Bash Scripting

`Bash` (Bourne Again Shell) is a specific type of shell. It was created as an improved replacement for the original Bourne shell(sh), which includes rich feature set like command line editing, job control, shell functions, etc.

`#` is the comment notation in Bash scripting

`.sh file` refers to **Bash scripting file** or **Shell executable file**

`#!/bin/bash` at the beginning of a **.sh file** tells the OS to use the Bash shell to interpret and execute the commands contained in the script file.

> **What's the main difference between 'shell' and 'bash'?**
>
> "shell" is a command-line interpreter(a program) that provides a user interface for accessing an OS's services, whereas "Bash" is a specific **implementation** of a shell, offering extended features and functionalities beyond those found in the traditional Bourne shell.

Comparison Operators

1. Numeria Comparison Operators

- `-eq` Equal to
- `-ne` Not equal to
- `-gt` Greater than
- `-lt` Less than
- `-ge` Greater than or equal to
- `-le` Less than or equal to

2. String Comparison Operators

- `=` Equal to
- `==` Equal to (same as `=`) (only for [[]])
- `!=` Not equal to
- `<` Less than, in ASCII alphabetical order
  - Used as `[[ "$a" < "$b" ]]` or in a **test command** with escaped syntax: `[ "$a" \< "$b" ]`
- `>` Greater than
  - Used as `[[ "$a" > "$b" ]]` or in a **test command** with escaped syntax: `[ "$a" \> "$b" ]`
- `-z` String is null, that is, has zero length
  - Used as `[ -z "$a" ]`

3. File Test Operators

- `-e` File exists (regardless of type)
- `-f` File exists and is a regular file

- `-d` File exists and is a directory
- `-r` File exists and is readable
- `-w` File exists and is writable
- `-x` File exists and is executable
- `-s` File exists and is not empty

Overall Structure

p.s. `$variable_name` is used to access the value stored in a variable named variable_name

```
greeting="Hello, World!"
echo $greeting # print "Hello, World!"
```

1. Conditional Statements (if, else, elif, fi)

- **Condition Evaluations**

  `[ "$a" \> "$b" ]` **Test Command** (sometime should use with escaped syntax when encounters **special characters**)
  `[[ "$a" > "$b" ]]` **Extended Test Command** without escaped syntax

  - Extended Test Command implements more operators like `== && || =~` and allows `> < >= >=` without escaping

  ```
  # Check if a file exists
  if [ -e "myfile.txt" ]; then
    echo "myfile.txt exists."
  else
    echo "myfile.txt does not exist."
  fi
  ```

  p.s. `[]` is POSIX compliant, `[[]]` is a Bash extension

- **Usage of `$?`**

  `$?` is a special built-in variable, which holds the **exit status**(zero or non-zero) of the last command executed in the shell. The exit status, also known as an exit code, is a numerical value returned by a command to the shell upon its completion.

  `0` **Success**: If a command executes successfully without errors, it typically returns an exit status of 0.

  `Non-zero` **Failure**: If a command fails due to any reason (such as a syntax error, a command not found, or an operational error), it returns a non-zero value, which usually indicates the type or category of error.

  ```
  grep "example" somefile.txt
  ```

```
if [ $? -eq 0 ]; then
  echo "The word 'example' was found."
else
  echo "The word 'example' was not found."
fi
```

> `grep` searches for the word "example" in the file somefile.txt
>
> if-statement first checks the **exit value** of `$?`
>
> if `$?` returns `0`, indicating `grep` successfully found the word, then it will prints a message saying the word was found.
>
> if `$?` is `not 0`, indicating `grep` did not find the word or the file does not exist, it prints a message saying the word was not found.

