

Linux Systems and Open Source Software

Bash and Shell Scripts

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Fall 2022



Outline

- Shell vs. Bash
- Useful Commands
  - Variable on shell (Bash)
  - Pipe commands
- Shell Script
  - Conditional expressions
  - Conditional statements
  - Loop

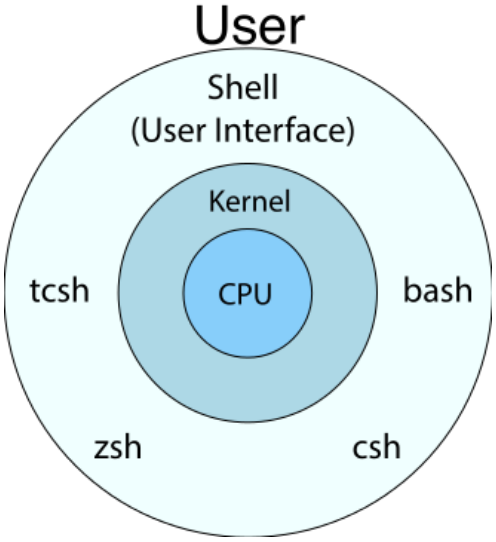


SHELL VS BASH

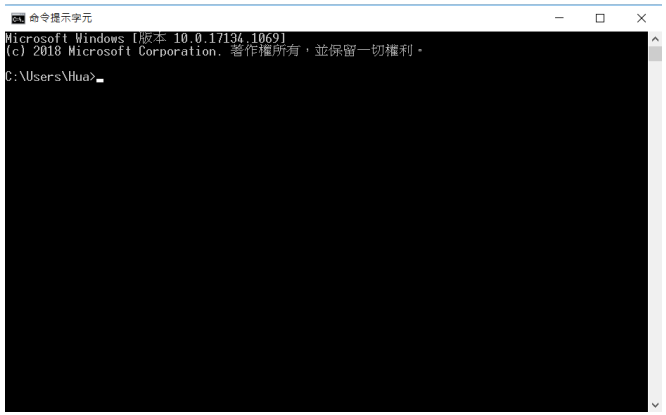


Shells

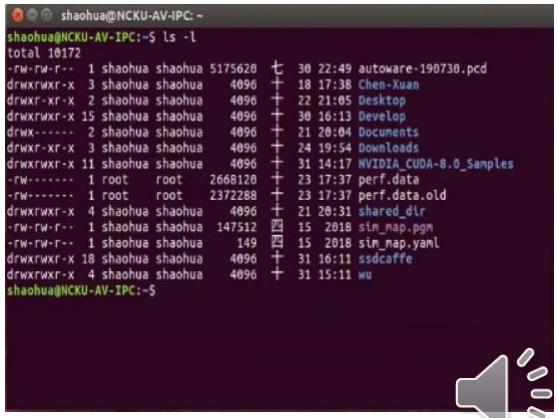
- A shell is a command-line interpreter
  - which acts as **a user interface** to send commands to the operating system (OS) requesting for its service
  - The OS then interprets the command and tells the CPU and other computer hardware how to carry out the user's commands
- In addition, the shell is both **an interactive command language** and **a scripting language**



Windows



Linux



Courtesy of https://code.snipcademy.com/tutorials/linux-command-line/environment/shells



Shells (Cont.)

- There are several advantages when using shell within the following occasions:
  - Faster transportation with texts when **working remotely** (e.g., ssh, ftp)
  - Most embedded devices operate in non-GUI environment
  - Manage resources on UNIX-like system (e.g., shell script for automation)



Relationship between Shell and Bash?

- A little bit of history
  - To commemorate Steven Bourne, the author of the shell, name it as *Bourne shell*, a new Unix shell, for Unix-like systems
- Nowadays, one of the most famous shells on Unix-like systems is **Bash** (Bourne-Again SHell), which is the remake version of Bourne shell
- You can check out the available shells on your system
  - **$ cat /etc/shells**

```
root@NCKU-AV-IPC:~
root@NCKU-AV-IPC:~# cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/tmux
root@NCKU-AV-IPC:~#
```

- This file uses Bash as an example to demonstrate the usages of shells
  - Most of the concepts and principles introduced in this file may be applied to other shells



