwise, you will get a test error.

15. To generate a **test error**, copy and paste the following **test** command:
    ```
    [$number1 -gt $number2]
                              -bash: [5: command not found
    ```
    The reason for the error was that you need **spaces** between the **square brackets** and the **test condition**.

16. Copy and paste the following (correct) **test** command:
    ```
    [ $number1 -gt $number2 ]
    ```

17. Issue a command to view the value of the **exit status** of the previously issued **test** command.
    You should notice that is works properly.   1   FALSE

    Now that we have learned how to test conditions, let's learn about **control-flow** statements.

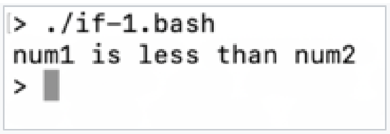
    LOGIC STATEMENTS are used to create **different paths** or directions that the shell script will take
    based on the result of the **test condition**. In this tutorial,we will only focus on the **if** and **if-else** logic statements.

18. Use a text editor like vi or nano to create the text file called **if-1.bash**
    (eg. `vi if-1.bash`)

19. Enter the following lines in your shell script:
    ```
    #!/bin/bash
    num1=5
    num2=10
    if [ $num1 -lt $num2 ]
    then
       echo "num1 is less than num2"
    fi
    ```

20. Save your editing session and exit the text editor
    (eg. with vi: press **ESC**, then type **:x** followed by **ENTER**).

21. Issue the following Linux command to add execute



```
> ./if-1.bash
num1 is less than num2
>
```

Output of a shell script using the **if** control-flow statement.

permissions for your shell script:
`chmod u+x if-1.bash`

22. Run your shell script by issuing:
`./if-1.bash`   num1 is less than num2

Confirm that the output indicates a correct result.

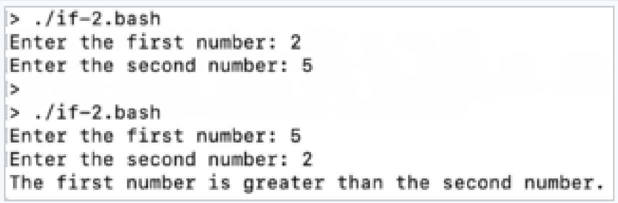
23. Use a text editor like vi or nano to create the text file called **if-2.bash**
(eg. `vi if-2.bash`)

24. Enter the following lines in your shell script:

```
#!/bin/bash
read -p "Enter the first number: " num1
read -p "Enter the second number: " num2
if [ $num1 -gt $num2 ]
then
   echo "The first number is greater than the second number."
fi
```

25. Save your editing session and exit the text editor
(eg. with vi: press **ESC**, then type **:x** followed by **ENTER**).

26. Issue the following Linux command to add execute permissions for your shell script:
`chmod u+x if-2.bash`



```
> ./if-2.bash
Enter the first number: 2
Enter the second number: 5
>
> ./if-2.bash
Enter the first number: 5
Enter the second number: 2
The first number is greater than the second number.
```

Output of a shell script using the **read** command and the **if** control-flow statement.

27. Run your shell script by issuing:
`./if-2.bash`

When prompted, make certain that the **first number** is greater than the **second number**. What happens?   **The first number is greater than the second number.**

28. Run the `./if-2.bash` Bash shell script again.

When prompted, make certain that the **first number** is less than or equal to the **second number**. What happens?   **nothing display**

Let's use an **if-else** statement to provide an **alternative** if the first number is less than or equal to the second number.

29. Use a text editor like vi or nano to create the text file called **if-3.bash**
(eg. `vi if-3.bash`)

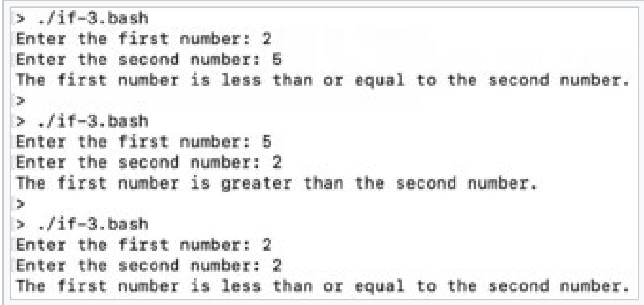
30. Enter the following lines in your shell script:

```
#!/bin/bash
read -p "Enter the first number: " num1
read -p "Enter the second number: " num2
```

```
if [ $num1 -gt $num2 ]
then
   echo "The first number is greater than the second number."
else
   echo "The first number is less than or equal to the second
number."
fi
```

31. Save your editing session and exit the text editor
(eg. with vi: press **ESC**, then type **:x** followed by **ENTER**).

32. Issue the following Linux command to add execute permissions for your shell script:
`chmod u+x if-3.bash`



```
> ./if-3.bash
Enter the first number: 2
Enter the second number: 5
The first number is less than or equal to the second number.
>
> ./if-3.bash
Enter the first number: 5
Enter the second number: 2
The first number is greater than the second number.
>
> ./if-3.bash
Enter the first number: 2
Enter the second number: 2
The first number is less than or equal to the second number.
```

Output of a shell script using the **if-else** control-flow statement.

33. Run your shell script by issuing:
`./if-3.bash`

Try running the script several times with numbers **different** and **equal** to each other to confirm that the shell script works correctly.

