Module 5 - Shell scripting

By Juan Medina

jmedina@collin.edu

COLLIN COLLEGE

Introduction to Shell Scripting

What Are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line. The shell is somewhat unique, in that it is both a powerful command-line interface to the system and a scripting language interpreter.

Everything you can do on the command line can be done in a scripts

How To Write A Shell Script

To successfully create and run a shell script, we need to do three things:

1. **Write a script**. Shell scripts are ordinary text files. It is recommended to use a text editor that provides syntax highlighting, allowing us to see a color-coded view of the elements of the script. (vim, gedit, kate, geany, visual studio code).
2. **Make the script executable**. We need to set the script file's permissions to allow execution.
3. **Put the script somewhere the shell can find it.** Place the script in a directory that can be find in the PATH.

Script File Format

The first line of our script is a little mysterious. It looks as if it should be a comment, since it starts with #, but it looks too purposeful to be just that. The #! character sequence is, in fact, a special construct called a shebang. The shebang is used to tell the system the name of the interpreter that should be used to execute the script that follows. **Every shell script should include this as its first line.**

```
#!/bin/bash
```

The second line is a comment line and they can be also set at the end of the line like our third line here

```
# This is our first script.
echo 'Hello World!'      # This is also a comment
```

Also in the third line of our script, we send an echo command with a string argument. Comments can also be used in the command line if needed

```
echo 'Hello World!' # This is a comment too
```

So our final script looks like this:

```
#!/bin/bash
# This is our first script.
echo 'Hello World!'      # This is also a comment
```

Let’s save our script file as hello_world.sh

Executable Permissions

The next thing we have to do is make our script executable. This is easily done using chmod command:

```
[jmedinar@localhost ~]$ chmod 755 hello_world.sh
```

or

```
[jmedinar@localhost ~]$ chmod +x hello_world.sh
```

With the permissions set, we can now execute our script:

```
[jmedinar@localhost ~]$ ./hello_world
Hello World!
```

Script File Location

In order for the script to run, we must let Linux know the location of our script. Otherwise we get the following error:

```
[jmedinar@localhost ~]$ hello_world
bash: hello_world: command not found
```

We can do this with the `.` symbol that indicates `current location` or `here`

```
[jmedinar@localhost ~]$ ./hello_world
Hello World!
```

We can also provide the full path to our script

```
[jmedinar@localhost ~]$ /home/jmedinar/hello_world
Hello World!
```

But for Linux to be able to find our script from any location in the system we have to add it’s current location to the `$PATH` environment variable or move our script to a location already recognized by `$PATH`. Let's understand `$PATH`.

```
[jmedinar@localhost ~]$ echo $PATH
/home/jmedinar/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
:/sbin:/bin:/usr/games
```

`$PATH` is an environment variable. This means it is always there and it is created everytime we boot Linux or we open a new shell. It contains routes to all the locations in the system where `executable` files can be found. It is a list separated by colons `:`

If we copy our script to any of the locations in `$PATH` it will be reachable from anywhere! and our problem would be solved.

But since this is a personal script and we don't want to share it with the rest of the users let's place it in a proper location and add it only to our own `$PATH`.

```
[jmedinar@localhost ~]$ cd ~
[jmedinar@localhost ~]$ mkdir bin
[jmedinar@localhost ~]$ mv hello_world.sh bin
```

We have created a new folder in our home directory called `bin` and moved our script there. Now let's add this new directory to our `$PATH` , we can easily do this by including this line in our `.bashrc` file, or running it as a command:

```
export PATH=~/bin:"$PATH"
```

What's happening?. We are exporting the environment variable `$PATH` with a new directory `~/bin` that points to our `$HOME/bin` directory and then adding `:"$PATH"` to include the already existing content of the variable otherwise we will loose it's original content. This will take effect on all the new sessions. But not in the current session! because our environment was already loaded!

To apply the change to the current session, we must force the shell re-read the .bashrc file. This can be done by "sourcing" it:

```
[jmedinar@localhost ~]$ source .bashrc
```

Limitations of Shell Scripts

The shell manipulates commands and files with relative ease. It can redirect output, one of the important elements of shell script programming. However, the shell script is only one tool for Linux programming, and although scripts have considerable power, they also have limitations.

One of the main strengths of shell scripts is that they can simplify and automate tasks that you can otherwise perform at the shell prompt, like manipulating batches of files. But if you're trying to pick apart strings, perform repeated arithmetic computations or access complex databases, or if you want functions and complex control structures, you're better off using a programming language.

Finally, be aware of your shell script size. Keep your shell scripts short. Scripts aren't meant to be big (though you will undoubtedly encounter some monstrosities). They are limited to 5000 lines but if your script is near 1000 you are doing something wrong!

Comments & Style

Making good comments in our scripts is extremely important. What is intuitively obvious to us may be total non-sense to others who follow in our footsteps. We have to write code that is readable and has an easy flow. This involves writing a script that is easy to read and easy to maintain, which means that it must have plenty of comments describing the steps.

There is nothing worse than having to hack through someone else's code that has no comments to find out what each step is supposed to do. It can be tough enough to modify the script in the first place, but having to figure out the mindset of the author of the script will sometimes make us think about rewriting the entire shell script from scratch.

We can avoid this by writing a clearly readable script and inserting plenty of comments describing what our philosophy is and how we are using the input, output, variables, and files.
For good style:

- Make it readable.
- break commands instead of stringing, or piping, everything together on the same line.
- or add the proper comments describing our thinking step by step.

