BASH scripting

1. What is Bash scripting?

A Bash script is a plain text file that contains many lines of Linux commands (e.g. echo, ls, cp) to be performed in a batch, as opposed to entering each command line individually in the Linux terminal. Bash scripting could be used to automate multiple or repetitive tasks on Linux. Bash scripts are written in the Bash programming language, which has its own syntaxes and structures, including loops, conditional constructions (if...else), and data containers, comparable to those of other programming languages.

2. Bash script execution

A Bash script file must be created and checked for the execution permission status before running.

Create bash script

For convenience, the name of script can follow this format.

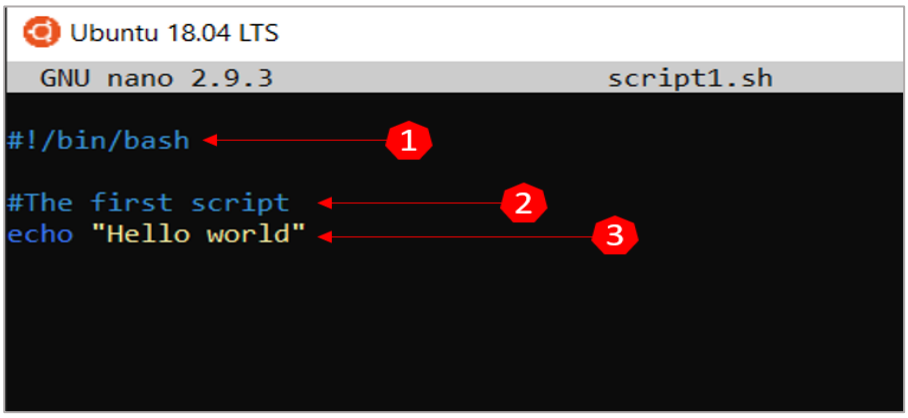
- Avoid adding spaces in the name, use underscore instead.
- Use alphanumerical [a-zA-Z0-9]
- File name has the extension ".sh"

Create and open file "script1.sh" for editing

```
nano script1.sh↵
```

*## The alternative way to create the empty file by using command **"touch"***

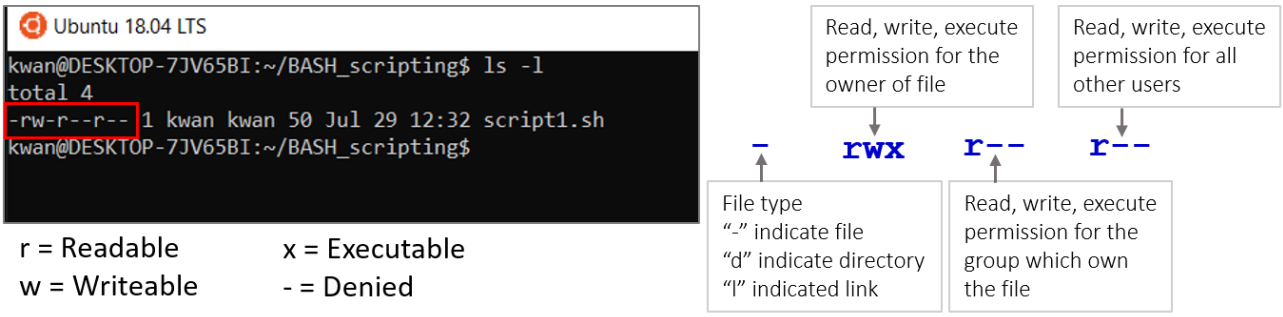
```
touch script1.sh↵
```



1. Shebang (#!) at the first line of script is used to instruct the OS to use bash as a command interpreter and specified the path of the interpreter.
2. The line starts with # will not be executed by interpreter. This line is referred to as a "comment" and is useful for describing the script.
3. Line of code. This code will print Hello world on the screen.

Set the execution permission

The execute permission of the bash script file can be checked by using “ls -l” command

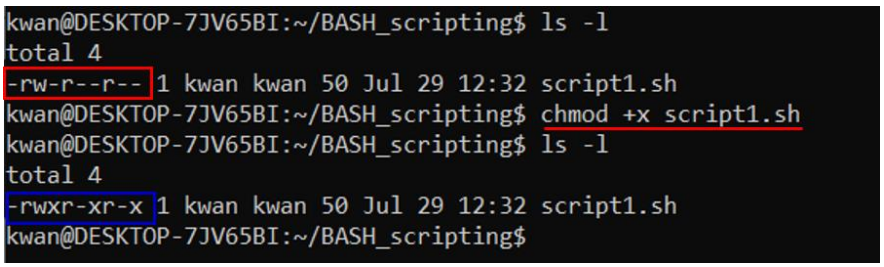


r = Readable x = Executable
w = Writeable - = Denied

The current status of the execute permission of `script1.sh` is “denied”. To change the execute permission, a command “`chmod`” , which is short for “change mode,” will be used.

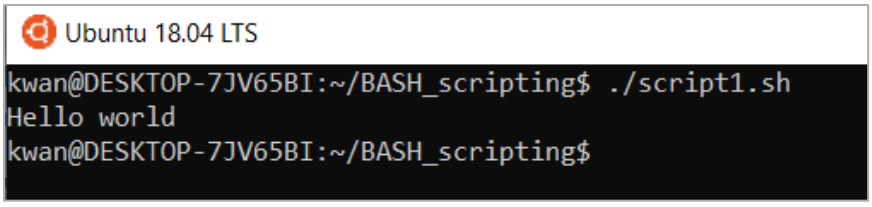
Make a script executable

```
chmod +x script1.sh↵
```



To run the script, just type “/path/to/file_script.sh”

```
./script1.sh↵
```



** *“./” is indicate that the file script is located here.*

3. Variables

Variables are important parts of programing. Variables store data to be use later in the script. Bash variables are untyped meaning the interpreter will define the data type automatically when assigning values to the variables. There are two types of bash variables in a shell or Linux system.

3.1 System-Defined Variables

These are the variables that are automatically assigned by LINUX operating system (i.e. built-in variables). They generally named in CAPITAL LETTER. An example list of System-Defined Variables is shown below.

Variables	Meaning	Example value
BASH	Return the bash path	/bin/bash
BASH_VERSION	Return the shell version	4.4.20(1)-release
HOME	Specifies the home directory	/home/kwan
PWD	Specifies the current working directory	/home/kwan/BASH_scripting
LOGNAME	Specifies the logging user name	kwan

3.2 User-Defined Variables

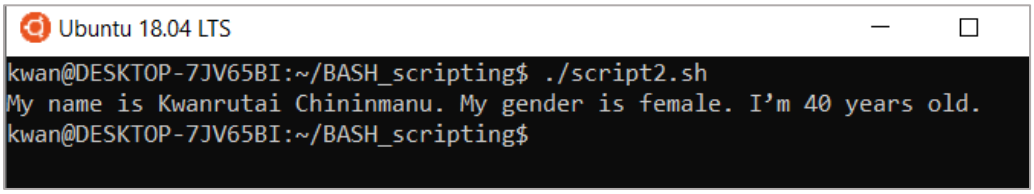
The variables created by user. This type of variables can be defined in either upper or lower case, but generally in lower cases. The rules for naming user-defined bash variables are as follows.

