CSI6203 Scripting Languages

Module 6

Functions

Contents

- Introducing functions and abstraction
- Function parameters
- Returning values from function
- Variable scope
- Command Substitution

Learning Objectives

After finishing this module, you should be able to:

- Understand and execute scripts that use multiple functions
- Send information into functions and retrieve results from functions
- Use command substitution to solve problems

Simple functions

Functions

- Functions are blocks of code that can be run at any time.
- In bash, a function is similar to a script but instead of being saved into a file, it is stored in memory.
- Functions are useful at breaking scripts up into neat logical modules and reducing the need for repeated code.

Functions

- Functions can be created outside of scripts by typing the function code into the bash command line

(The uptime command shows how long the computer has been on)

func_script.sh

CSI6203 ▸ func_script.sh

```bash
#!/bin/bash

displayMemmory()
{
    echo "Mem details"
    free -m
}

displayUptime()
{
    echo "Uptime details"
    uptime
}

displayCPUMemInfo()
{
    echo "CPU Mem info"
    cat /proc/meminfo
}


displayMemmory
displayUptime
displayCPUMemInfo
```

Functions

- To execute a function from the command line, type the name of the function (similar to running a script).

```
-rwxrwxr-x 1 student student   250 Aug 26 09:10 func_script.sh
student@csi6203:~/CSI6203/CSI6203$ ./func_script.sh
Mem details
              total        used        free      shared  buff/cache   available
Mem:           1966         988         131          43         845         767
Swap:          2047           2        2045
Uptime details
 09:11:03 up 15 min,  1 user,  load average: 0.08, 0.19, 0.25
CPU Mem info
MemTotal:        2013464 kB
MemFree:          134472 kB
MemAvailable:     786016 kB
Buffers:          149200 kB
Cached:           605168 kB
SwapCached:          396 kB
Active:           888728 kB
Inactive:         521340 kB
Active(anon):     506332 kB
Inactive(anon):   176752 kB
Active(file):     382396 kB
Inactive(file):   344588 kB
Unevictable:       18548 kB
Mlocked:           18548 kB
SwapTotal:       2097148 kB
SwapFree:        2095100 kB
Dirty:               144 kB
Writeback:             0 kB
AnonPages:        673992 kB
Mapped:           315016 kB
Shmem:             45000 kB
KReclaimable:     111912 kB
Slab:             215288 kB
SReclaimable:     111912 kB
SUnreclaim:       103376 kB
KernelStack:        9976 kB
PageTables:        14836 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
```

Functions

- Functions can be created inside of scripts to allow for easy code re-use.

- Instead of needing to copy-paste large sections of scripts, functions can allow the code to be executed by name

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Create the function for later use

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Create the function for later use

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Print text

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Execute the “colour_green” function

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Print green colour code

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Print text

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Execute the “colour_reset” function

```
#!/bin/bash
colour_green()
{
    echo -e -n "Green Text: \033[32m"
}

colour_reset()
{
    echo -n -e "\033[0m"
}

echo "this is some normal text"
colour_green
echo "this is some green text"
colour_reset
echo "back to normal"
```

Print reset colour code

Output

```
this is some normal text
Green Text: this is some green text
back to normal
```

Function Arguments

Function Arguments

- Functions can have arguments, just like scripts.
- The $1, $2 etc. variables work the same way as they do in scripts

Function Arguments

```
#!/bin/bash
greet_name()
{
    echo "Hello $1"
}

greet_name "Geoff"
greet_name "Sally"
greet_name "Control"
```