A beautiful style:

```
#!/bin/bash
# SCRIPT: NAME_of_SCRIPT
# AUTHOR: AUTHORS_NAME
# DATE: DATE_of_CREATION
# PURPOSE: Give a clear, and if necessary, long, description of the purpose of the shell script.
#          This will also help you stay focused on the task at hand.

# REVISIONS:
#                       DATE: DATE_of_REVISION
#                       BY:  AUTHOR_of_MODIFICATION
#                       MODIFICATION: Describe what was modified, new features, etc--
#

# set -n                   # Uncomment to check script syntax, without execution.
# set -x                   # Uncomment to debug this shell script

# === Variable Definitions ===

# ===  Functions ===

# ===  Main ===

# End of script
```

Proper Indentation

It will be hard to understand and read a script that looks like this:

```
#!/bin/bash
MAX_NO=0;echo -n "Enter a Number between (5 to 9) : ";read MAX_NO
if ! [ $MAX_NO -ge 5 -a $MAX_NO -le 9 ]then;echo "Seriously!... I ask to enter number between 5 and 9, Try Again";exit 1
fi
clear;for (( i=1; i<=MAX_NO; i++ ));do;echo -e "Whenever you need help, LINUX is always there";done
```

This script will be a lot easier to understand if written this way:

```
#!/bin/bash
MAX_NO=0
echo -n "Enter Number between (5 to 9) : "
read MAX_NO
if ! [ $MAX_NO -ge 5 -a $MAX_NO -le 9 ]
then
      echo "Seriously!... I ask to enter a number between 5 and 9, Try Again"
      exit 1
fi
clear
for (( i=1; i<=MAX_NO; i++ ))
do
      echo -e "Whenever you need help, LINUX is always there"
done
```

A good structure and format will always help! **There are no restrictions on how you write your script but it will be part of your presentation card among IT professionals!**

Variables

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data. A variable is nothing more than a pointer to the actual data.

There are two actions we may perform for variables:

- Setting a value for a variable.
- Reading the value for a variable.

For example, the following creates a shell variable and then prints it:

```
[jmedinar@localhost ~]$ variable ="Hello"
[jmedinar@localhost ~]$ echo $variable
```

Here are a few quick points on syntax.

- When reading a variable we place a `$` sign before the variable name. **We call this referencing!**
- When setting a variable we leave out the `$` sign. **We call this assigning!**
- They can be all uppercase, all lowercase, or a mixture. But is a good practice to leave UPPERCASE only for Environment variables and a combination for Script Variables. It's your preference however.
- A variable may be placed anywhere in a script and, when run, Bash will replace it with the value of the variable. This is made possible as the substitution is done before the command is run.

Quoting & Literals

How do we know which quoting to use in our scripts, functions, and command statements?. This decision causes the most confusion in writing scripts. We are going to set this straight now. Depending on what the task is and the output desired, it is very important to use the correct enclosure. Failure to use these correctly will give unpredictable results.

Double Quotes

We use **" "** **, double quotes,** in a statement where we want to **allow special characters**. Double quotes are required when defining a variable with data that contains white space, as shown here.

```
[jmedinar@localhost ~]$ NAME="Bob Squarepants"
```

If the double quotes are missing we will get an error.

We also use this when we want **command substitution**.

```
[jmedinar@localhost ~]$ echo "Who lives in a pineapple under the
sea?: $NAME"
Who lives in a pinneapple under the sea?: Bob Squarepants
```

Single Quotes

We use **''**, **forward tics (single quotes)**, in a statement where we **do not want character or command substitution**. Enclosing in forward tics is intended to use the literal text in the variable or command statement, without any substitution. **All special meanings and functions are removed**. It is also used when you want a variable reread each time it is used; for example

```
[jmedinar@localhost ~]$ echo 'Who lives in a pineapple under the
sea?: $NAME'
Who lives in a pinneapple under the sea?: $NAME
```

Back Quotes

We use **``** **back tics**, in a statement where we want to **execute a command** or script and have its output substituted instead; this is command substitution.

```
[jmedinar@localhost ~]$ echo "Who lives in a pinneapple under the
sea?: `echo $NAME`"
Who lives in a pinneapple under the sea?: Bob Squarepants
```

The above can also be accomplished in the following way

```
[jmedinar@localhost ~]$ echo "Who lives in a pinneapple under the
sea?: $(echo $NAME)"
Who lives in a pinneapple under the sea?: Bob Squarepants
```

Basically what is happening here is that everything under the backtics or the `$()` will be executed in a separated shell first and the resulting output will be appended to our initial `echo` command as a string.

Exit Codes

Whenever we run a command there is a response back from the system about the last command that was executed, known as the `exit code`. If the command was successful the exit code will always be `0 , zero`. If it was not successful the return will be something else. To check the return code we look at the value of the **$? shell variable**. Example:

We check for a file that exist:

```
[jmedinar@localhost ~]$ ls -l /etc/passwd
-rw-r--r--. 1 root root 2662 Nov 18 17:12 /etc/passwd
[jmedinar@localhost ~]$ echo $?
0
```

Now we check for one that doesn't exist and will fail

```
[jmedinar@localhost ~]$ ls -l /non/existing/file
ls: cannot access '/non/existing/file': No such file or directory
[jmedinar@localhost ~]$ echo $?
2
```

Knowing the `exit code` can help you verify that everything you are executing in your script is correct before you continue. Example:

```
#!/bin/bash
ls -l /etc/passwd
if [ $? -eq 0 ]
then
  echo "The script ran ok"
  exit 0
else
  echo "The script failed" >&2
  exit 1
fi
```

Notice how besides checking the exit code of our `ls` command we are also exiting our own script with a proper exit code by using the `exit 0` if success or `exit 1` if failure.

What exit code should I use?

The Linux Documentation Project has a list of reserved codes that also offers advice on what code to use for specific scenarios.

These are the standard error codes in Linux or UNIX.

- `1` - Catchall for general errors
- `2` - Misuse of shell builtins (according to Bash documentation)
- `126` - Command invoked cannot execute.

- `127` - “command not found”.
- `128` - Invalid argument to exit.
- `128+n` - Fatal error signal “n”.
- `130` - Script terminated by Control-C.
- `255\*` - Exit status out of range.

Or use whatever you want! But zero!

Command Line Arguments

The basis for the shell script is the automation of a series of commands. Automate commands that are executed often and reduce the chance of error by eliminating typos and missed definitions, and we can do conditional tests. Besides all the commands we have reviewed and many more that are available there are some commands that are only available for shell scripts.