1) A variable name can include alphabets, digit, and underscore (_).
   a. Valid names:
      level, level1, _level, level_1
   b. Names cannot start with digit:
      1level, 1_level
2) The variable name might be in all CAPS, all lowercase, or a mixture of both..
3) The variable name is case-sensitive. For example, “Sequence” and “sequence” are considered as two separate variables.
4) The equal sign (=) is used for assigning a value to a variable. The variable is located on the left of equal sign while value is on the right. The whitespace **should not** be added on either side of qual sign.
5) When referring to a previously defined variable, the dollar sign ($) is prefixed to the variable's name.

Setting variables:

```
#!/bin/bash

name="Kwanrutai Chininmanu"
gender=female
age=40

echo "My name is $name. My gender is $gender. I'm
$age years old."
```

Output:



4. String manipulation

Bash scripting supports various string manipulations. This lecture will show the example of string operation Length, Substring, and Find and Replace.

4.1 String Length

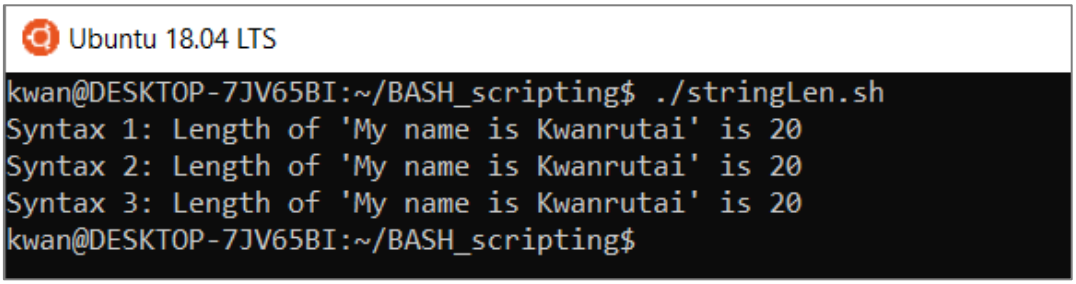
There are many ways to calculate the string length.

1) A simple way to calculate the length of the string is to use # symbol.
   Syntax:
   `${#string_variable_name}`
2) Calculate the length of the string using an "`expr`" command with an option "`length`".
   Syntax:
   `expr length "$string_variable_name"`
3) Use an "awk" command to calculate the length of the string
   Syntax:
   `echo $string_variable_name | awk '{print length}'`

Bash script: stringLen.sh

```
#!/bin/bash


str=”My name is Kwanrutai”
##Syntax 1
length1=${#str}
echo "Syntax 1: Length of '$str' is $length1"

##Syntax 2
length2=$(expr length "$str")
echo "Syntax 2: Length of '$str' is $length2"

##Syntax 3
length3=$(echo $str | awk '{print length}')
echo "Syntax 3: Length of '$str' is $length3"
```

Output:



4.2 Substring

Bash scripting provide an option to extract a substring from a string.

Syntax:

`${string:position:length}`

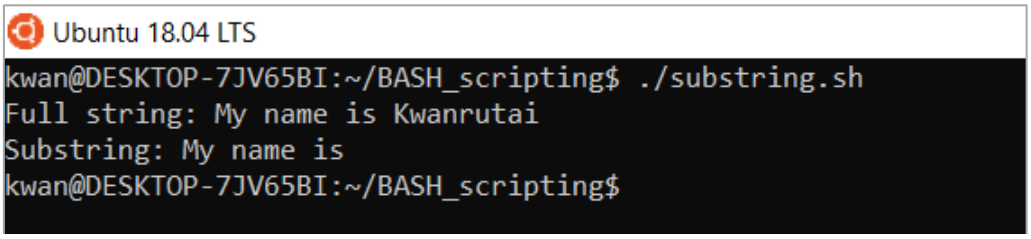
Extract Length characters of substring from String at Position.

Example 1: Extract substring from start until specific length

Extract first 10 characters of string (position = 0, length = 10)

```
#!/bin/bash

str=”My name is Kwanrutai”
substr=”${str:0:10}”

echo "Full string: $str"
echo "Substring: $substr"
```

Output:

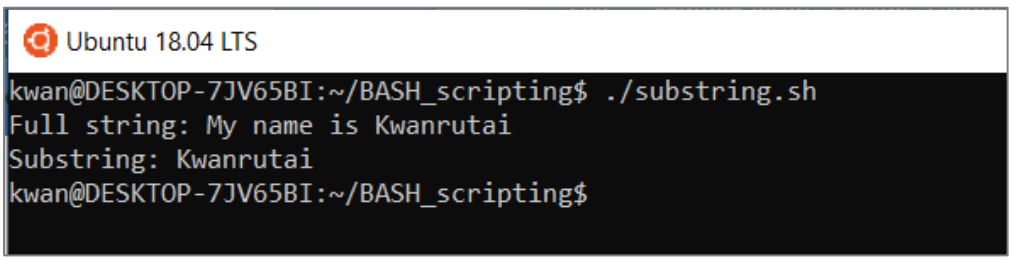


Example 2: Extract substring from specific character onwards

Extract substring 11th character onwards (position = 11, length = end of string)

```
#!/bin/bash

str=”My name is Kwanrutai”
substr=”${str:11}”

echo "Full string: $str"
echo "Substring: $substr"
```

Output:

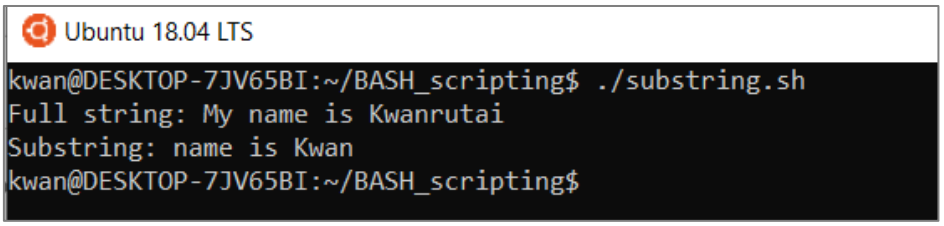


Example 3: Delete the first 3 characters and then print 12 subsequent characters

Extract substring at the middle of string (position = 3, length = 12)

```
#!/bin/bash

str="My name is Kwanrutai"
substr="${str:3:12}"

echo "Full string: $str"
echo "Substring: $substr"
```

