Scripting Languages

Module 6

Writing Functions

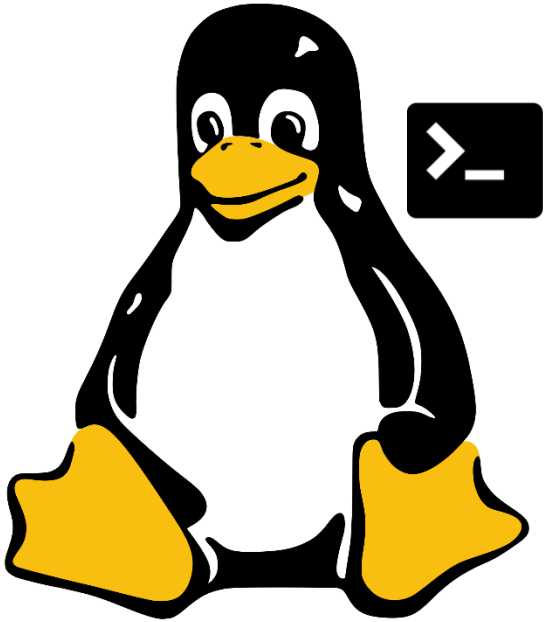
Today’s Concepts

1. Functions Defined
2. Function Basics
3. Variable Scope
4. Command Substitution
5. Writing a Function

Learning Objectives

By the end of this Module you will:

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



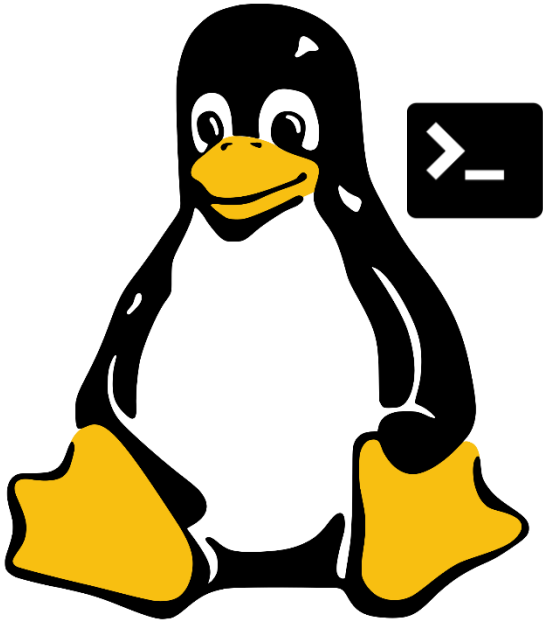
Functions Defined

What are Functions

- Functions are aggregated blocks of code that exist under a declared name, and in combination, perform a highly specific task
- They can be stored in the scripts that calls them to execute or in a separate file that is called in when the parent script is run
- When a bash script is run, any Functions attached to it will be stored in memory ready for use when called upon
- Functions are particularly useful for aggregating set of commands that are frequently required in bash scripts

Advantages of Functions

- ✓ Avoid time-consuming repetition of commonly used code.
- ✓ Make scripts, especially complex scripts, more readable.
- ✓ Allow complex coding tasks to be broken down into much simpler ones.
- ✓ Reduce the likelihood of error – get right the first time, then reuse.
- ✓ Maintaining and updating scripts is much easier when core parts of it are comprised by common functions.



Function Basics

Where Do Functions Go

func_script.sh

CSI6203 ▸ func_script.sh

```
#!/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 are often placed into the head region of a script that will call upon them
- They can also be placed in a separate file and called into a script using the **`source`** command, i.e. **`source func_script.sh`**
- The **`uptime`** command shows how long the computer has been on
- Functions do *not* execute unless they are called by name

How to Call a Function

```bash
#!/bin/bash
source ccodes.sh

ktm() {
    toMiles=1.60934
    echo "You want $1 km converted to miles..."
    miles=$(echo "scale=2; $1/$toMiles" | bc)
    echo -e "${YELLOW}$1${NCOL} km is equivalent to ${GREEN}$miles${NCOL} miles."
}

read -p 'Enter kilometres to convert to miles: ' kvar
ktm $kvar
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshop/week $ ./func2.sh
Enter kilometres to convert to miles: 25
You want 25 km converted to miles...
25 km is equivalent to 15.53 miles.
```

- 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

Function Arguments

