**What are functions in shell script**

**A function is a block of code that can be called multiple times from different parts of the script. It's a way to reuse code, make the script more modular, and improve readability.**

**syntax:**

**function function\_name () {**

**# code to be executed**

**}**

**or**

**function\_name() {**

**# code to be executed**

**}**

**To call function :**

**function\_name**

**Example 1: Simple Function**

**#! /bin/bash**

**greet () {**

**echo "Hello, $1!"**

**}**

**greet "John" # Output: Hello, John!**

**greet "Jane" # Output: Hello, Jane!**

**Example 2: Function with Parameters**

**#! /bin/bash**

**add () {**

**echo "$1 + $2 = $(( $1 + $2 ))"**

**}**

**add 2 3 # Output: 2 + 3 = 5**

**Example 3: Function with Return Value**

**#! /bin/bash**

**get\_square () {**

**local num=$1**

**echo $(( num \* num ))**

**}**

**result=$(get\_square 4)**

**echo "Square of 4 is: $result" # Output: Square of 4 is: 16**

**Example 4**

**#! /bin/bash**

**add () {**

**local num1=$1**

**local num2=$2**

**local result=$(( num1 + num2 ))**

**return $result**

**}**

**add 3 5**

**# Output: 8**

**Key Points:**

**- Functions can take arguments, just like scripts.**

**- Use local to declare variables inside a function.**

**- Functions can return values using echo or return.**

**- Call functions by simply typing their name, followed by arguments (if any).**

**Using functions in shell scripts makes the code more:**

**- Modular**

**- Reusable**

**- Readable**

**- Maintainable**

**In shell scripting, local is used to declare variables within a function. Here's why:**

**1. Scope: Without local, variables declared inside a function are global, meaning they can be accessed and modified outside the function. By using local, you restrict the variable's scope to within the function, making it private.**

**2. Avoid naming conflicts: If you have a global variable with the same name as a variable inside a function, using local ensures that the function's variable doesn't overwrite the global one.**

**3. Improve code readability: By declaring variables as local, you clearly indicate that they are only used within the function, making the code easier to understand.**

**4. Prevent unexpected side effects: Without local, a function can inadvertently modify global variables, leading to unexpected behavior. Using local helps prevent such side effects.**

**Shifting Arguments**

**In shell scripting, shift is a built-in command that:**

**1. Shifts the positional parameters (arguments) passed to a script or function.**

**2. Removes the first parameter ($1) and shifts the remaining parameters one position to the left.**

**Example:**

**#! /bin/bash**

**# Initial parameters: $1=apple $2=banana $3=cherry**

**echo "$1" # Output: apple**

**shift**

**echo "$1" # Output: banana**

**shift**

**echo "$1" # Output: cherry**

**After each shift, the first parameter is removed, and the remaining parameters are shifted one position to the left.**

**Common use cases for shift:**

**1. Processing arguments: Iterate through arguments passed to a script or function.**

**2. Removing options: Remove options or flags from the argument list.**

**3. Parsing input: Parse input data, like command-line arguments or file contents.**

**Example:**

**#! /bin/bash**

**set – $(date)**

**echo “count: $#”**

**echo “$1 $2 $3 $4 $5”**

**shift**

**echo “$1 $2 $3 $4 $5”**

**shift**

**echo “$1 $2 $3 $4 $5”**

**shift**

**echo “$1 $2 $3 $4 $5”**

**Example:**

**1. Parsing CSV data:**

**#!/bin/bash**

**while read -r line; do**

**set -- $line**

**shift 2 # Skip first two columns**

**echo "Remaining columns: $@"**

**done < data.csv**

**Other useful concept**

**Break Statement**

**The break statement will exit out of the loop and control is passed to the next statement in the loop.**

**Example:**

**for i in {1..5}; do**

**if [ $i -eq 3 ]; then**

**break**

**fi**

**echo "Iteration $i"**

**done**

**Output:**

**Iteration 1**

**Iteration 2**

**#!/bin/bash**

**for val in {1..20..2}**

**do**

**If [[ $val -eq 9 ]]**

**then**

**break**

**else**

**echo "printing ${val}"**

**fi**

**done**

**The conditional statement will evaluate the expression and when it is true($val = 9) then it will run the break statement and the loop will be terminated skipping the remaining iterations.**