Output:

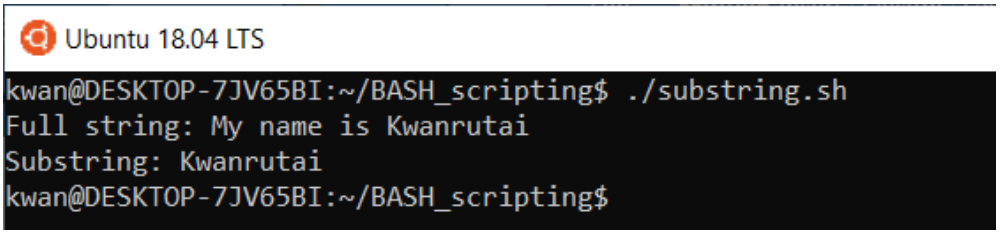


Example 4: Extract a specific number of characters counting from the end of the string

Extract last 9 character (position = -9, length = end of string)

```
#!/bin/bash

str="My name is Kwanrutai"
substr="${str:(-9)}"

echo "Full string: $str"
echo "Substring: $substr"
```

Output:



4.3 Shortest (non-greedy) substring match

The syntax for deleting the shortest match of the substring from the string

Syntax: Delete matched **substring** from the beginning of **string**

```
${string#substring}
```

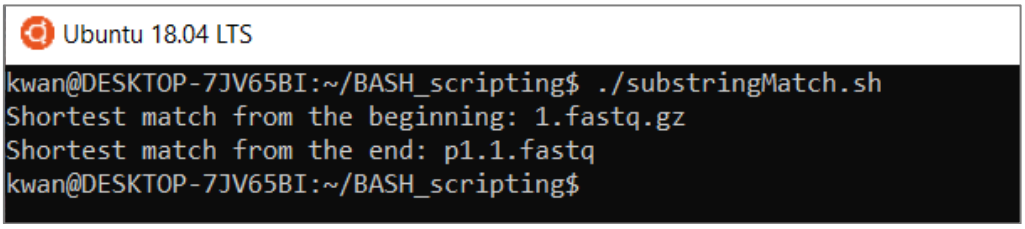
Syntax: Delete matched **substring** from the end of **string**

```
${string%substring}
```

Delete matched substring from full string

```
#!/bin/bash

filename="p1.1.fastq.gz"

begin=${filename#*.} #Delete from the beginning
end=${filename%.*} #Delete from the end

echo "Shortest match from the beginning: $begin"
echo "Shortest match from the end: $end"
```

Output:



4.4 Longest (greedy) substring match

The syntax for deleting the longest match of substring from string

Syntax: Delete match **substring** from the beginning of **string**

```
${string##substring}
```

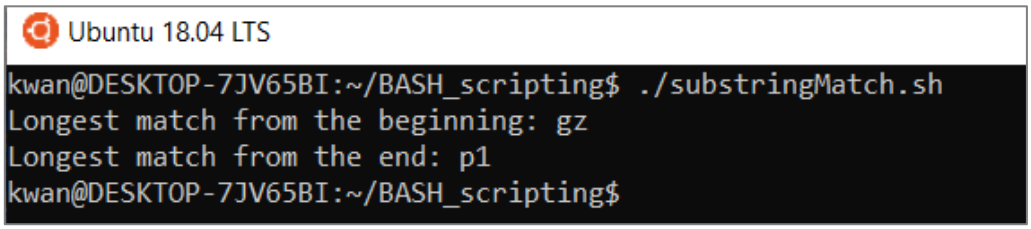
Syntax: Delete match **substring** from the end of **string**

```
${string%%substring}
```

Delete matched substring from full string

```
#!/bin/bash

filename=”p1.1.fastq.gz”

begin=${filename##*.} #Delete from the beginning
end=${filename%%.*} #Delete from the end

echo “Longest match from the beginning: $begin”
echo “Longest match from the end: $end”
```

Output:



4.5 Find and replace

1) Replace only the first match

Find the **`pattern`** in **`string`** and replace only the first match by **`replacement`**.

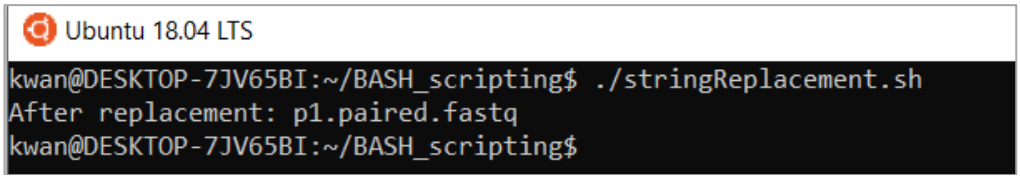
Syntax:

`${string/pattern/replacement}`

Replace only the first match

```
#!/bin/bash

filename=”p1_1.fastq.gz”
replacement=${filename/_*.gz/.paired.fastq}

echo “After replacement: $replacement”
```

Output:



2) Replace all the matches

Find the **`pattern`** in **`string`** and replace all matches by **`replacement`**.

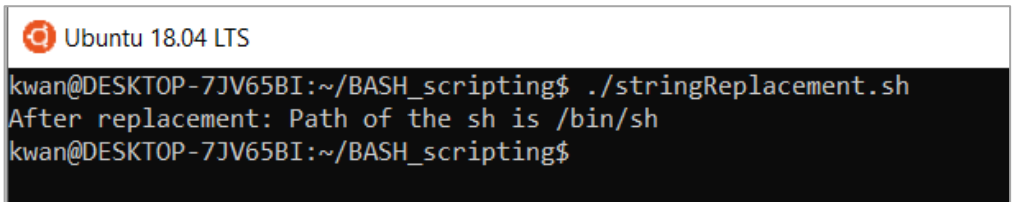
Syntax:

`${string//pattern/replacement}`

Replace all matches

```
#!/bin/bash

filename="Path of the bash is /bin/bash"
replacement=${filename//bash/sh}

echo “After replacement: $replacement”
```

Output:



3) Replace at the beginning or the end

Find the **pattern** in **string** and replace only first match by **replacement**.