Function Arguments

- Output:

```
Hello Geoff
Hello Sally
Hello Control
```

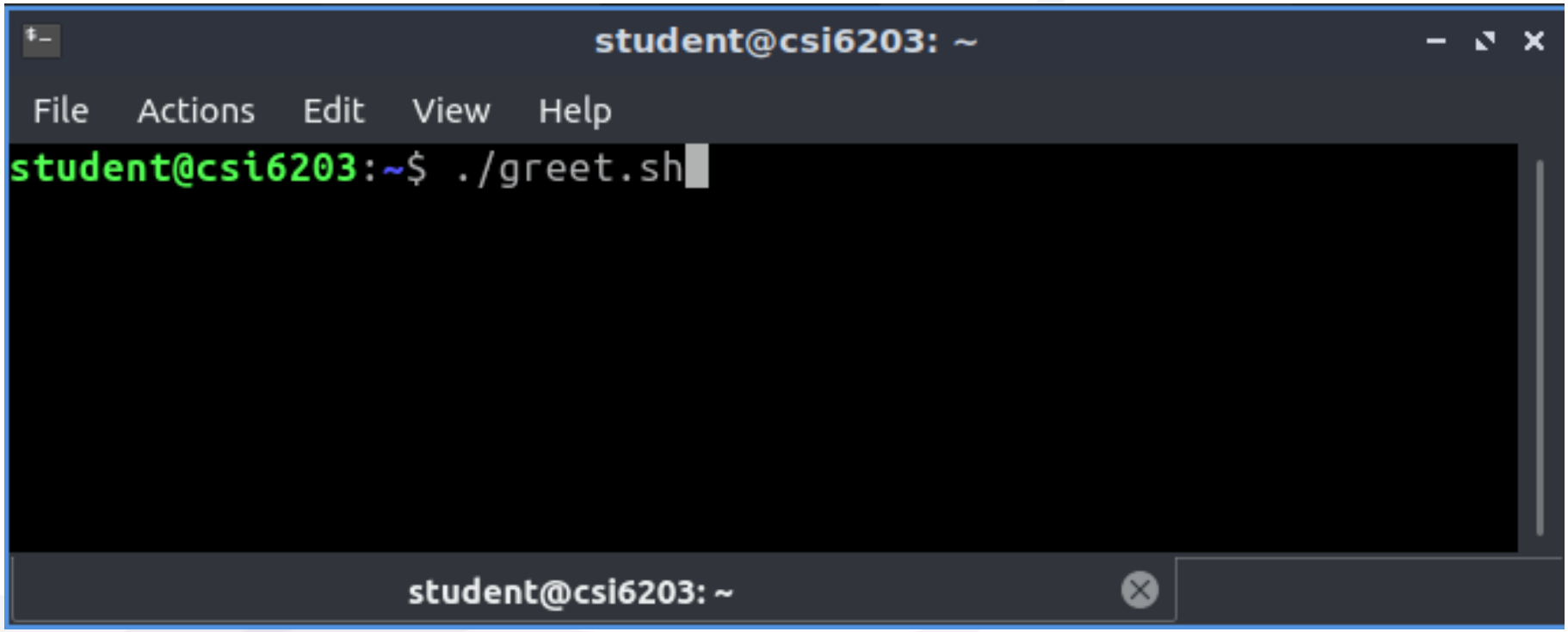
- In many ways, functions can act as scripts within scripts

A more complex example

```
#!/bin/bash
greet_name()
{
    echo "Hello $1"
}

while true
do
    read -p "please type your name or q to quit: "
    if [ "$REPLY" = "q" ] ; then
        break;
    else
        greet_name "$REPLY"
    fi
done
echo "Goodbye!"
exit 0
```

Output


student@csi6203: ~
File Actions Edit View Help
student@csi6203:~$ ./greet.sh
student@csi6203: ~

Scope and Return Values

Variable Scope

- “Scope” refers to the parts of code where a variable can be used
- A variable used before it has a value is considered to be “out of scope”

Variable Scope

- By default, variables are stored globally
- This means any variable created in a function can be used anywhere in the script
- This can lead to problems with large scripts that use the same variable name for different things

Variable Scope

```
#!/bin/bash
func()
{
    word="Toast"
}

word="Test"
echo $word
func
echo $word
```

- The “word” variable here can be changed by the function, even if that is not intended behaviour

Output

```
Test
Toast
```

- The “word” variable here can be changed by the function, even if that is not intended behaviour

Variable Scope

- A better option is to use local variables inside functions.
- A local variable will only exist within the function and will go out of scope as soon as the function is finished

Variable Scope

```
#!/bin/bash
func()
{
    local word="Toast"
}

word="Test"
echo $word
func
echo $word
```

- The “word” variable in func only exists inside the function. It is not the same as the one outside

Output

```
Test
Test
```

- The “word” variable here cannot be changed by the function. This lets the same variable name to be reused in multiple places

Return values

- Often functions will have local variables and echo the results to send data back to the script
- This allows functions to be treated like mathematical functions which have a single result
- This is done by using command substitution

Return values

```
calculate_volume()
{
    local volume=$(( $1*$2*$3 ))
    echo $volume
}

#command substitution
swimmingPool=$(calculate_volume 3 5 10)
echo "The volume of the pool is: $swimmingPool"
bedroom=$(calculate_volume 5 6 12)
echo "The volume of the bedroom is: $bedroom"
```

Command Substitution

- Command Substitution allows the output of a command or function will be stored in variables instead of printed to the screen

```
swimmingPool=$(calculate_volume 3 5 10)
```

Output

```
The volume of the pool is: 150
The volume of the bedroom is: 360
```

Return values

- Bash does have a “return” command similar to other languages.

- However, the return command sets the “exit_status” of the function, so it can only return numeric values and can be accessed using $?

Return values

```
calculate_volume()
{
    local volume=$(( $1*$2*$3 ))
    return $volume
}

#using return
calculate_volume 3 5 10
echo "The volume of the pool is: $?"
calculate_volume 5 6 12
echo "The volume of the bedroom is: $?"
```

Summary

- Terms to review and know include:
  - Functions
  - abstraction
  - Function parameters/arguments
  - Variable scope
  - Command Substitution
  - Return

References and Further Reading

- Ebrahim, M. and Mallet, A. (2018) Mastering Linux Based Scripting (2nd Ed) Chapter 7, pp 125-140
- http://tldp.org/LDP/abs/html/functions.html
- http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-8.html
- https://likegeeks.com/bash-functions