Swift - Functions

A function is a set of statements organized together to perform a specific task. A Swift 4 function can be as simple as a simple C function to as complex as an Objective C language function. It allows us to pass local and global parameter values inside the function calls.

- **Function Declaration** − tells the compiler about a function's name, return type, and parameters.
- **Function Definition** − It provides the actual body of the function.

Swift 4 functions contain parameter type and its return types.

Function Definition

In Swift 4, a function is defined by the "func" keyword. When a function is newly defined, it may take one or several values as input 'parameters' to the function and it will process the functions in the main body and pass back the values to the functions as output 'return types'.

Every function has a function name, which describes the task that the function performs. To use a function, you "call" that function with its name and pass input values (known as arguments) that match the types of the function's parameters. Function parameters are also called as 'tuples'.

A function's arguments must always be provided in the same order as the function's parameter list and the return values are followed by →.

Syntax

```
func funcname(Parameters) -> returntype {
   Statement1
   Statement2
   ---
   Statement N
   return parameters
}
```

Take a look at the following code. The student's name is declared as string datatype declared inside the function 'student' and when the function is called, it will return student's name.

Live Demo

```
func student(name: String) -> String {
   return name
}
```

```
print(student(name: "First Program"))
print(student(name: "About Functions"))
```

When we run the above program using playground, we get the following result −

```
First Program
About Functions
```

Calling a Function

Let us suppose we defined a function called 'display' to Consider for example to display the numbers a function with function name 'display' is initialized first with argument 'no1' which holds integer data type. Then the argument 'no1' is assigned to argument 'a' which hereafter will point to the same data type integer. Now the argument 'a' is returned to the function. Here display() function will hold the integer value and return the integer values when each and every time the function is invoked.

Live Demo

```
func display(no1: Int) -> Int {
   let a = no1
   return a
}

print(display(no1: 100))
print(display(no1: 200))
```

When we run above program using playground, we get the following result −

```
100
200
```

Parameters and Return Values

Swift 4 provides flexible function parameters and its return values from simple to complex values. Similar to that of C and Objective C, functions in Swift 4 may also take several forms.

Functions with Parameters

A function is accessed by passing its parameter values to the body of the function. We can pass single to multiple parameter values as tuples inside the function.

Live Demo

```
func mult(no1: Int, no2: Int) -> Int {
   return no1*no2
}

print(mult(no1: 2, no2: 20))
print(mult(no1: 3, no2: 15))
print(mult(no1: 4, no2: 30))
```

When we run above program using playground, we get the following result −

```
40
45
120
```

Functions without Parameters

We may also have functions without any parameters.

Syntax

```
func funcname() -> datatype {
   return datatype
}
```

Following is an example having a function without a parameter −

Live Demo

```
func votersname() -> String {
   return "Alice"
}
print(votersname())
```

When we run the above program using playground, we get the following result −

```
Alice
```

Functions with Return Values

Functions are also used to return string, integer, and float data type values as return types. To find out the largest and smallest number in a given array function 'ls' is declared with large and small integer datatypes.

An array is initialized to hold integer values. Then the array is processed and each and every value in the array is read and compared for its previous value. When the value is lesser than the previous one it is stored in 'small' argument, otherwise it is stored in 'large' argument and the values are returned by calling the function.

Live Demo

```
func ls(array: [Int]) -> (large: Int, small: Int) {
   var lar = array[0]
   var sma = array[0]
   for i in array[1..<array.count] {
      if i < sma {
         sma = i
      } else if i > lar {
         lar = i
      }
   }
   return (lar, sma)
}

let num = ls(array: [40,12,-5,78,98])
print("Largest number is: \(num.large) and smallest number is: \(num.small)"
```

When we run the above program using playground, we get the following result −

```
Largest number is: 98 and smallest number is: -5
```

Functions without Return Values

Some functions may have arguments declared inside the function without any return values. The following program declares **a** and **b** as arguments to the sum() function. inside the function itself the values for arguments **a** and **b** are passed by invoking the function call sum() and its values are printed thereby eliminating return values.

Live Demo

```
func sum(a: Int, b: Int) {
   let a = a + b
   let b = a - b
   print(a, b)
}

sum(a: 20, b: 10)
sum(a: 40, b: 10)
sum(a: 24, b: 6)
```

When we run the above program using playground, we get the following result −

```
30 20
50 40
30 24
```

Functions with Optional Return Types

Swift 4 introduces 'optional' feature to get rid of problems by introducing a safety measure. Consider for example we are declaring function values return type as integer but what will happen when the function returns a string value or either a nil value. In that case compiler will return an error value. 'optional' are introduced to get rid of these problems.

Optional functions will take two forms 'value' and a 'nil'. We will mention 'Optionals' with the key reserved character '?' to check whether the tuple is returning a value or a nil value.

Live Demo

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
   if array.isEmpty { return nil }
   var currentMin = array[0]
   var currentMax = array[0]

   for value in array[1..<array.count] {
      if value < currentMin {
         currentMin = value
      } else if value > currentMax {
         currentMax = value
      }
   }
   return (currentMin, currentMax)
}

if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
   print("min is \(bounds.min) and max is \(bounds.max)")
}
```

When we run above program using playground, we get following result −

```
min is -6 and max is 109
```

"Optionals' are used to check 'nil' or garbage values thereby consuming lot of time in debugging and make the code efficient and readable for the user.

Functions Local Vs External Parameter Names

Local Parameter Names

Local parameter names are accessed inside the function alone.