Symbol Commands

Command	Description
`$( )`	Runs the enclosed command in a sub-shell
`(( ))`	Evaluates and assigns value to a variable and does math in a shell
`$(())`	Evaluates the enclosed expression
`[ ]`	Same as the test command
`< >`	Used for string comparison
`‘command’`	Command substitution same as $()

Command-Line Arguments

The command-line arguments `$1, $2, $3,. . .$9` are positional parameters, with `$0` pointing to the actual command, program, shell script, or function name.

```
#!/bin/sh
echo First argument: $1
echo Third argument: $3
```

We can execute

```
[jmedinar@localhost ~]$ ./pshow one two three
First argument: one
Third argument: three
```

Special Parameters

There are special parameters that allow accessing all the command-line arguments at once. `$*` and `$@` both will act the same unless they are enclosed in double-quotes, " "

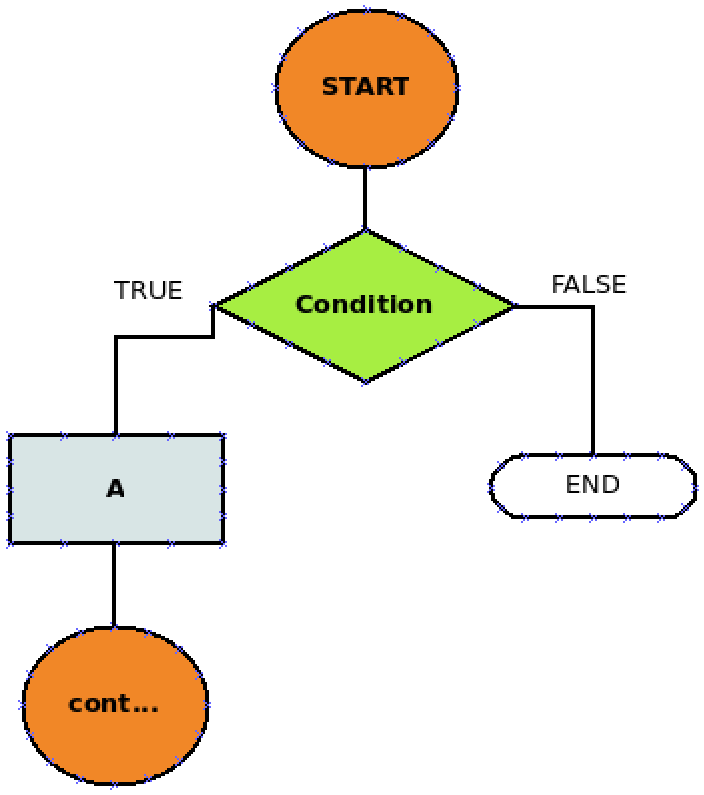
Parameter	**Description**
`$0`	The name of the script.
`$1 - $16`	The arguments passed to the script.
`$#`	How many arguments were passed to the script.
`$@`	All the arguments supplied to the script.
`$?`	The exit status of the most recently run process.
`$$`	The process ID of the current script.
`$USER`	The username of the user running the script.
`$HOSTNAME`	The hostname of the machine the script is running on.
`$SECONDS`	The number of seconds since the script was started.
`$RANDOM`	Returns a different random number each time is it referred to.
`$LINENO`	Returns the current line number in the script.

Control Structures

Most of our programming languages today are able to make decisions based on the conditions we set. A condition is an expression that evaluates to a Boolean value - **true or false.** Any programmer can make his program smart based on the decision and logic he puts into his program.

> *Executing for testing:*
>
> *I have set all the following examples as simple functions that you can actually copy paste into your Linux terminal and execute without having to actually create an script. But feel free to run the example in whatever way you find more effective.*

if statement



A basic if statement effectively says, **if** a particular condition is true, then perform a given set of actions and that code looks like this:

```
if [ condition ]
then
   commands
fi
```

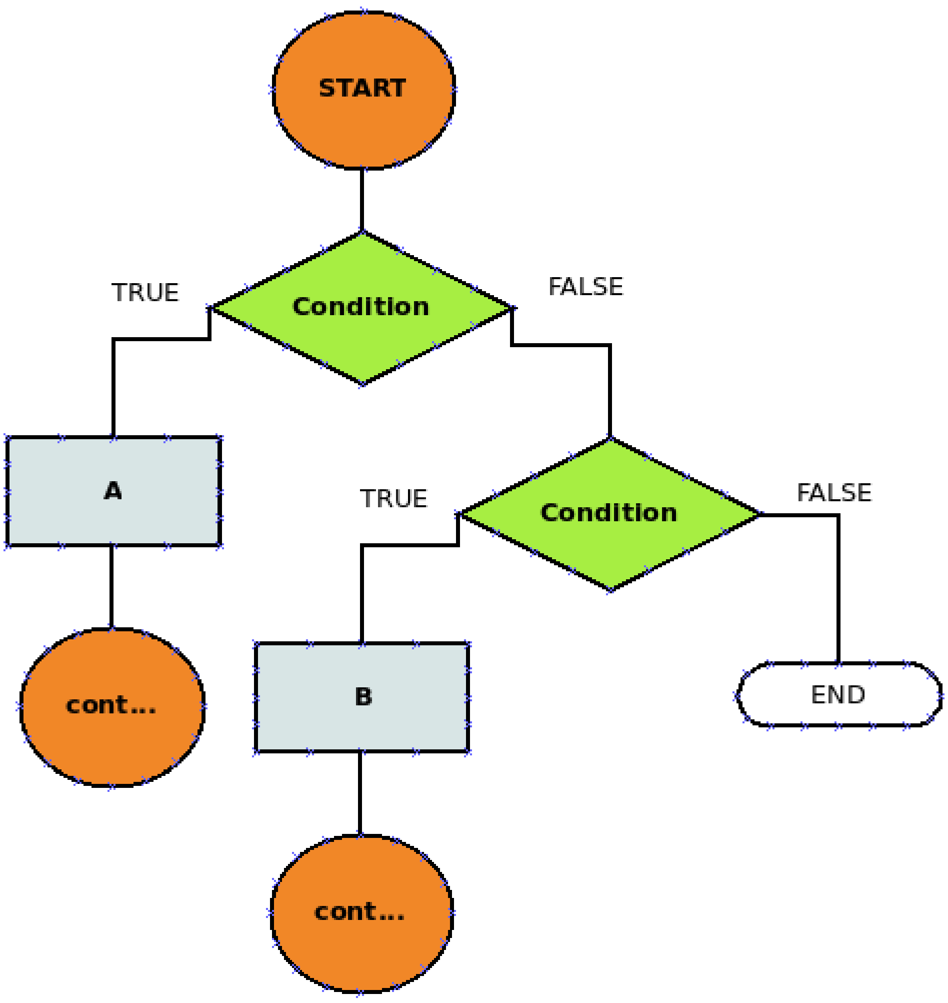
Anything between **then** and **fi** (if backwards) will be executed only if the condition (between the square brackets) is true.

```
function basic_if() {
   if [[ $1 -gt 100 ]]
   then
      echo Hey that\'s a large number.
   fi
   echo "done"
}
```

