School of
Science

ECU AUSTRALIA EDITH COWAN UNIVERSITY

Scripting Languages

Module 4

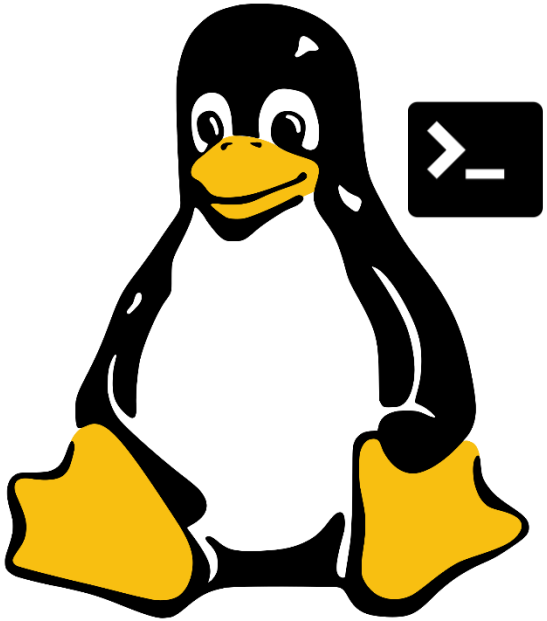
Managing Repetition
with Loops

Contents

Learning Objectives

By the end of this Module you should:

- Understand and execute scripts that require iteration
- Write scripts that iterate through content using a range of loop structures



For Loops

for loops

- **for** is a shell keyword used to control iteration
- Iteration allows one or more commands to be executed for each item within a list of items
- These items may be contained within a *variable*, an *array* or an *external file*

FOR LOOP BASIC STRUCTURE

```
for i in items_list; do
    command(s) to be executed
    for each item
done
```

for loop example - array

- In a for loop, we read each item in the list from left to right
- If the list is a string of text, the items are separated by spaces by default
- Each value in the list is assigned to the variable on the left one at a time

```
1   #!/bin/bash
2
3   declare -a prof_array
4   prof_array=($USER $HOME $EUID $HOSTNAME $HOSTTYPE)
5
6   for i in "${prof_array[@]}"; do
7       echo -n "The current value of "
8       echo -n '$i'
9       echo " is now $i"
10  done
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week    $ ./forloop1.sh
The current value of $i is now vbrown
The current value of $i is now /home/vbrown
The current value of $i is now 1000
The current value of $i is now LAPTOP-N6EFE714
The current value of $i is now x86_64
```

The IFS

- The **Internal Field Separator** (IFS) variable is used by the system to tell where one item in a list ends and the next one starts
- By default, this is a space so that structures such as *for loops* will count through each word in a list
- By setting this to something else, we can make it split each item a list in a different way e.g. newlines (\n)

Change IFS value

```bash
#!/bin/bash

orig_ifs=IFS # save the deafult IFS (blank space) to a variable $orig_ifs
IFS=$'\n' # set $IFS value to newline \n
cnt=1 # create a counter and initialise to 1
for line in $(cat datafile.txt); do # read in each line of datafile.txt into for loop variable $line
    if [[ $line == *"SRAM"* ]]; then # check if current line contains the substring SRAM
        echo "Line $cnt: $line" # if yes, echo the line numver ($cnt) and the line itself
    else
        echo "Line $cnt: no match" # otherwise echo no match
    fi
    ((cnt++)) # increment counter by 1
done
IFS=orig_ifs # restite #IFS with its orginal value
exit 0 # exit program
```

shell script

```
Random access memory (RAM) is a general-purpose memory that usually stores the user data
in a program. RAM is volatile in the sense that it cannot retain data in the absence of power;
i.e., data is lost after the removal of power. The RAM in a system is either static RAM (SRAM)
or dynamic RAM (DRAM). The SRAMs are fast, with access time in the range of a few nanoseconds,
which makes them ideal memory chips in computer applications. DRAMs are slower and because they
are capacitor based they require refreshing every several milliseconds. DRAMs have the advantage
that their power consumption is less than that of SRAMs.
```