Workflow of the Bash

1. Read the instructions given by user when **Enter** is pressed
2. Call **fork()** to create a new process, referred to as *child process*
3. Analyze the input string
   - e.g., disassemble "**ls -l hello.c**" into "**ls**", "**-l**" and "**hello.c**"
4. The child process calls **execve()** system call to execute target program (e.g., **ls**) and put in the argument (e.g., **-l hello.c**)
5. Parent process (the shell itself) call **wait4()** to wait for the completion of the child process and **exit()**
6. Repeat step (1) iteratively until the shell is terminated

Courtesy of https://www.itread01.com/content/1546274376.html



Login and Non-login Shells

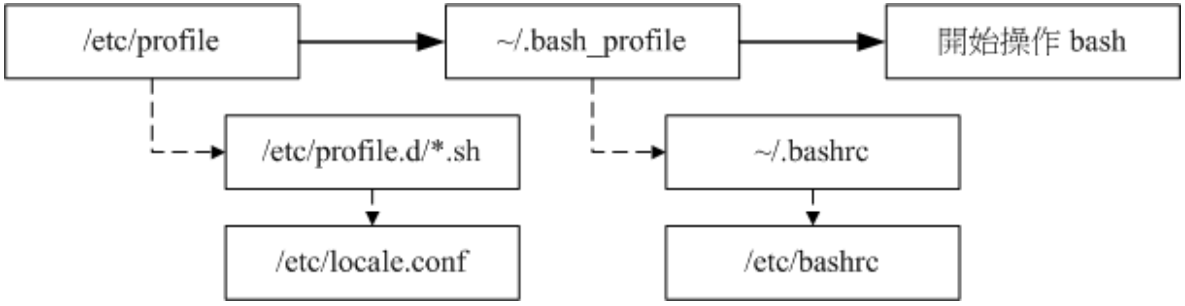
- Login shell
  - Before you can use shell, you need to do the login process
  - E.g., when you login the system (remotely) through tty1 ~ tty6, you are asked to type the user name and password

- Non-login shell
  - You do not perform the login process before you can use shell
  - E.g., you have got in the Linux system through the X window interface, where the login process has been done, and you can turn on the *terminal* without typing the account and password



The Configuration Files for Bash

- **/etc/profile**
  - To set up the **global (system-wide) environment variables** for the user based on the user identifier (UID), e.g., **PATH**, **HOSTNAME**
  - It reads **/etc/profile.d/*.sh** to further read the scripts to help set up the user interface, language, aliases of the commands (**ll** for **ls –l**), etc.
  - For example, **/etc/locale.conf** is used by **/etc/profile.d/lang.sh** to set up the default language used by the system
- **~/.bash_profile**
  - To set up the **user-specific configurations**
  - In fact, only one of the three files, **~/.bash_profile**, **~/.bash_login**, and **~/.profile**, will be read during the login process
- During the login process, the sequence of the configurations being loaded is as below



Courtesy of https://unix.stackexchange.com/questions/129143/what-is-the-purpose-of-bashrc-and-how-does-it-work
Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#settings_bashrc



Load the User Specific Settings

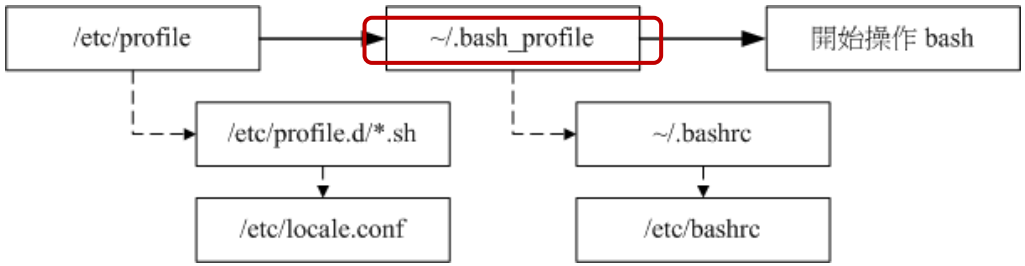
if **~/.bashrc** exists, then applies its settings

- Apply the user-specific setting for the **PATH** variable
- Note the new setting appends at the tail of the original setting

```
[dmtsai@study ~]$ cat ~/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/.local/bin:$HOME/bin
export PATH
```



Courtesy of https://unix.stackexchange.com/questions/129143/what-is-the-purpose-of-bashrc-and-how-does-it-work
Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#settings_bashrc