Testing...

```
[jmedinar@localhost ~]$ basic_if 10
done
[jmedinar@localhost ~]$ basic_if 1000
Hey that's a large number.
done
```

Nested If statements



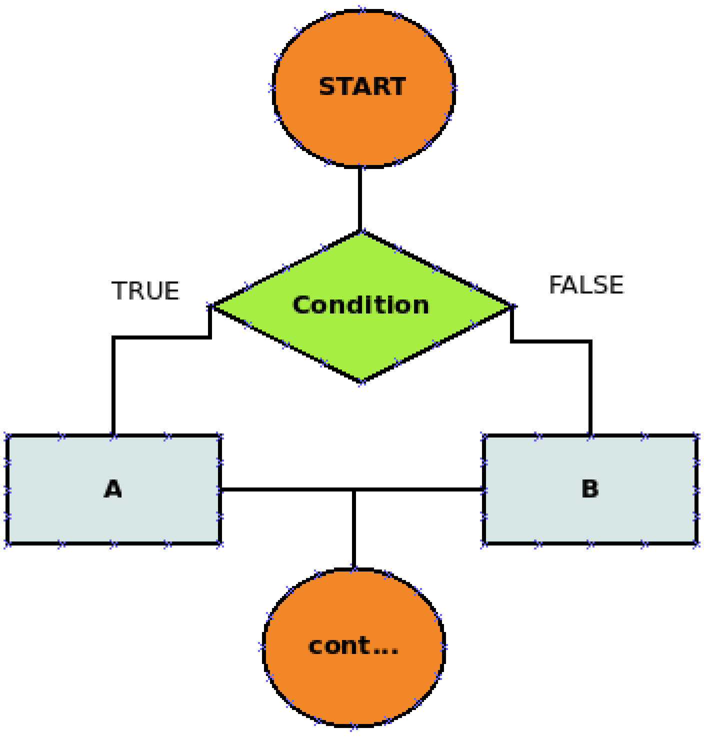
You may have as many **if** statements as necessary inside your script. It is also possible to have an if statement inside of another if statement.

```
function nested_if(){
   if [[ $1 -gt 100 ]]
   then
      echo Hey that\'s a large number.
      if (( $1 % 2 == 0 ))
      then
         echo And is also an even number.
      fi
   fi
}
```

Testing...

```
[jmedinar@localhost ~]$ nested_if 10
[jmedinar@localhost ~]$ nested_if 1000
Hey that's a large number.
And is also an even number.
[jmedinar@localhost ~]$ nested_if 1111
Hey that's a large number.
```

if . . . then . . . else statement



Sometimes we want to perform a certain set of actions if a statement is true, and another set of actions if it is false. We can accommodate this with the **else** mechanism.

```
if [ condition ]
then
   commands
else
   commands
fi
```

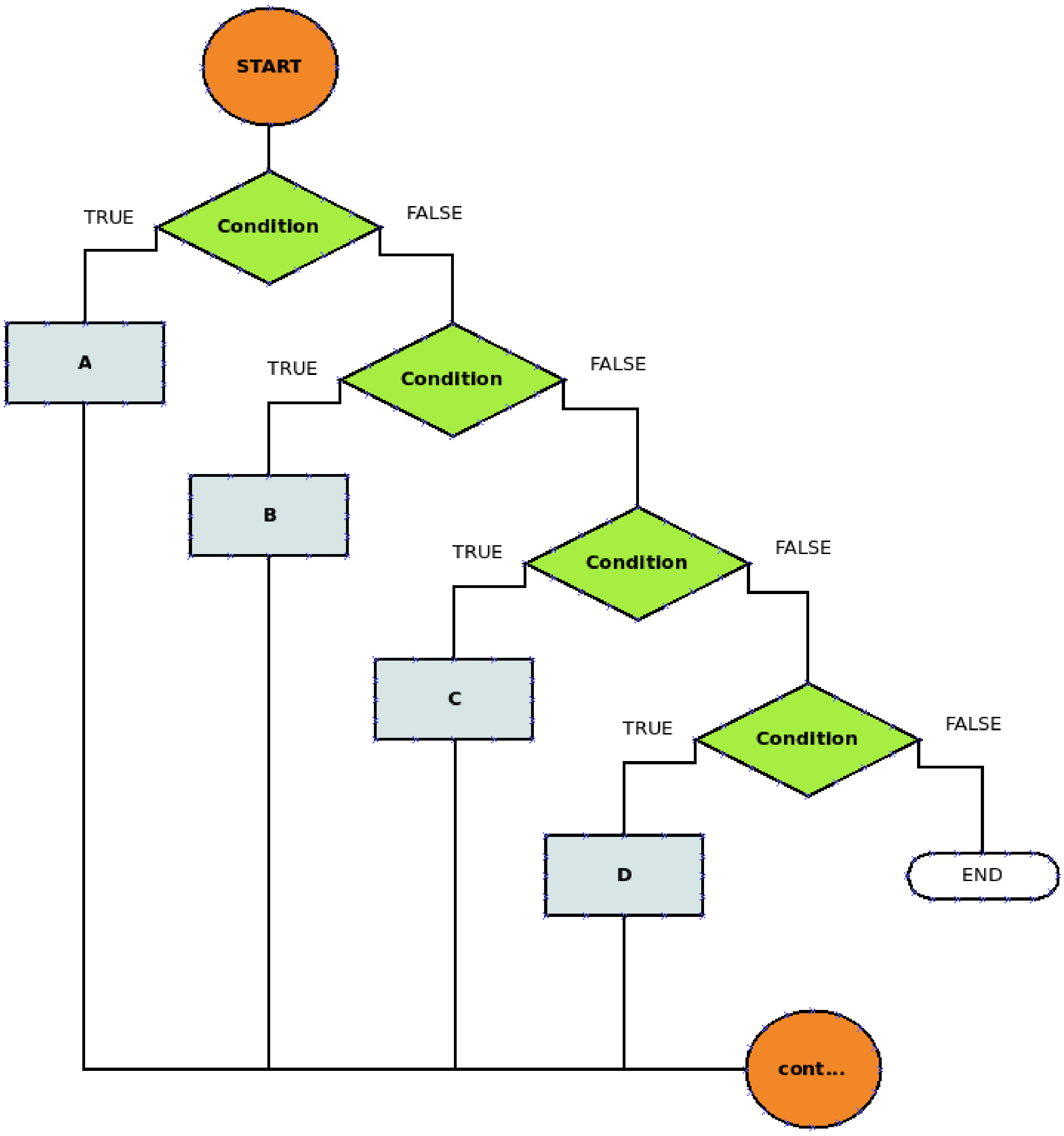
Now we could easily check if an argument was supplied by the user, **else** notify.

```
function if_else(){
   if [[ $# -eq 1 ]]
   then
      echo The argument supplied was: $1
   else
      echo No arguments
   fi
}
```

