# Bash scripting

*Presenter: Nguyen Le Duc Minh*

# Contents

- What scripts are
- The components that make up a script
- How to use variables in your scripts
- How to perform tests and make decisions
- How to accept command line arguments
- How to accept input from a user.

# Scripts

- Contain a series of commands
- An interpreter executes commands in the script
- Anything you can type at the command line, you can put in a script.
- Great for automating tasks.

# script.sh

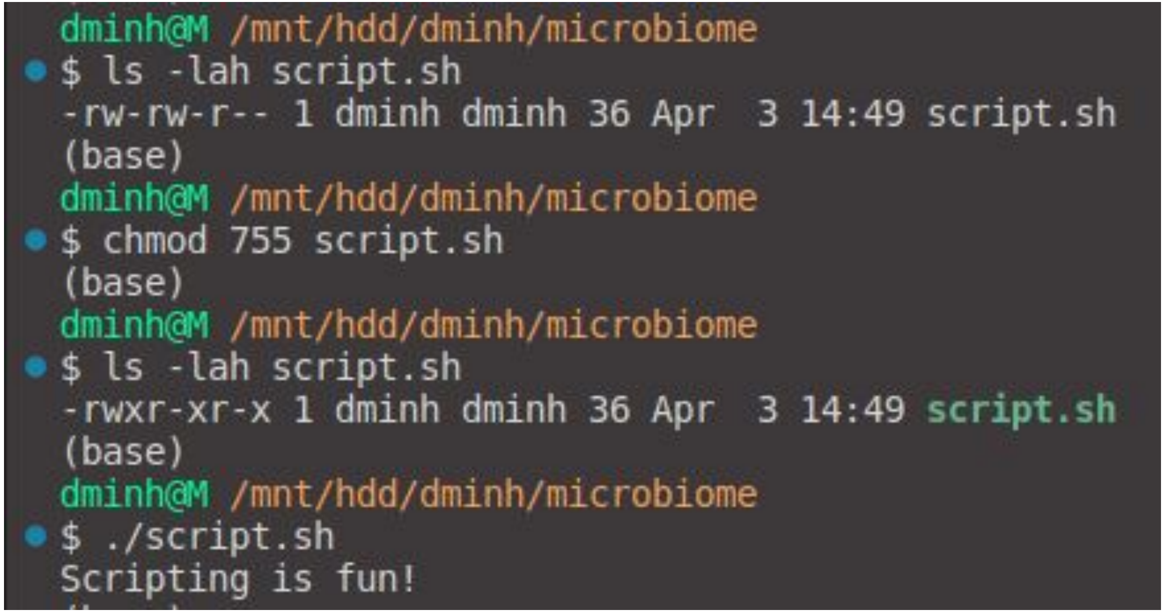
#!/bin/bash

echo "Scripting is fun!"

$ chmod 755 script.sh

$ ./script.sh

Scripting is fun!



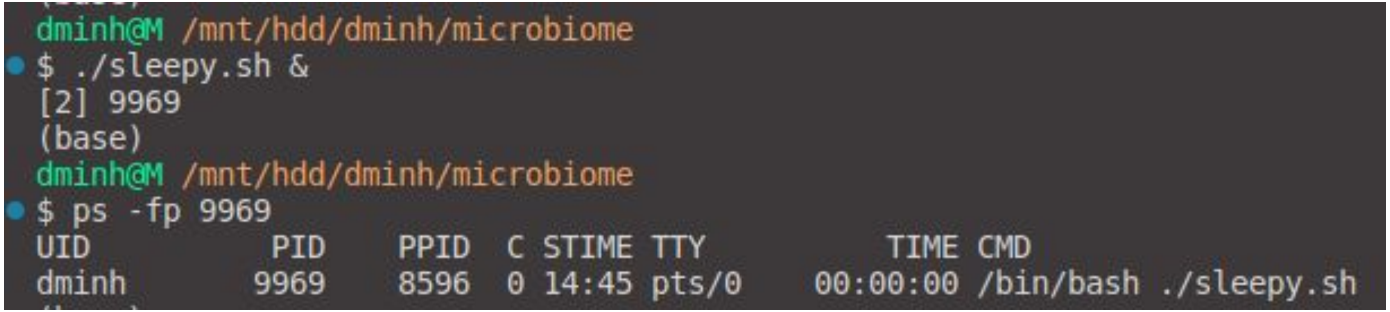
```
dminh@M /mnt/hdd/dminh/microbiome
$ ls -lah script.sh
-rw-rw-r-- 1 dminh dminh 36 Apr  3 14:49 script.sh
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ chmod 755 script.sh
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ls -lah script.sh
-rwxr-xr-x 1 dminh dminh 36 Apr  3 14:49 script.sh
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ./script.sh
Scripting is fun!
```

# Shebang

```
#!/bin/csh
echo "This script uses csh as the interpreter."
#!/bin/ksh
echo "This script uses ksh as the interpreter."
#!/bin/zsh
echo "This script uses zsh as the interpreter."
```

# sleepy.sh

```
#!/bin/bash

sleep 90

$ ./sleepy.sh &

[1] 9969

$ ps -fp 9969
```



```
dminh@M /mnt/hdd/dminh/microbiome
$ ./sleepy.sh &
[2] 9969
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ps -fp 9969
UID          PID    PPID  C STIME TTY          TIME CMD
dminh       9969    8596  0 14:45 pts/0    00:00:00 /bin/bash ./sleepy.sh
```

# The interpreter executes the script

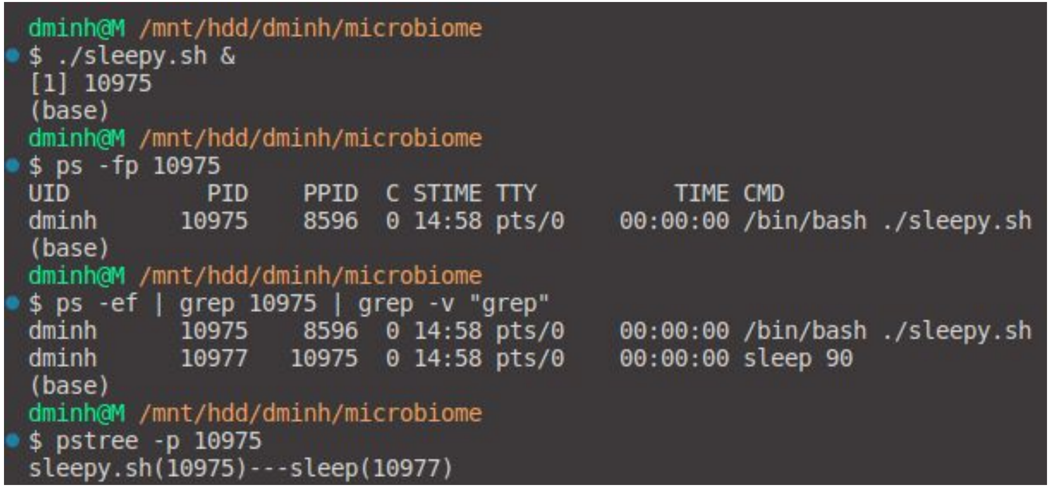
$ ./sleepy.sh &

[1] 10975

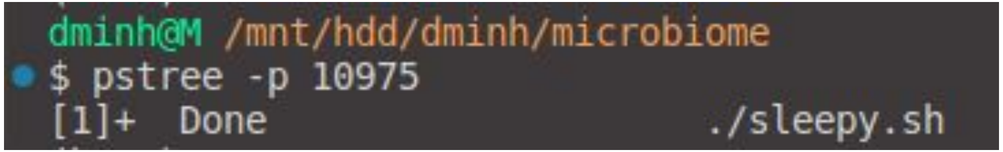
$ ps -fp 10975

$ ps -ef | grep 10975 | grep -v grep

$ pstree –p 10975



```
dminh@M /mnt/hdd/dminh/microbiome
$ ./sleepy.sh &
[1] 10975
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ps -fp 10975
UID          PID    PPID  C STIME TTY          TIME CMD
dminh      10975    8596  0 14:58 pts/0    00:00:00 /bin/bash ./sleepy.sh
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ps -ef | grep 10975 | grep -v "grep"
dminh      10975    8596  0 14:58 pts/0    00:00:00 /bin/bash ./sleepy.sh
dminh      10977   10975  0 14:58 pts/0    00:00:00 sleep 90
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ pstree -p 10975
sleepy.sh(10975)---sleep(10977)
```



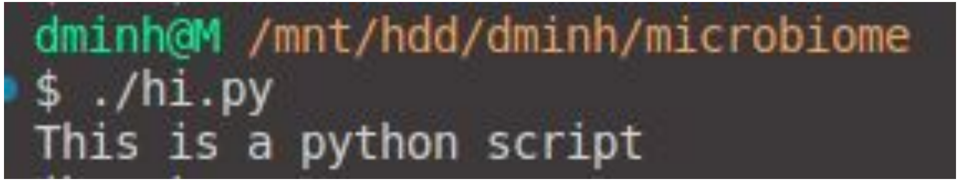
```
dminh@M /mnt/hdd/dminh/microbiome
$ pstree -p 10975
[1]+  Done                    ./sleepy.sh
```

# Shebang or Not to Shebang

- If a script does not contain a shebang the commands are executed using your shell.
- You might get lucky. Maybe. Hopefully.
- Different shells have slightly varying syntax.

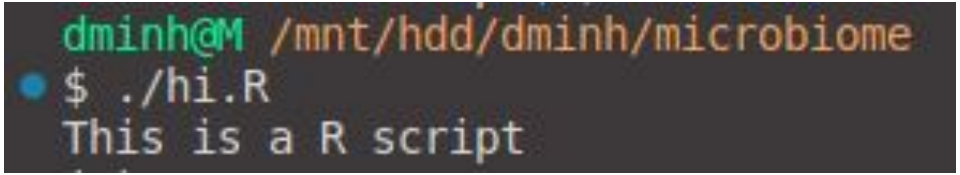
# More than just shell scripts

```
#!/usr/bin/python

print("This is a Python script.")
```



```
dminh@M /mnt/hdd/dminh/microbiome
$ ./hi.py
This is a python script
```

```
#!/usr/bin/Rscript

cat("This is an R script.")
```



```
dminh@M /mnt/hdd/dminh/microbiome
$ ./hi.R
This is a R script
```

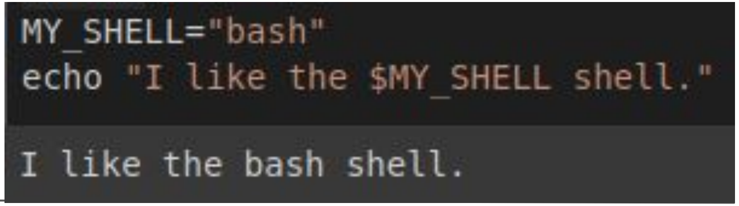
```
$ chmod 755 <hi.py/hi.R>

$ ./hi.py or $ ./hi.R
```

# Variables

- Storage locations that have a name
- Name-value pairs
- Syntax:
    - VARIABLE_NAME="Value"
- Variables are case sensitive
- By convention variables are uppercase

# Variable Usage

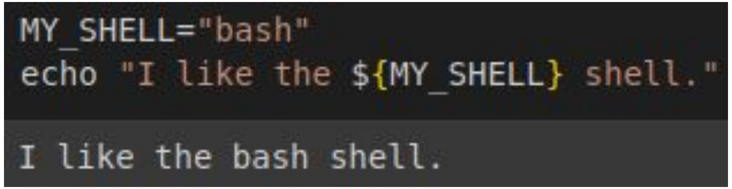
```
#!/bin/bash

MY_SHELL="bash"

echo "I like the $MY_SHELL shell."
```



```
MY_SHELL="bash"
echo "I like the $MY_SHELL shell."
```

```
I like the bash shell.
```

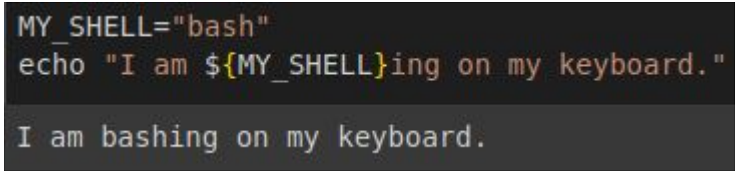
```
#!/bin/bash

MY_SHELL="bash"

echo "I like the ${MY_SHELL} shell."
```



```
MY_SHELL="bash"
echo "I like the ${MY_SHELL} shell."
```

```
I like the bash shell.
```

```
#!/bin/bash

MY_SHELL="bash"

echo "I am ${MY_SHELL}ing on my keyboard."
```

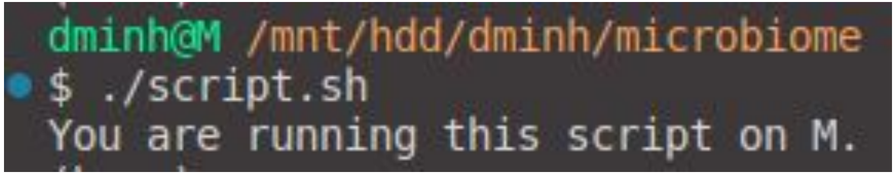


```
MY_SHELL="bash"
echo "I am ${MY_SHELL}ing on my keyboard."
```

```
I am bashing on my keyboard.
```

# Assign command output to a variable

```
#!/bin/bash

SERVER_NAME=$(hostname)

echo "You are running this script on ${SERVER_NAME}."
```



```
dminh@M /mnt/hdd/dminh/microbiome
$ ./script.sh
You are running this script on M.
```

```
#!/bin/bash

SERVER_NAME=`hostname`

echo "You are running this script on ${SERVER_NAME}."
```

# Variable Names

**Valid**:

- FIRST3LETTERS="ABC"
- FIRST_THREE_LETTERS="ABC"
- firstThreeLetters="ABC"

**Invalid**:

- 3LETTERS="ABC"
- first-three-letters="ABC"
- first@Three@Letters="ABC"

# Tests

Syntax:

- [ condition-to-test-for ]

Example:

- [ -e /etc/passwd ]

# File operators (tests)

**-d** FILE True if file is a directory.

**-e** FILE True if file exists.

**-f** FILE True if file exists and is a regular file.

**-r** FILE True if file is readable by you.

**-s** FILE True if file exists and is not empty.

**-w** FILE True if the file is writable by you.

**-x** FILE True if the file is executable by you

# String operators (tests)

- **-z** STRING True if string is empty.
- **-n** STRING True if string is not empty.
- STRING1 **=** STRING2 True if the strings are equal.
- STRING1 **!=** STRING2 True if the strings are not equal

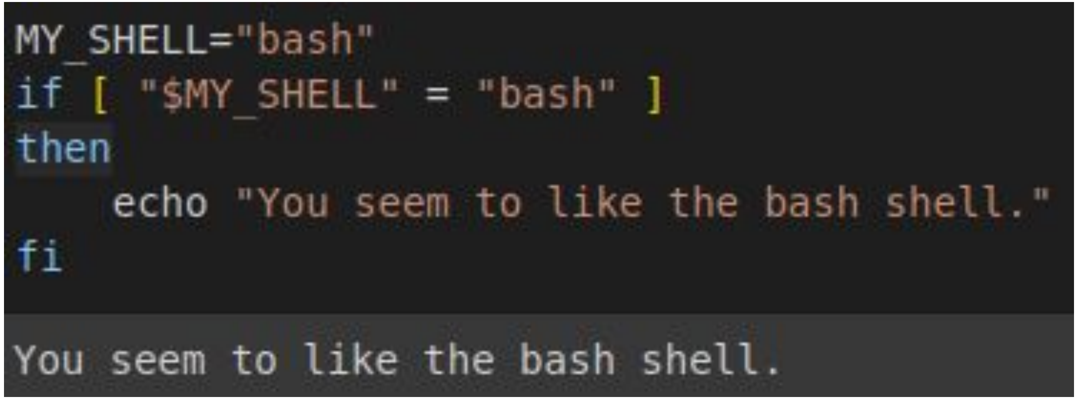
# Arithmetic operators (tests)