Variables used by (Bash) shell

Pipe command

MORE ON BASH



Variables

- Rules of variable assignment
  - Correct format
    - (O) variable=hello , (O) variable='hello world' , (O) variable="hello world"
  - The double quotes (") can reserve variable properties with ($) sign.
    - (O) variable="$VAR world" (variable=hello world, if $VAR=hello)
  - Cannot use "SPACE" character between the variable and value.
    - (X) variable = hello , (X) variable=hello world
  - Cannot use number as leading character of variable name
    - (X) 2variable=hello
- Recall the variable, use **echo** command
  - E.g., **echo $PATH, echo ${PATH}**
- Erase the variable, use **unset** command
  - E.g., **unset variable**



Environment Variables: env and set

- Use **env** to list the environment variables
  - **HOME**: the path to home directory
  - **SHELL**: the shell we used
  - **HISTSIZE**: the maximum number of command recorded by ***history***
  - **PATH**: where the executable be, using colon sign (:) to fence off each path

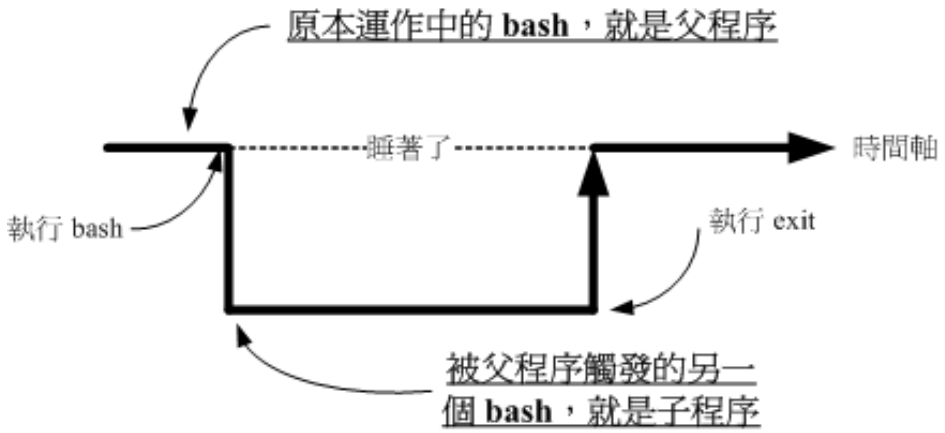
- Using **set** to list all variable (environment + user defined)
  - **PS1**: the primary prompt style of command line

```
shaohua@NCKU-AV-IPC: ~
shaohua@NCKU-AV-IPC:~$echo $PS1
\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$
shaohua@NCKU-AV-IPC:~$PS1='\[\033[01;32m\]\u@\h<Extend Text in Prompt>\[\033[00m
\]:\[\033[01;34m\]\w\[\033[00m\]\$'
shaohua@NCKU-AV-IPC<Extend Text in Prompt>:~$
```



export Environment Variable

- **export**: the command to activate *the variable setting* by adding *the setting* as part of the environment variables
  - This command usually be used in **.bashrc** file
  - It is a common practice that we **export** variable settings (e.g., user-defined variables) before we create a child process in the shell script since
    - the child process will inherit the environment variables of parent process



Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#export



Variable Declaration: read, array

- **read**
  - Read the data from the keyboard and set the data to the given variable

```
[dmtsai@study ~]$ read [-pt] variable
選項與參數：
-p ：後面可以接提示字元
-t ：後面可以接等待的『秒數！』

<example>
[dmtsai@study ~]$ read -p "Please keyin your name: " named
Please keyin your name: Linux
[dmtsai@study ~]$ echo ${named}
Linux
```

- **array**
  - To set up the one dimensional array
  - Pay attention to the ***data types***

```
[dmtsai@study ~]$  var[index]=content

<example>
[dmtsai@study ~]$ var[1]="small min"
[dmtsai@study ~]$ var[2]="big min"
[dmtsai@study ~]$ var[3]="nice min"
[dmtsai@study ~]$ echo "${var[1]}, ${var[2]}, ${var[3]}"
small min, big min, nice min
```

Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#read



Variable Declaration: declare

- declare
  - to declare the variable and its type
  - By default, the variable is assumed to be ***String*** in Bash
  - In addition, it assumes the **integer-based arithmetic operations**
    - E.g., the result of the division 1/3 is 0

```
[dmtsai@study ~]$ declare [-aixr] variable
選項與參數：
-a ：將後面名為 variable 的變數定義成為陣列 (array) 類型
-i ：將後面名為 variable 的變數定義成為整數數字 (integer) 類型
-x ：用法與 export 一樣，就是將後面的 variable 變成環境變數
-r ：將變數設定成為 readonly 類型，該變數不可被更改內容，也不能 unset

<example1>
[dmtsai@study ~]$ sum=100+300+50
[dmtsai@study ~]$ echo ${sum}
100+300+50 <- String variable
[dmtsai@study ~]$ declare -i sum=100+300+50
[dmtsai@study ~]$ echo ${sum}
450 <- Integer variable

<example2>
[dmtsai@study ~]$ declare -r sum
[dmtsai@study ~]$ sum=testing
-bash: sum: readonly variable
```

Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#read



Command-line Operators: ; && ||

- ';'
  - is used to chain the commands
  - Does not consider the dependencies of the issued commands
- '&&' and '||'
  - are logical operators, where the execution of the next command depends entirely on the execution result of the previous command
  - E.g., **$/>ls /tmp/abc && touch /tmp/abc/hehe**
    - Check if the directory **/tmp/abc** exists, and if yes, use **touch** to create the file **/tmp/abc/hehe**

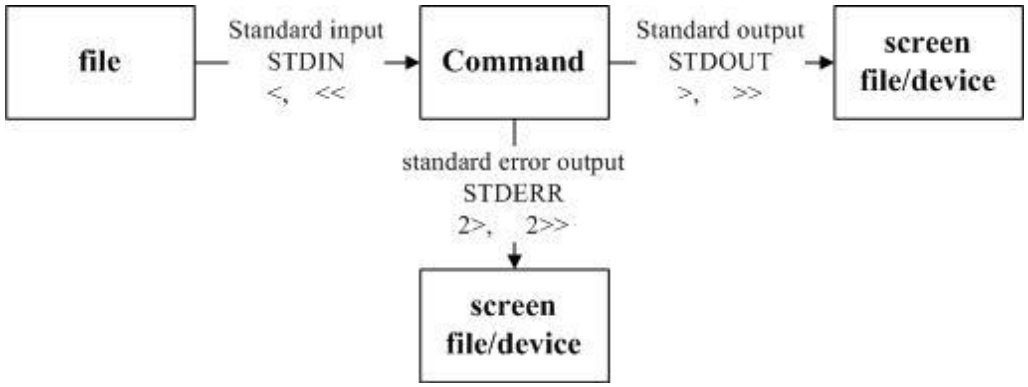
Commands	Description
cmd1 && cmd2	1. if cmd1 is terminated successfully and correctly ($?=0), then runs cmd2 2. if cmd1 is terminated incorrectly ($? ≠ 0), then does not run cmd2
cmd1 || cmd2	1. if cmd1 is terminated successfully and correctly ($?=0), then does not run cmd2 2. if cmd1 is terminated incorrectly ($?≠0), then runs cmd2

Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#redirect_com



Redirection: stdin, stdout, stderr

- Standard input (**stdin**)
  - use **<** and **<<** to redirect
  - **<** overwrites original data and **<<** appends to original data
- Standard output (**stdout**)
  - use **>** and **>>** to redirect
- Standard error output (**stderr**)
  - use **2>** and **2>>** to redirect
- Using **/dev/null** to ignore the redirect data to the *black hole*




```
<example>
[dmtsai@study ~]$ find /home -name .bashrc
find: '/home/arod': Permission denied <== Standard error output
find: '/home/alex': Permission denied <== Standard error output
/home/dmtsai/.bashrc <== Standard output
<example: redirect the stdout and stderr data to the file list>
[dmtsai@study ~]$ find /home -name .bashrc 2>&1 list
<example: redirect the stderr to /dev/null>
[dmtsai@study ~]$ find /home -name .bashrc 2> /dev/null
/home/dmtsai/.bashrc
```

Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#redirect



History and Alias of Commands

history and alias

- history
  - View the previously executed commands

```
[dmtsai@study ~]$ history [n]
[dmtsai@study ~]$ history [-c]
[dmtsai@study ~]$ history [-raw] histfiles
選項與參數：
n ：數字，意思是『要列出最近的 n 筆命令列表』的意思！
-c ：將目前的 shell 中的所有 history 內容全部消除
-a ：將目前新增的 history 指令新增入 histfiles 中，若沒有加 histfiles， 則預設寫入 ~/.bash_history
-r ：將 histfiles 的內容讀到目前這個 shell 的 history 記憶中；
-w ：將目前的 history 記憶內容寫入 histfiles 中！
```