datafile.txt

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week    $ ./forloop2.sh
Line 1: no match
Line 2: no match
Line 3: i.e., data is lost after the removal of power. The RAM in a system is either static RAM (SRAM)
Line 4: or dynamic RAM (DRAM). The SRAMs are fast, with access time in the range of a few nanoseconds,
Line 5: no match
Line 6: no match
Line 7: that their power consumption is less than that of SRAMs.
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week    $
```

output

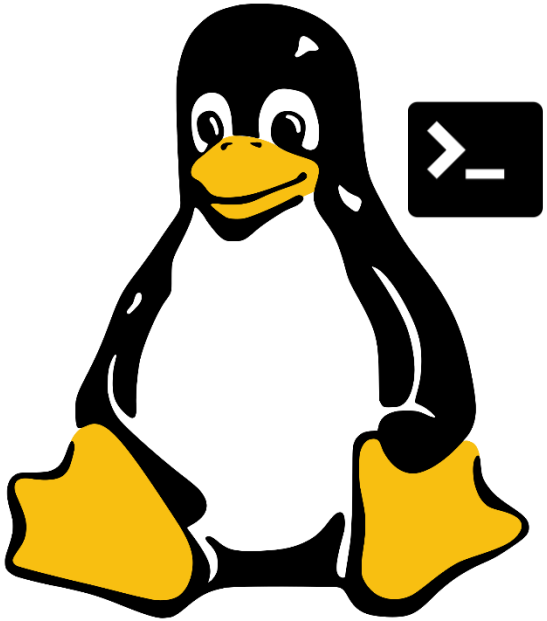
for loops with files directories

```bash
#!/bin/bash

for item in .* *; do
    if [ -d $item ]; then
        echo -e "$item is a folder"
    elif [ -f $item ]; then
        echo -e "$item is a file"
    else
        echo "Item type unknown"
    fi
done
exit 0
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week $ ./forloop3.sh
. is a folder
.. is a folder
archive is a folder
cstyleloops.sh is a file
forloop3.sh is a file
sl is a folder
untildemo.sh is a file
videos is a folder
```

- For loops are also often used to iterate through the contents of files and directories
- If the **-d** test evaluates to true, then echo that the item is a *folder*
- If the **-f** test evaluates to true, then echo that the item is a *file*
- If neither the -d or the -f test evaluate true, echo that the item is of an indeterminate type



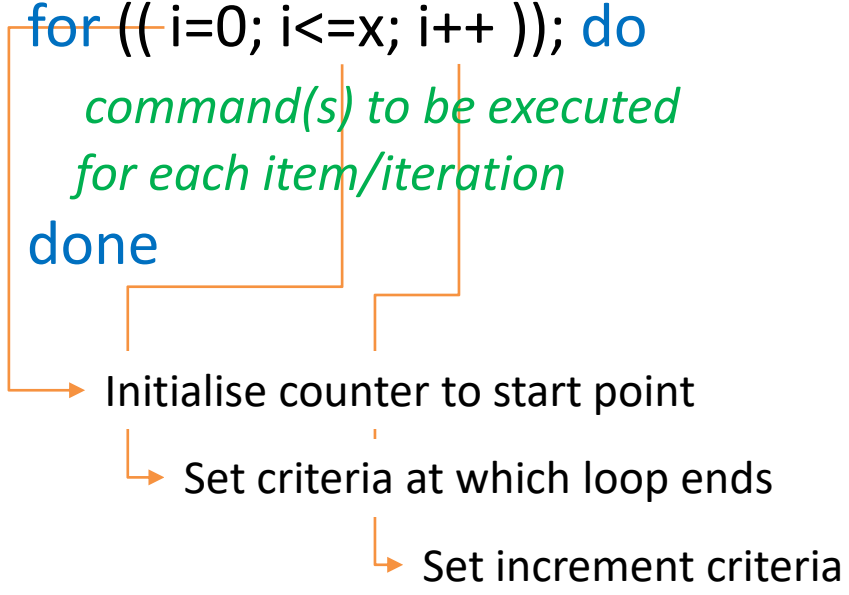
C-Style Loops

C-style for loops

- Bash also supports C-style for loops that count a specified number of times
- The *C-style for loop* sets an *initial value*, a *guard* and an *increment* within the loop
- This is very similar to for loops in other programming languages such as java, C# and C++

FOR C-STYLE LOOP BASIC STRUCTURE