Syntax: Replace matched **pattern** with the **replacement** from the beginning of the **string**

`${string/#pattern/replacement}`

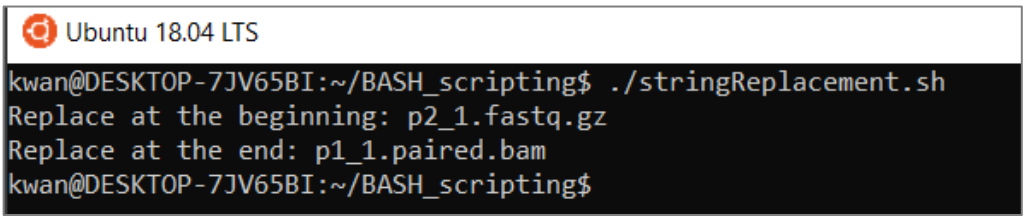
Syntax: Replace matched **pattern** with the **replacement** from the end of the **string**

`${string/%pattern/replacement}`

Delete matched substring from full string

```
#!/bin/bash

filename="p1_1.fastq.gz"

begin=${filename/#*_/p2_} #Replace from the beginning
end=${filename/%.*/.paired.bam} #Replace from the end

echo "Replace at the beginning: $begin"
echo "Replace at the end: $end"
```

Output:



5. Arrays

An array is a data container comprised of two parts including keys and values.

5.1 Create indexed or associative arrays using `declare` command

Syntax:

1) Bash indexed array: the keys of array are ordered integers.

```
declare -a array_name

array_name=(value1 value2)
```

2) Bash associative array: the keys of array are strings.

```
declare -A array_name

array_name=(["key1"]="value1" ["key2"]="value2")
```

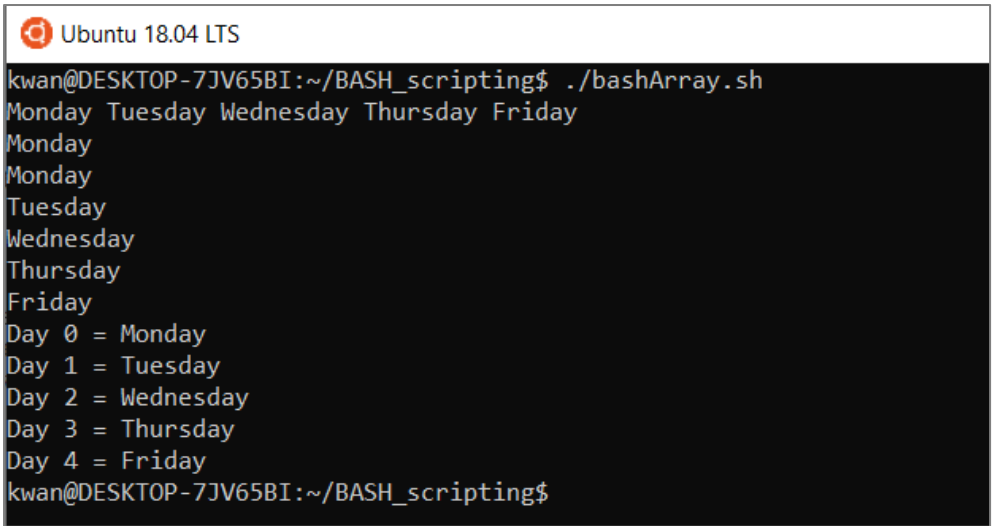
5.2 Access values of an array

1) Access all data in the array
   `${array_name[@]}`
2) Show all index of the array
   `${!array_name[@]}`
3) Access to the data of the index **n** of the array
   `${array_name[n]}`
4) Show the length of the array
   `${#array_name[@]}`
5) Remove both index and data at the index **n**
   `unset array_name[n]`
6) Add new data to the array at the index **n**
   `array_name[n]=”new_value”`

Accessing data in the array

```
#!/bin/bash

wkday=(Monday Tuesday Wednesday Thursday Friday)

echo ${wkday[@]}
echo ${wkday[0]}

for i in ${wkday[@]}
do
     echo $i
done

for index in ${!wkday[@]}
do
     echo "Day $index = ${wkday[index]}"
done
```

Output:



6. Arithmetic operators

Arithmetic operator is a mathematical function that used to perform an arithmetic operation. The following 11 arithmetic operators are supported by bash.

Operator	Name	Description	Example
+	Addition	It adds two operands	`x=$((10+3))` Result: x = 13
-	Subtraction	It subtracts the second operand from the first one	`x=$((10-3))` Result: x = 7
*	Multiplication	Multiply two operands	`x=$((10*3))` Result: x = 30
/	Division	Divide first operand from second operands and return quotient	`x=$((10/3))` Result: x = 3
**	Exponentiation	The second operand raised to the power of the first operand.	`x=$((10**3))` Result: x = 1000
%	Modulo	Divide the first operand from the second operand and return the remainder	`x=$((10%3))` Result: x = 1
+=	Increment by constant	Increment value of the first operand with a given constant value	x=10 ((x+=3)) Result: x=13
-=	Decrement by constant	Decrement value of the first operand with a given constant value	x=10 ((x-=3)) Result: x=7
*=	Multiply by constant	Multiply value of the first operand with a given constant value	x=10 ((x*=3)) Result: x=30
/=	Divide by constant	Divide value of the first operand with a given constant value and return the quotient	x=10 ((x/=3)) Result: x=3
%=	Remainder by dividing with constant	Divide value of the first operand with a given constant value and return the remainder	x=10 ((x%=3)) Result: x=1

Double parentheses can be used to specify arithmetic operation in Bash.

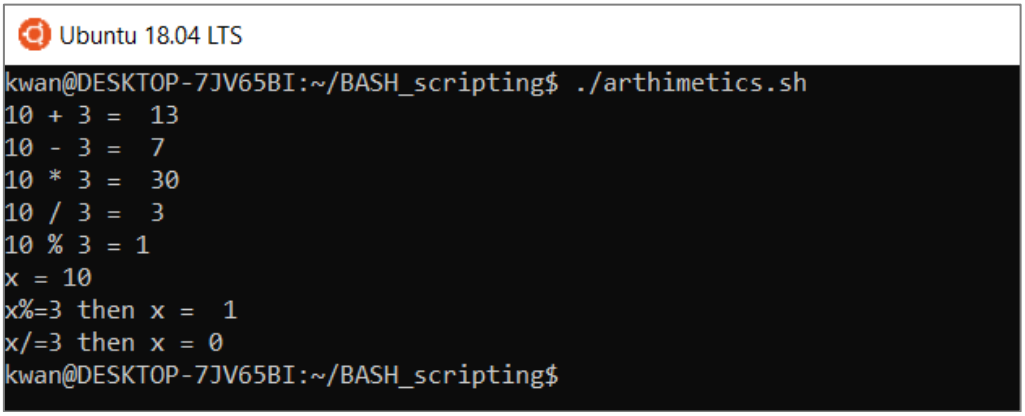
Syntax:

```
((expression))
```

Perform arithmetic operations by Double parentheses

```
#!/bin/bash

echo "10 + 3 = " $((10+3))
echo "10 - 3 = " $((10-3))
echo "10 * 3 = " $((10*3))
echo "10 / 3 = " $((10/3))
a=$((10%3))
echo "10 % 3 = $a”

x=10
echo "x = $x"
echo "x%=3 then x = " $((x%=3))

b=$((x/=3))
echo "x/=3 then x = $b"
```

Output:



7. Script Input (STDIN)

7.1 Command line arguments

The arguments are input that necessary for processing the script. The command line arguments are passed in a positional way.