- arg1 **–eq** arg2 True if arg1 is **equal** to arg2.
- arg1 **–ne** arg2 True if arg1 is **not equal** to arg2.
- arg1 **–lt** arg2 True if arg1 is **less** than arg2.
- arg1 **–le** arg2 True if arg1 is **less** than or **equal** to arg2.
- arg1 **–gt** arg2 True if arg1 is **greater** than arg2.
- arg1 **–ge** arg2 True if arg1 is **greater** than or **equal** to arg2

# Making Decisions - The if statement

```
if [ condition-is-true ]
then
    command 1
    command 2
    command N
fi
```

```
#!/bin/bash

MY_SHELL="bash"

if [ "$MY_SHELL" = "bash" ]

then

        echo "You seem to like the bash shell."

fi
```



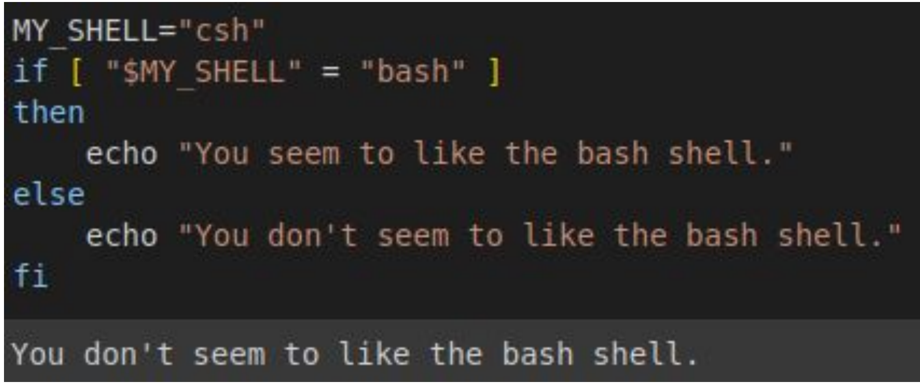
```
MY_SHELL="bash"
if [ "$MY_SHELL" = "bash" ]
then
    echo "You seem to like the bash shell."
fi
```

```
You seem to like the bash shell.
```

# if/else

```
if [ condition-is-true ]
then
    command M
else
    command N
fi
```

```
#!/bin/bash

MY_SHELL="csh"

if [ "$MY_SHELL" = "bash" ]

then

        echo "You seem to like the bash shell."

else

        echo "You don't seem to like the bash shell."

fi
```



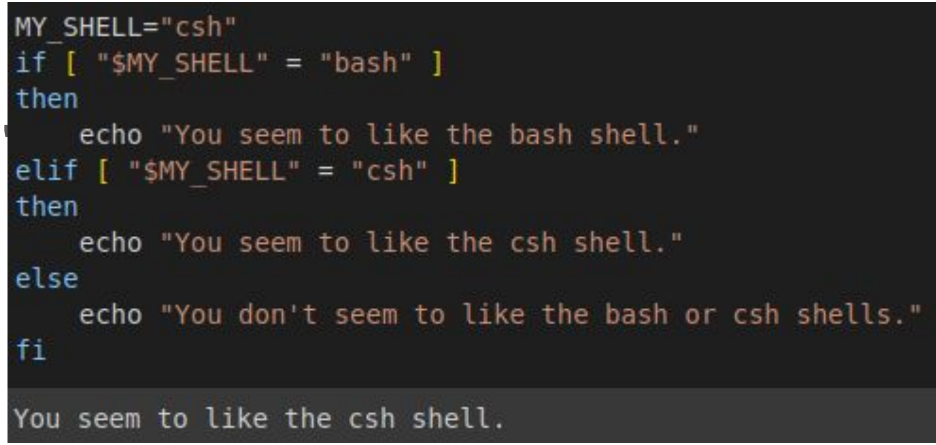
```
MY_SHELL="csh"
if [ "$MY_SHELL" = "bash" ]
then
    echo "You seem to like the bash shell."
else
    echo "You don't seem to like the bash shell."
fi
```

```
You don't seem to like the bash shell.
```

# if/elif/else

```
if [ condition1-is-true ]
then
    command 1
elif [ condition2-is-true ]
then
    command 2
else
    command 3
fi
```

```
#!/bin/bash

MY_SHELL="csh"

if [ "$MY_SHELL" = "bash" ]

then

        echo "You seem to like the bash shell."

elif [ "$MY_SHELL" = "csh" ]

then

        echo "You seem to like the csh shell."

else

        echo "You don't seem to like the bash or csh shells."

fi
```



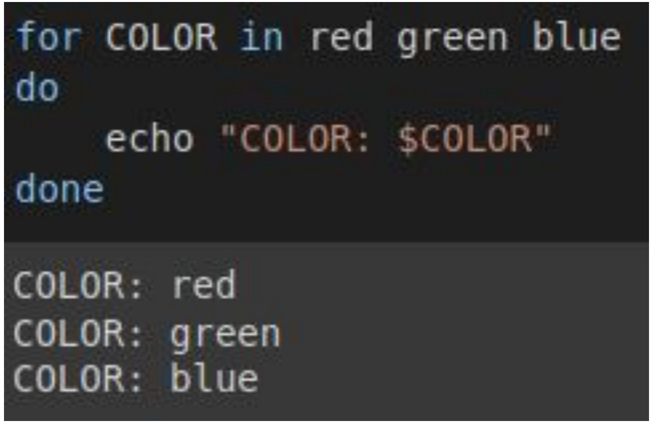
```
MY_SHELL="csh"
if [ "$MY_SHELL" = "bash" ]
then
    echo "You seem to like the bash shell."
elif [ "$MY_SHELL" = "csh" ]
then
    echo "You seem to like the csh shell."
else
    echo "You don't seem to like the bash or csh shells."
fi
```

```
You seem to like the csh shell.
```

# For loop

```
for VARIABLE_NAME in ITEM_1 ITEM_N

do

    command 1

    command 2

    command N

done
```

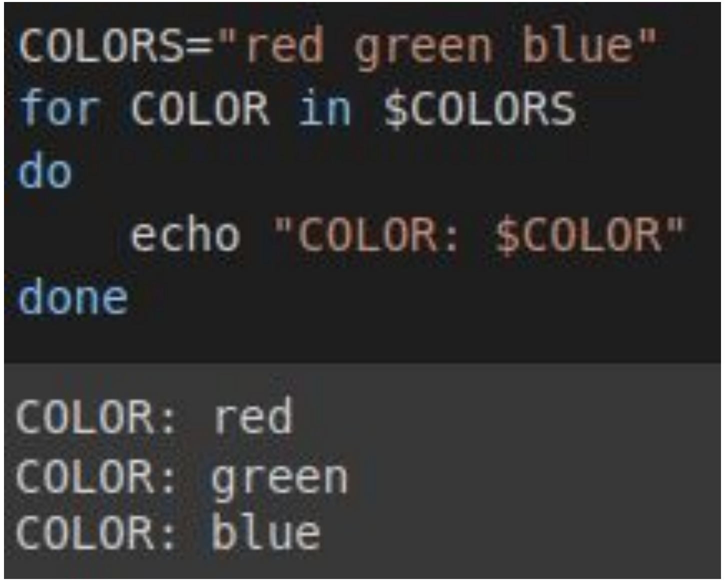
```
#!/bin/bash

for COLOR in red green blue

do

        echo "COLOR: $COLOR"

done
```



```
for COLOR in red green blue
do
    echo "COLOR: $COLOR"
done
```

```
COLOR: red
COLOR: green
COLOR: blue
```

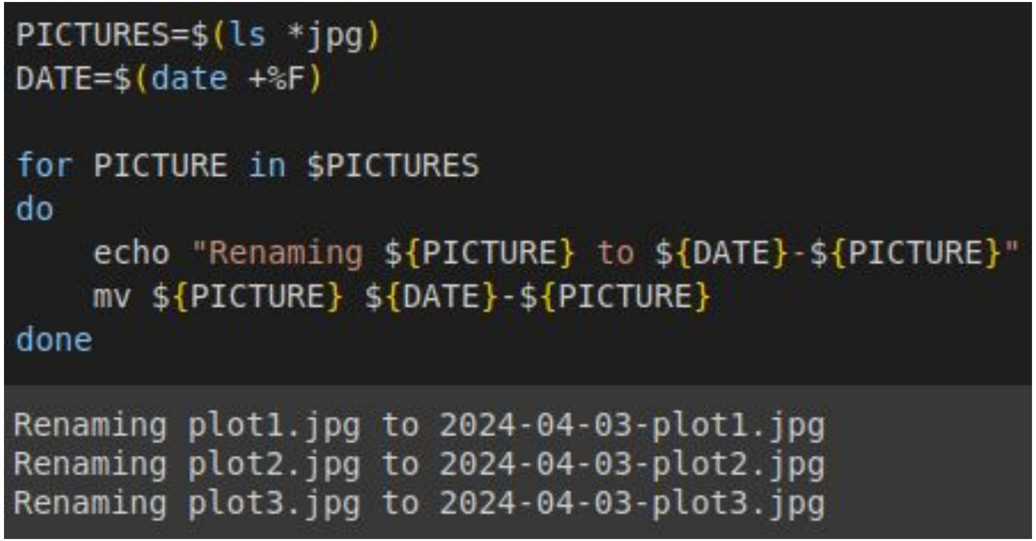
```
#!/bin/bash

COLORS="red green blue"

for COLOR in $COLORS

do

	echo "COLOR: $COLOR"

done
```



```
COLORS="red green blue"
for COLOR in $COLORS
do
    echo "COLOR: $COLOR"
done
```

```
COLOR: red
COLOR: green
COLOR: blue
```

```
#!/bin/bash

PICTURES=$(ls *jpg)

DATE=$(date +%F)


for PICTURE in $PICTURES

do

    echo "Renaming ${PICTURE} to ${DATE} -${PICTURE}"

    mv ${PICTURE} ${DATE}-${PICTURE}

done
```



```
PICTURES=$(ls *jpg)
DATE=$(date +%F)

for PICTURE in $PICTURES
do
    echo "Renaming ${PICTURE} to ${DATE}-${PICTURE}"
    mv ${PICTURE} ${DATE}-${PICTURE}
done
```

```
Renaming plot1.jpg to 2024-04-03-plot1.jpg
Renaming plot2.jpg to 2024-04-03-plot2.jpg
Renaming plot3.jpg to 2024-04-03-plot3.jpg
```

# Positional Parameters

```
$ script.sh parameter1 parameter2 parameter3
```

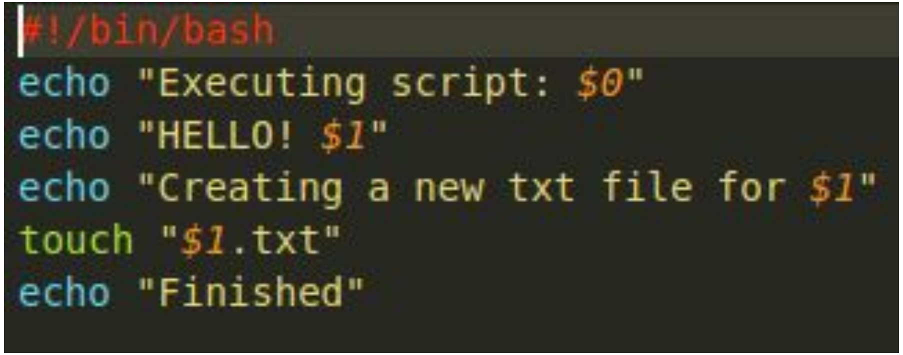
**$0 : "script.sh"**

**$1 : "parameter1"**

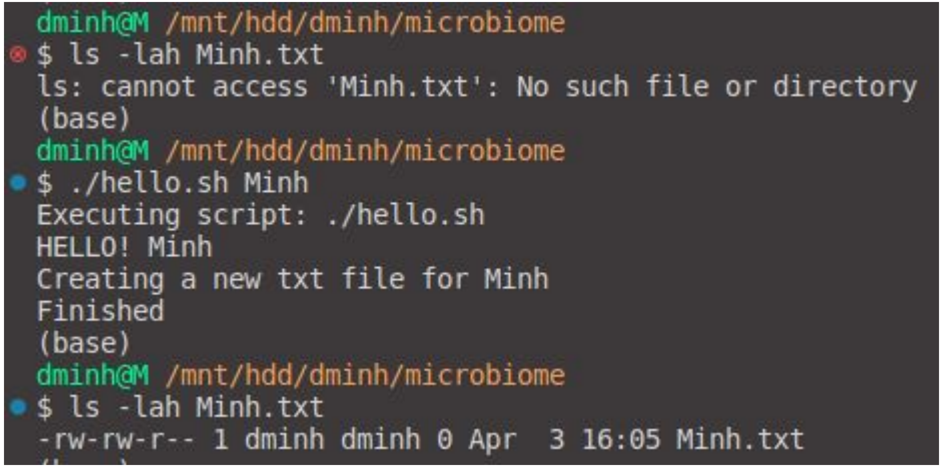
**$2 : "parameter2"**

**$3 : "parameter3"**

```
#!/bin/bash

echo "Executing script: $0"

echo "HELLO! $1"

echo “Creating a new txt file for $1”

touch “$1.txt”

echo “Finished”
```

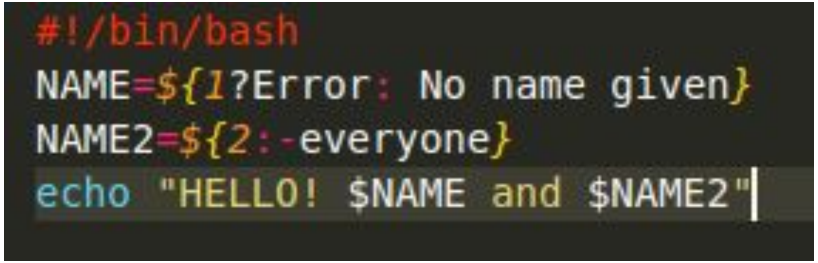


```
#!/bin/bash
echo "Executing script: $0"
echo "HELLO! $1"
echo "Creating a new txt file for $1"
touch "$1.txt"
echo "Finished"
```

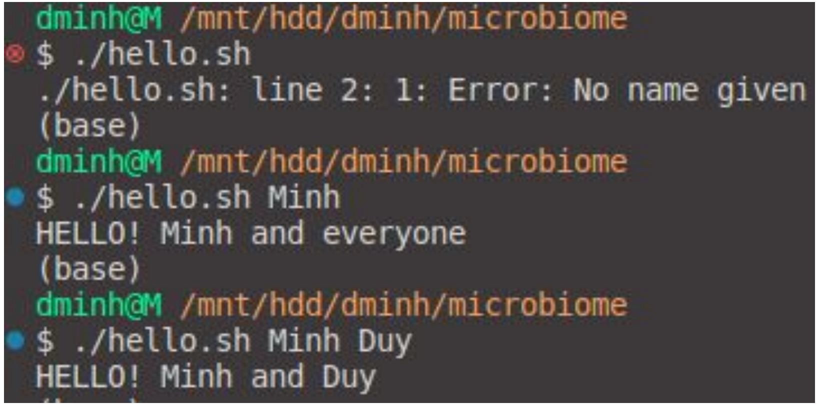


```
dminh@M /mnt/hdd/dminh/microbiome
⊗ $ ls -lah Minh.txt
ls: cannot access 'Minh.txt': No such file or directory
(base)
dminh@M /mnt/hdd/dminh/microbiome
● $ ./hello.sh Minh
Executing script: ./hello.sh
HELLO! Minh
Creating a new txt file for Minh
Finished
(base)
dminh@M /mnt/hdd/dminh/microbiome
● $ ls -lah Minh.txt
-rw-rw-r-- 1 dminh dminh 0 Apr  3 16:05 Minh.txt
```