```bash
#!/bin/bash

source ccodes.sh

gwtc() {
    curl $2 > wpage.txt
    wcnt=`grep -c -i $1 wpage.txt`
    echo -e "This web page has ${YELLOW}$wcnt${NCOL} instances of the term ${GREEN}$1${NCOL}"
}

read -p 'Enter a search term: ' sterm
echo -e "The search will be conducted for ${YELLOW}$sterm${NCOL}"
read -s -p 'Enter a web page address: ' url
gwtc $sterm $url
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714 ~/CSI6203/workshop/week $ ./func1.sh
Enter a search term: tutorial
The search will be conducted for tutorial
This web page has 15 instances of the term tutorial
vbrown@LAPTOP-N6EFE714 ~/CSI6203/workshop/week $
```

- Functions can have arguments, just like scripts.
- The $#, $1 and $2 variables work the same way as they do in scripts
- In many ways, functions can act as scripts within scripts

Variable Scope Within Functions

- **Scope** refers to the parts of code where a variable can be used
- By default, functions variables are considered global and can be used anywhere in the script
- The danger of this is a variable name being used multiple times for different things
- A better option is to declare variables inside functions as `local`
- A local variable will only exist within the function and will go *out of scope* as soon as the function is finished

```bash
#!/bin/bash

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

room1=$(ccsp 4 3 2)
room2=$(ccsp 5 4 2)
room3=$(ccsp 6 5 2)

echo "Room 1 is $room1 cubic metres"
echo "Room 2 is $room2 cubic metres"
echo "Room 3 is $room3 cubic metres"

exit 0
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714 ~/CSI6203/workshop/week $ ./func4.sh
Room 1 is 24 cubic metres
Room 2 is 40 cubic metres
Room 3 is 60 cubic metres
```

Command Substitution

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

```bash
#!/bin/bash

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

room1=$(ccsp 4 3 2)
room2=$(ccsp 5 4 2)
room3=$(ccsp 6 5 2)

echo "Room 1 is $room1 cubic metres"
echo "Room 2 is $room2 cubic metres"
echo "Room 3 is $room3 cubic metres"

exit 0
```

Command Substitution

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshop/week $ ./func4.sh
Room 1 is 24 cubic metres
Room 2 is 40 cubic metres
Room 3 is 60 cubic metres
```

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 accessible using the **$?** variable

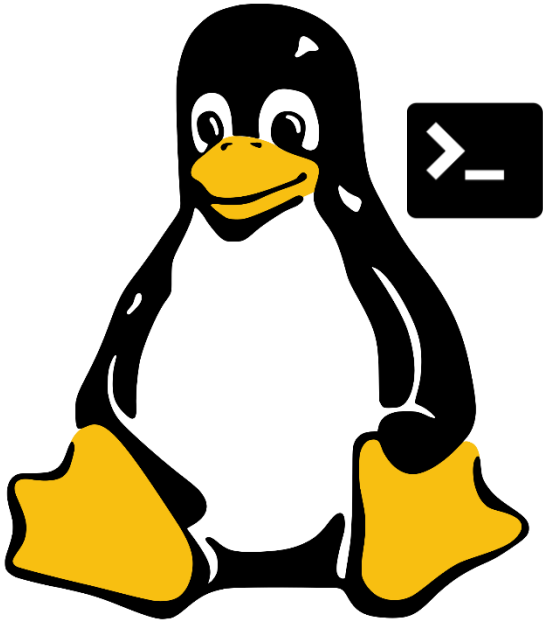
```bash
#!/bin/bash
source ccodes.sh

getpr() {
    local payrate=0
    if [[ $1 =~ ^[1-3] ]]; then
        case $1 in
            1 ) payrate=25 ;;
            2 ) payrate=35 ;;
            3 ) payrate=45 ;;
        esac
    fi
    return $payrate
}

read -p 'Enter the employees pay scale ID: ' pscale
getpr $pscale
pph=$?
if [[ $pph -eq 0 ]]; then
    echo "Invalid Pay Rate ID entered"
else
    echo -e "The employee's hourly rate is ${YELLOW}$pph${NCOL}"
fi
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE71  ~/CSI6203/workshop/week $ ./func5.sh
Enter the employees pay scale ID: 2
The employee's hourly rate is 35
```



Writing a Function

Plan/Write a Function

- Write a function that that calculates the average temperature for each week of daily maximums contained within an external *.csv* file
- The output for each week should be - `The average temperature in Week1 was 23 degrees Celsius`
- The function will need to first strip out the file's header which does not form part of average temperature calculations
- The original *.csv* file is **not** to be altered, so the modified data is to be placed into a temporary **.txt** file for processing
- The user is to be prompted for the *.csv* input file, at which point the function will be invoked
- Name the function **`calcstats`**
- The script is to be named **exf1.sh**