2. Loop Structures (for, while, until)

- For-loop

```
# Print numbers from 1 to 5
for i in {1..5}; do
  echo “Number $i”
done
```

- While-loop

```
# Repeat while a condition is true
count=1
while [ $count -le 5 ]; do
  echo “Count: $count”
  count=$((count + 1)) # $((...)) is used for arithmetic expansion (we will see it later)
done
```

3. Functions

- In shell scripting, we **don’t specify parameters inside parentheses** while declaring, parameters are passed to the function when it is called instead.
- While calling the function great() below, we use ”$1” to get the parameters (“$1, $2, $3…” is positional parameters similar in Python). This “1” refers to the first argument passed to the function.
- “$0” is a special case: it refers to the name of the script itself, not a function parameter
- "$@" is a list of all the command line arguments

```
# function declaration and definition
great() {
```

```
  echo “Hello, $1! Your UCLA_ID is $2.”
}

# function calling
great “Alice” 123456789
```

Variable Expansion

1. Basic Variable Expansion

Simply prefix the variable name with a `$`

```
name=“Alice”
echo “Hello, $name!”
```

2. Brace Expansion

Useful for when you need to clearly delimit the variable name

```
greeting="hello"
echo "${greeting}world"
# Without braces, it would try to expand `greetingworld`
```

Command Substitution

Command substitution, denoted by `$(command)`, allows you to use the output of a command as an argument of another command

```
today=$(date)
echo "The current date and time are: $today"
# `date` is a command outputs the current date and time
```

```
echo "The current directory is: $(basename $(pwd))"
# `pwd` returns the current directory path, `basename` extracts the last
part of that path (the current directory name), and echo prints it out.
```

Arithmetic Expansion

Arithmetic Expansion allows for the evaluation of an arithmetic expression

- denoted by `$((expression))`
- only support for integer

- common operator like +, -, *, /, %
- Variables inside the $(( )) are treated as integers and their values are used in the arithmetic operation.

```
a=10
b=5
sum=$((a + b))
echo "Sum: $sum"  # Outputs: Sum: 15

product=$((a * b))
echo "Product: $product"  # Outputs: Product: 50
```

> **What's the difference between `$(...)` and `$((...))`?**
>
> `$(command)` is used for command substitution
> `$((expression))` is used for performing arthemetic

Special Characters & Quoting Mechanisms

Quoting in shell scripting is a critical aspect for controlling how the shell interprets special characters and whitespace

1. Special Characters in shell

- `$`, `$()`, `$(())` - Dollar Sign
- `\` - Backslash
- `‘’` - Single Quotes
- `“”` - Double Quotes
- `*` - Asterisk
  `ls *.txt`
- `?` - Question Mark
  `ls file?.txt`
- `;` - Semicolon
  `echo "Start"; ls; echo "End"`
  `# Running multiple commands sequentially`
- `|` - Pipe
  `cat file.txt | grep "search term"`
  `# Used for piping the output of first command as input to later one`
- `&` Ampersand
- `>`, `<`, `=`
- `()`, `[]`, `{}`

2. Quoting Mechanisms

Some techniques to deal with **Special Characters**: `\`, `''`, `""`

1. Backslash `\` - Escape Character

```
echo \$PATH
# Outputs: $PATH

echo "\\"
# Outputs: \
```

In these examples, the backslash is used to prevent the shell from interpreting $ as the start of a variable and \ as a special character backflash itself.

2. Single Quotes '' - Strong Quoting

```
echo '$HOME is where the heart is'
# Outputs: $HOME is where the heart is
```

Here, $HOME is not expanded to the user's home directory, as it is enclosed in single quotes.

3. Double Quotes "" - Partial Quoting
Double Quotes allow for **Variable Expansion, Command Substitution, Arithmetic Expansion and Escape Sequence Processing**, whereas Backslash and Single Quotes do not.

```
echo "Your home directory is $HOME"  # Variable Expression
# Outputs: Your home directory is /home/username

echo "Current path is $(pwd)"  # Command Substitution
# Outputs: Current path is /home/username/desktop

echo "2 + 3 is $((2+3))"  # Arithmetic Expansion
# Outputs: 2 + 3 is 5

echo "Line 1\nLine 2" # Escape Sequence Processing
# Output:
# Line1
# Line2
```

In this example, `$HOME` is expanded to the user's home directory, it will return the **Absolute Path** of HOME.