Syntax:

```
./bash_script.sh arg1 arg2 arg3..
```

where arg1 = $1 arg2 = $2 arg3 = $3

Special variable	Detail
$0	Name of bash script
$1 … $n	Positional argument indicated from 1 to n.
$@	All arguments that are passed in to the script
$#	The total number of arguments passed to script
$?	The exit status of the most recently run process
$$	The process ID of the current script

7.2 Read command

A read command is built-in command that takes the user input into a variable.

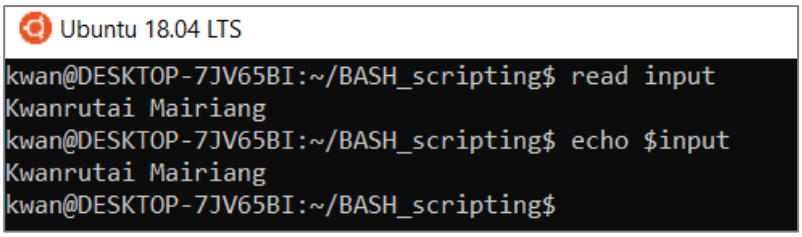
Syntax:

```
read OPTIONS ARGUMENT
```

Try read command

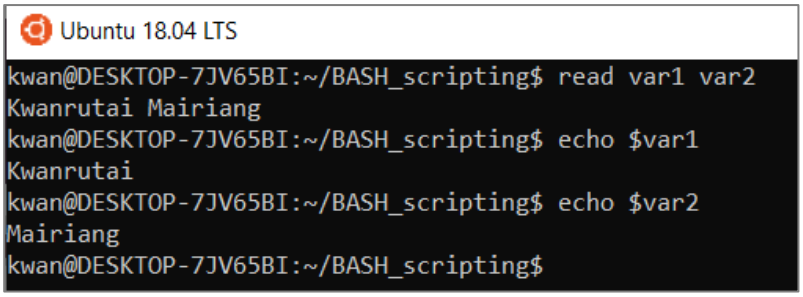
1). Save the user input into a specified variable

```
read input
echo $input
```



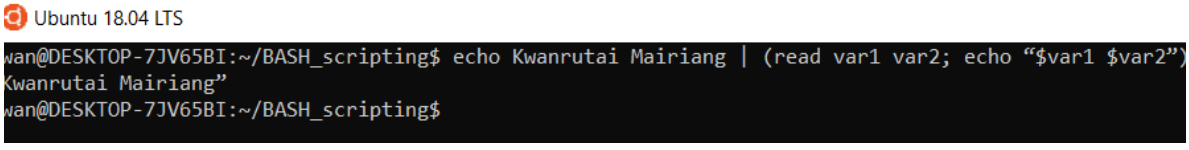
2).Split the user input into different variables by adding multiple argument

```
read var1 var2
echo var1
echo var2
```



3). Piping: pipe a standard output from one command and pass it as an input for the other command

```
echo Kwanrutai Mairiang | (read var1 var2; echo "$var1 $var2")
```



8. Condition statement

A condition statement is used for decision making in any programing language. Bash scripting also use this statement for making some decisions in an automated task.

Comparison operators

Operator	Syntax	Description
-eq	INTEGER1 -eq INTEGER2	Return true if two numbers are equal
-ne	INTEGER1 -ne INTEGER2	Return true if two numbers are not equal
-lt	INTEGER1 -lt INTEGER2	Return true if integer1 less than integer2
-gt	INTEGER1 -gt INTEGER2	Return true if integer1 greater than integer2
==	STRING1 == STRING2	Return true if STRING1 is equal to STRING2
!=	STRING1 != STRING2	Return true if STRING1 is not equal to STRING2
!	! EXPRESSION	Return true if the expression is false
-d	-d FILE	Check the existence of a directory
-e	-e FILE	Check the existence of a file
-r	-r FILE	Check the existence of a file and read permission
-w	-w FILE	Check the existence of a file and write permission
-x	-x FILE	Check the existence of a file and execute permission

8.1 If statement

The basic if statement contains one level of condition and action. The syntax consisting of **`if`** follow by **`EXPRESSION`** in square brackets. If the **`EXPRESSION`** is true, **`then ACTION`** will be performed. The statement ends with **`fi`**. One if statement can contain one (single condition) or more expressions (multiple conditions).

1) Single condition

Syntax:

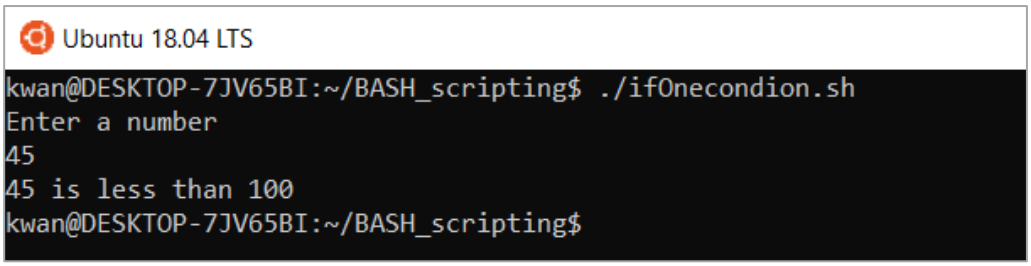
```
if [ EXPRESSION ]; then
ACTION
fi
```

The following example show the basic “if statement” with single condition.

Check if input number is less than 100

```
#!/bin/bash

#Get input number from user input
echo "Enter a number"
read n

#Check if input number less than 100
if [ $n -lt 100 ]; then
echo "$n is less than 100"
fi
```

Output:



2) Multiple conditions

Multiple conditions in "if statement" need BOOLEAN operator for joining between conditions.

Operator	Symbol	Description
AND	&&	Return TRUE when both Expression_1 and Expression_2 are TRUE
OR	||	Return TRUE when one of Expression_1 or Expression_2 is TRUE

Syntax:

AND operator

```
if [ EXPRESSION_1 ] && [ EXPRESSION_2 ]; then
ACTION
fi
```

OR operator

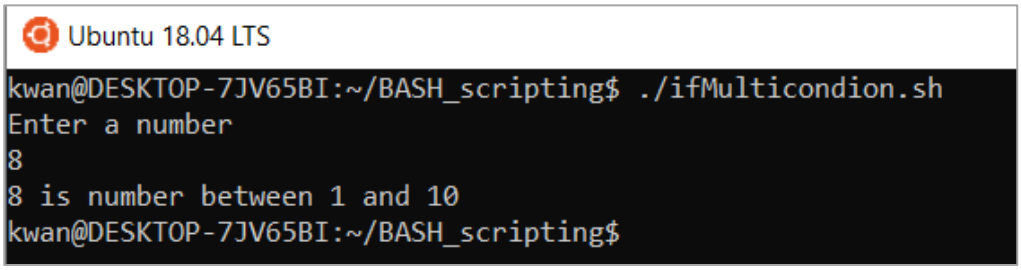
```
if [ EXPRESSION_1 ] || [ EXPRESSION_2 ]; then
ACTION
fi
```

The following example shows the basic "if statement" with multiple conditions.

