Functions

Creating a function is fairly easy. They may be written in two different formats:

**function\_name () {  
<commands>  
}**

or

**function function\_name {  
<commands>  
}**

A few points to note:

* Either of the above methods of specifying a function is valid. Both operate the same and there is no advantage or disadvantage to one over the other. It's really just personal preference.
* In other programming languages it is common to have arguments passed to the function listed inside the brackets (). In Bash they are there only for decoration and you never put anything inside them.
* The function definition ( the actual function itself) must appear in the script before any calls to the function.

Let's look at a simple example:

1. #!/bin/bash
2. *# Basic function*
3. print\_something () {
4. echo Hello I am a function
5. }
6. print\_something
7. print\_something

### **Passing Arguments**

It is often the case that we would like the function to process some data for us. We may send data to the function in a similar way to passing command line arguments to a script. We supply the arguments directly after the function name. Within the function they are accessible as **$1, $2, etc**.

1. #!/bin/bash
2. *# Passing arguments to a function*
3. print\_something () {
4. echo Hello $1
5. }
6. print\_something Mars
7. print\_something Jupiter

### **Return Values**

Most other programming languages have the concept of a return value for functions, a means for the function to send data back to the original calling location. Bash functions don't allow us to do this. They do however allow us to set a return status. Similar to how a program or command exits with an exit status which indicates whether it succeeded or not. We use the keyword **return** to indicate a return status.

1. #!/bin/bash
2. *# Setting a return status for a function*
3. print\_something () {
4. echo Hello $1
5. return 5
6. }
7. print\_something Mars
8. print\_something Jupiter
9. echo The previous function has a return value of $?

One way to get around this is to use [Command Substitution](https://ryanstutorials.net/bash-scripting-tutorial/bash-variables.php#commandsubstitution) and have the function print the result (and only the result).

return\_hack.sh

1. #!/bin/bash
2. *# Setting a return value to a function*
3. lines\_in\_file () {
4. cat $1 | wc -l
5. }
6. num\_lines=$( lines\_in\_file $1 )
7. echo The file $1 has $num\_lines lines in it

Variable Scope

Scope refers to which parts of a script can see which variables. By default a variable is **global**. This means that it is visible everywhere in the script. We may also create a variable as a **local** variable. When we create a local variable within a function, it is only visible within that function. To do that we use the keyword **local** in front of the variable the first time we set it's value.

**local var\_name=<var\_value>**

It is generally considered good practice to use local variables within functions so as to keep everything within the function contained. This way variables are safer from being inadvertently modified by another part of the script which happens to have a variable with the same name (or vice versa).

local\_variables.sh

1. #!/bin/bash
2. *# Experimenting with variable scope*
3. var\_change () {
4. local var1='local 1'
5. echo Inside function: var1 is $var1 : var2 is $var2
6. var1='changed again'
7. var2='2 changed again'
8. }
9. var1='global 1'
10. var2='global 2'
11. echo Before function call: var1 is $var1 : var2 is $var2
12. var\_change
13. echo After function call: var1 is $var1 : var2 is $var2