Computer Practicum 1

Introduction to Bash scripts

Vida Groznik

Source: Ryan‘s tutorials.

Functions

```
1. #!/bin/bash
2. # Basic function
3.
4. print_something () {
5.     Echo Hello, I am a function
6. }
7.
8. print_something
9. print_something
```

```
user@bash: ./function.sh
Hello, I am a function
Hello, I am a function
user@bash:
```

Functions can be written in two different formats:

```
Function_name () {
    <commands>
}
```

Or

```
function function_name {
    <commands>
}
```

- In other programming languages it is common to have arguments passed to the function listed inside the brackets ().
  In Bash they are there for decoration only and you don‘t put anything inside them.
- The function definition (the actual function itself) must appear in the script **before** any calls to the function.

Functions – passing arguments

```
1. #!/bin/bash
2. # Passing arguments to a function
3.
4. print_something () {
5.     Echo Hello, $1
6. }
7.
8. print_something Mark
9. print_something Nina
```

```
user@bash: ./function_arguments.sh
Hello, Mark
Hello, Nina
user@bash:
```

To send the data to the function, we do it in the similar way as when passing command line arguments to a script.

We supply the arguments directly after (calling) the function name. Within the function they are accessible as **$1, $2, etc**.

Functions – return values

```
1.  #!/bin/bash
2.  # Setting a return status for a function
3.
4.  print_something () {
5.      Echo Hello, $1
6.      return 5
7.  }
8.  print_something Mark
9.  print_something Nina
10. echo The previous function has a return value of $?
```

Bash functions don‘t allow us to send (return) the data back to the original calling location.

But they do allow us to set a return status in a similar way to how a program or command exits with an exit status which indicates whether it succeeded or not.

We use the **`return`** keyword to indicate a return status.

```
user@bash: ./return_status.sh
Hello, Mark
Hello, Nina
The previous function has a return
value of 5
user@bash:
```

If you want to return a number (eg. the result of a calculation) then you can consider using the return status to achieve this.
(It is not it's intended purpose but it will work.)

Functions – return values (2)

```
1.  #!/bin/bash
2.  # Setting a return value to a function
3.
4.  lines_in_file () {
5.      cat $1 | wc -l
6.  }
7.
8.  num_lines=$( lines_in_file $1 )
9.
10. echo The file $1 has $num_lines lines in it.
```

```
user@bash: ./return2.sh myfile.txt
The file myfile.txt has 5 lines in it.
user@bash:
```

One way to get around the problem of returning the values is to use *Command Substitution* and have the function print the result (and only the result).

Line 5 - This command will print the number of lines in the file referred to by `$1`.

Line 8 - We use command substitution to take what would normally be printed to the screen and assign it to the variable `num_lines`.

Variable Scope

```
1.  #!/bin/bash
2.  # Experimenting with variable scope
3.
4.  var_change () {
5.      Local var1='local 1'
6.      echo Inside function: var1 is $var1
7.      echo Inside function: var2 is $var2
8.      var1='changed again'
9.      var2='2 changed again'
10. }
11.
12. var1='global 1'
13. var2='global 2'
14.
15. echo Before function call: var1 is $var1
16. echo Before function call: var2 is $var2
17.
18. var_change
19.
20. echo After function call: var1 is $var1
21. echo After function call: var2 is $var2
```

Scope refers to **which parts of a script can see which variables**.

By **default** a variable is **global**. This means that it is **visible everywhere** in the script.

We may also create a variable as a **local** variable. When we create a local variable within a function, it is only **visible within** that **function**.

To do that we use the keyword **`local`** in front of the variable the first time we set it's value.

```
local var_name=<var_value>
```

```
user@bash: ./local_variables.sh
Before function call: var1 is global1
Before function call: var2 is global 2
Inside function: var1 is local 1
Inside function: var2 is global 2
After function call: var1 is global1
After function call: var2 is 2 changed again
user@bash:
```

Overriding Commands

```
1.  #!/bin/bash
2.  # Create a wrapper around the command ls
3.
4.  ls () {
5.      command ls -lh
6.  }
7.
8. ls
```

It is possible to name a function as the same name as a command you would normally use on the command line.

This allows us to create a wrapper.

eg. Maybe every time we call the command **`ls`** in our script, what we actually want is **`ls -lh`**.

Line 5 - When we have a function with the same name as a command we need to put the keyword **`command`** in front of the the name when we want the command as opposed to the function as the function normally takes precedence.

Exercise

Write a bash script to implement getopts statement.

Your script should understand following command line argument called this script options.sh:

```
./options.sh -c –d -u
```

Where options work as

-c clear the screen

-d show list of files in current working directory

-u print the details of a user running the programme

- show help for running the command if no switches or unknown switches are used

Exercise

1. Write a bash script that produces its source code as output when executed.

2. Write a script that takes a filename and 3 keywords. It should grep in the file for all 3 keywords and display for each keyword the number of matches followed by the line numbers where the matches did occur.
   - No other output on stdout should be produced by the script
   - If the file cannot be read the script should exit with a return code 1, else with code 0 (see help exit if you do not know the exit command)
   - Count the number of characters excluding comments