LOOP STATEMENTS are a series of steps or sequence of statements executed repeatedly zero or more times satisfying the given condition is satisfied.
Reference: *https://www.chegg.com/homework-help/definitions/loop-statement-3*

*There are several loops, but we will look at a **for** loop using a **list**.*

34. Use a text editor like vi or nano to create the text file called **for-1.bash**
(eg. `vi for-1.bash`)

35. Enter the following lines in your shell script:

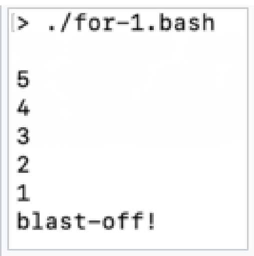
```
#!/bin/bash
echo
for x in 5 4 3 2 1
do
   echo $x
done
echo "blast-off!"
echo
```

36. Save your editing session and exit the text editor
(eg. with vi: press **ESC**, then type **:x** followed by **ENTER**).

37. Issue the following Linux command to add execute permissions for your shell script:
`chmod u+x for-1.bash`

```
5
4
3
2
1
blast-off!
```

38. Run your shell script by issuing:
    `./for-1.bash`



```
> ./for-1.bash

5
4
3
2
1
blast-off!
```

Output of a shell script using the **for** loop with a **list**.

39. Use a text editor like vi or nano to create the text file called **for-2.bash**
    (eg. `vi for-2.bash`)

40. Enter the following lines in your shell script:

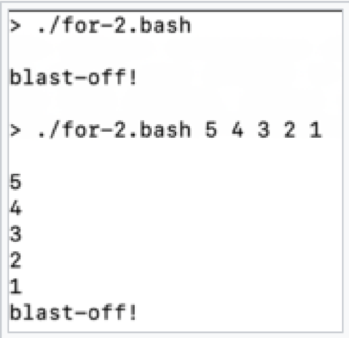
```
#!/bin/bash
echo
for x
do
   echo $x
done
echo "blast-off!"
echo
```

41. Save your editing session and exit the text editor
    (eg. with vi: press **ESC**, then type **:x** followed by **ENTER**).

42. Issue the following Linux command to add execute permissions for your shell script:
    `chmod u+x for-2.bash`

43. Run your shell script by issuing:
    `./for-2.bash 10 9 8 7 6 5 4 3 2 1`

```
10
9
8
7
6
5
4
3
2
1
blast-off!
```



```
> ./for-2.bash

blast-off!

> ./for-2.bash 5 4 3 2 1

5
4
3
2
1
blast-off!
```

Output of a shell script using the **for** loop without a **list**.

How does this differ from the previous shell script?
You will learn in a couple of weeks more examples of using loop statements.

Let's run a **checking-script** to confirm that both your **for-1.bash** and **for-2.bash**

44. Issue the following Linux command to run a checking script:
    `~uli101/week11-check-4 | more`

    If you encounter errors, make corrections and **re-run** the checking script until you receive a congratulations message, then you can proceed.

LINUX PRACTICE QUESTIONS

The purpose of this section is to obtain **extra practice** to help with **quizzes**, your **midterm**, and your **final exam**.

Here is a link to the MS Word Document of ALL of the questions displayed below but with extra room to answer on the document to simulate a quiz:

https://github.com/ULI101/labs/raw/main/uli101_week10_practice.docx

Your instructor may take-up these questions during class. It is up to the student to attend classes in order to obtain the answers to the following questions. Your instructor will NOT provide these answers in any other form (eg. e-mail, etc).

Review Questions:

PART A: WRITE BASH SHELL SCRIPT CODE

Write the answer to each question below the question in the space provided.

1. Write a Bash shell script that clears the screen and displays the text Hello World on the screen.

   What **permissions** are required to run this Bash shell script?
   What are the different methods that you can run this Bash shell script from the command line?

2. Write a Bash shell script that clears the screen, prompts the user for their **full name** and then prompts the user for their **age**, then clears the screen again and welcomes the user by their name and tells them their age.

   What **comments** would you add to the above script's contents to properly document this Bash shell script to be understood for those users that would read / edit this Bash shell script's contents?

3. Write a Bash shell script that will first set the value of a read-only variable called **dogFactor** to **7**. The script will then clear the screen and prompt the user to enter the age of a dog in human years (which will be stored into a variable called **humanYears**).

   The script will store in a variable called **dogYears** the value of *humanYears x dogFactor*
   The script will then clear the screen a second time and then display the age of the dog in *"dog years"*.

4. Write a Bash shell script that will clear the screen and then display all **arguments** that were entered after your Bash shell script when it was run. Also have the Bash shell script display the **number of arguments** that were entered after your Bash shell script.

PART B: WALK-THRUS

Write the expected output from running each of the following Bash shell scripts You can assume that these Bash shell script files have execute permissions. Show your work.

Walkthru #1:

cat walkthru1.bash

```
#!/usr/bin/bash
word1=”counter”
word2=”clockwise”
echo “The combined word is: $word2$word1”
```

WRITE ROUGH WORK AND OUTPUT FROM ISSUING:

./walkthru1.bash

ROUGH WORK:

OUTPUT:

Walkthru #2:

cat walkthru2.bash

```
#!/usr/bin/bash
echo “result1: $1”
echo “result2: $2”
echo “result3: $3”
echo “result 4:”
echo “$*”
```

WRITE ROUGH WORK AND OUTPUT FROM ISSUING:

./walkthru2.bash apple orange banana

ROUGH WORK:

OUTPUT:

Walkthru #3:

cat walkthru2.bash

```
#!/usr/bin/bash

for x in 1 2 3 4 5
do

  if [ $((x % 2)) -eq 0 ]
  then
    echo "this"
  else
    echo "that"
  fi

done
```

WRITE ROUGH WORK AND OUTPUT FROM ISSUING:

./walkthru3.bash apple orange banana

ROUGH WORK:

OUTPUT:

Author: Murray Saul

Category: ULI101