# Raise error, default value option

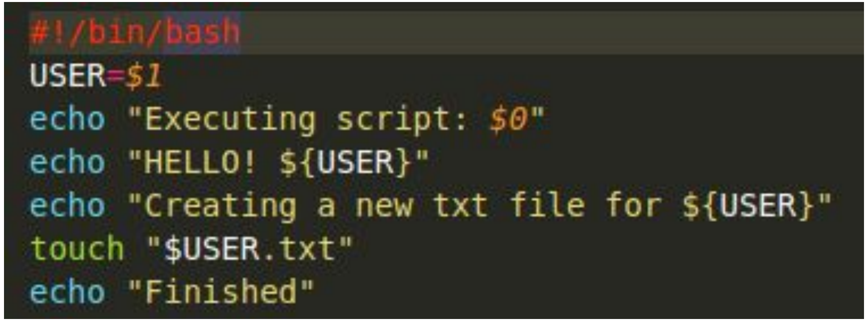


```bash
#!/bin/bash
NAME=${1?Error: No name given}
NAME2=${2:-everyone}
echo "HELLO! $NAME and $NAME2"
```

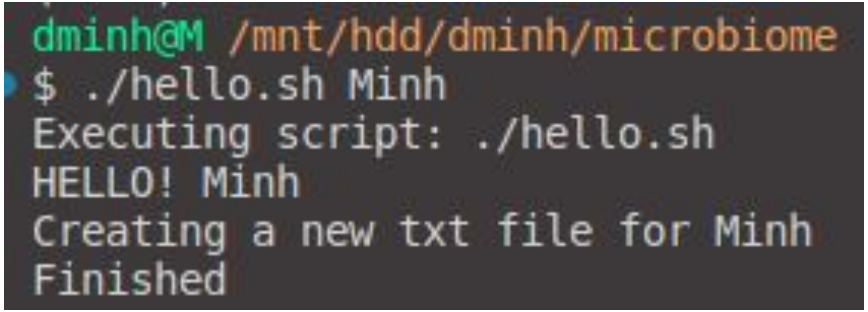


```
dminh@M /mnt/hdd/dminh/microbiome
$ ./hello.sh
./hello.sh: line 2: 1: Error: No name given
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ./hello.sh Minh
HELLO! Minh and everyone
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ./hello.sh Minh Duy
HELLO! Minh and Duy
```

```
#!/bin/bash

USER=$1

echo "Executing script: $0"

echo "HELLO! $USER"

echo “Creating a new txt file for $USER”

touch “$USER.txt”

echo “Finished”
```

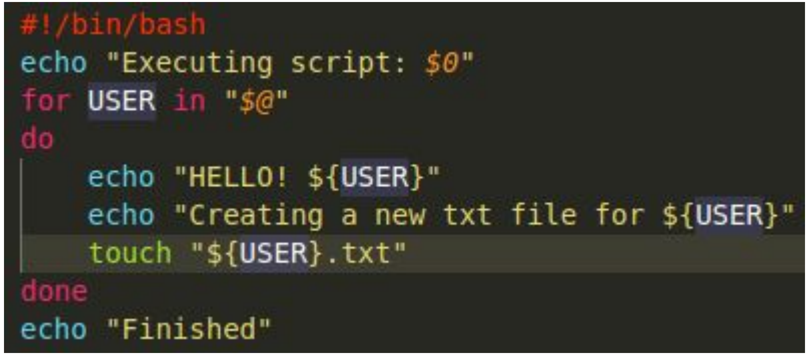


```
#!/bin/bash
USER=$1
echo "Executing script: $0"
echo "HELLO! ${USER}"
echo "Creating a new txt file for ${USER}"
touch "$USER.txt"
echo "Finished"
```

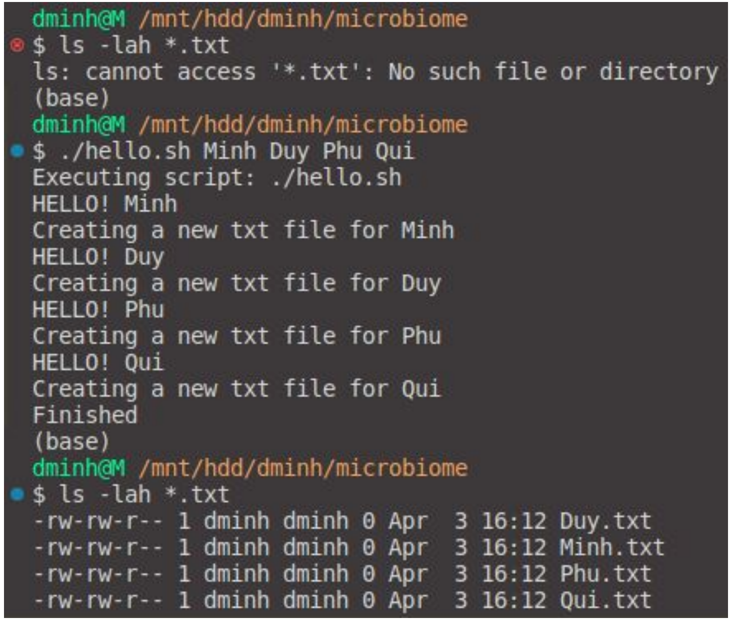


```
dminh@M /mnt/hdd/dminh/microbiome
$ ./hello.sh Minh
Executing script: ./hello.sh
HELLO! Minh
Creating a new txt file for Minh
Finished
```

```
#!/bin/bash

echo "Executing script: $0"

for USER in "$@"

do

    echo "HELLO! ${USER}"

    echo "Creating a new txt file for ${USER}"

    touch "${USER}.txt"

done

echo "Finished"
```



```
#!/bin/bash
echo "Executing script: $0"
for USER in "$@"
do
    echo "HELLO! ${USER}"
    echo "Creating a new txt file for ${USER}"
    touch "${USER}.txt"
done
echo "Finished"
```



```
dminh@M /mnt/hdd/dminh/microbiome
$ ls -lah *.txt
ls: cannot access '*.txt': No such file or directory
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ./hello.sh Minh Duy Phu Qui
Executing script: ./hello.sh
HELLO! Minh
Creating a new txt file for Minh
HELLO! Duy
Creating a new txt file for Duy
HELLO! Phu
Creating a new txt file for Phu
HELLO! Qui
Creating a new txt file for Qui
Finished
(base)
dminh@M /mnt/hdd/dminh/microbiome
$ ls -lah *.txt
-rw-rw-r-- 1 dminh dminh 0 Apr  3 16:12 Duy.txt
-rw-rw-r-- 1 dminh dminh 0 Apr  3 16:12 Minh.txt
-rw-rw-r-- 1 dminh dminh 0 Apr  3 16:12 Phu.txt
-rw-rw-r-- 1 dminh dminh 0 Apr  3 16:12 Qui.txt
```

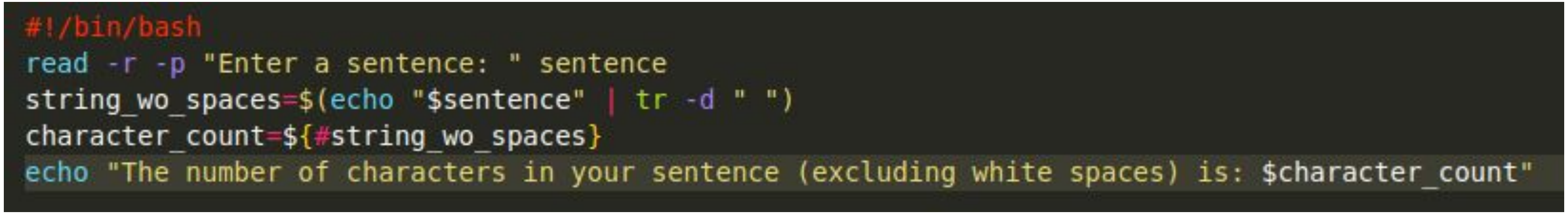
# Accepting User Input (STDIN)

The read command accepts STDIN.

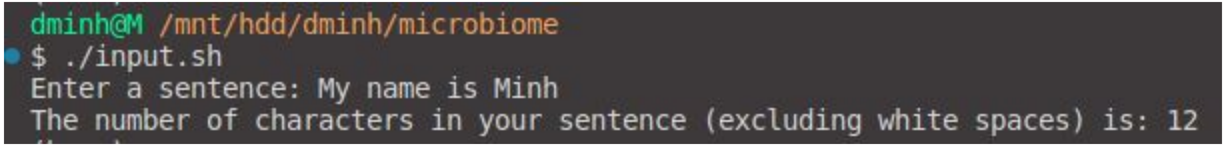
Syntax:

```
	read -p "PROMPT" VARIABLE
```

```
#!/bin/bash

read -r -p "Enter a sentence: " sentence

string_wo_spaces=$(echo "$sentence" | tr -d " ")

character_count=${#string_wo_spaces}

echo "The number of characters in your sentence (excluding white spaces) is: $character_count"
```



```
#!/bin/bash
read -r -p "Enter a sentence: " sentence
string_wo_spaces=$(echo "$sentence" | tr -d " ")
character_count=${#string_wo_spaces}
echo "The number of characters in your sentence (excluding white spaces) is: $character_count"
```



```
dminh@M /mnt/hdd/dminh/microbiome
$ ./input.sh
Enter a sentence: My name is Minh
The number of characters in your sentence (excluding white spaces) is: 12
```

# Summary

```
#!/path/to/interpreter

VARIABLE_NAME="Value"

$VARIABLE_NAME

${VARIABLE_NAME}

VARIABLE_NAME=$(command)
```

# if/else statement

```
if [ condition1-is-true ]

then

      Command 1

elif [ condition2-is-true ]

then

      Command 2

else

      Command 3

fi
```

# For Loop

```
for VARIABLE_NAME in ITEM_1 ITEM_N

do

        command 1

        command 2

        command N

done
```

# Parameters input

- Positional Parameters:
  - $0, $1, $2 … $9
  - $@
- Comments start with #.
- Use read to accept input.

# Exit Status

# Contents

- How to check the exit status of a command.
- How to make decisions based on the status.
- How to use exit statuses in your own scripts.

# Exit Status / Return Code

- Every command returns an exit status
- Range from 0 to 255
- 0 = success
- Other than 0 = error condition
- Use for error checking
- Use man or info to find meaning of exit status

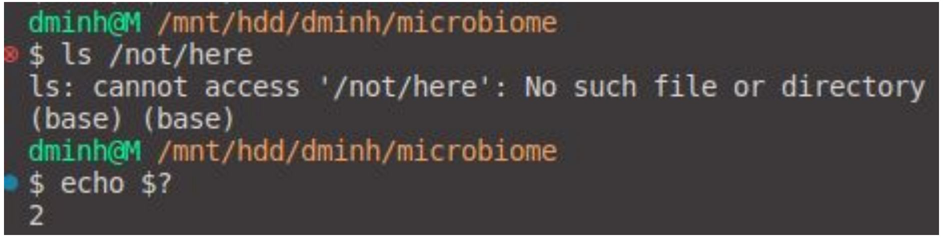
# Checking the Exit Status

- $? contains the return code of the previously executed command.

ls /not/here

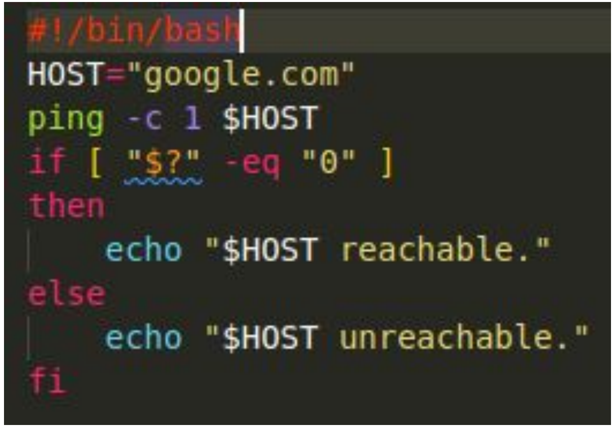
echo "$?"

Output: ?

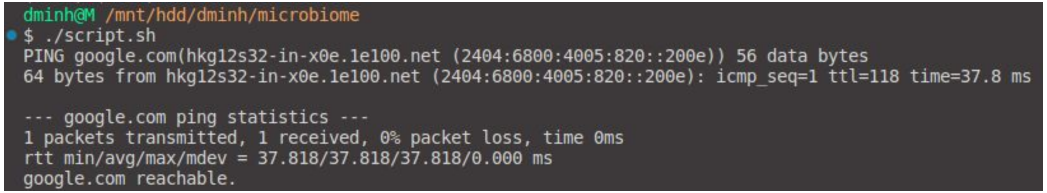


```
dminh@M /mnt/hdd/dminh/microbiome
$ ls /not/here
ls: cannot access '/not/here': No such file or directory
(base) (base)
dminh@M /mnt/hdd/dminh/microbiome
$ echo $?
2
```

```
HOST="google.com"

ping -c 1 $HOST

if [ "$?" -eq "0" ]

then

        echo "$HOST reachable."

else

        echo "$HOST unreachable."

fi
```

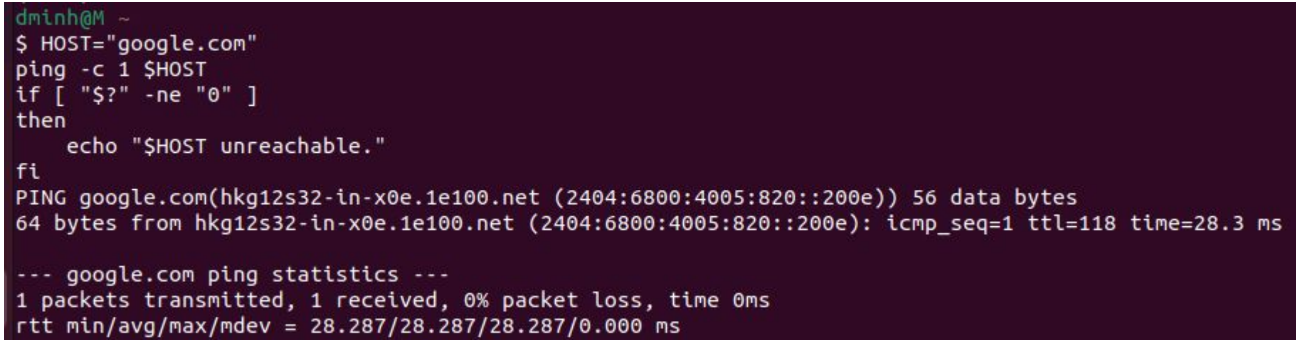


```
#!/bin/bash
HOST="google.com"
ping -c 1 $HOST
if [ "$?" -eq "0" ]
then
    echo "$HOST reachable."
else
    echo "$HOST unreachable."
fi
```