- alias, unalias
  - Instructs the shell to replace one string with another string while executing the commands; i.e., to create shortcuts

```
[dmtsai@study ~]$ alias command='other command'
<example>
[dmtsai@study ~]$ alias lm='ls -al | more'
[dmtsai@study ~]$ unalias lm
```

Courtesy of http://linux.vbird.org/linux_basic/0320bash.php#history
https://www.geeksforgeeks.org/history-command-in-linux-with-examples/
https://www.geeksforgeeks.org/alias-command-in-linux-with-examples/



Variable on shell (Bash) & Commands

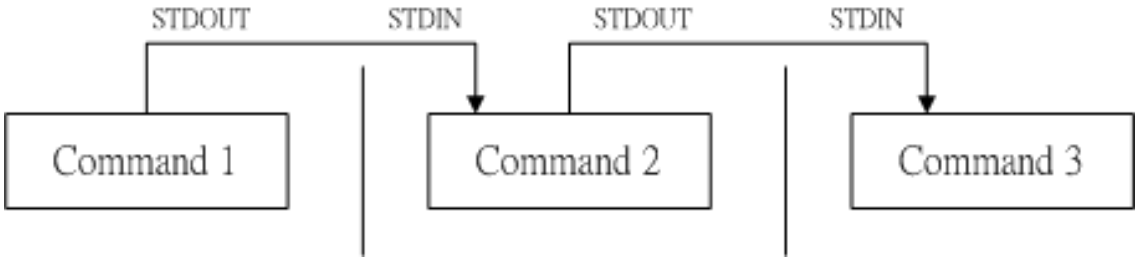
Pipe commands

MORE ON BASH



What is Pipe?

- The symbol '|'denotes a **pipe** command
- **Pipe** redirects the output of a process as the input to another one like a **pipeline**
- Pipe only handles the *correct* data (i.e., data from standard output)
  - By default, it cannot handle the data from standard error
  - You can do it by using **2>&1** wisely





grep

- **grep**
  - It searches the given file for *lines* containing a match to the given strings or words
  - The results are sent to standard output
- A new choice: **ag**
  - Silver Searcher, which is said to be way faster than **grep** and **ack**
  - You have to install it by yourself

- Search in the file **/etc/man_db.conf** for **MANPATH**
- Matched strings are highlighted automatically because the option "**—color=auto**" is used

```
[dmtsai@study ~]$ grep [-acinv] [--color=auto] '搜尋字串' filename
選項與參數：
-a ：將 binary 檔案以 text 檔案的方式搜尋資料
-c ：計算找到 '搜尋字串' 的次數
-i ：忽略大小寫的不同，所以大小寫視為相同
-n ：順便輸出行號
-v ：反向選擇，亦即顯示出沒有 '搜尋字串' 內容的那一行
--color=auto ：將找到的關鍵字部分加上顏色的顯示

範例一：將 last 當中，有出現 root 的那一行就取出來；
[dmtsai@study ~]$ last | grep 'root'

範例二：與範例一相反，只要沒有 root 的就取出！
[dmtsai@study ~]$ last | grep -v 'root'

範例三：取出 /etc/man_db.conf 內含 MANPATH 的那幾行
[dmtsai@study ~]$ grep --color=auto 'MANPATH' /etc/man_db.conf
....(前面省略)....
MANPATH_MAP /usr/games /usr/share/man
MANPATH_MAP /opt/bin /opt/man
MANPATH_MAP /opt/sbin /opt/man
```