```
for (( i=0; i<=x; i++ )); do
    command(s) to be executed
    for each item/iteration
done
```



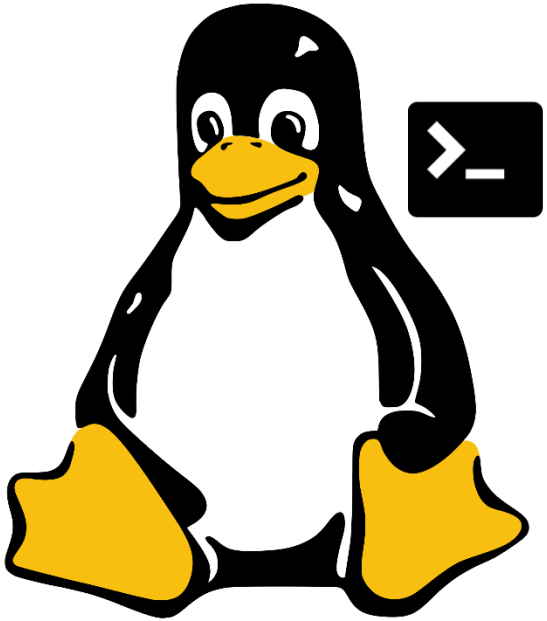
C-style for loops

```
1   #!/bin/bash
2
3   declare -a ldistro # declare an array named ldistro to hold my favourite distros
4   ldistro=(Ubuntu Mint Elementary Zorin SUSE CentOS Debian RedHat Gentoo Arch Manjaro Slackware Fedora OpenSUSE Solus Peppermint)
5   len=${#ldistro[*]} # get the total number of elements in the ldistro array
6
7   echo "MY FAVOURITE LINUX DISTROS" # echo a header to the terminal
8   for (( i=0; i<${len}; i++ )); do # set counter to 1, set end condition to length of array, increment by 1
9       echo -e "$(($i+1))\t${ldistro[$i]}" # echo distro number and distro name
10  done
11
12  exit 0
```

CODE EXPLAINED:

1. Declare an array [Line 3]
2. Populate array with values [Line 4]
3. Get length of array [Line 5]
4. Print each array item to terminal with its ordinal positon [Lines 8-10]

```
N6EFE714:~/ ./workshops/ws5$ ./csl.sh
MY FAVOURITE LINUX DISTROS
1       Ubuntu
2       Mint
3       Elementary
4       Zorin
5       SUSE
6       CentOS
7       Debian
8       RedHat
9       Gentoo
10      Arch
11      Manjaro
12      Slackware
13      Fedora
14      OpenSUSE
15      Solus
16      Peppermint
```



While Loops

While loops

- For loops are mostly useful when we know exactly how many times we want commands to repeat
- In many cases however, we need to keep looping until a certain condition is met, for example:
  - *repeat while* the user has not chosen to exit
  - *repeat until* a correct value is entered
  - *repeat while* there is still additional information being written
- This is when **while** loops come in handy

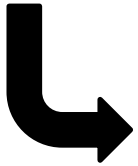
WHILE LOOP BASIC STRUCTURE

```
while (( x -gt y )); do
    command(s) to be executed
    for each item/iteration
done
```

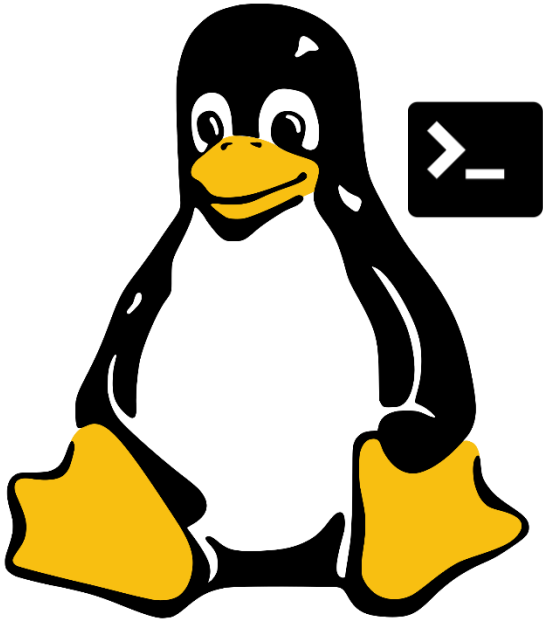
Loop end criteria

While loop example