Example Functions

temps.csv

```csv
WEEK,Mon,Tue,Wed,Thu,Fri,Sat,Sun
Week 1,25,22,26,30,25,20,18
Week 2,20,18,24,26,32,32,30
Week 3,32,35,36,40,41,35,35
Week 4,40,39,39,35,35,30,28
Week 5,25,24,30,32,35,33,32
```

```
calcstats() {
 # your code here
}
```

This code is not to be altered

```
read -p 'Enter temperature report file name: '
tmprpt
if ! [[ -f $tmprpt ]]; then
    echo "File not found"
    exit 1
else
    calcstats $tmprpt
fi

exit 0
```

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week $ ./exf1.sh
Enter temperature report file name: temps.csv
The average temperature in Week1 was 23 degrees celcius
The average temperature in Week2 was 26 degrees celcius
The average temperature in Week3 was 36 degrees celcius
The average temperature in Week4 was 35 degrees celcius
The average temperature in Week5 was 30 degrees celcius
```

Function Breakdown

```
calcstats() {

}
```

- Begin by inserting the fundamental function structure and giving it a name, in this case **`calcstats()`**

Function Breakdown

```
lcnt=0
  while IFS= read -r line
    do
      if ! [[ $line =~ [0-9] ]]; then
        (( lcnt++ ))
      fi
  done < "$1“
```

- This code block looks for lines in the source file (provided to the function and held in the default variable **$1**) that do not contain any numeric values.
- If the file contains a header, which contains no numbers, then the variable **`$lcount`** variable will be set to **1**.
- The status of **`$lcount`** (0 or 1) can then be used to determine subsequent script actions

Function Breakdown

```
if [[ $lcnt -eq 1 ]]; then
    printf "%s\n\n" "$(tail -n +2 $1)" > tempraw.txt
fi
```

- If **`$lcount`** is equal to 1, then the source file *does* contain a header, so we want to remove it before copying the data to a temporary file named **`tempraw.txt`**
- To do this, the **`tail`** command has been used to print the contents of the source file to the temporary file from Line 2 (**+2**) onwards, this ensuring the header on Line 1 is not copied across to **`tempraw.txt`**

Function Breakdown

```
while IFS="," read -r week mon tue wed thu fri sat sun
    do
        tmpsum=$(($mon+$tue+$wed+$thu+$fri+$sat+$sun))
        let tmpavg=$tmpsum/7
        echo "The average temperature in $week was $tmpavg degrees celcius"
    done < tempraw.txt 2>/dev/null
```

- Setting the IFS to comma (,), setting tempraw.txt as the source and using a **`while`** loop, **`read`** each value on each line into the ordinally corresponding variable name, i.e. **`week mon tue`** etc.
- Get the sum of each variable populated, excluding **`$week`** and assign to variable **`tmpsum`**, then get the average by dividing **`tmpsum`** by 7 and assign to variable **`tmpavg`**
- Echo output regarding average temperature to the terminal

Function Breakdown

```
read -p 'Enter temperature report file name: ' tmprpt
if ! [[ -f $tmprpt ]]; then
    echo "File not found"
    exit 1
else
    calcstats $tmprpt
fi

exit 0
```

1

```
calcstats() {

    lcnt=0
    while IFS= read -r line
        do
            if ! [[ $line =~ [0-9] ]]; then
                (( lcnt++ ))
            fi
    done < "$1"

    if [[ $lcnt -eq 1 ]]; then
        printf "%s\n\n" "$(tail -n +2 $1)" > tempraw.txt
    fi

    while IFS="," read -r week mon tue wed thu fri sat sun
        do
            tmpsum=$(($mon+$tue+$wed+$thu+$fri+$sat+$sun))
            let tmpavg=$tmpsum/7
            echo "The average temperature in $week was $tmpavg degrees celcius"
        done < tempraw.txt 2>/dev/null

}
```

2

3

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/worksheet $ ./exf1.sh
Enter temperature report file name: temps.csv
The average temperature in Week1 was 23 degrees celcius
The average temperature in Week2 was 26 degrees celcius
The average temperature in Week3 was 36 degrees celcius
The average temperature in Week4 was 35 degrees celcius
The average temperature in Week5 was 30 degrees celcius
```

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

Terms to Review and Know

- Functions
- Function parameters/arguments
- Variable scope
- Command Substitution
- Return