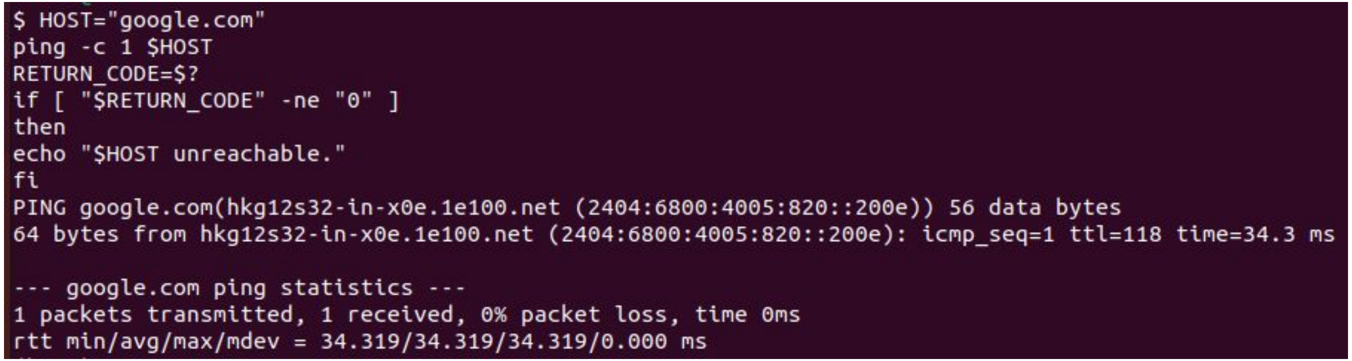
```
dminh@M /mnt/hdd/dminh/microbiome
$ ./script.sh
PING google.com(hkg12s32-in-x0e.1e100.net (2404:6800:4005:820::200e)) 56 data bytes
64 bytes from hkg12s32-in-x0e.1e100.net (2404:6800:4005:820::200e): icmp_seq=1 ttl=118 time=37.8 ms

--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 37.818/37.818/37.818/0.000 ms
google.com reachable.
```

```
HOST="google.com"

ping -c 1 $HOST

if [ "$?" -ne "0" ]

then

        echo "$HOST unreachable."

fi
```



```
dminh@M ~
$ HOST="google.com"
ping -c 1 $HOST
if [ "$?" -ne "0" ]
then
    echo "$HOST unreachable."
fi
PING google.com(hkg12s32-in-x0e.1e100.net (2404:6800:4005:820::200e)) 56 data bytes
64 bytes from hkg12s32-in-x0e.1e100.net (2404:6800:4005:820::200e): icmp_seq=1 ttl=118 time=28.3 ms

--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 28.287/28.287/28.287/0.000 ms
```

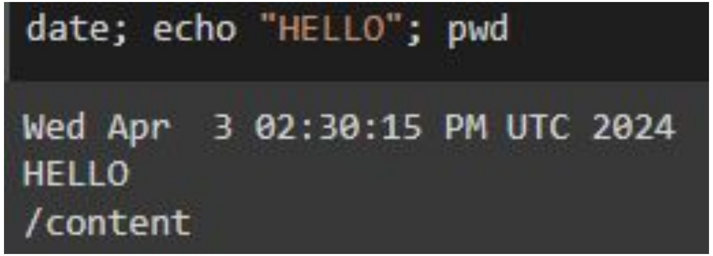
```
HOST="google.com"

ping -c 1 $HOST

RETURN_CODE=$?

if [ "$RETURN_CODE" -ne "0" ]

then

        echo "$HOST unreachable."

fi
```



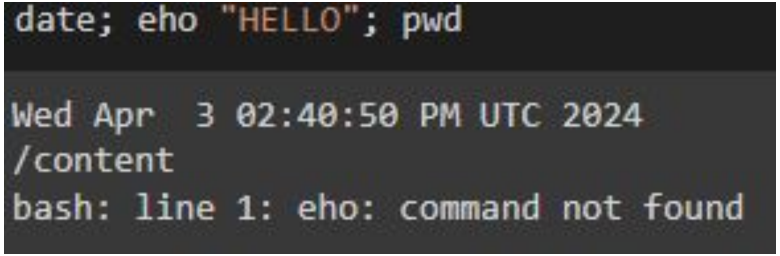
```
$ HOST="google.com"
ping -c 1 $HOST
RETURN_CODE=$?
if [ "$RETURN_CODE" -ne "0" ]
then
echo "$HOST unreachable."
fi
PING google.com(hkg12s32-in-x0e.1e100.net (2404:6800:4005:820::200e)) 56 data bytes
64 bytes from hkg12s32-in-x0e.1e100.net (2404:6800:4005:820::200e): icmp_seq=1 ttl=118 time=34.3 ms

--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 34.319/34.319/34.319/0.000 ms
```

# Semicolon (;) operator

- The Semicolon (;) operator: execute all commands



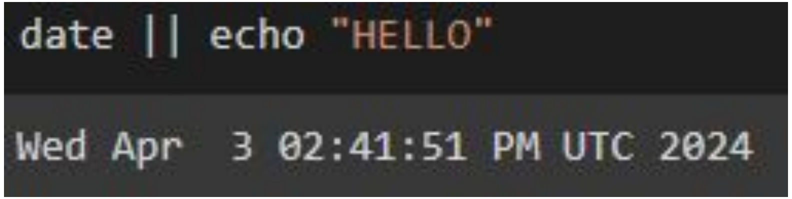
```
date; echo "HELLO"; pwd

Wed Apr  3 02:30:15 PM UTC 2024
HELLO
/content
```



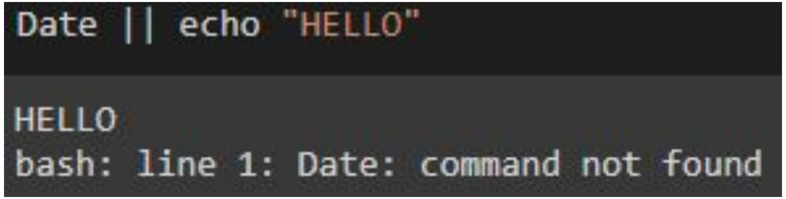
```
date; eho "HELLO"; pwd

Wed Apr  3 02:40:50 PM UTC 2024
/content
bash: line 1: eho: command not found
```

# The logical OR (||) operator

- It execute only *one command between the two*.
- Command B will only execute if command A fails and vice versa.
- Syntax: command A || command B



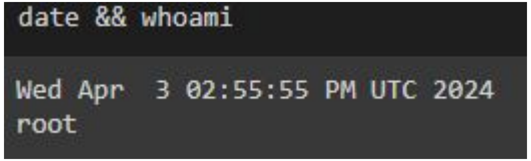
```
date || echo "HELLO"

Wed Apr  3 02:41:51 PM UTC 2024
```



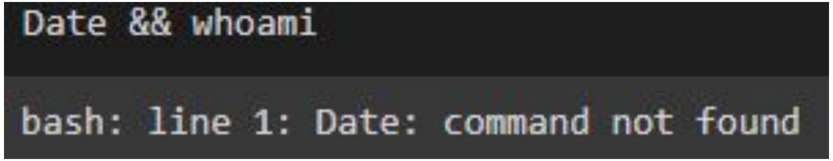
```
Date || echo "HELLO"

HELLO
bash: line 1: Date: command not found
```

# The logical AND (&&) operator

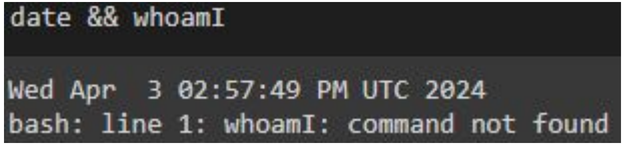
- With AND (&&) operator, the second command only runs if the first command is successful
- Syntax: command A && command B



```
date && whoami

Wed Apr  3 02:55:55 PM UTC 2024
root
```



```
Date && whoami

bash: line 1: Date: command not found
```

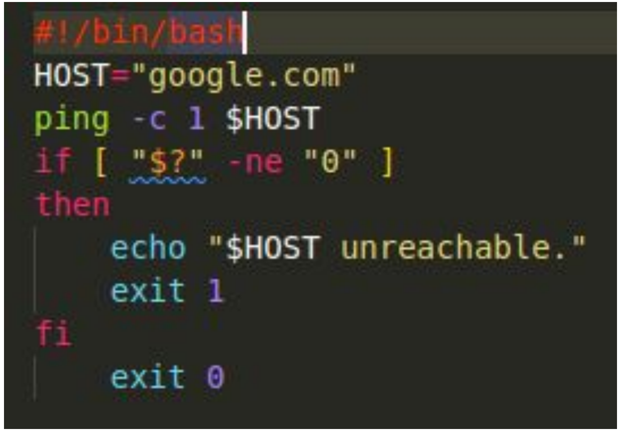


```
date && whoamI

Wed Apr  3 02:57:49 PM UTC 2024
bash: line 1: whoamI: command not found
```

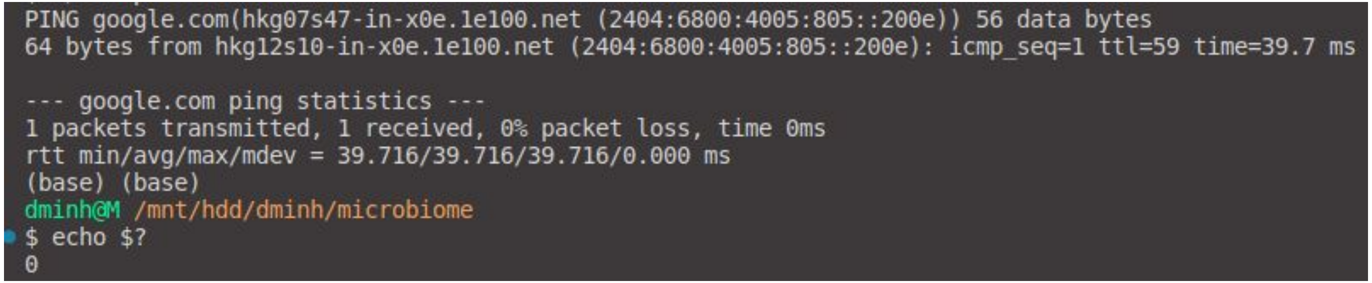
# Exit Command

- Explicitly define the return code
  - exit 0
  - exit 1
  - exit 2
  - exit 255
  - etc…
- The default value is that of the last command executed.

```
#!/bin/bash

HOST="google.com"

ping -c 1 $HOST

if [ "$?" -ne "0" ]

then

        echo "$HOST unreachable."

        exit 1

fi

exit 0
```



```
#!/bin/bash
HOST="google.com"
ping -c 1 $HOST
if [ "$?" -ne "0" ]
then
    echo "$HOST unreachable."
    exit 1
fi
    exit 0
```



```
PING google.com(hkg07s47-in-x0e.1e100.net (2404:6800:4005:805::200e)) 56 data bytes
64 bytes from hkg12s10-in-x0e.1e100.net (2404:6800:4005:805::200e): icmp_seq=1 ttl=59 time=39.7 ms

--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 39.716/39.716/39.716/0.000 ms
(base) (base)
dminh@M /mnt/hdd/dminh/microbiome
$ echo $?
0
```

# Summary

- All command return an exit status
- 0 - 255
- 0 = success
- Other than 0 = error condition
- $? contains the exit status
- Decision making - if, &&,||
- exit

# Shell Functions

# Contents

- Why to use functions
- How to create them
- How to use them
- Variable scope
- Function Parameters
- Exit statuses and return codes.

# Why use functions ? (Keep it DRY!)

- Don't repeat yourself! Don't repeat yourself!
- Write once, use many times.
- Reduces script length.
- Single place to edit and troubleshoot.
- Easier to maintain.

# Functions

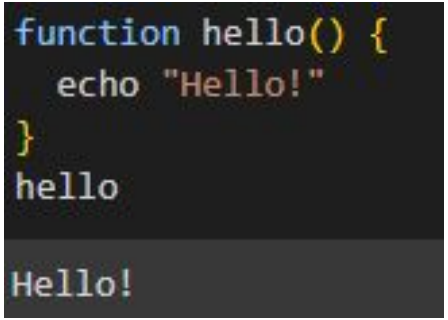
- If you're repeating yourself, use a function
- Reusable code
- Must be defined before use
- Has parameter support.

# Creating a function

```
function function-name() {
        # Code goes here.
}


function-name () {
        # Code goes here.
}
```

# Calling a function

```
#!/bin/bash

function hello() {

        echo "Hello!"

}

hello
```

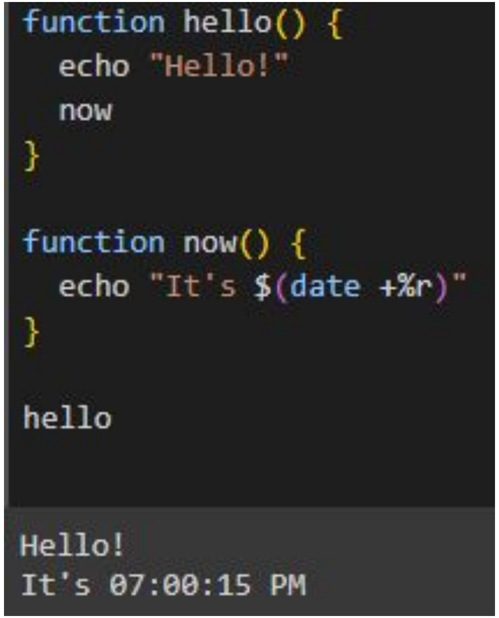


```
function hello() {
  echo "Hello!"
}
hello
```

```
Hello!
```

# Functions can call other functions

```
#!/bin/bash
function hello() {
        echo "Hello!"
        now
}
function now() {
        echo "It's $(date +%r)"
}
hello
```

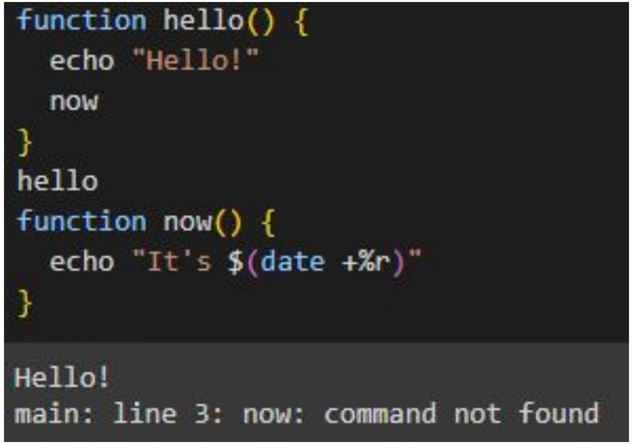


```
function hello() {
  echo "Hello!"
  now
}

function now() {
  echo "It's $(date +%r)"
}

hello
```

```
Hello!
It's 07:00:15 PM
```

# Do NOT do this…

```
#!/bin/bash
function hello() {
	echo "Hello!"
	now
}
hello
function now() {
	echo "It's $(date +%r)"
}
```



```
function hello() {
  echo "Hello!"
  now
}
hello
function now() {
  echo "It's $(date +%r)"
}
```

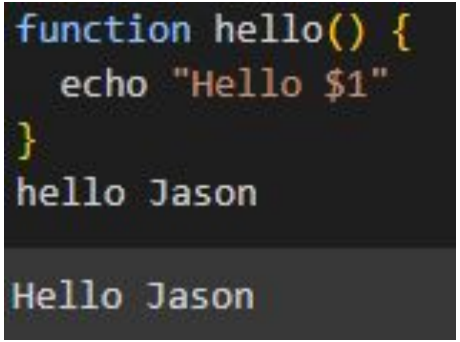
```
Hello!
main: line 3: now: command not found
```

# Positional Parameters

- Functions can accept parameters.
- The first parameter is stored in $1.
- The second parameter is stored in $2, etc.
- $@ contains all of the parameters.
- Just like shell scripts.
    - $0 = the script itself, not function name.