```
1   #!/bin/bash
2
3   value1=1 # set a variable named value1 to 1
4   read -p 'Enter a value between 5 and inclusive: ' value2 # prompt user for a
    value and assign to variable value2
5   while [ $value2 -gt 0 ] # set while loop end criteria
6   do
7       value1=$(( $value1 * $value2 )) # code to be iterated until loop end criteria
        is reached
8       value2=$(( $value2 - 1 ))
9       echo "Value 1 is now $value1 and Value 2 is $value2"
10  done
11  echo $value1 # echo final value now stored in variable $value1
12  exit 0
```



```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week $ ./wl1.sh
Enter a value between 5 and inclusive: 5
Value 1 is now 5 and Value 2 is 4
Value 1 is now 20 and Value 2 is 3
Value 1 is now 60 and Value 2 is 2
Value 1 is now 120 and Value 2 is 1
Value 1 is now 120 and Value 2 is 0
120
```



Until Loops

Until Loops

- An **until** loop is used to execute a given set of commands as long as the given condition evaluates to *false*
- The condition is evaluated *before* executing the commands
- If the condition evaluates to false, commands are executed
- Otherwise, if the condition evaluates to *true*, the loop will be terminated and program control will be passed to whatever code follows

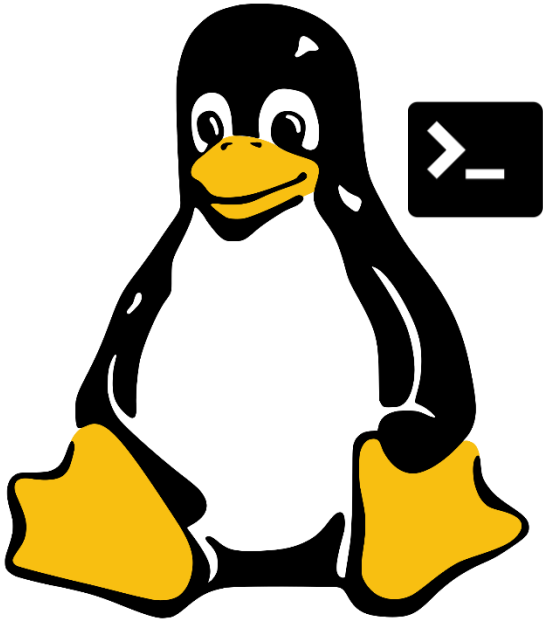
UNTIL LOOP BASIC STRUCTURE

```
until [ conditional_test ]; do
    command(s) to execute if
    condition is false
done
```

Until Loop Example

```bash
#!/bin/bash

floor=10 # set value below which until loop will exit
i=20 # set the counter
result=0 # initialize a variable to hold cumulative sum of counter

# run until loop with a single condition
until [ $i -lt $floor ]; do # set the test
    result=$(($result+$i)) # add the current value of the counter to the result variable
    echo "The counter is set at $i and result is set at $result" # print the current values of $i and $result to terminal
    ((i--)) # decrement counter by 1
done
```

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week    $ ./w6until.sh
The counter is set at 20 and result is set at 20
The counter is set at 19 and result is set at 39
The counter is set at 18 and result is set at 57
The counter is set at 17 and result is set at 74
The counter is set at 16 and result is set at 90
The counter is set at 15 and result is set at 105
The counter is set at 14 and result is set at 119
The counter is set at 13 and result is set at 132
The counter is set at 12 and result is set at 144
The counter is set at 11 and result is set at 155
The counter is set at 10 and result is set at 165
```



Break and Continue

Loop Controls – Break and Continue

- The loop controls break and continue can be use to change the behaviour of loops
- These are primarily useful for error handling or to skip unwanted items
- The break statement allows *exit* from a loop when a condition is met
- The continue statement *skips* the current iteration and moves on to the next one