Testing...

```
[jmedinar@localhost ~]$ if_else
No arguments
[jmedinar@localhost ~]$ if_else 23
The argument supplied was: 23
```

if . . . then . . . elif . . . (else) statement



Sometimes we may have a series of conditions that may lead to different paths. This is also a way of managing multiple IF statements without having to worry about the nesting.

```
if [ condition ]
then
   commands
elif [ condition ]
then
   commands
elif [ condition ]
then
   commands
else
   commands
fi
```

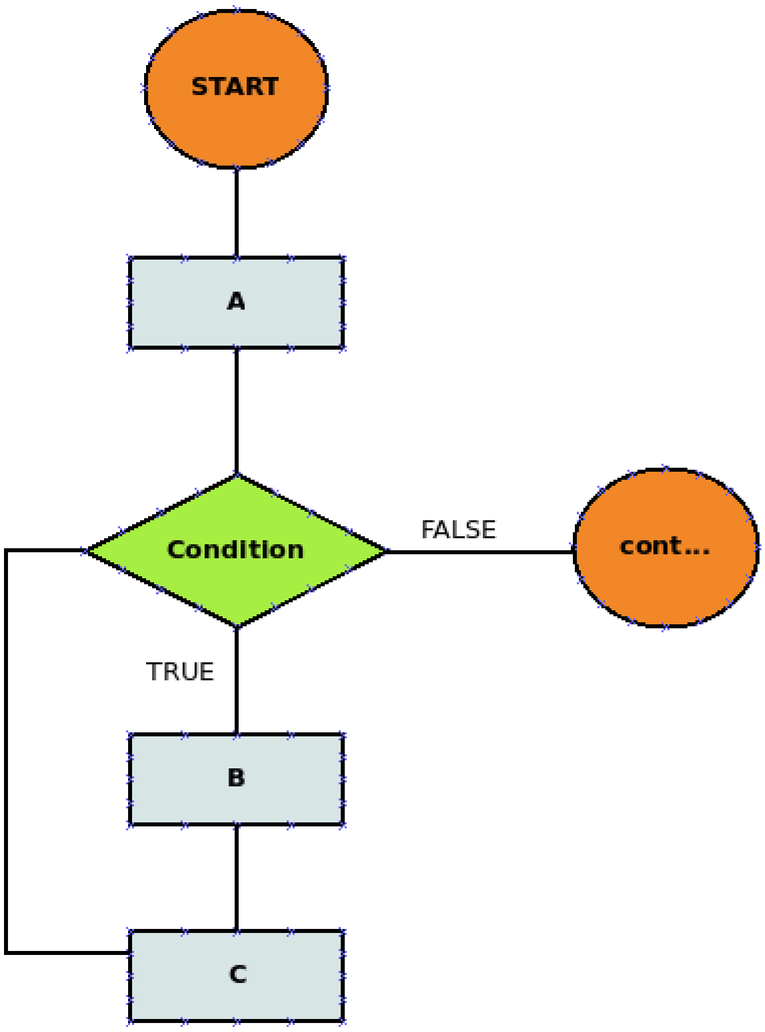
For example, it may be the case that if you are 18 or over you may go to the party. If you aren't but you have a letter from your parents you may go but must be back before midnight. Otherwise, you cannot go.

```
function party(){
   if [[ $1 -ge 18 ]]
   then
      echo You may go to the party.
   elif [[ $2 == 'yes' ]]
   then
      echo You may go to the party but be back before midnight.
   else
      echo You are a minor and will not go to the party.
   fi
}
```

Testing...

```
[jmedinar@localhost ~]$ party 10
You may not go to the party.
[jmedinar@localhost ~]$ party 10 yes
You may go to the party but be back before midnight.
[jmedinar@localhost ~]$ party 20
You may go to the party.
[jmedinar@localhost ~]$ party 20 yes
You may go to the party.
```

for . . . in statement



In the for statement the for loop will iterate once for every item in a pre-defined range or a list sequentialy, and execute the commands between do and done then go back to the top, grab the next item in the list and repeat.