# Positional Parameters

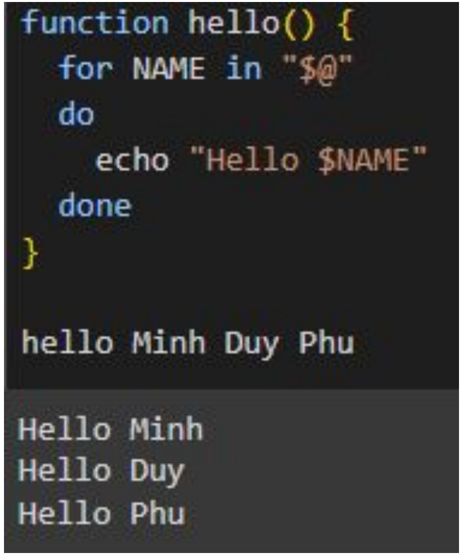
```
#!/bin/bash

function hello() {

      echo "Hello $1"

}

hello Jason
```



```
function hello() {
  echo "Hello $1"
}
hello Jason
```

```
Hello Jason
```

```
#!/bin/bash

function hello() {

        for NAME in $@

        do

                echo "Hello $NAME"

        done

}

hello Minh Duy Phu
```



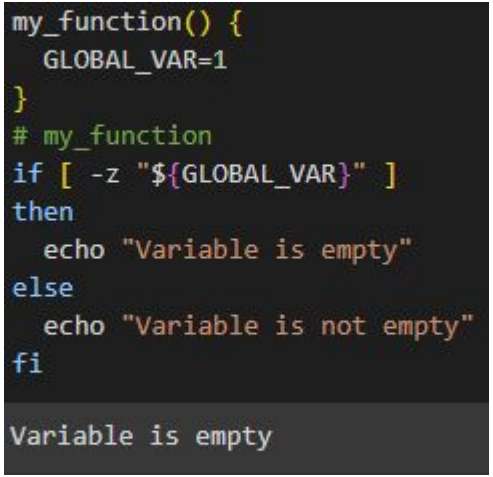
```
function hello() {
  for NAME in "$@"
  do
    echo "Hello $NAME"
  done
}

hello Minh Duy Phu
```

```
Hello Minh
Hello Duy
Hello Phu
```

# Variable Scope

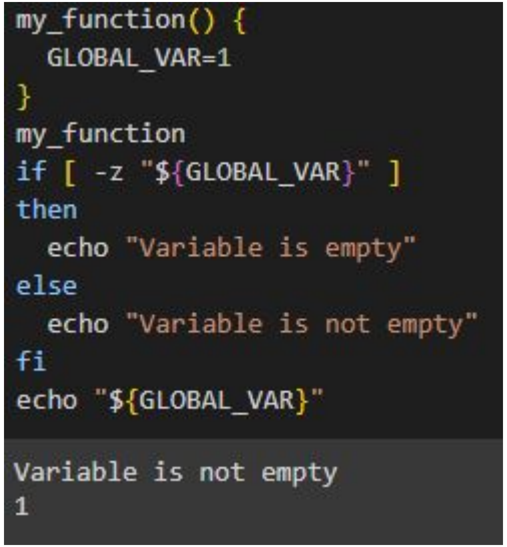
- By default, variables are global
- Variables have to be defined before used.

```
#!/bin/bash

my_function() {

GLOBAL_VAR=1

}

# GLOBAL_VAR not available yet.

echo $GLOBAL_VAR

my_function

# GLOBAL_VAR is NOW available.

echo $GLOBAL_VAR
```



```
my_function() {
  GLOBAL_VAR=1
}
# my_function
if [ -z "${GLOBAL_VAR}" ]
then
  echo "Variable is empty"
else
  echo "Variable is not empty"
fi
```

```
Variable is empty
```



```
my_function() {
  GLOBAL_VAR=1
}
my_function
if [ -z "${GLOBAL_VAR}" ]
then
  echo "Variable is empty"
else
  echo "Variable is not empty"
fi
echo "${GLOBAL_VAR}"
```

```
Variable is not empty
1
```

# Local Variables

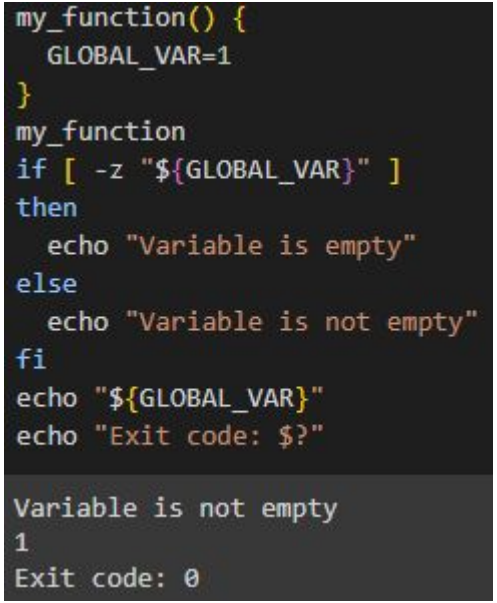
- Can only be accessed within the function.
- Create using the local keyword.
  - **local** LOCAL_VAR=1
- Only functions can have local variables.
- Best practice to keep variables local in functions.

# Exit Status (Return Codes)

- Functions have an exit status
- Explicitly
  - **return** <RETURN_CODE>
- Implicity
  - The exit status of the last command executed in the function

# Exit Status (Return Codes)

- Valid exit codes range from 0 to 255
- 0 = success
- $? = the exit status

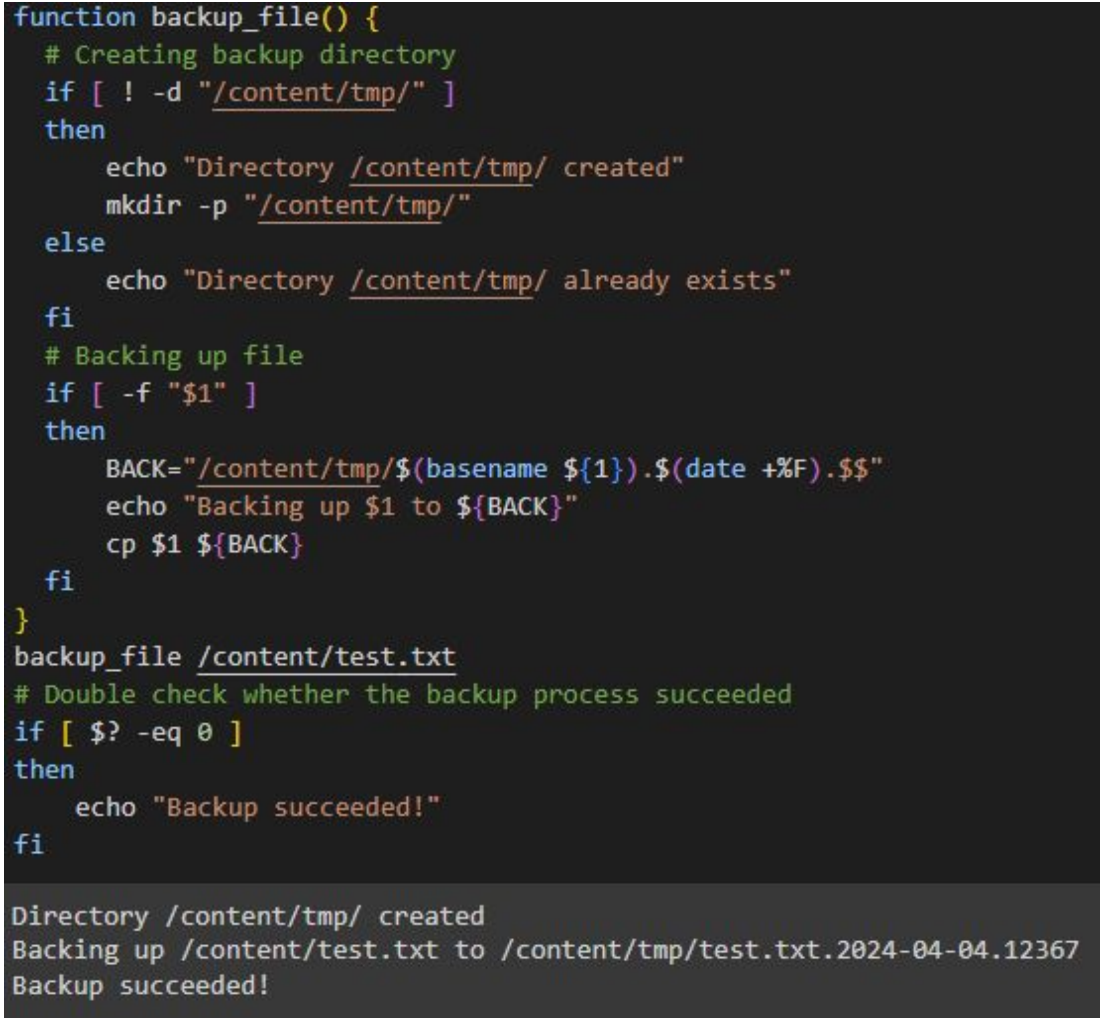


```
my_function() {
  GLOBAL_VAR=1
}
my_function
if [ -z "${GLOBAL_VAR}" ]
then
  echo "Variable is empty"
else
  echo "Variable is not empty"
fi
echo "${GLOBAL_VAR}"
echo "Exit code: $?"
```

```
Variable is not empty
1
Exit code: 0
```

Example:

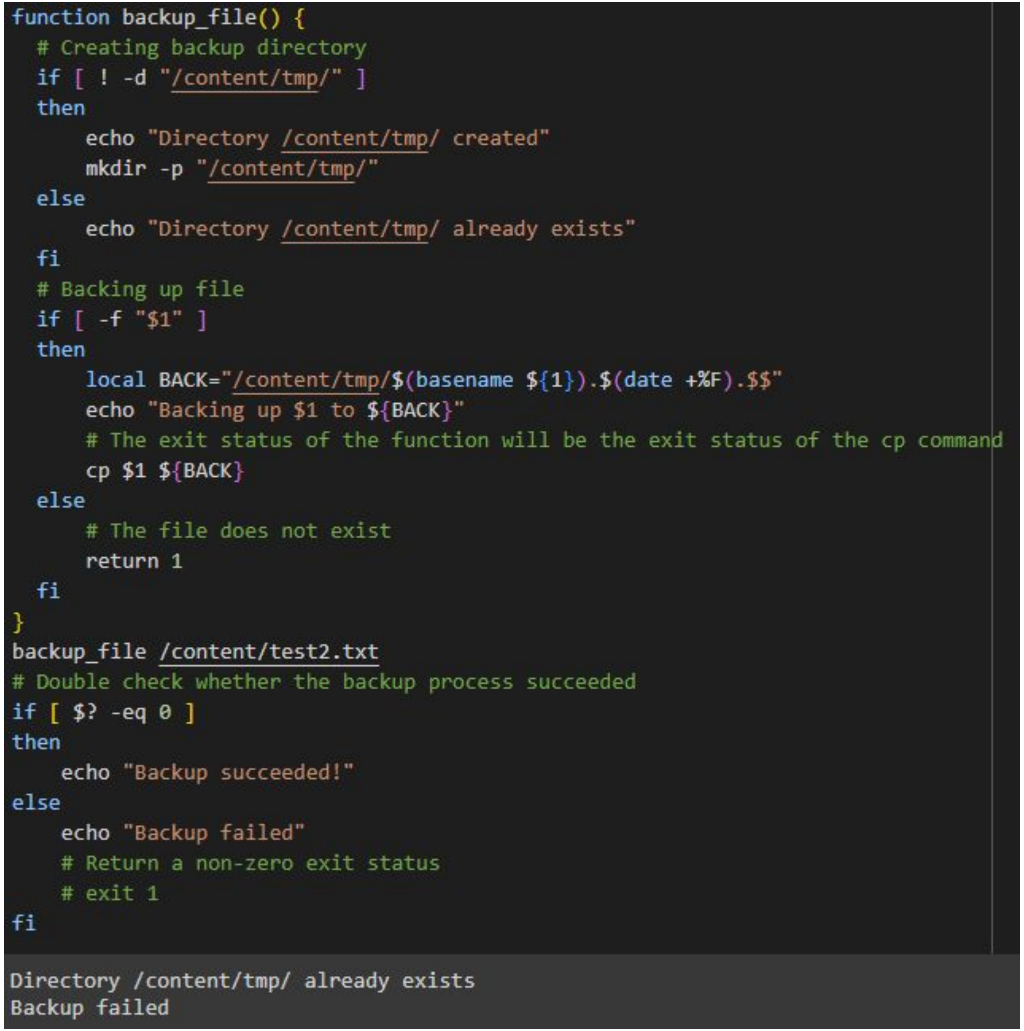
## Successful backup



```
function backup_file() {
  # Creating backup directory
  if [ ! -d "/content/tmp/" ]
  then
      echo "Directory /content/tmp/ created"
      mkdir -p "/content/tmp/"
  else
      echo "Directory /content/tmp/ already exists"
  fi
  # Backing up file
  if [ -f "$1" ]
  then
      BACK="/content/tmp/$(basename ${1}).$(date +%F).$$"
      echo "Backing up $1 to ${BACK}"
      cp $1 ${BACK}
  fi
}
backup_file /content/test.txt
# Double check whether the backup process succeeded
if [ $? -eq 0 ]
then
    echo "Backup succeeded!"
fi
```

```
Directory /content/tmp/ created
Backing up /content/test.txt to /content/tmp/test.txt.2024-04-04.12367
Backup succeeded!
```

Example:
# Fail backup



```
function backup_file() {
  # Creating backup directory
  if [ ! -d "/content/tmp/" ]
  then
      echo "Directory /content/tmp/ created"
      mkdir -p "/content/tmp/"
  else
      echo "Directory /content/tmp/ already exists"
  fi
  # Backing up file
  if [ -f "$1" ]
  then
      local BACK="/content/tmp/$(basename ${1}).$(date +%F).$$"
      echo "Backing up $1 to ${BACK}"
      # The exit status of the function will be the exit status of the cp command
      cp $1 ${BACK}
  else
      # The file does not exist
      return 1
  fi
}
backup_file /content/test2.txt
# Double check whether the backup process succeeded
if [ $? -eq 0 ]
then
    echo "Backup succeeded!"
else
    echo "Backup failed"
    # Return a non-zero exit status
    # exit 1
fi
```

```
Directory /content/tmp/ already exists
Backup failed
```

# Summary

- DRY
- Global and local variables
- Parameters
- Exit statuses

# Shell Script Order

1. Shebang
2. Comments / file header
3. Global variables
4. Functions
   - Use local variables
5. Main script contents
6. Exit with an exit status
   - exit <STATUS> at various exit points

# Introduction to Wildcards

# Contents

- What wildcards are.
- When and where they can be used.
- The different types of wildcards.
- How to use wildcards with various commands.

# Wildcards

- A character or string used for pattern matching.
- Globbing expands the wildcard pattern into a list of files and/or directories. (paths)
- Wildcards can be used with most commands.
    - ls
    - rm
    - cp

# Wildcards

- ***** - matches zero or more characters.
  - *.txt
  - a*
  - a*.txt
- **?** - matches exactly one character
  - ?.txt
  - a?
  - a?.txt

# More Wildcards - Character Classes

- [ ] - A character class.
  - Matches any of the characters included between the brackets. Matches exactly **one** character.
  - [aeiou]
  - ca[nt]*
    - can
    - cat
    - candy
    - catch

# More Wildcards - Character Classes

- [ ! ] - Matches any of the characters **NOT** included between the brackets. Matches exactly **one** character.
    - [!aeiou]*
        - baseball
        - cricket

# More Wildcards - Ranges

- Use two characters separated by a hyphen to create a range in a character class.
- [a-g]*
    - Matches all files that start with a, b, c, d, e, f, or g.
- [3-6]*
    - Matches all files that start with 3, 4, 5 or 6.

# Named Character Classes

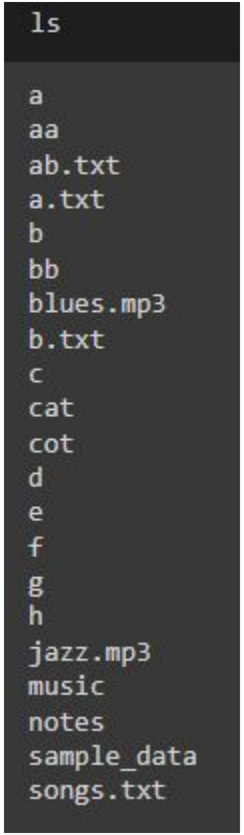
- [[:alpha:]]: alphabetic letters (lower + upper case letters)
- [[:alnum:]]: alphanumeric characters (alpha + digits)
- [[:digit:]]: numbers and decimal from 0 to 9
- [[:lower:]]: any lowercase letters
- [[:space:]]: wide space (spaces, tabs, newline characters)
- [[:upper:]]: any uppercase letters

# Matching Wildcard patterns

- \ - escape character. Use if you want to match a wildcard character.
    - Match all files that end with a question mark:
        - *\?