Check if input number is between 1 and 10

```
#!/bin/bash

#Get input number from user input
echo "Enter a number"
read n

#Check if input number is greater than 1 and less
than 10
if [ $n -gt 1 ] && [ $n -lt 10 ]; then
echo "$n is number between 1 and 10 "
fi
```

Output:



8.2 If-else statement

This pattern of conditional statement is used to execute one action with a true condition and the other action with a false condition.

Syntax:

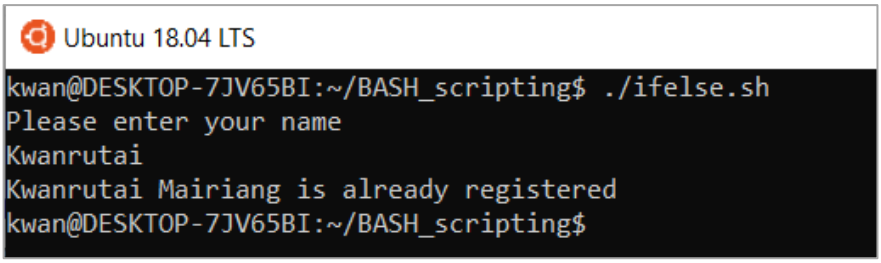
```
if [ EXPRESSION ]; then
ACTION_1
else
ACTION_2
fi
```

Check if input name is already in "users" array

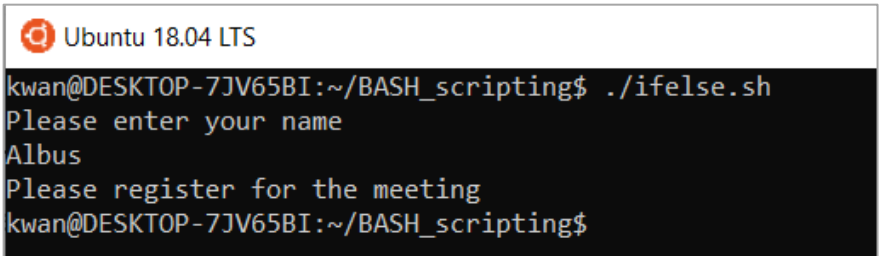
```
#!/bin/bash

declare -A users

users=(["Harry"]="Harry Potter"
["Hermione"]="Hermione Granger"
["Ron"]="Ron Weasley"
["Kwanrutai"]="Kwanrutai Mairiang")

echo "Please enter your name"
read name

if [[ -n "${users[$name]}" ]]; then
   printf '%s is already registered\n' "${users[$name]}"
else
   echo "Please register for the meeting"
fi
```

Output:

1. Input: **Kwanrutai**



2. Input: **Albus**



8.3 If..elif..else statement (if-else in ladder)

This pattern of conditional statement is used for a series of conditions. The set of **ACTION** in **if** statement is executed, when the **EXPRESSION** is TRUE. If there is no TRUE **EXPRESSION**, the **ACTION** in **else** statement will be executed.

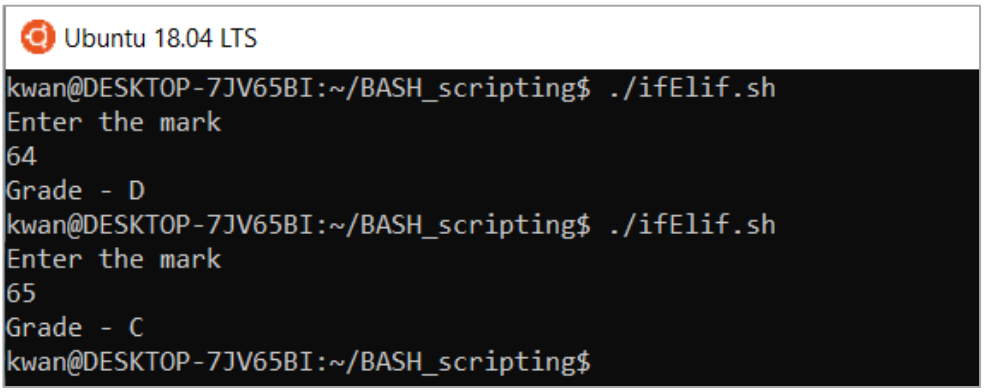
Syntax:

```
if [ EXPRESSION_1 ]; then
ACTION_1
elif [ EXPRESSION_2 ]; then
ACTION_2
…
else
ACTION_3
Fi
```

Check grade using the input score

```
#!/bin/bash

echo "Enter the mark"
read mark

if (( $mark >= 85 )); then
echo "Grade - A"
elif (( $mark < 85 && $mark >= 75 )); then
echo "Grade - B"
elif (( $mark < 75 && $mark >= 65 )); then
echo "Grade - C"
elif (( $mark < 65 && $mark >= 55 )); then
echo "Grade - D"
else
echo "Grade - F"
fi
```

Output:



8.4 Nested if statement

This pattern of conditional statement is used when one condition is true, then the next condition is checked. Two example syntax are shown below.

Syntax:

1) In syntax 1, if the **EXPRESSION_1** is true, then another expression, **EXPRESSION_2** is checked. If **EXPRESSION_2** also true, **ACTION** will be executed.

```
if [ EXPRESSION_1 ]; then
    if [ EXPRESSION_2 ]; then
    ACTION
    fi
fi
```

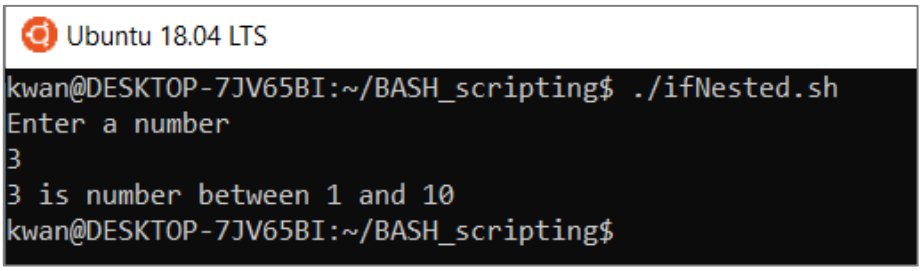
2) In syntax 2, if **EXPRESSION_1** is true, then the **ACTION_1** will be performed. But, if **EXPRESSION_1** is false, the **EXPRESSION_2** in **else** will be checked. If **EXPRESSION_2** is true, the **ACTION_2** will be executed.

```
if [ EXPRESSION_1 ]; then
ACTION_1
else
    if [ EXPRESSION_2 ]; then
    ACTION_2
    fi
fi
```