**Continue Statement**

**The continue statement skips the current iteration of a loop and moves on to the next one.**

**What if you don’t want to completely exit out of the loop but skip the block of code when a certain condition is met? This can be done with a continue statement. The continue statement will skip the execution of the code block when a certain condition is met and the control is passed back to the** [**loop statement**](https://www.tecmint.com/bash-until-loop/) **for the next iteration.**

**Example:**

**bash**

**for i in {1..5}; do**

**if [ $i -eq 3 ]; then**

**continue**

**fi**

**echo "Iteration $i"**

**done**

**Output:**

**Iteration 1**

**Iteration 2**

**Iteration 4**

**Iteration 5**

**#!/bin/bash**

**for val in {1..20..2}**

**do**

**If [[ $val -eq 9 ]]**

**then**

**continue**

**fi**

**echo "printing ${val}"**

**done**

**when Val is evaluated to nine then the continue statement will skip all the remaining blocks of code and pass the control to for loop for the next iteration.**

**Exit Statement**

**The exit statement terminates the script entirely.**

**Each shell command returns an exit code when it terminates, either successfully or unsuccessfully.**

**By convention, an exit code of zero indicates that the command completed successfully, and non-zero means that an error was encountered.**

**#!/bin/bash**

**echo 'Starting the script'**

**exit 0**

**echo 'This will not be printed'**

**# Output:**

**# 'Starting the script'**

**In this script, we first print ‘Starting the script’, then we execute the exit 0 command. This command terminates the script, so ‘This will not be printed’ is never executed.**

**The exit 0 command signals successful execution of the script. The number 0 is used to denote successful execution, while other numbers (1 to 255) are typically used to indicate various types of errors.**

**#!/bin/bash**

**if [ -f /etc/passwd ]; then**

**echo 'File exists'**

**exit 0**

**else**

**echo 'File does not exist'**

**exit 1**

**fi**

**# Output (if file exists):**

**# 'File exists'**

**# Output (if file does not exist):**

**# 'File does not exist'**

**In this script, we’re checking if the file ‘/etc/passwd’ exists. If it does, we print ‘File exists’ and exit with a status of 0, signaling success. If the file doesn’t exist, we print ‘File does not exist’ and exit with a status of 1, signaling an error.**

**Sleep command**

**The sleep command in shell scripting is used to pause the execution of a script for a specified amount of time.**

**Syntax:**

**sleep [number] [suffix]**

**Arguments:**

**- [number]: The amount of time to sleep.**

**- [suffix]: Optional, specifies the unit of time (s, m, h, d).**

**Examples:**

**- sleep 5: Sleep for 5 seconds.**

**- sleep 1m: Sleep for 1 minute.**

**- sleep 2h: Sleep for 2 hours.**

**- sleep 3d: Sleep for 3 days.**

**Use cases:**

**- Delaying execution of a command or script.**

**- Creating a pause between tasks.**

**- Simulating a delay in a test or demo environment.**

**- Allowing time for a process to complete before proceeding.**

**basename command**

**basename is a command in shell scripting that removes the directory path from a file name, leaving only the file name.**