Break Example

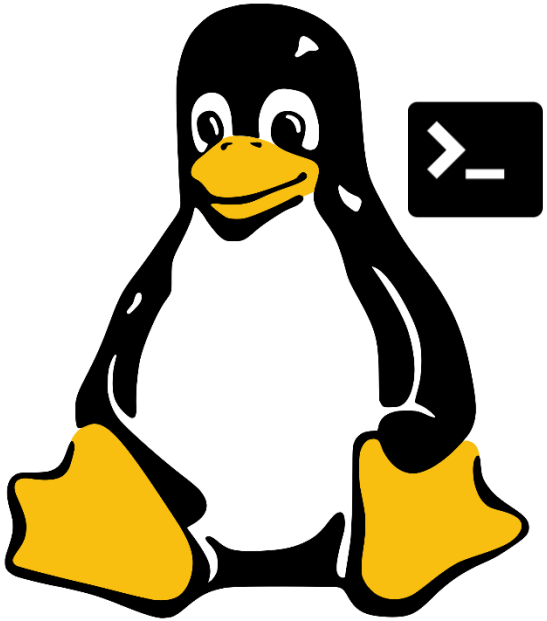
```
1   #!/bin/bash
2
3   while true; do # begin loop
4       read -p 'Enter a number between 5 and 10 inclusive: ' var # prompt user for
        a number between 1 and 10 inclusive
5           if [[ $var -lt 5 ]] || [[ $var -gt 10 ]]; then # if invalid number
            given, loop back to prompt
6               echo "Invalid input, please try again"
7           else
8               break # if valid number given, exit the loop
9           fi
10  done
11
12  echo "Thank you, you have entered $var" # echo the input number to terminal
13  exit 0
```

```
$ ./brk.sh
Enter a number between 5 and 10 inclusive: 4
Invalid input, please try again
Enter a number between 5 and 10 inclusive: 11
Invalid input, please try again
Enter a number between 5 and 10 inclusive: 8
Thank you, you have entered 8
```

Continue Example

```bash
#!/bin/bash

declare -a numlist # declare an array named numlist to hold a range of integers
numlist=(12 15 18 21 23 27 30 33 36 40 48 51 56 60 63)
len=${#numlist[*]} # get the total number of elements in the numlist array

for (( i=0; i<${len}; i++ )); do # set counter to 0, set end condition to length of array, increment by 1
    if ! [[ $((${numlist[$i]} % 2)) -eq 0 ]]; then # if there's a remainder, integer is odd so skip it
        continue
    else
        echo "${numlist[$i]} is an even number" # otherwise integer is even so echo to
    fi
done
exit 0
```

```
vbrown@LAPTOP-N6EFE714  ~/CSI6203/workshop/week $ ./cont.sh
12 is an even number
18 is an even number
30 is an even number
36 is an even number
40 is an even number
48 is an even number
56 is an even number
60 is an even number
```

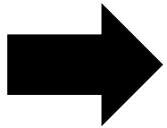


Nested Loops

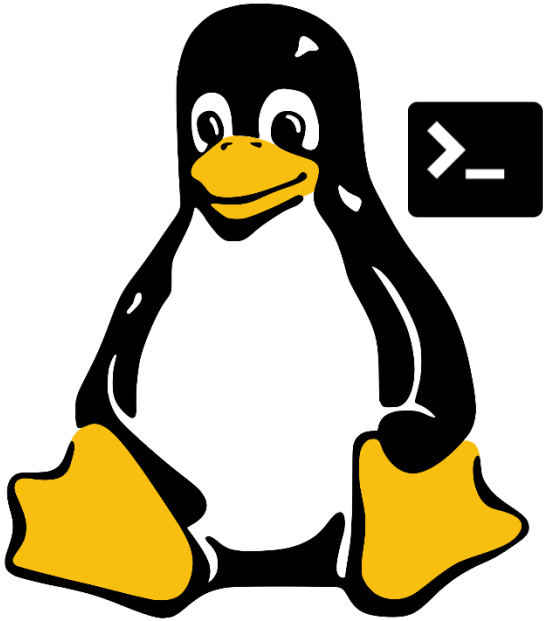
Nested loops

```bash
#!/bin/bash

outerloop=1 # Set outer loop counter

# Beginning of outer loop
for a in 1 2 3
do
  echo "Iteration $outerloop of outer loop"
  innerloop=1 # Set inner loop counter

  # Beginning of inner loop
  for b in 1 2 3 4 5
  do
    echo -e "\tInteration $innerloop of inner loop."
    let "innerloop=$innerloop+1"  # Increment inner loop counter
  done
  # End of inner loop

  let "outerloop=$outerloop+1"    # Increment outer loop counter
done
# End of outer loop

exit 0
```