Check if input number is between 1 and 10 using nested if condition

```
#!/bin/bash

#Get input number from user input
echo "Enter a number"
read n

#Check if input number is greater than 1 and less
than 10
if [ $n -gt 1 ]; then
        if [ $n -lt 10 ]; then
        echo "$n is number between 1 and 10"
        fi
fi
```

Output:

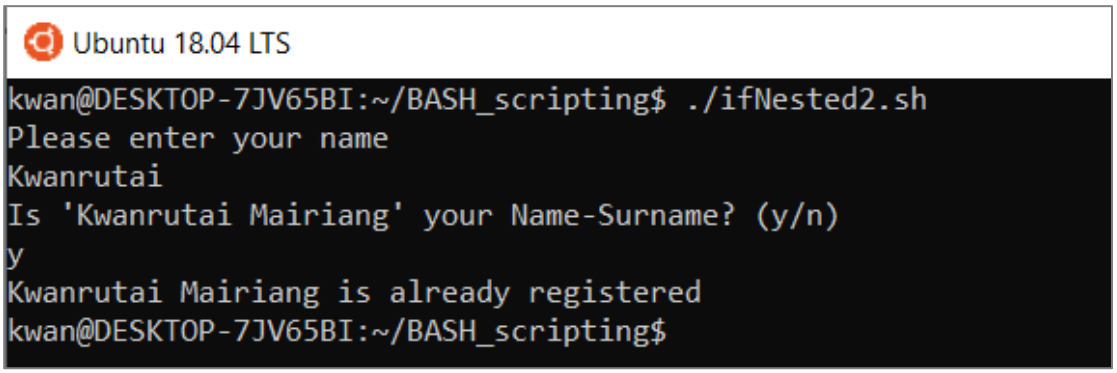


Check if input name is already in “users” array

```
#!/bin/bash

declare -A users

users=(["Harry"]="Harry Potter"
["Hermione"]="Hermione Granger"
["Ron"]="Ron Weasley"
["Kwanrutai"]="Kwanrutai Mairiang")

echo "Please enter your name"
read name

if [[ -n "${users[$name]}" ]]; then
     echo "Is '${users[$name]}' your Name-Surname? (y/n)"
     read check
     if [ $check == y ]; then
          printf '%s is already registered\n' "${users[$name]}"
     else
          echo "Please register for the meeting"
     fi
else
     echo "Please register for the meeting"
fi
```

Output:

Input: **Kwanrutai**



9. For loop

For loop is used for iterating item in the list of items. An item from each round is assigned to the variable which is then used to perform any action in loop. The syntax of "**For** loop "consisting of **LIST** of data and variable (**ITEM**). For loop starts with **do** and ends with **done**.

Syntax:

```
for ITEM in [LIST]
do
  ACTION
done
```

The list of items can be a series of strings separated by spaces, a range of numbers, output of a command, an array.

9.1 Loop over a series of strings

For loop over series of string: Sunday … Saturday

```
#!/bin/bash

count=0
for day in Sunday Monday Tuesday Wednesday Thursday Friday Saturday
do
      count+=1
      echo "Day $count = $day"
done
```

Output:

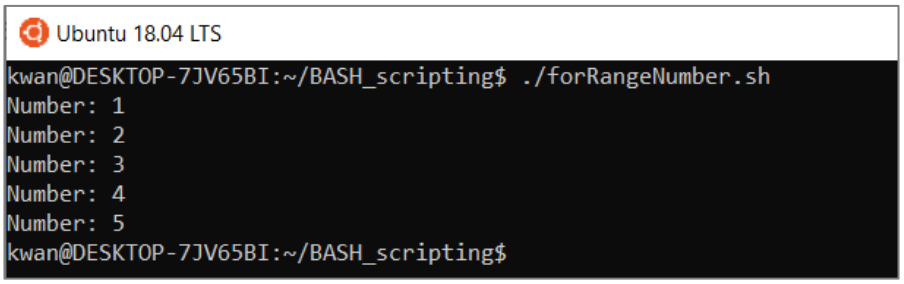
9.2 Loop over a number range

1) Loop over the specified range, {START..END}, of numbers.

For loop over specified range of number 1 to 5

```
#!/bin/bash

for i in {1..5}
do
      echo "Number: $i"
done
```

Output:

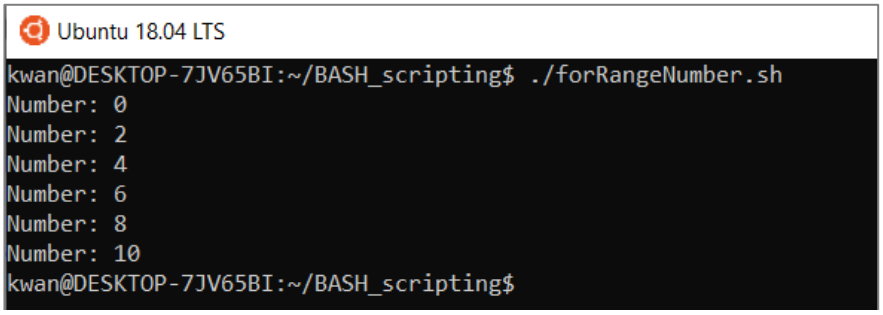


2) Loop over the specified range with increment, {START..END..INCREMENT}

For loop over specified range of number 0 to 10 with increment 2

```
#!/bin/bash

for i in {0..10..2}
do
      echo "Number: $i"
done
```

Output:



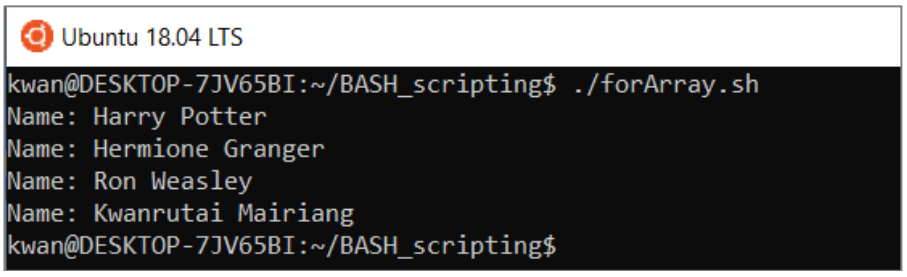
9.3 Loop over array elements

Use for loop for iterating item in array.

