# Function

|  |
| --- |
| Functions are very nifty inside Bash scripts. They **are blocks of commands**, much like normal scripts you might write, except **they don’t reside in separate files**, **and they don’t cause a separate process to be executed**.  However, they take arguments just like scripts -- and unlike scripts, **they can affect variables inside your script, if you want them to** |

## Simple Example

|  |
| --- |
| Example without writing in bash:  # sum() {  > echo "$1 + $2 = $(($1 + $2))"  > }  # sum 2 4  2 + 4 = 6  Example with writing in bash:  # vim F\_sum.sh  #!/bin/bash  sum (){  echo "$1 + $2 = $(($1 + $2))"  }  sum "$1" "$2"  [root@TESTBED-VOD-CMS test]# bash -x F\_sum.sh 1 2  + sum 1 2  + echo '1 + 2 = 3'  1 + 2 = 3  **A note on scope**: if you choose to embed functions within script files, as many will find more convenient, then you need to understand that the parameters you pass to the script are not necessarily the parameters that are passed to the function.  As you can see, *we passed the script’s two parameters to the function within*, but we could have passed anything we wanted (though, doing so in this situation would only confuse users trying to use the script) |

# Pass arguments into a function

|  |
| --- |
| * Shell functions have their own command line argument. * Use variable $1, $2..$n to access argument passed to the function.: **position variable** * The syntax is as follows:   name(){  arg1=$1  arg2=$2  command on $arg1  }   * To invoke the the function use the following syntax:   name foo bar  Where,   1. **name** = function name. 2. **foo** = Argument # 1 passed to the function (**positional parameter # 1**). 3. **bar** = Argument # 2 passed to the function. |

# Function shell variables

|  |
| --- |
| * All function parameters or arguments can be accessed via $1, $2, $3,..., $N. * [$0](https://bash.cyberciti.biz/guide/$0) always point to the shell script name. * [$\*](https://bash.cyberciti.biz/guide/$*) or [$@](https://bash.cyberciti.biz/guide/$@) holds all parameters or arguments passed to the function. * [$#](https://bash.cyberciti.biz/guide/$) holds the number of positional parameters passed to the function.  How Do I Display Function Name? [$0](https://bash.cyberciti.biz/guide/$0) always point to the shell script name. However, you can use an array variable called [FUNCNAME](https://bash.cyberciti.biz/wiki/index.php?title=FUNCNAME&action=edit&redlink=1) which contains the names of all shell functions currently in the execution call stack. The element with index 0 is the name any currently-executing shell function.This variable *exists only* when a shell function is executing. |

# Function Purpose

|  |
| --- |
| Function Purpose:  Functions serve a few purposes in a script.   * The first is to isolate a block of code that performs a specific task, so that it doesn’t clutter up other code. This helps you make things more readable, when done in moderation. (Having to jump all over a script to track down 7 functions to figure out what a single command does has the opposite effect, so make sure you do things that make sense.) * The second is to allow a block of code to be reused with slightly different arguments. |

## Example

|  |
| --- |
| # vim F\_play.sh  #!/bin/bash  open() {  case "$1" in  \*.mp3|\*.ogg|\*.wav|\*.flac|\*.wma) xmms "$1";;  \*.jpg|\*.gif|\*.png|\*.bmp) display "$1";;  \*.avi|\*.mpg|\*.mp4|\*.wmv) mplayer "$1";;  esac  }  for file; do  open "$file"  done  Here, we define a *function* named open.   * This function is a block of code that takes a single argument, and based on the *pattern* of that argument, it will either run xmms, display or mplayer with that argument. * Then, a for loop iterates over all of the *script’s* positional parameters. (Remember, **for file is equivalent to for file in "$@"** and both of them iterate over the full set of positional parameters.) *The for loop calls the open function for each parameter*.   \* The function’s parameters are different from the script’s parameters. |

# The function’s parameters are different from the script’s parameters.

# Function local variable usage

|  |
| --- |
| Functions may also have *local variables*, declared with the *local* or *declare* **keywords**.   * *This lets you do work without potentially overwriting important variables from the caller’s namespace*. |

## Example

|  |  |  |  |
| --- | --- | --- | --- |
| # vim F\_count.sh  #!/bin/bash  count(){  local i  for ((i=1; i<=$1; i++)); do echo $i; done  echo 'Ah, ah, ah!'  }  for ((i=1; i<=3; i++)); do count $i; done  [root@TESTBED-VOD-CMS test]# bash -x F\_count.sh   |  | | --- | | + (( i=1 ))  + (( i<=3 ))  + count 1  + local i  + (( i=1 ))  + (( i<=1 ))  + echo 1  1  + (( i++ ))  + (( i<=1 ))  + echo 'Ah, ah, ah!'  Ah, ah, ah! |  |  | | --- | | + (( i++ ))  + (( i<=3 ))  + count 2  + local i  + (( i=1 ))  + (( i<=2 ))  + echo 1  1  + (( i++ ))  + (( i<=2 ))  + echo 2  2  + (( i++ ))  + (( i<=2 ))  + echo 'Ah, ah, ah!'  Ah, ah, ah! |  |  | | --- | | + (( i++ ))  + (( i<=3 ))  + count 3  + local i  + (( i=1 ))  + (( i<=3 ))  + echo 1  1  + (( i++ ))  + (( i<=3 ))  + echo 2  2  + (( i++ ))  + (( i<=3 ))  + echo 3  3  + (( i++ ))  + (( i<=3 ))  + echo 'Ah, ah, ah!'  Ah, ah, ah!  + (( i++ ))  + (( i<=3 )) | |

|  |
| --- |
| The local variable i inside the function is stored differently from the variable i in the outer script. This allows the two loops to operate without interfering with each other’s counters. Functions may also call themselves *recursively*, but we won’t show that today. *Maybe later!* |

# Function Template: Menu driven scripts

|  |
| --- |
| #Variable  #Function handle  One()  Two()  Show\_menu()  Read\_option()  #Scripts  1.Show\_menu()  2.Read\_option()  3.Infinite Loop: 1+2 |