- Loops can be placed inside each other.
- The entire inner loop will be repeated by the outer loop



```
vbrown@LAPTOP-N6EFE714 ~/CSI6203/wor
Iteration 1 of outer loop
        Interation 1 of inner loop.
        Interation 2 of inner loop.
        Interation 3 of inner loop.
        Interation 4 of inner loop.
        Interation 5 of inner loop.
Iteration 2 of outer loop
        Interation 1 of inner loop.
        Interation 2 of inner loop.
        Interation 3 of inner loop.
        Interation 4 of inner loop.
        Interation 5 of inner loop.
Iteration 3 of outer loop
        Interation 1 of inner loop.
        Interation 2 of inner loop.
        Interation 3 of inner loop.
        Interation 4 of inner loop.
        Interation 5 of inner loop.
```



Infinte Loops

Infinite Loops

- There is nothing in bash that stops you from creating loops that cannot finish.
- These can be created by using a *guard* that:
  - Has a boolean expression that can never be false
  - Has a boolean expression that can be false but doesn't reach that case
  - Has an error that causes the loop to not execute the statements within

```
1  #!/bin/bash
2
3  while true; do
4      echo 'Use CTRL+C to escape infinite loop'
5      sleep 1
6  done
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week6$ ./inf.sh
Use CTRL+C to escape infinite loop
Use CTRL+C to escape infinite loop
Use CTRL+C to escape infinite loop
Use CTRL+C to escape infinite loop
Use CTRL+C to escape infinite loop
Use CTRL+C to escape infinite loop
```

Infinite Loop Example

- Write a script that when run, prompts the user to enter a three-digit integer that is **> 1000** but **< 2000**
- In this case, an infinite loop structure is used to ensure user cannot proceed until a valid input is provided
- The **break** keyword allows the loop to be escaped when a valid value has been provided

```bash
#!/bin/bash

# infinite loop wrapper to ensure usr cannot proceed unless valid int provided
while true; do
# prompt for input from user, being clear in what is required and assign to variable
read -p 'Enter a three-digit integer greater than 1000 and less than 2000: ' usrint
    # test that user inout value is an int within the required range
    if [[ $usrint -gt 1000 ]] && [[ $usrint -lt 2000 ]]; then
        break # if yes, break out of infinte loop and proceed to next logic block
    else
    # if no, inform user of issue then loop them back to original prompt
        echo "Invalid input! Please try again."
    fi
done
# once valid int is provided, inform user of such
echo "Success. You have entered a valid integer - $usrint"
# exit the program with success code
exit 0
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws5$ ./validint.sh
Enter a three-digit integer greater than 1000 and less than 2000: 2000
Invalid input! Please try again.
Enter a three-digit integer greater than 1000 and less than 2000: 1000
Invalid input! Please try again.
Enter a three-digit integer greater than 1000 and less than 2000: 999
Invalid input! Please try again.
Enter a three-digit integer greater than 1000 and less than 2000: 2001
Invalid input! Please try again.
Enter a three-digit integer greater than 1000 and less than 2000: helloworld
Invalid input! Please try again.
Enter a three-digit integer greater than 1000 and less than 2000: Just pressed Enter
Invalid input! Please try again.
Enter a three-digit integer greater than 1000 and less than 2000: 1500
Success. You have entered a valid integer - 1500
```

References and Further Reading

Ebrahim, M. and Mallet, A. (2018) *Mastering Linux Based Scripting* (2nd Ed), Chapter 6, pp 102-120

Terms to Know

- Iteration
- For Loops
- C-Style Loops
- While Loops
- Until Loops
- Break and Continue
- Nested Loops
- Infinite Loops