For loop over item in array

```
#!/bin/bash

users=("Harry Potter" "Hermione Granger" "Ron Weasley"
"Kwanrutai Mairiang")

for name in "${users[@]}"
do
     echo "Name: $name"
done
```

Output:



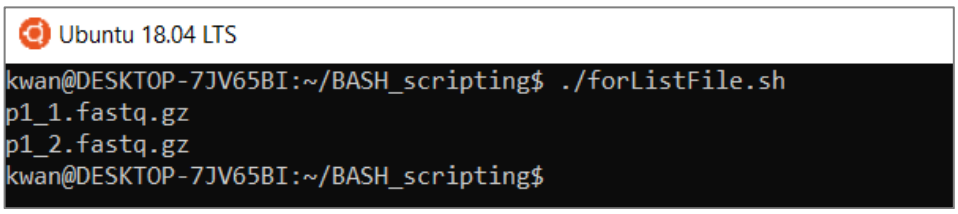
9.4 Loop over output of a command

The following example showing how to iterate filename with specific extension in current folder.

For loop over the list of files with extension ".gz"

```
#!/bin/bash

for file in *.gz
do
     echo $file
done
```

Output:



10. While loop

Another type of loop is while loop. While loop will iterate while the specified condition is true. While loop is useful when exact times for looping is not known. The syntax of "**`While`**" loop contains **`CONDITION`** that made the loop keep iterate. Then, **`UPGRADE CONDITION`** until condition becomes false for stopping the iteration.

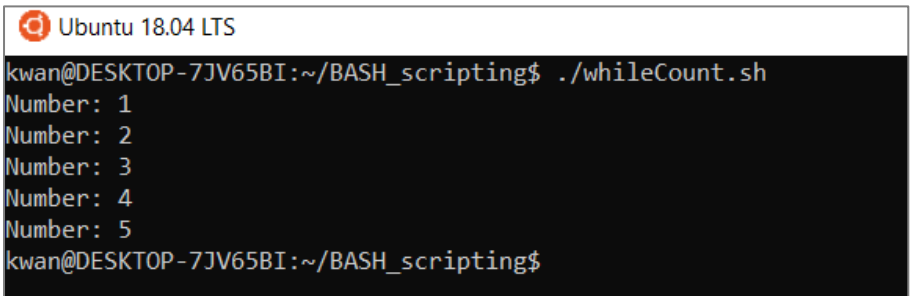
Syntax:

```
while [ CONDITION ]
do
      ACTION
      UPGRADE_CONDITION    Ex.((number ++))
Done
```

Loop and print out the number from 1 to 5

```
#!/bin/bash

count=1
while [ $count -le 5 ]
do
      echo "Number: $count"
      ((count++))
done
```

Output:



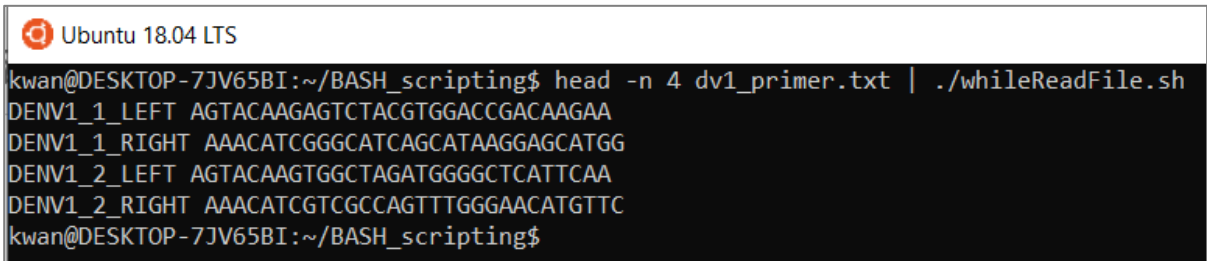
Reading file using while loop

Read data or file from standard input

```
#!/bin/bash


while read line
do
      echo $line  #Print out each line in file or input data
done < "${1:-/dev/stdin}"  #Get filename or data from standard input
```

Output:

Pipe 4 lines of data from "dv1_primer.txt" to Bash script



Bash scripting practical

1. Write a script to read a tab delimited file containing primer names and sequences. Primer sequences contain 8nt-index at position 1 to 8. Remove the 8nt-index from primer sequences and print out both primer names and edited sequences in FASTA format.
    a. Input: dv1_primer.txt

2. Write a script to read a genome sequence from a FASTA file. Split the genome sequence into each gene using the following gene positions. Pipe all gene sequences in a FASTA format to an output file.
    a. Input: reference.fasta
    b. Output: dv1_gene.fasta
    c. Gene position

Gene	Start	End
capsid	95	436
prM	437	934
envelope	935	2419
ns1	2420	3475
ns2a	3476	4129
ns2b	4130	4519
ns3	4520	6376
ns4a	6377	6826
ns4b	6827	7573
ns5	7574	10270

Group practical

1. Create a folder 'p1' , and then move files 'p1_1.fastq.gz' and 'p1_2. fastq.gz' into the newly created folder.
2. Write a script "`run_analysis.sh`" to build an automated pipeline to run the following processes:
    1) Run "Trimmometic" program to trim low quality base
        a. Input: p1_1.fastq.gz, p1_2. fastq.gz in the p1 folder
        b. Trimming parameter:
            i. Length >= 40
            ii. Score >= 20
        c. Trimmometic command:

```
java -jar /path/to/trimmomatic/trimmomatic-0.39.jar PE -phred33
p1/p1_1.fastq.gz p1/p1_2.fastq.gz p1/p1_1.trim.fastq.gz
p1/p1_1.unpair.fastq.gz p1/p1_2.trim.fastq.gz p1/p1_2.unpair.fastq.gz
LEADING:20 TRAILING:20 SLIDINGWINDOW:5:20 MINLEN:40
```

2) Align trimmed sequences to a reference genome using minimap2
    a. Reference file: reference.fasta
    b. Minimap2 command:

```
/path/to/minimap2/minimap2 -ax sr -o p1/p1.sam
reference.fasta p1/p1_1.trim.fastq.gz p1/p1_2.trim.fastq.gz
```

3) Convert a SAM file (from step2) to a BAM file, then sort BAM file and filter only paired mapped
    a. Samtools command
        i. SAM to BAM:

```
samtools view -Shb -o p1/p1.bam p1/p1.sam
```

        ii. Sort BAM:

```
samtools sort -o p1/p1.sorted.bam p1/p1.bam
```

        iii. Filter paired mapped

```
samtools view -hb -f 2 -o p1/p1.sorted.pair.bam
p1/p1.sorted.bam
```

4) Run samtools flagstat
    a. Flagstat command:

```
samtools flagstat p1/p1.sorted.pair.bam
```