The list is defined as a series of strings, separated by spaces.

```
for loop_variable in range_or_list
do
   commands
done
```

Here is a simple example to illustrate:

```
function simpsons(){
   names='Bart Lisa Maggie'
   for name in ${names}
   do
      echo Hello ${name}
   done
   echo All done
}
```

Testing...

```
[jmedinar@localhost ~]$ simpsons
Hello Bart
Hello Lisa
Hello Maggie
All done
```

We can also process a range of numbers

```
function for_range() {
   for number in {1..5}
   do
      echo ${number}
   done
   echo All done
}
```

Testing...

```
[jmedinar@localhost ~]$ for_range
1
2
3
4
5
All done
```

It is also possible to specify a different value to **increase** or **decrease** each time.

```
function for_range2() {
   for value in {10..0..2}
   do
      echo $value
   done
   echo All done
}
```

Testing...

```
[jmedinar@localhost ~]$ for_range2
10
8
6
4
2
0
All done
```

One of the more useful applications of **for** loops is in the processing of a set of files. To do this we may use wildcards. Let's say we want to change the extension of some files.

Let’s start by creating our test scenario

```
[jmedinar@localhost ~]$ mkdir html_files; cd html_files
[jmedinar@localhost html_files]$ touch file{1..10}.html
[jmedinar@localhost html_files]$ ls
file10.html  file1.html  file2.html  file3.html  file4.html
file5.html  file6.html  file7.html  file8.html  file9.html
```

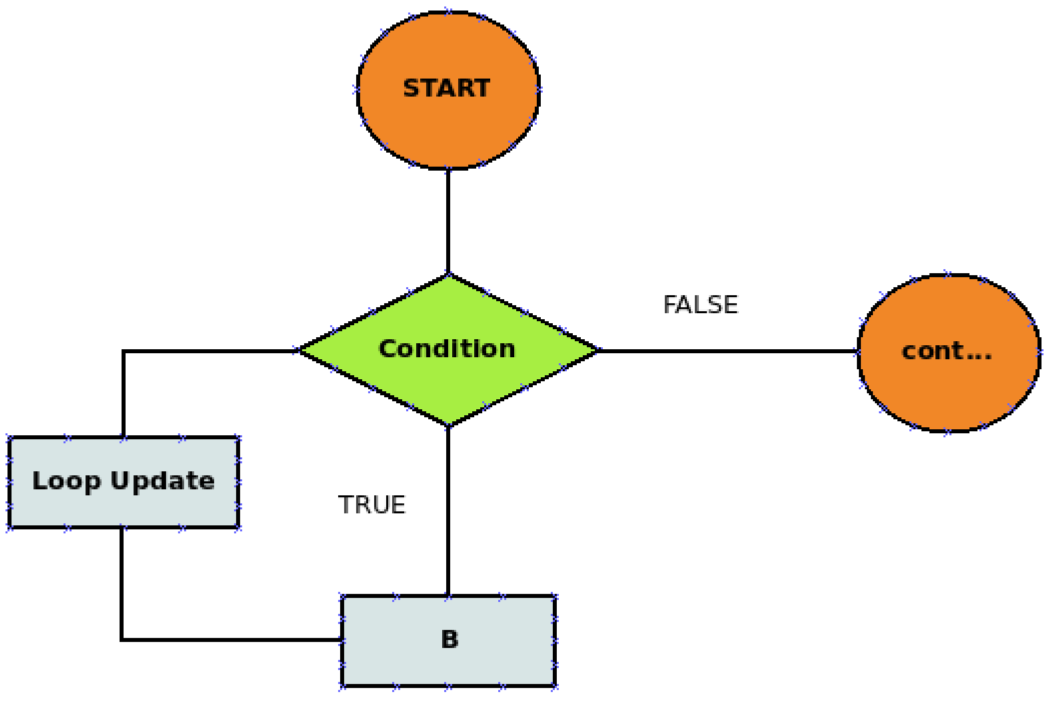
Now let’s do a quick code to change the extension of the files

```
function chext(){
   # This function receives 2 parameters:
   # 1. The extension to change
   # 2. The desired new extension
   for file in $(find . -type f -name "*.${1}" -exec basename {} \;)
   do
      mv $file $(basename -s ".$1" $file).${2}
   done
}
```

Testing...

```
[jmedinar@localhost html_files]$ ls
file10.html  file1.html  file2.html  file3.html  file4.html
file5.html  file6.html  file7.html  file8.html  file9.html
[jmedinar@localhost html_files]$ chext html php
[jmedinar@localhost html_files]$ ls
file10.php  file1.php  file2.php  file3.php  file4.php  file5.php
file6.php  file7.php  file8.php  file9.php
[jmedinar@localhost html_files]$
```

while statement



One of the easiest statements to work with is **while** loops. In very simple terms they just say, “while an expression is true, keep executing these lines of code”. They have the following format:

```
while condition
do
   commands
done
```

In the example below we will print the numbers while the number is lower than 10:

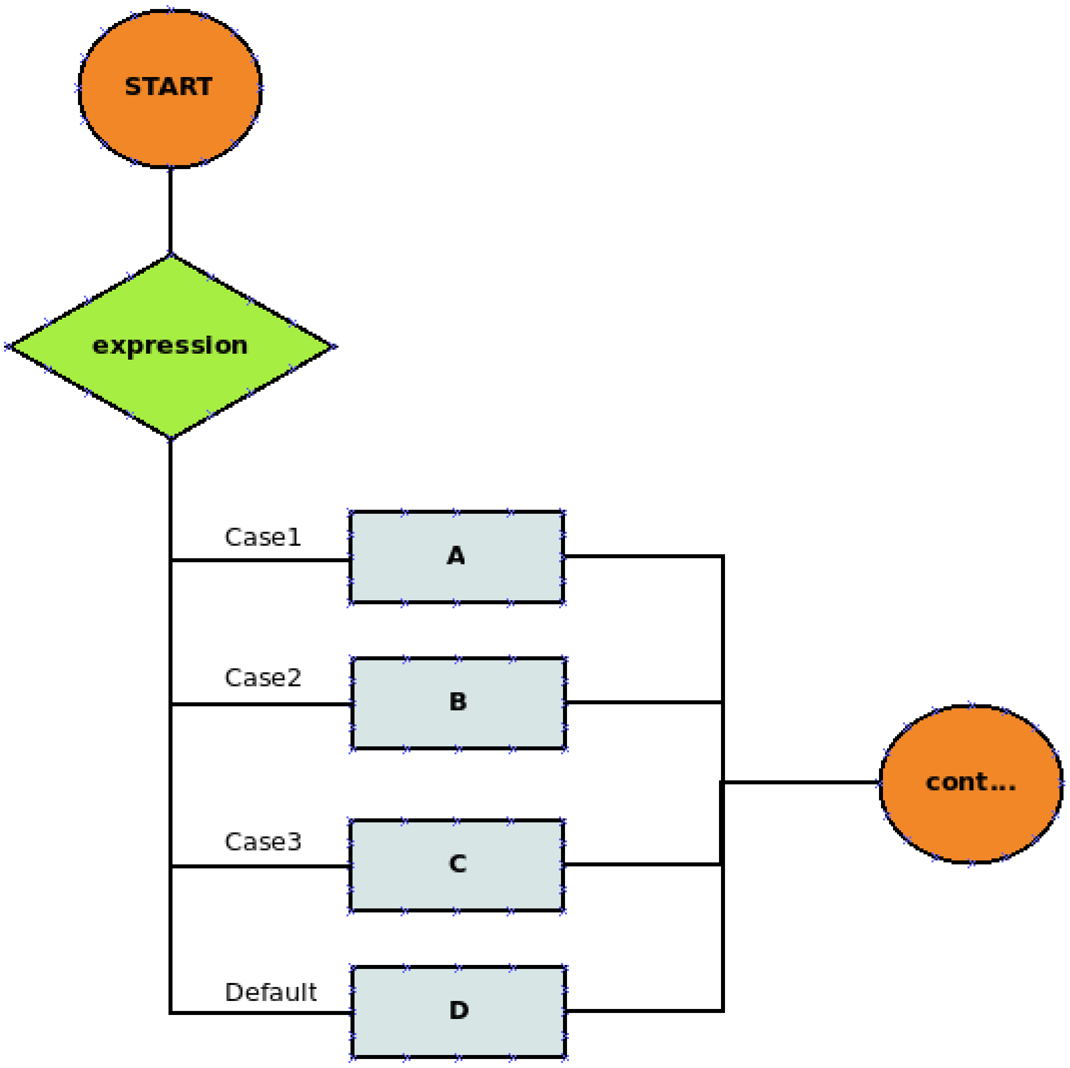
```
function while_counter(){
   counter=1
   while [ $counter -le 10 ]
   do
      echo $counter
      ((counter++))
   done
   echo "All done"
}
```

Testing...

```
[jmedinar@localhost html_files]$ while_counter
1
2
3
4
5
6
7
8
9
10
All done
```

Important difference between a while statement and a for statement is that in the first we will never know how many times the iteration will occur as we do not control the condition while in the for statement the number of iterations are pre-defined.

case statement



Sometimes we may wish to take different paths based upon a variable matching a series of patterns. We could use a series of **if** and **elif** statements but that would soon grow to be hard to mantain. Fortunately, there we can use a **case** statement.

```
case $variable in
   match_1)
      block of code
   ;;
   match_2)
      block of code
   ;;
   match_3)
      block of code
   ;;
   *)
      block of code to execute as default
   ;;
esac
```

It compares the content of a variable with the possible cases or uses the default if there is no match. We can have different actions for each block of code.

Example

```
function menu(){
   echo "Welcome to my program!"
   echo "Select an action to perform:"
   echo "   start: 1"
   echo "   stop: 2"
   echo "   restart: 3"
   echo "   exit: x"
   read -p "Answer: " answer
   case $answer in
      '1') echo starting...
         ;;
      '2') echo stoping...
         ;;
      '3') echo restarting...
         ;;
      'x') echo exiting...
         ;;
      *) echo Invalid option :P
         ;;
   esac
}
```

Testing...

```
[jmedinar@localhost html_files]$ menu
Welcome to my program!
Select an action to perform:
   start: 1
   stop: 2
   restart: 3
   exit: x
Answer: 1
starting...
[jmedinar@localhost html_files]$ menu
Welcome to my program!
Select an action to perform:
   start: 1
   stop: 2
   restart: 3
   exit: x
Answer: x
exiting...
[jmedinar@localhost html_files]$ menu
Welcome to my program!
Select an action to perform:
   start: 1
   stop: 2
   restart: 3
   exit: x
Answer: F
Invalid option :P
```

Operators

File Test Operators

Operator	Description	Example
-b file	Checks if file is a block special file	[-b $file] is false.
-c file	Checks if file is a character special file	[-c $file] is false.
-d file	Checks if file is a directory	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set	[-g $file] is false.
-k file	Checks if file has its sticky bit set	[-k $file] is false.
-p file	Checks if file is a named pipe	[-p $file] is false.

Operator	Description	Example
-t file	Checks if file descriptor is open and associated with a terminal	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set	[-u $file] is false.
-r file	Checks if file is readable	[-r $file] is true.
-w file	Checks if file is writable	[-w $file] is true.
-x file	Checks if file is executable	[-x $file] is true.
-s file	Checks if file has size greater than 0	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

String Operators

Operator	Description	Example
=	Checks if the value of two operands are equal or not	[a =b] is not true.
!=	Checks if the value of two operands are equal or not	[$a!$ =b] is true.
-z	Checks if the given string operand size is zero	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero	[-n $a] is not false.
str	Checks if **str** is not the empty string	[$a] is not false.

Boolean Operators

Operator	Description	Example
!	This is logical negation	[! false] is true.
-o	This is logical **OR**.	[$a - lt20 - o$b -gt 100] is true.