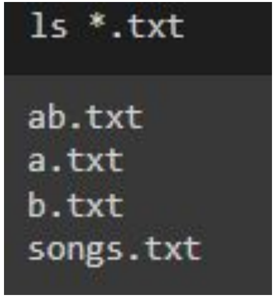
          - done?

# Examples(1)



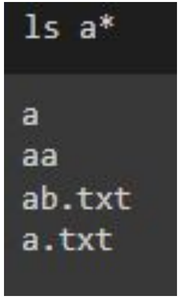
```
ls
```

```
a
aa
ab.txt
a.txt
b
bb
blues.mp3
b.txt
c
cat
cot
d
e
f
g
h
jazz.mp3
music
notes
sample_data
songs.txt
```



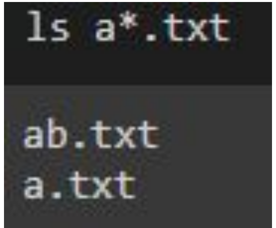
```
ls *.txt
```

```
ab.txt
a.txt
b.txt
songs.txt
```



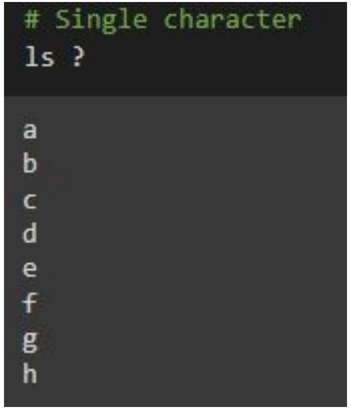
```
ls a*
```

```
a
aa
ab.txt
a.txt
```



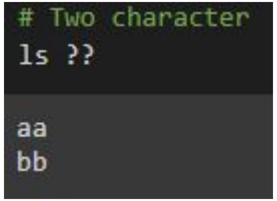
```
ls a*.txt
```

```
ab.txt
a.txt
```



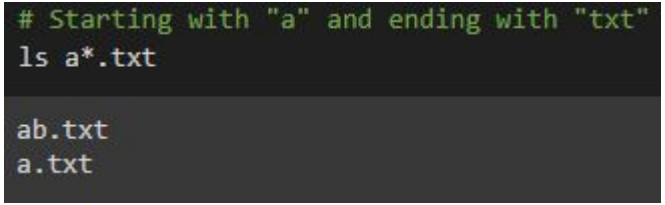
```
# Single character
ls ?
```

```
a
b
c
d
e
f
g
h
```



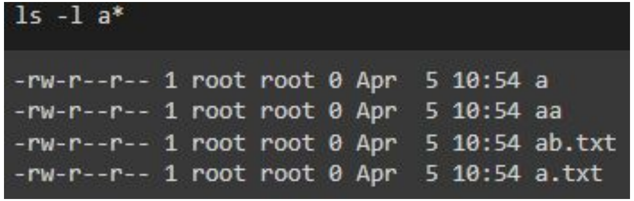
```
# Two character
ls ??
```

```
aa
bb
```



```
# Starting with "a" and ending with "txt"
ls a*.txt
```

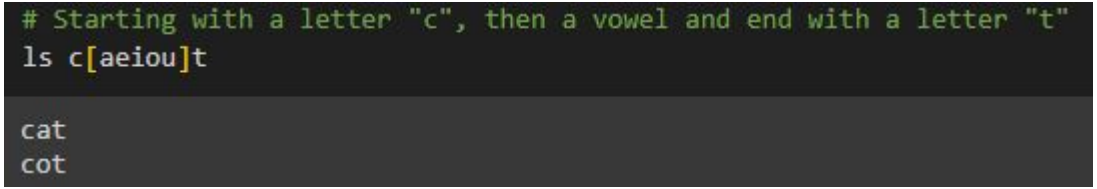
```
ab.txt
a.txt
```



```
ls -l a*
```

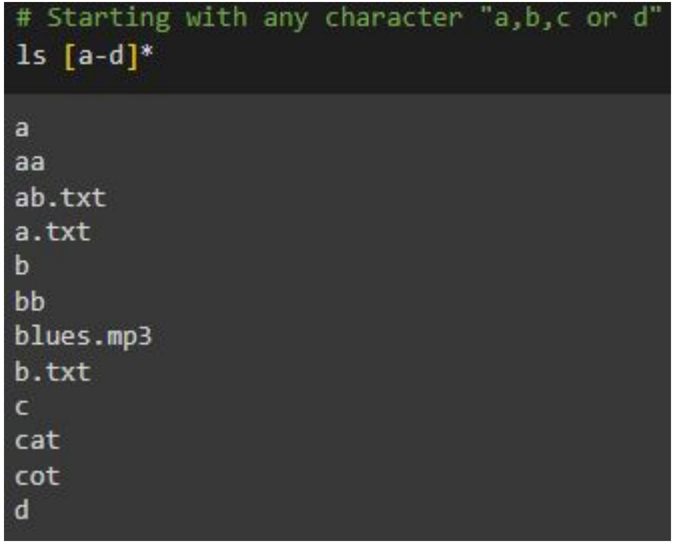
```
-rw-r--r-- 1 root root 0 Apr  5 10:54 a
-rw-r--r-- 1 root root 0 Apr  5 10:54 aa
-rw-r--r-- 1 root root 0 Apr  5 10:54 ab.txt
-rw-r--r-- 1 root root 0 Apr  5 10:54 a.txt
```

# Examples(2)



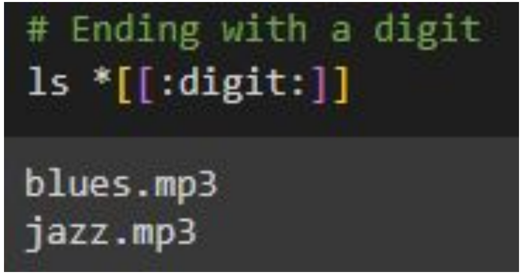
```
# Starting with a letter "c", then a vowel and end with a letter "t"
ls c[aeiou]t
```

```
cat
cot
```



```
# Starting with any character "a,b,c or d"
ls [a-d]*
```

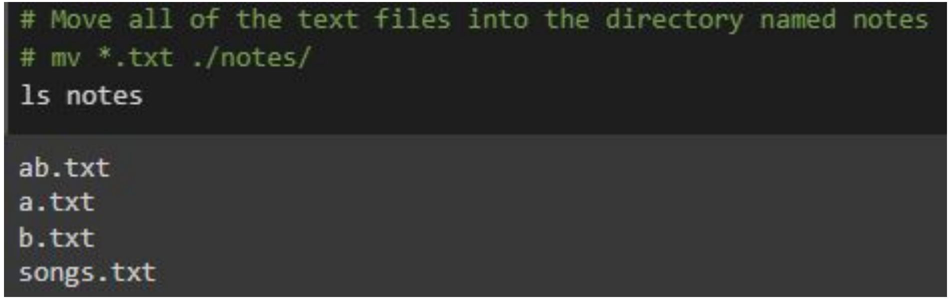
```
a
aa
ab.txt
a.txt
b
bb
blues.mp3
b.txt
c
cat
cot
d
```



```
# Ending with a digit
ls *[[:digit:]]
```

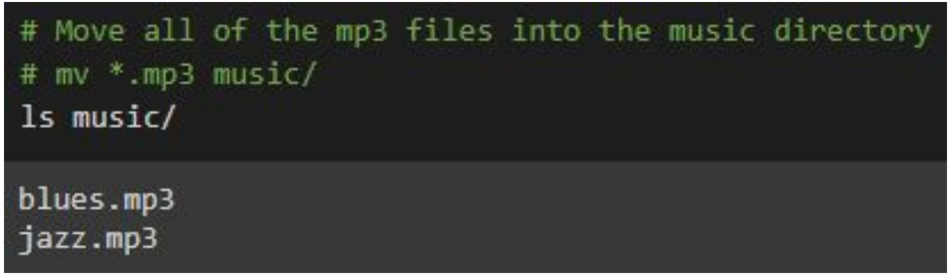
```
blues.mp3
jazz.mp3
```

# Examples(3)



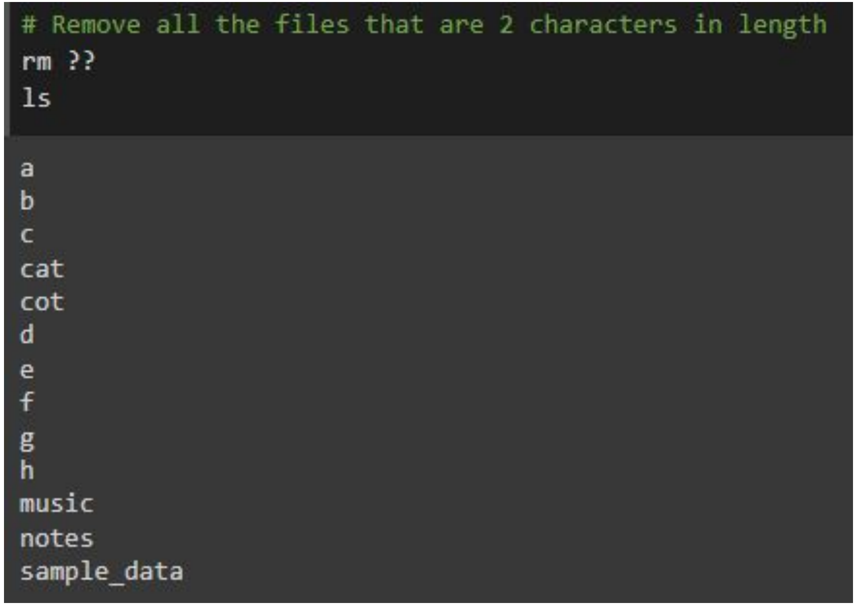
```
# Move all of the text files into the directory named notes
# mv *.txt ./notes/
ls notes
```

```
ab.txt
a.txt
b.txt
songs.txt
```



```
# Move all of the mp3 files into the music directory
# mv *.mp3 music/
ls music/
```

```
blues.mp3
jazz.mp3
```



```
# Remove all the files that are 2 characters in length
rm ??
ls
```

```
a
b
c
cat
cot
d
e
f
g
h
music
notes
sample_data
```

# Wildcards in shell scripts

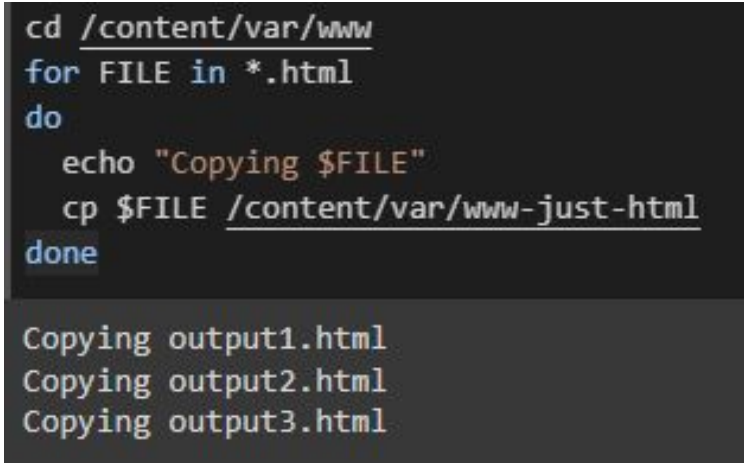
- Wildcards are great when you want to work on a group of files or directories.

# Just like a regular command line

```
#!/bin/bash

cd /var/www

cp *.html /var/www-just-html
```

# In a for loop

```
#!/bin/bash

cd /var/www

for FILE in *.html

do

        echo "Copying $FILE"

        cp $FILE /var/www-just-html

done
```

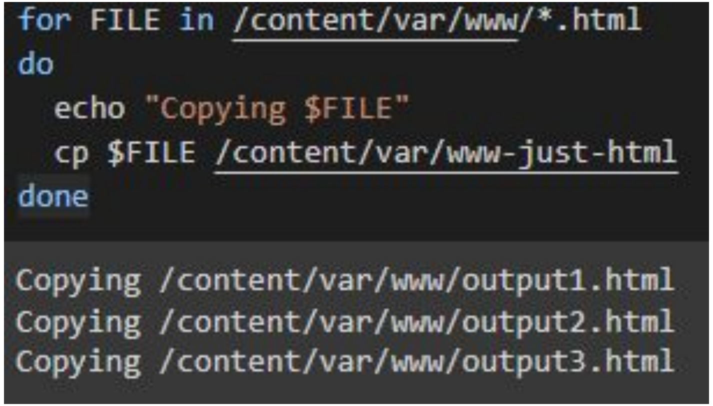


```
cd /content/var/www
for FILE in *.html
do
  echo "Copying $FILE"
  cp $FILE /content/var/www-just-html
done
```

```
Copying output1.html
Copying output2.html
Copying output3.html
```

# In a for loop

```
#!/bin/bash

for FILE in /var/www/*.html

do

        echo "Copying $FILE"

        cp $FILE /var/www-just-html

done
```



```
for FILE in /content/var/www/*.html
do
  echo "Copying $FILE"
  cp $FILE /content/var/www-just-html
done
```

```
Copying /content/var/www/output1.html
Copying /content/var/www/output2.html
Copying /content/var/www/output3.html
```

# Notes

- Just like on the command line.
- In loops
- Supply a directory in the wildcard or use the cd command to change the current directory

# Summary

- *
- ?
- [ ]
- [0-3]
- [[:digit:]]

# Case Statements

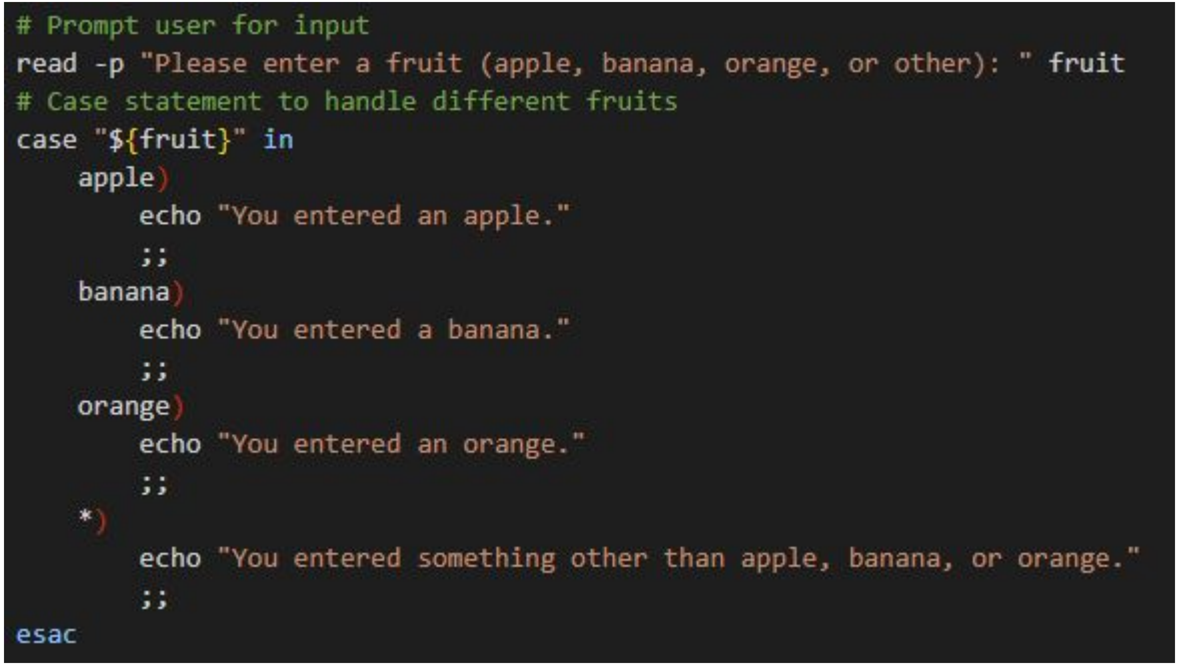
# Case Statements

- Alternative to if statements
  - if [ "$VAR" = "one" ]
  - elif [ "$VAR" = "two" ]
  - elif [ "$VAR" = "three" ]
  - elif [ "$VAR" = “four" ]
- May be easier to read than complex if statements.