**Syntax:**

**basename [path]**

**Examples:**

**- basename /path/to/file.txt outputs file.txt**

**- basename ~/Documents/report.pdf outputs report.pdf**

**Use cases:**

**- Extracting the file name from a full path.**

**- Removing directory paths to get a clean file name.**

**- Preparing file names for further processing.**

**Example script:**

**#!/bin/bash**

**file\_path="/path/to/file.txt"**

**file\_name=$(basename "$file\_path")**

**echo "File name: $file\_name"**

**This script outputs "File name: file.txt".**

**dirname command**

**dirname is a command in shell scripting that removes the file name from a path, leaving only the directory path.**

**Syntax:**

**dirname [path]**

**Examples:**

**- dirname /path/to/file.txt outputs /path/to**

**- dirname ~/Documents/report.pdf outputs ~/Documents**

**Use cases:**

**- Extracting the directory path from a full file path.**

**- Removing file names to get a clean directory path.**

**- Preparing directory paths for further processing.**

**Example script:**

**#!/bin/bash**

**file\_path="/path/to/file.txt"**

**dir\_path=$(dirname "$file\_path")**

**echo "Directory path: $dir\_path"**

**This script outputs "Directory path: /path/to".**

**/dev/null File**

**In case you don’t want to print the output of command on terminal or write in a file. we can redirect the output to /dev/null**

* **/dev/null is a special file in Unix-like systems that discards all data written to it.**
* **It's commonly referred to as a "black hole" because any output sent to it is permanently deleted.**

**Usage:**

* **Suppress unnecessary or unwanted output in shell scripts.**

**Example 1: Suppressing Standard Output**

**#!/bin/bash**

**# Redirect standard output to /dev/null**

**ls /path/to/directory > /dev/null**

* **In this example, the output of ls is discarded, so nothing is displayed on the terminal.**

**Example 2: Suppressing Standard Error**

**#!/bin/bash**

**# Redirect standard error to /dev/null**

**ls /nonexistent/path 2> /dev/null**

* **Here, only the error messages from the ls command are suppressed.**

**Example 3: Suppressing Both Standard Output and Standard Error**

**#!/bin/bash**

**# Redirect both stdout and stderr to /dev/null**

**ls /path/to/directory &> /dev/null**

* **Both the regular output and any error messages are discarded.**

#### **2. What is 2>&1?**

* **2>&1 is used to redirect the standard error (stderr, file descriptor 2) to the same location as standard output (stdout, file descriptor 1).**
* **This is useful when you want both stdout and stderr to be combined and redirected together.**

**Usage:**

* **Combine stderr with stdout, then redirect them both to a file or another command.**

**Example 1: Redirecting Both stdout and stderr to a File**

**#!/bin/bash**

**# Redirect stdout and stderr to a file**

**ls /path/to/directory > output.log 2>&1**

* **> output.log: Redirects stdout to output.log.**
* **2>&1: Redirects stderr to the same location as stdout, so both outputs go to output.log.**

**Example 2: Redirecting stdout and stderr to Another Command**

**#!/bin/bash**

**# Pipe both stdout and stderr to another command**

**ls /path/to/directory 2>&1 | grep "something"**

* **This pipes both stdout and stderr to grep for further processing.**

#### **3. What is &>?**

* **&> is a shorthand for redirecting both stdout and stderr to a specified file or location.**
* **It’s more concise than using 2>&1 and is often preferred in modern scripts.**

**Usage:**

* **Redirect both stdout and stderr simultaneously in a simplified manner.**

**Example 1: Redirecting Both stdout and stderr to a File**

**#!/bin/bash**

**# Redirect stdout and stderr to a file using &>**

**ls /path/to/directory &> output.log**

* **This example achieves the same result as the 2>&1 example but uses the more concise &> syntax.**

**Example 2: Redirecting Both stdout and stderr to /dev/null**

**#!/bin/bash**

**# Suppress both stdout and stderr using &>**

**ls /nonexistent/path &> /dev/null**

* **Here, both stdout and stderr are redirected to /dev/null, suppressing all output.**

### **Summary: When to Use Each**

* **/dev/null: Use when you want to discard output (e.g., suppress stdout or stderr).**
* **2>&1: Use when you want to combine stderr with stdout and then redirect both together (e.g., logging both to the same file).**
* **&>: Use when you want a concise way to redirect both stdout and stderr to the same location.**