Operator	Description	Example
-a	This is logical **AND**.	[$a - lt20 - a$b -gt 100] is false.

Relational Operators

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not	[$a - eq$b] is not true.
-ne	Checks if the value of two operands are equal or not	[$a - ne$b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand	[$a - gt$b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand	[$a - lt$b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand	[$a - ge$b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand	[$a - le$b] is true.

Arithmetic Operators

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a

Operator	Description	Example
== (Equality)	Compares two numbers, if both are same then returns true.	[a ==b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a!$ =b] would return true.

Functions

We have been using functions already directly from the command line only for testing purposes. Now let's understand them better.

A function is basically a tiny little script that performs a single action or a set of related actions that we need to repeat within our script multiple times.

So instead of having some code to do something and having to repeat it over an over.

```
block of code...
print message in a special format for message AAA
block of code...
print message in a special format for message BBB
block of code...
print message in a special format for message CCC
block of code...
print message in a special format for message DDD
```

we better create a function to accomplish this repetitive task.

```
function print_message(){
   print message in special format for $message
}

block of code...
print_message AAA
block of code...
print_message BBB
block of code...
print_message CCC
block of code...
print_message DDD
```

A function is written in much the same way as a shell script but it is defined and referenced within the script. This way we can write a piece of code just once, and use it over and over.

> *NOTE: You can also define functions at the system level that are always available in your environment by adding them to your* `~./bashrc` *file.*

A function has the following form:

```
function_name() {
   actions
}
```

The name of your function is **function_name**, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

When we write functions into our scripts we must remember to declare, or write, the function before we use it. The function must appear above the command statement calling the function. We can’t use something that doesn’t exist right?.

```
#!/bin/bash
# Define your function here
function hello(){
   echo "Hello Jack, how are you?"
   echo "Hello Rose, how are you?"
   echo "Hello John, how are you?"
}

# Invoke your function
hello
```

Testing...

```
[jmedinar@localhost html_files]$ ./hello.sh
Hello Jack, how are you?
Hello Rose, how are you?
Hello John, how are you?
```

Passing Parameters to a Function

Passing parameters to our function will allow us to have more freedom while using the code. These parameters would be represented by **$1**, **$2** and so on.

```
#!/bin/bash
# Define your function here
function hello(){
   echo "Hello $1, how are you?"
}

# Invoke your function
hello Jack
hello Rose
hello John
```

No we can run the same code but change the names at will. Testing...

```
[jmedinar@localhost html_files]$ ./hello.sh
Hello Jack, how are you?
Hello Rose, how are you?
Hello Jhon, how are you?
```

Better right?

We can do it even better by allowing the parameter passing directly from the user.

```
#!/bin/bash
# Define your function here
function hello(){
   echo "Hello $1, how are you?"
}

# Invoke your function
hello $1
```

Now we can call it and send parameters from the command line.

```
[jmedinar@localhost html_files]$ ./hello.sh Juan
Hello Juan, how are you?
[jmedinar@localhost html_files]$ ./hello.sh Brad
Hello Brad, how are you?
[jmedinar@localhost html_files]$ ./hello.sh Nadia
Hello Nadia, how are you?
```

Returning Values from Functions

Now we know how to put something in our functions but how can we get something from it?

If you execute an **exit** command from within a function, its effect is not only to terminate the execution of the function but also of the shell program that called the function, that is effectively killing your whole script!.

If you instead want to just terminate the execution of the function and continue with the script, then you must **return**. This basically means you need to send a `return code` to indicate the state in which your function finished!

You **return code** can be anything you want, but obviously you should choose something that is meaningful or useful in the context of your script as a whole. Following function returns the value 10

```
#!/bin/bash
function Hello() {
   echo "Hello World $1 $2"
   return 10
}
Hello Linux Student
ret=$?
echo "Return value is $ret"
```

Testing...

```
[jmedinar@localhost html_files]$ ./hello.sh
Hello World Linux Student
Return value is 10
```

Nested Functions

One of the more interesting features of functions is that they can call other functions. A function that calls itself is known as a **recursive function**. Following example demonstrates nesting of two functions

```
#!/bin/bash
function one(){
   echo "This is the first function speaking..."
   two
}
function two(){
   echo "This is now the second function speaking..."
}
one
```

Testing...

```
[jmedinar@localhost html_files]$ ./test.sh
This is the first function speaking...
This is now the second function speaking...
```

So if you think on the possibilities, now you can create re-usable code to solve problems and break your problems into little sections that solve little things efficiently Just like in the **Linux philosophy**.

Scheduling

You will write a lot of shell scripts that we want to execute on a timed interval or run once at a specific time.

Crontab

A cron table is a system file in which we can create entries of commands we want to have executed according to a schedule with an specific format.

```
|--------------min (0-59)
| |----------------hour (0-23)
| | |-----------------day of the month (1-31)
| | | |-------------------month (1-12)
| | | | |--------------------day of the week (0-6)(Sun-Sat)
| | | | |
| | | | |
| | | | |
* * * * * The command to execute
```

So an entry in crontab will look something like this:

```
0 0 1 1 * /usr/bin/banner "Happy New Year" > /dev/console
```

This file is read every minute. By default, any user can create a cron table with the **crontab -e command,** but the only root can control which users are allowed to create and edit cron tables with the cron.allow and cron.deny files.

Need more?

🎥 Linux/Mac Tutorial: Cron Jobs - How to Schedule Commands with crontab

at Command

Like the cron table, the `at` command executes commands based on time but this is used for one time executions. When the job is executed an email will be send with the content of the standard output and standard error to the user who scheduled the job to run, unless the output is redirected. Only root can control which users are allowed to schedule jobs with the `at.allow` and `at.deny` files.

To execute a command in 10 minutes, use the following syntax:

```
[jmedinar@localhost html_files]$ at now + 10 minutes
warning: commands will be executed using /bin/sh
at> ./hello.sh
at> <EOT>
job 29 at Fri Dec 25 17:38:00 2020
```