cut

- **cut**
  - It cuts out the **sections** from each line of files and writing the result to standard output

```
[dmtsai@study ~]$ cut -d '分隔字元' -f fields <==用於有特定分隔字元
[dmtsai@study ~]$ cut -c 字元區間 <==用於排列整齊的訊息
選項與參數：
-d ：後面接分隔字元。與 -f 一起使用；
-f ：依據 -d 的分隔字元將一段訊息分割成為數段，用 -f 取出第幾段的意思；
-c ：以字元 (characters) 的單位取出固定字元區間；

範例一：將 PATH 變數取出，找出第三，五個路徑。
[dmtsai@study ~]$ echo ${PATH}
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/dmtsai/.local/bin:/home/dmtsai/bin
#|    1     |  2   |     3      |  4  |          5          |     6     |
[dmtsai@study ~]$ echo ${PATH} | cut -d ':' -f 5
/home/dmtsai/.local/bin
[dmtsai@study ~]$ echo ${PATH} | cut -d ':' -f 3,5
/usr/local/sbin:/home/dmtsai/.local/bin

範例二：將 export 輸出的訊息，取得第 12 字元以後的所有字串
[dmtsai@study ~]$ export
declare -x HISTCONTROL="ignoredups"
declare -x HISTSIZE="1000"
.....(其他省略).....

# 刪除『 declare -x 』：
[dmtsai@study ~]$ export | cut -c 12-
HISTCONTROL="ignoredups"
HISTSIZE="1000"
.....(其他省略).....
```



Conditional expressions

Conditional statements

Loop

SHELL SCRIPT



What Is Shell Script?

- It is a computer program run by the shell
  - The program is written in the scripting language recognized by the shell
  - It often sets up the environment, runs the program, and does any necessary cleanup, logging, etc.
    - E.g., typical operations performed by shell scripts include file manipulation, program execution, and printing text
- Sometimes, shell script refers to the **automated mode of running the shell commands**
  - The shell script example is as below
  - **#!/bin/bash** means the bash shell is used, and # means a comment in Bash

```
#!/bin/bash
# Program:
# This program shows "Hello World!" in your screen.
# History: # 2015/07/16 VBird First release

echo -e "Hello World! \a \n"
exit 0
```

Courtesy of https://en.wikipedia.org/wiki/Shell_script



Conditional expressions

Conditional statements

Loop

SHELL SCRIPT



Conditional Expressions: test and [ ]

- *Check file types and compare values
  - Often used as part of the conditional execution of shell commands

Flag	Meaning
Test the type, existence or property of a file/directory; e.g., **test -e filename**	
-e	if the filename exists?
-f	if the filename exists and it is a *file*
-d	if the filename exists and it is a *directory*
-s	if the filename exists and it is an non-empty file
Test the relationship between two integers; e.g., **test n1 -eq n2**	
-eq	if the numbers are equal
-ne	if the numbers are not equal
-gt	if n1 is greater than n2
-lt	if n1 is less than n2
-ge	if n1 is greater than or equal to n2
-le	if n1 is less than or equal to n2
Test multiple conditions; e.g., **test -r filename -a -x filename**	
-a	(and) both conditions are true; e.g., **test -r file -a -x file**, it returns true when the file is both readable and executable
-o	(or) either of the two conditions is true; e.g., **test -r file -o -x file**, it returns true when the file is either readable or executable
!	Opposite state; e.g., **test ! -x file**, it returns true when the file is not executable

*Courtesy of http://linux.vbird.org/linux_basic/0340bashshell-scripts.php#test



Examples of test and [ ]

- Placing the expression between square brackets '[' and ']' is the same as testing the expression with **test**
  - Note there should be a **blank** after and before '[' and ']', respectively
- The example script reads user input (i.e., **Y** or **N**) and print out the corresponding strings

```
<Check if the filename exists and print out the result>
[dmtsai@study ~]$ test -e testing && echo "exist" || echo "Not exist"
Not exist

<The above command can be rewritten in a different way>
[dmtsai@study ~]$ [ -e testing ] && echo "exist" || echo "Not exist"
Not exist
```

```
#!/bin/bash
# Program:
# This program shows the user's choice
# History:
# 2015/07/16 VBird First release

read -p "Please input (Y/N): " yn
[ "${yn}" == "Y" -o "${yn}" == "y" ] && echo "OK, continue" && exit 0
[ "${yn}" == "N" -o "${yn}" == "n" ] && echo "Oh, interrupt!" && exit 0
echo "I don't know what your choice is" && exit 0
```



Conditional expressions

Conditional statements

Loop

SHELL SCRIPT



Conditional Statements: if ... then

- Three different styles

```
#1.
# Single condition (If ... then)
if [ 條件判斷式 ]; then
    Statements when the condition holds.
fi   <==End of the if ... then

#2.
# Single condition (if ... then ... else)
if [ 條件判斷式 ]; then
    Statements when the condition holds.
else
    Statements when the condition fails.
fi

#3.
# Multiple conditions
if [ Condition#1 ]; then
    Statements when the condition#1 holds.
elif [ Condition#2 ]; then
    Statements when the condition#2 holds.
else
    Statements when both of the condition#1 and
    condition#2 fail.
fi
```