# Syntax

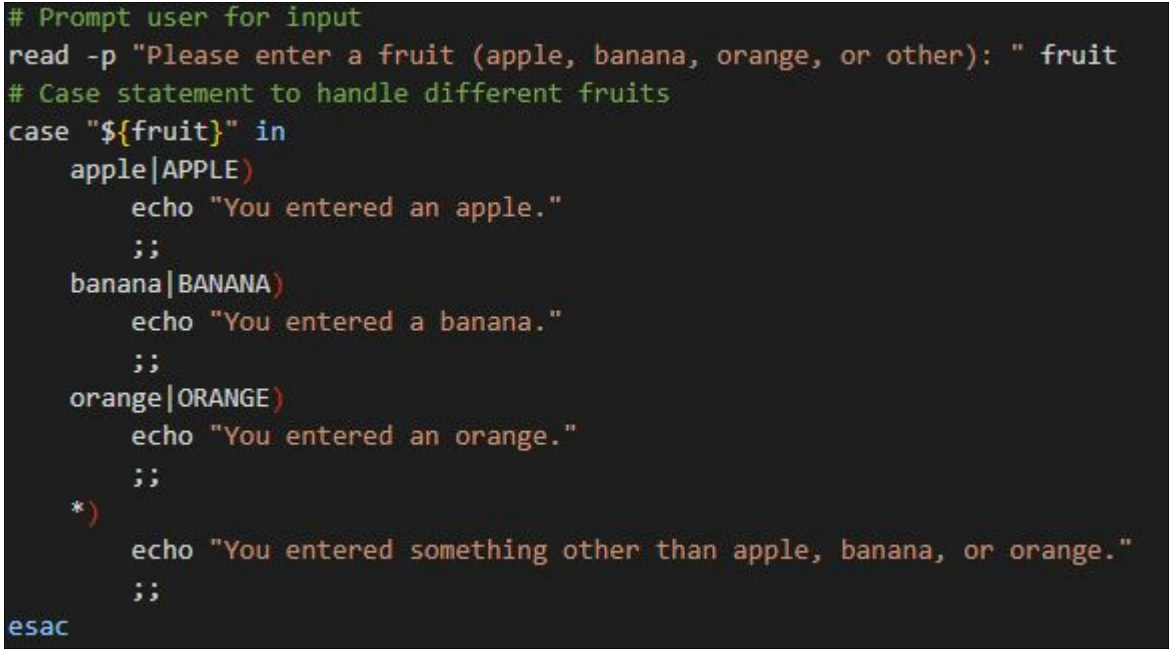
```
case "$VAR" in
    pattern_1)
        # Commands go here.
        ;;
    pattern_N)
        # Commands go here.
        ;;
esac
```

```
#!/bin/bash

# Prompt user for input

read -p "Please enter a fruit (apple, banana, orange, or other): " fruit

# Case statement to handle different fruits

case "$fruit" in

    apple)

        echo "You entered an apple."

        ;;

    banana)

        echo "You entered a banana."

        ;;

    orange)

        echo "You entered an orange."

        ;;

    *)

        echo "You entered something other than apple, banana, or orange."

        ;;

esac
```



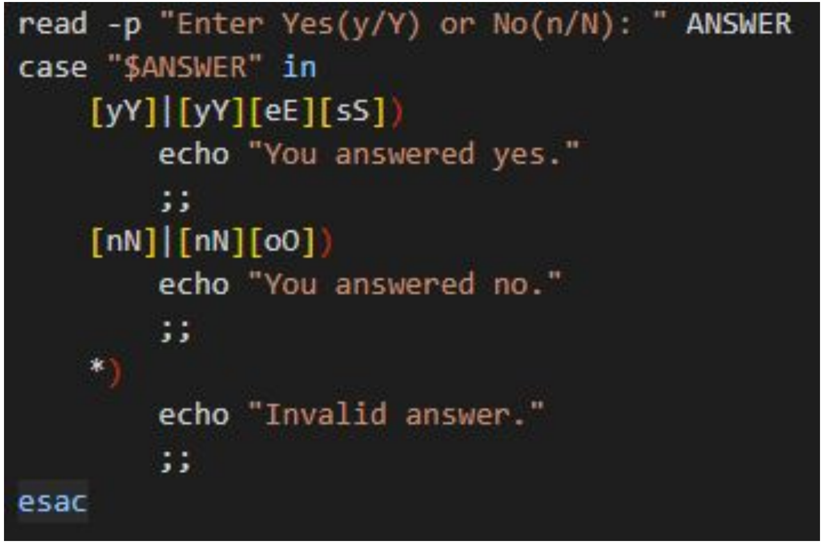
```
# Prompt user for input
read -p "Please enter a fruit (apple, banana, orange, or other): " fruit
# Case statement to handle different fruits
case "${fruit}" in
    apple)
        echo "You entered an apple."
        ;;
    banana)
        echo "You entered a banana."
        ;;
    orange)
        echo "You entered an orange."
        ;;
    *)
        echo "You entered something other than apple, banana, or orange."
        ;;
esac
```

```
#!/bin/bash
# Prompt user for input
read -p "Please enter a fruit (apple, banana, orange, or other): " fruit
# Case statement to handle different fruits
case "$fruit" in
  apple|APPLE)
    echo "You entered an apple."
    ;;
  banana|BANANA)
    echo "You entered a banana."
    ;;
  orange|ORANGE)
    echo "You entered an orange."
    ;;
  *)
    echo "You entered something other than apple, banana, or orange."
    ;;
esac
```

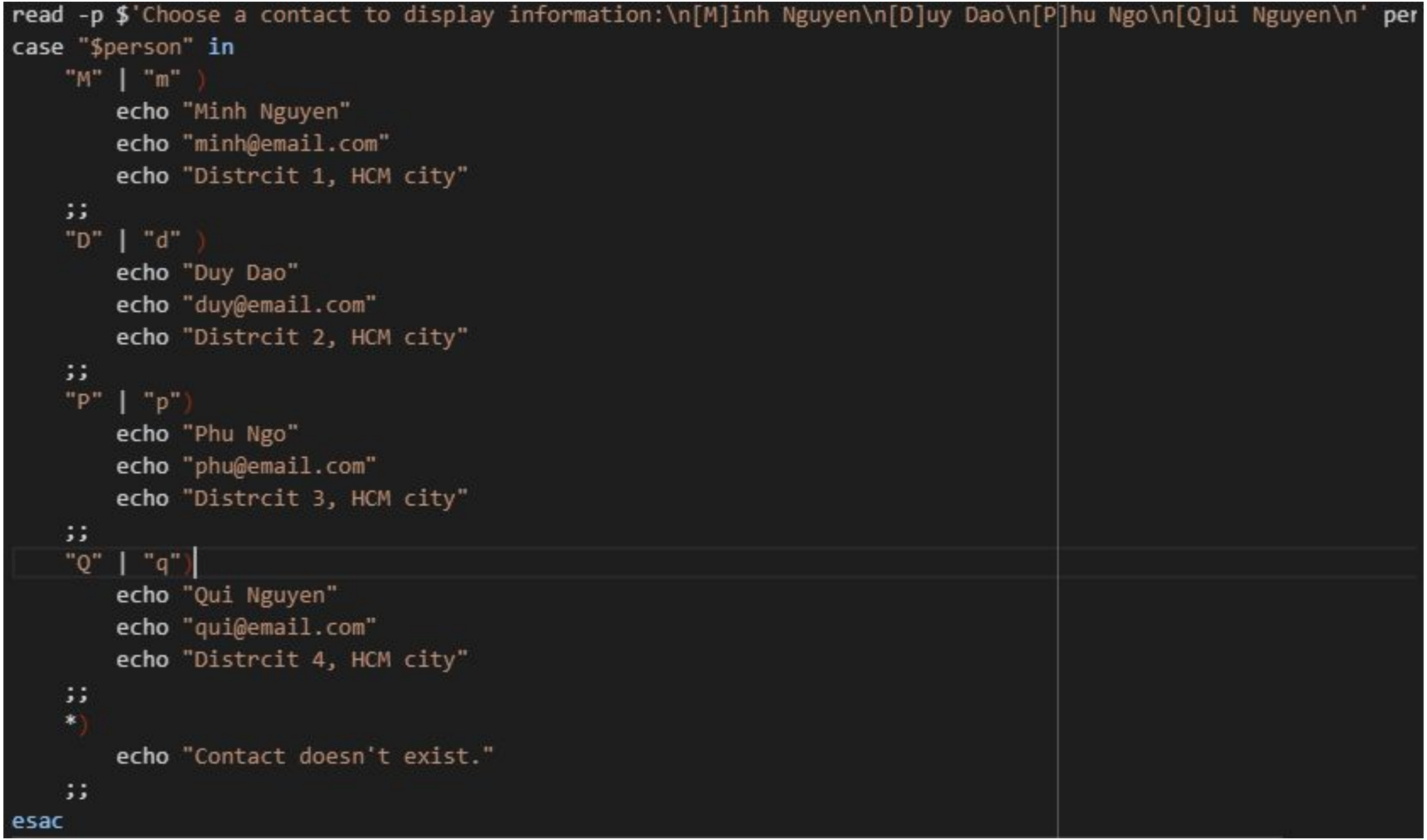


```
# Prompt user for input
read -p "Please enter a fruit (apple, banana, orange, or other): " fruit
# Case statement to handle different fruits
case "${fruit}" in
    apple|APPLE)
        echo "You entered an apple."
        ;;
    banana|BANANA)
        echo "You entered a banana."
        ;;
    orange|ORANGE)
        echo "You entered an orange."
        ;;
    *)
        echo "You entered something other than apple, banana, or orange."
        ;;
esac
```

```
read -p "Enter y or n: " ANSWER
case "$ANSWER" in
	[yY]|[yY][eE][sS])
		echo "You answered yes."
		;;
	[nN]|[nN][oO])
		echo "You answered no."
		;;
	*)
		echo "Invalid answer."
		;;
esac
```



```
read -p "Enter Yes(y/Y) or No(n/N): " ANSWER
case "$ANSWER" in
    [yY]|[yY][eE][sS])
        echo "You answered yes."
        ;;
    [nN]|[nN][oO])
        echo "You answered no."
        ;;
    *)
        echo "Invalid answer."
        ;;
esac
```



```
read -p $'Choose a contact to display information:\n[M]inh Nguyen\n[D]uy Dao\n[P]hu Ngo\n[Q]ui Nguyen\n' per
case "$person" in
    "M" | "m" )
        echo "Minh Nguyen"
        echo "minh@email.com"
        echo "Distrcit 1, HCM city"
    ;;
    "D" | "d" )
        echo "Duy Dao"
        echo "duy@email.com"
        echo "Distrcit 2, HCM city"
    ;;
    "P" | "p")
        echo "Phu Ngo"
        echo "phu@email.com"
        echo "Distrcit 3, HCM city"
    ;;
    "Q" | "q")
        echo "Qui Nguyen"
        echo "qui@email.com"
        echo "Distrcit 4, HCM city"
    ;;
    *)
        echo "Contact doesn't exist."
    ;;
esac
```

# Summary

- Can be used in place of if statements.
- Patterns can include wildcards.
- Multiple pattern matching using a pipe.

# Logging

# Contents

- Why log
- Syslog standard
- Generating log messages
- Custom logging functions

# Logging

- Logs are the who, what, when, where, and why.
- Output may scroll off the screen.
- Script may run unattended (via cron, etc.)

# Syslog

- The syslog standard uses facilities and severities to categorize messages.
- **Facilities**: kern, user, mail, daemon, auth, local0, local7
- **Severities**: emerg, alert, crit, err, warning, notice, info, debug
- Log file locations are configurable:
    - /var/log/messages
    - /var/log/syslog

# Logging with logger

- The logger utility
- By default creates user.notice messages.

*logger "Message"*

*logger -p local0.info "Message":* changing facilities/severities

*logger -t myscript -p local0.info "Message":* adding tag

*logger -i -t myscript "Message" :* include PID in log

```
$ logger "Message"

Aug 2 01:22:34 linuxsvr jason: Message


$ logger -p local0.info "Message"

Aug 2 01:22:41 linuxsvr jason: Message


$ logger -s -p local0.info "Message"

jason: Message # <-- Displayed on screen.
```

```
$ logger -t myscript -p local0.info "Message"

Aug 2 01:22:44 linuxsvr myscript: Message
```

```
$ logger -i -t myscript "Message"

Aug 2 01:22:53 linuxsvr myscript[12986]: Message
```

# Custom log functions

```
logit () {
	local LOG_LEVEL=$1
	shift
	MSG=$@
	TIMESTAMP=$(date +"%Y-%m-%d %T")
	if [ $LOG_LEVEL = 'ERROR' ] || $VERBOSE
	then
		echo "${TIMESTAMP} ${HOST} ${PROGRAM_NAME}[${PID}]: ${LOG_LEVEL} ${MSG}"
	fi
}
```

```
logit INFO "Processing data."

fetch-data $HOST || logit ERROR "Could not fetch data from $HOST"
```

# While Loops

# Contents

- While loops
- Infinite loops
- Loop control
    - Explicit number of times
    - User input
    - Command exit status
- Reading files, line-by-line
- break and continue

# While Loop Format

```
while [ CONDITION_IS_TRUE ]
do
    command 1
    command 2
    command N
done
```

# While Loop Format

```
while [ CONDITION_IS_TRUE ]
do
    # Commands change the condition
    command 1
    command 2
    command N
done
```

# Infinite Loops

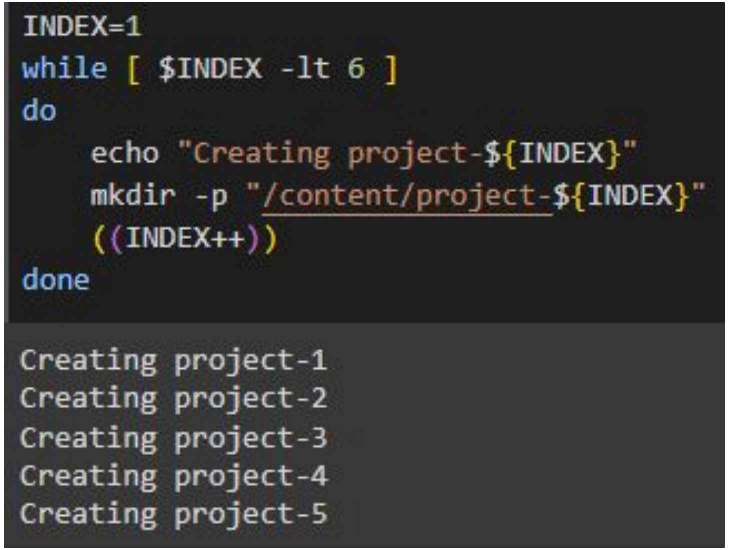
```
while [ CONDITION_IS_TRUE ]
do
    # Commands do NOT change the condition
    command N
done
```

# Infinite Loops

```
while [ true ]
do
    command N
    sleep 1
done
```

# Example - Loop 5 Times

```
INDEX=1

while [ $INDEX -lt 6 ]

do

        echo "Creating project-${INDEX}"

        mkdir -p “/content/project-${INDEX}”

        ((INDEX++))

done
```

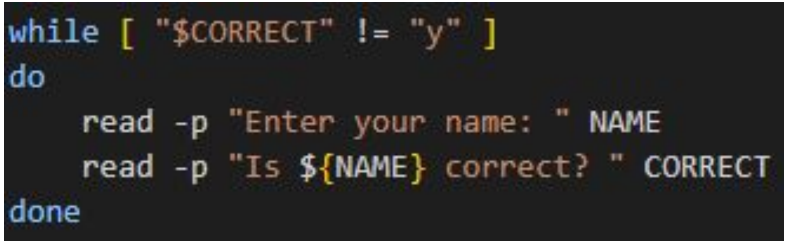


```
INDEX=1
while [ $INDEX -lt 6 ]
do
    echo "Creating project-${INDEX}"
    mkdir -p "/content/project-${INDEX}"
    ((INDEX++))
done
```

```
Creating project-1
Creating project-2
Creating project-3
Creating project-4
Creating project-5
```