```
func sample(number: Int) {
   print(number)
}
```

Here, the **func** sample argument number is declared as internal variable since it is accessed internally by the function sample(). Here the 'number' is declared as local variable but the reference to the variable is made outside the function with the following statement −

Live Demo

```
func sample(number: Int) {
   print(number)
}
```

```
sample(number: 1)
sample(number: 2)
sample(number: 3)
```

When we run the above program using playground, we get the following result −

```
1
2
3
```

External Parameter Names

External parameter names allow us to name a function parameters to make their purpose more clear. For example below you can name two function parameters and then call that function as follows −

Live Demo

```
func pow(firstArg a: Int, secondArg b: Int) -> Int {
   var res = a
   for _ in 1..<b {
      res = res * a
   }
   print(res)
   return res
}

pow(firstArg:5, secondArg:3)
```

When we run the above program using playground, we get the following result −

```
125
```

Variadic Parameters

When we want to define function with multiple number of arguments, then we can declare the members as 'variadic' parameters. Parameters can be specified as variadic by (···) after the parameter name.

Live Demo

```
func vari<N>(members: N...){
   for i in members {
      print(i)
   }
}

vari(members: 4,3,5)
vari(members: 4.5, 3.1, 5.6)
vari(members: "Swift 4", "Enumerations", "Closures")
```

When we run the above program using playground, we get the following result −

```
4
3
5
4.5
3.1
5.6
Swift 4
Enumerations
Closures
```

Constant, Variable and I/O Parameters

Functions by default consider the parameters as 'constant', whereas the user can declare the arguments to the functions as variables also. We already discussed that 'let' keyword is used to declare constant parameters and variable parameters is defined with 'var' keyword.

I/O parameters in Swift 4 provide functionality to retain the parameter values even though its values are modified after the function call. At the beginning of the function parameter definition, 'inout' keyword is declared to retain the member values.

It derives the keyword 'inout' since its values are passed 'in' to the function and its values are accessed and modified by its function body and it is returned back 'out' of the function to modify the original argument.

Variables are only passed as an argument for in-out parameter since its values alone are modified inside and outside the function. Hence no need to declare strings and literals as in-out parameters. '&' before a variable name refers that we are passing the argument to the in-out parameter.

Live Demo

```
func temp(a1: inout Int, b1: inout Int) {
   let t = a1
   a1 = b1
   b1 = t
}

var no = 2
var co = 10
temp(a1: &no, b1: &co)
print("Swapped values are \(no), \(co)")
```

When we run the above program using playground, we get the following result −

```
Swapped values are 10, 2
```

Function Types & its Usage

Each and every function follows the specific function by considering the input parameters and outputs the desired result.

```
func inputs(no1: Int, no2: Int) -> Int {
   return no1/no2
}
```

Following is an example −

Live Demo

```
func inputs(no1: Int, no2: Int) -> Int {
   return no1/no2
}

print(inputs(no1: 20, no2: 10))
print(inputs(no1: 36, no2: 6))
```

When we run the above program using playground, we get the following result −

```
2
6
```

Here the function is initialized with two arguments **no1** and **no2** as integer data types and its return type is also declared as 'int'

```
Func inputstr(name: String) -> String {
   return name
}
```

Here the function is declared as **string** datatype.

Functions may also have **void** data types and such functions won't return anything.

Live Demo

```
func inputstr() {
   print("Swift 4 Functions")
   print("Types and its Usage")
}
inputstr()
```

When we run the above program using playground, we get the following result −

```
Swift 4 Functions
Types and its Usage
```

The above function is declared as a void function with no arguments and no return values.

Using Function Types

Functions are first passed with integer, float or string type arguments and then it is passed as constants or variables to the function as mentioned below.

```
var addition: (Int, Int) -> Int = sum
```

Here sum is a function name having 'a' and 'b' integer variables which is now declared as a variable to the function name addition. Hereafter both addition and sum function both have same number of arguments declared as integer datatype and also return integer values as references.

Live Demo

```
func sum(a: Int, b: Int) -> Int {
   return a + b
}
var addition: (Int, Int) -> Int = sum
print("Result: \(addition(40, 89))")
```

When we run the above program using playground, we get the following result −

```
Result: 129
```

Function Types as Parameter Types & Return Types

We can also pass the function itself as parameter types to another function.

```
func sum(a: Int, b: Int) -> Int {
   return a + b
}
var addition: (Int, Int) -> Int = sum
print("Result: \(addition(40, 89))")

func another(addition: (Int, Int) -> Int, a: Int, b: Int) {
   print("Result: \(addition(a, b))")
}
another(sum, 10, 20)
```

When we run the above program using playground, we get the following result −

```
Result: 129
Result: 30
```

Nested Functions

A nested function provides the facility to call the outer function by invoking the inside function.

Live Demo

```
func calcDecrement(forDecrement total: Int) -> () -> Int {
   var overallDecrement = 0
   func decrementer() -> Int {
      overallDecrement -= total
      return overallDecrement
   }
   return decrementer
}

let decrem = calcDecrement(forDecrement: 30)
print(decrem())
```

When we run the above program using playground, we get the following result −

```
-30
```