- Rewrite the previous script with the conditional statement

```
#!/bin/bash
# Program:
# This program shows the user's choice
# History:
# 2015/07/16 VBird First release

read -p "Please input (Y/N): " yn
if [ "${yn}" == "Y" ] || [ "${yn}" == "y" ]; then
    echo "OK, continue“
elif [ "${yn}" == "N" ] || [ "${yn}" == "n" ]; then
    echo "Oh, interrupt!“
else
    echo "I don't know what your choice is“
fi
```

Courtesy of http://linux.vbird.org/linux_basic/0340bashshell-scripts.php#ifthen



Conditional Statements: case ..... esac

- A good alternative to multilevel **if-then-else-fi** statement
  - Enable you to match several values against one variable
  - Is easier to read and write

```
case $Variable in
   "Value#1")
        Statements when $Variable == "Value#1".
        ;;
   " Value#2")
        Statements when $Variable == "Value#2".
        ;;
   *)  # Other values...
        Statements when the content of $Variable is
        not the same as the above values.
        exit 1
        ;;
esac
```

- Example: Read and print the user's choice

```
#!/bin/bash
# Program:
# This script only accepts the flowing parameter: one, two or three.
# History:
# 2015/07/17 VBird First release
echo "This program will print your selection !"
read -p "Input your choice: " choice # Read input; save in choice
case ${choice} in
   "one") # User types "one"
      echo "Your choice is ONE"
      ;;
   "two") # User types "two"
      echo "Your choice is TWO"
      ;;
   "three") # User types "three"
      echo "Your choice is THREE"
      ;;
   *)
      echo "Usage ${0} {one|two|three}"
      ;;
esac
```

Courtesy of https://bash.cyberciti.biz/guide/The_case_statement
http://linux.vbird.org/linux_basic/0340bashshell-scripts.php#case



Conditional expressions

Conditional statements

Loop

WHAT IS SHELL SCRIPT?



Loop: while and until

- Repeat particular instruction again and again, until particular condition satisfies
  - The number of iterations is usually unknown for **while, until** loops
  - The number is known for **for** loops
- The example script is used to calculate the sum of "1+2+3+...+100"

```
# while...do...done
while [ condition ]
do
   Statements
done

# until...do...done
until [ condition ]
do
   Statements
done
```

```
#!/bin/bash
# Program:
# Use loop to calculate "1+2+3+...+100" result.
# History: # 2015/07/17 VBird First release

s=0 # 這是加總的數值變數
i=0 # 這是累計的數值，亦即是 1, 2, 3....
while [ "${i}" != "100" ]
do
   i=$(($i+1)) # 每次 i 都會增加 1
   s=$(($s+$i)) # 每次都會加總一次！
done

echo "The result of '1+2+3+...+100' is ==> $s"
```

Courtesy of https://bash.cyberciti.biz/guide/For_loop



Loop: for...do...done

- The for loop executes its body once for every new value assigned to a **var** (variable) in specified list (**con1, con2, ...**)
  - Repeat all statement between do and done till condition is not satisfied
  - The lists or values are normally: strings, numbers, command line arguments, file names, Linux command output

```
# for...do...done
for var in con1 con2 con3 ...
do
   Statements
done

# Another format
# EXP1: init_value; EXP2: limiting_value;
# EXP3: increment value;
for (( EXP1; EXP2; EXP3 ))
do
   Statements
done
```

- Prints the contents in the list
- Calculate the summation

```
# example 1
#!/bin/bash
# Program:
# Using for .... loop to print 3 animals
# History:
# 2015/07/17 VBird First release
for animal in dog cat elephant
do
   echo "There are ${animal}s.... "
done

# example 2
#!/bin/bash
# Program:
# Try do calculate 1+2+....+${your_input}
# History:
# 2015/07/17 VBird First release
read -p "Please input a number, I will count for 1+2+...+your_input: " nu
s=0
for (( i=1; i<=${nu}; i=i+1 )) do
   s=$(($s}+${i}))
done
echo "The result of '1+2+3+...+${nu}' is ==> ${s}"
```

Courtesy of https://bash.cyberciti.biz/guide/For_loop
https://bash.cyberciti.biz/guide/For_loop



THANK YOU!