while loop

Executes one or more instructions while a condition is true.
It stops when the control condition is true or when the execution is intentionally stopped by the programmer with an explicit interruption instruction (break or continue)

Syntax:

```
while CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=0
while [  ${counter} -lt 10 ]; do
    echo The counter is ${counter}
    let counter=counter+1
done
```

Example of break statement in while loop

Interrupt the loop at number … (try)

```
#!/bin/bash

num=1
while [ ${num} -lt 10 ]
do
   if [ ${num} -eq 5 ]
   then
      echo “${num} equal to 5 so I interrupt the loop”
   break
   fi
   echo ${num}
   let num+=1
done
echo “Loop is complete”
```

until loop

Executes one or more instructions until a condition is false.

Syntax:

```
until CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=20
until [  ${counter} -lt 10 ]; do
    echo counter ${counter}
    let counter-=1
done
```

until vs. while

Until is similar to while, but it is a slightly difference:
Until is executed while the condition is false,
While is executed while the condition is true.
What means it?

Try the following code and check the output:

```
num=1
while [[ $num -lt 10 ]]
do
  if [[ $num -eq 5 ]]
  then
    break
  fi
  echo $num
  let num=num+1
done
echo “Loop while is complete”
```

```
num1=1
until [[ $num1 -lt 10 ]]
do
  if [[ $num1 -eq 5 ]]
  then
    break
  fi
  echo $num1
  let num1=num1+1
done
echo “Loop until is complete”
```

Loop through array elements

${myArray[@]} return all the elements of an array
replace the numeric index with the @ symbol can be thought as standing for all.

Example: Loop on all elements of the array:

```
myArray=(1, 3, 5, ”try” , ”this” ,1)

for t in ${myArray[@]}; do
  echo array element ${t}
done
```

Loop through array indices

${!allThreads[@]} returns all the indexes in an array.

Example: Loop on all indexes of the array:

```
myArray=(1 3 5 ”try”  ”this” 1)

for i in ${!myArray[@]}; do
  echo “Array element ${i} is = ${myArray[$i]}”
done
```

Functions

Functions are use to group sets of commands logically related making them reusable without the need to re-write them.

A function does not need to be declared (like in compiled languages).

Function example:

```
#!/bin/bash

function quit {
 exit 9
}

function hello {
    echo Hello!
}
hello
quit
exit 0
```

Syntax:

```
function func_name {
    command1
    command2
    …..
}
```

How to call the function in a script:

```
func_name
```

Functions parameters/arguments

Parameters does not need to be declared.

It is good practice

- to put a comment before the function definition describing parameters and their meaning
- Read the parameters at the beginning of the function

Function with parameters example:

```
#!/bin/bash
function quit {
    exit 0
}
# input parameter msg=”a message”
function my_func {
    msg=$1
    echo $msg
}
my_func Hello
my_func World
quit
echo foo
```

Syntax with parameters:

```
function func_name {
        command1
        command2
        …..
}
```

How to call the function with parameters in a script:

```
func_name param1 param2 ...
```

Add help to a script

```
cat usage.sh

#!/bin/bash

function display_usage() {

     echo -e "\n Usage:\n $0 [arguments] \n"
}

# if less than two arguments supplied, display usage
if [[  $# -le 1 ]]
  then
    display_usage
    exit 1
fi
```

Add help to a script

Example

```
#!/bin/bash
if [ -z "$1" ]; then        # check if one parameter exists
    echo Usage: $0 directory
    exit 1
fi
srcd=$1
bakd="/tmp/"
mkdir ${bakd}
of=home-$(date +%Y%m%d).tgz
tar -czf ${bakd}${of} ${srcd}
```

Positional parameters

Positional parameters are a series of special variables ($0 through $9) that contain the contents of the command line.
If my_script is a bash shell script, we could read each item on the command line because the positional parameters contain the following:
$0 would contain "some_program"
$1 would contain "parameter1"
$2 would contain "parameter2"
…..

This way, if I call my_script with two parameters:
my_script Hello world
Then inside the script I can read them with:

```
#!/bin/bash
script_name=$0
first_word=$1
second_word=$2
Echo “$script_name says $first_word $second_word
```

The mechanism is the same to read functions parameters.

Read the user’s input examples

- Example on how to read the user’s input:

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

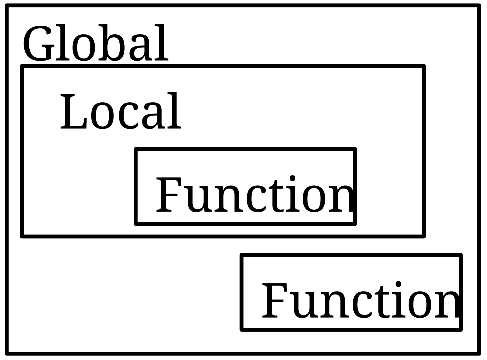
- Example on how to read multiple user’s input:

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
echo "How are you?"
```

Scope of variables

In general you can distinguish between

Global
Local — Scope
Function



Bash (like Python) doesn't have block scope in conditionals.

It has local scope within functions, it is also possible to use the ‘local’ modifier which is a keyword to declare the local variables.
Local variables are visible only within the block of code.

Variable scope (visibility) is related mainly to the shell.
Exported variables are visible in all subshells.

Scope of variables

A variable exported is a global variable.

A variable defined in the main body of the script:
• it will be visible throughout the script;
• it will be visible and accessible inside functions within the script;
• a variable which is defined “local” inside a function is local to that function; it is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.

• Global variables can have unintended consequences because of their wide-ranging effects: we should almost never use them

Exercise: Scope of variables

```
#!/bin/bash
e=2
echo At beginning e = $e
function test1() {
  e=4
  echo "hello. Now in the function1 e = $e"
}
function test2() {
  local e=4
  echo "hello. Now in the function2 e = $e"
}
test1
echo "After calling the function1 e = $e"

e=2
echo In the file before to call func2 reassign e = $e
test2
echo "After calling the function2 e = $e"
```

Justify the result !