# Example - Checking User Input

```
while [ "$CORRECT" != "y" ]
do
	read -p "Enter your name: " NAME
	read -p "Is ${NAME} correct? " CORRECT
done
```



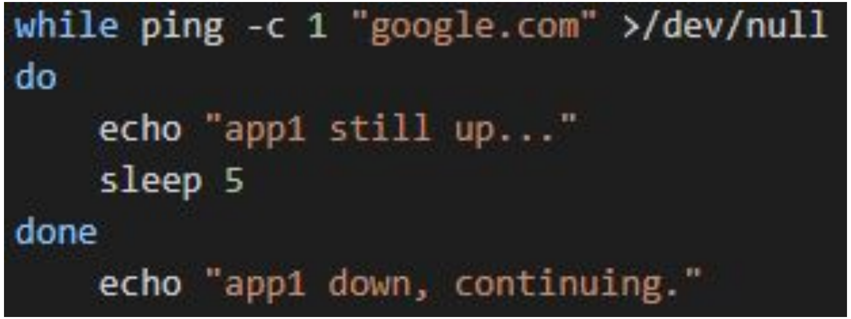
```
while [ "$CORRECT" != "y" ]
do
    read -p "Enter your name: " NAME
    read -p "Is ${NAME} correct? " CORRECT
done
```

# Output - Checking User Input

```
Enter your name: Minh

Is Minh correct? n

Enter your name: Duy

Is Duy correct? y
```

# Example - Return Code of Command

```
while ping -c 1 app1 >/dev/null

do

        echo "app1 still up..."

        sleep 5

done

echo "app1 down, continuing."
```



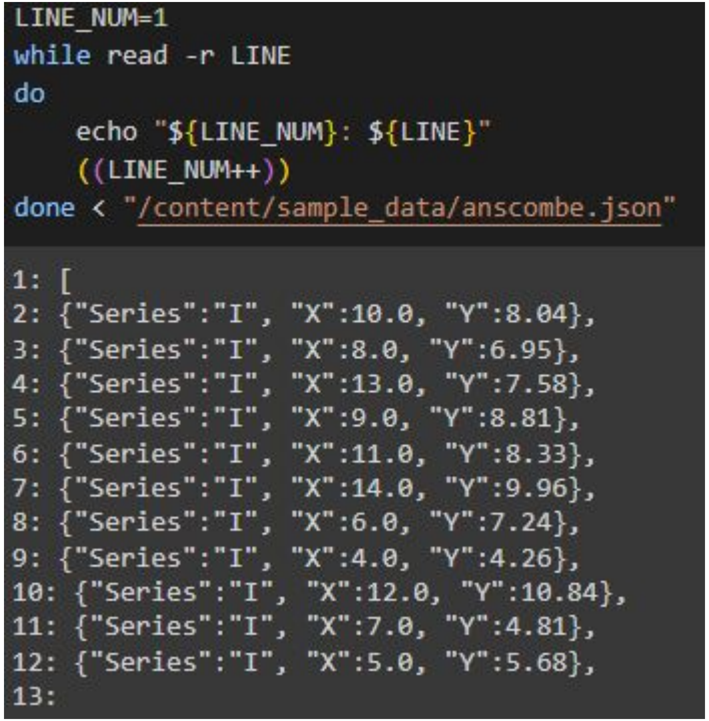
```
while ping -c 1 "google.com" >/dev/null
do
    echo "app1 still up..."
    sleep 5
done
    echo "app1 down, continuing."
```

# Output - Return Code of Command

```
app1 still up...
app1 still up…
app1 still up…
app1 still up…
app1 still up…
app1 down, continuing.
```

# Reading a file, line-by-line

```
LINE_NUM=1
while read LINE
do
    echo "${LINE_NUM}: ${LINE}"
    ((LINE_NUM++))
done < /etc/fstab
```

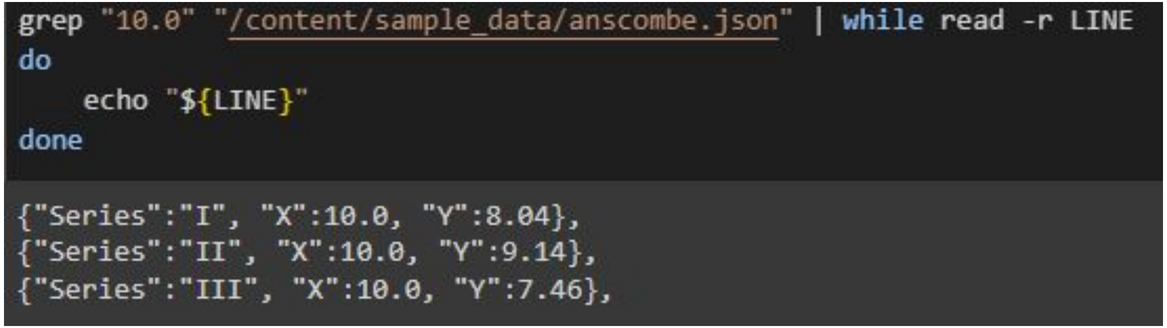


```
LINE_NUM=1
while read -r LINE
do
    echo "${LINE_NUM}: ${LINE}"
    ((LINE_NUM++))
done < "/content/sample_data/anscombe.json"
```

```
1: [
2: {"Series":"I", "X":10.0, "Y":8.04},
3: {"Series":"I", "X":8.0, "Y":6.95},
4: {"Series":"I", "X":13.0, "Y":7.58},
5: {"Series":"I", "X":9.0, "Y":8.81},
6: {"Series":"I", "X":11.0, "Y":8.33},
7: {"Series":"I", "X":14.0, "Y":9.96},
8: {"Series":"I", "X":6.0, "Y":7.24},
9: {"Series":"I", "X":4.0, "Y":4.26},
10: {"Series":"I", "X":12.0, "Y":10.84},
11: {"Series":"I", "X":7.0, "Y":4.81},
12: {"Series":"I", "X":5.0, "Y":5.68},
13:
```

# Reading a file, line-by-line from a command

```
grep “10.0” /content/sample_data/anscombe.json | while read -r LINE

do

    echo "${LINE}"

done
```

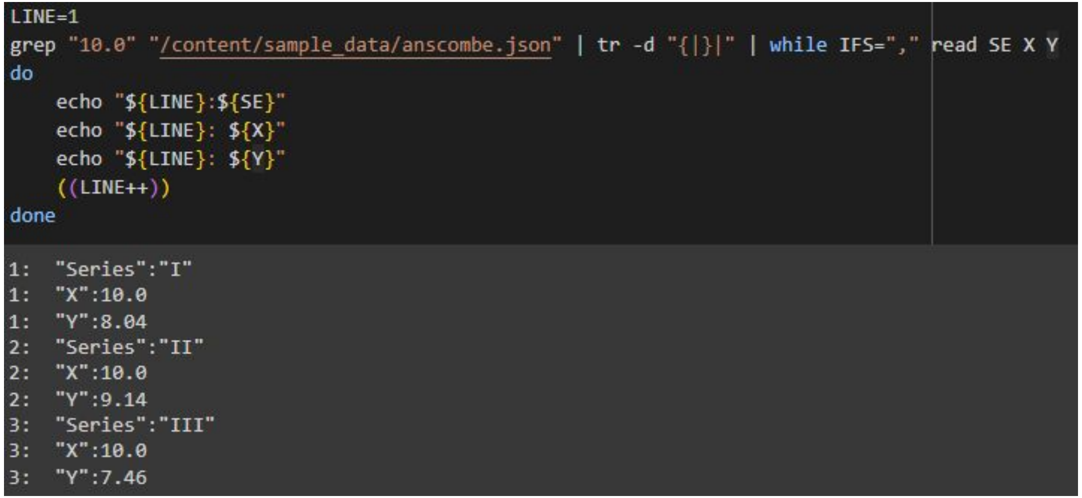


```
grep "10.0" "/content/sample_data/anscombe.json" | while read -r LINE
do
    echo "${LINE}"
done
```

```
{"Series":"I", "X":10.0, "Y":8.04},
{"Series":"II", "X":10.0, "Y":9.14},
{"Series":"III", "X":10.0, "Y":7.46},
```

# read command with multiple variables

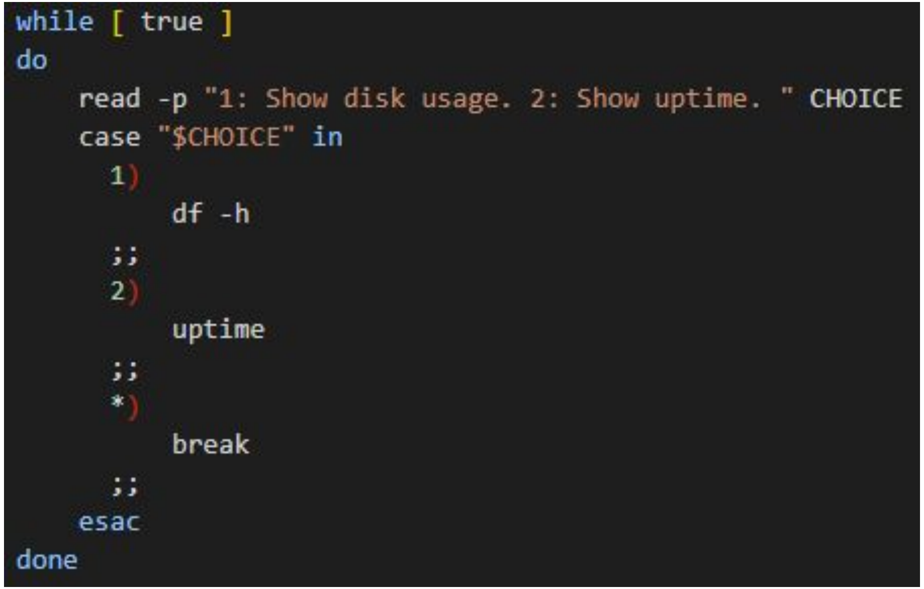
```
#!/bin/bash

LINE=1

grep "10.0" "/content/sample_data/anscombe.json" | tr -d "{|}|" | while IFS="," read SE X Y

do

    echo "${LINE}:${SE}"

    echo "${LINE}: ${X}"

    echo "${LINE}: ${Y}"

    ((LINE++))

done
```



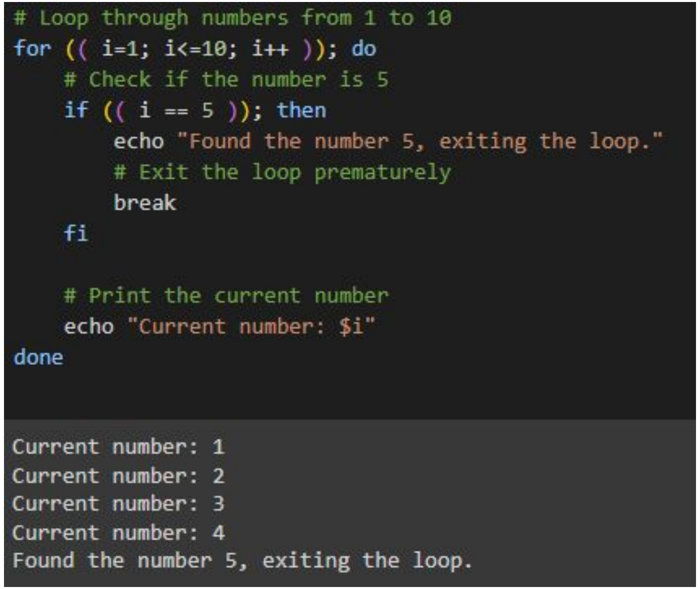
```
LINE=1
grep "10.0" "/content/sample_data/anscombe.json" | tr -d "{|}|" | while IFS="," read SE X Y
do
    echo "${LINE}:${SE}"
    echo "${LINE}: ${X}"
    echo "${LINE}: ${Y}"
    ((LINE++))
done
```

```
1:  "Series":"I"
1:  "X":10.0
1:  "Y":8.04
2:  "Series":"II"
2:  "X":10.0
2:  "Y":9.14
3:  "Series":"III"
3:  "X":10.0
3:  "Y":7.46
```

# “**break**” statement



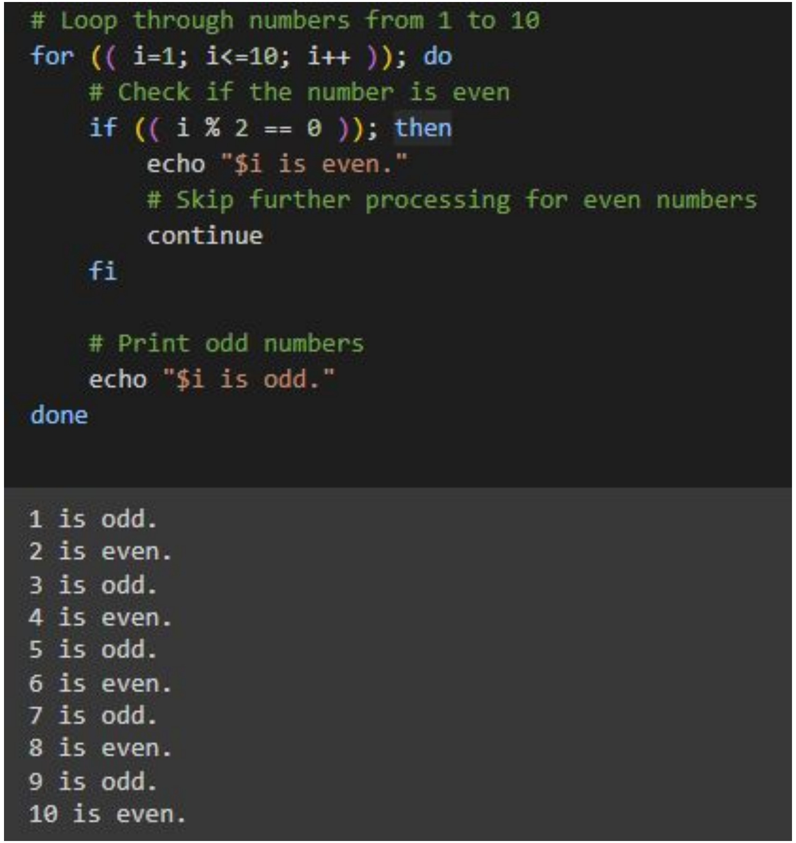
```
while [ true ]
do
    read -p "1: Show disk usage. 2: Show uptime. " CHOICE
    case "$CHOICE" in
      1)
          df -h
      ;;
      2)
          uptime
      ;;
      *)
          break
      ;;
    esac
done
```



```
# Loop through numbers from 1 to 10
for (( i=1; i<=10; i++ )); do
    # Check if the number is 5
    if (( i == 5 )); then
        echo "Found the number 5, exiting the loop."
        # Exit the loop prematurely
        break
    fi

    # Print the current number
    echo "Current number: $i"
done
```

```
Current number: 1
Current number: 2
Current number: 3
Current number: 4
Found the number 5, exiting the loop.
```

# “**continue**” statement



```
# Loop through numbers from 1 to 10
for (( i=1; i<=10; i++ )); do
    # Check if the number is even
    if (( i % 2 == 0 )); then
        echo "$i is even."
        # Skip further processing for even numbers
        continue
    fi

    # Print odd numbers
    echo "$i is odd."
done
```

```
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
6 is even.
7 is odd.
8 is even.
9 is odd.
10 is even.
```

# Summary

- While loops
- Infinite loops
- Loop control
  - Explicit number of times
  - User input
  - Command exit status
- Reading files, line-by-line
- break and continue