**# Function:**

**Function contain:**

1. **Func keyword**
2. **Function name**
3. **Parameter**
4. **Return type --> like string, integer etc.**

Note:

* A function’s arguments must always be provided in the same order as the function’s parameter list.
* function’s return type with the *return arrow* ->

1. **Function without parameter**

Functions are not required to define input parameters. Here’s a function with no input parameters, which always returns the same String message whenever it is called:

1. func sayHelloWorld() -> String {
2. return "hello, world"
3. }
4. print(sayHelloWorld())
5. // Prints "hello, world"

The function definition still needs parentheses after the function’s name, even though it does not take any parameters. The function name is also followed by an empty pair of parentheses when the function is called.

1. **Function with multiple parameter**

Functions can have multiple input parameters, which are written within the function’s parentheses, separated by commas.

func detailRecord(name: String, dob: String) -> String

{

print(name)

print(dob)

// print(detailRecord(name: "jony", dob: "15August 1989"))

return (name)

}

// print(detailRecord(name: "anil", dob: "september"))

detailRecord(name: "jony", dob: "15 August 1987")

1. **Function without return type:**

Functions are not required to define a return type. Here’s a version of the greet(person:) function, which prints its own String value rather than returning it:

func withoutReturnType(name: String) {

print("Mention name \(name)")

}

print(withoutReturnType(name: "Vikrant"))

Because it does not need to return a value, the function’s definition does not include the return arrow (->) or a return type.

NOTE

Strictly speaking, this version of the greet(person:) function *does* still return a value, even though no return value is defined. Functions without a defined return type return a special value of type Void. This is simply an empty tuple, which is written as ().

1. **Function with multiple return type:**

You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.

func calculateResult(result: [Int]) -> (min:Int, max:Int, sum:Int) {

let min = 10

let max = 20

let sum = min + max

// print(sum)

return (min, max, sum)

}

print(calculateResult(result: [10]))

### Optional Tuple Return Types

If the tuple type to be returned from a function has the potential to have “no value” for the entire tuple, you can use an optional tuple return type to reflect the fact that the entire tuple can be nil. You write an optional tuple return type by placing a question mark after the tuple type’s closing parenthesis, such as (Int, Int)? or (String, Int, Bool)?.

// Functions with Multiple Return Values -- optional

func calculateResultWithOptional(EmailId: String, password: String) -> (String, String)? {

// var min

// var max

// let sum = min + max

// // print(sum)

return (EmailId, password)

}

// Functions with Multiple Return Values -- optional

print(calculateResultWithOptional(EmailId: "", password: "") )

NOTE

An optional tuple type such as (Int, Int)? is different from a tuple that contains optional types such as (Int?, Int?). With an optional tuple type, the entire tuple is optional, not just each individual value within the tuple.

1. **Function argument labels and parameter names:**

Each function parameter has both an *argument label* and a *parameter name*. The **argument label is used when calling the function;** **each argument is written in the function call with its argument label before it.** The **parameter name is used in the implementation of the function.** By default, parameters use their parameter name as their argument label.

//Argument Label With Parameter Name

func argumentLabelWithParameterName(argumentName paramterName: String, argumentFrom parameterCity: String) -> String

{

return ("Hello \(paramterName) which city you are \(parameterCity)")

}

print(argumentLabelWithParameterName(argumentName: "Jony", argumentFrom: "Tohana"))

1. **Specifying argument labels:**

You write an argument label before the parameter name, separated by a space:

func argumentLabelWithParameterName(argumentName paramterName: String, argumentFrom parameterCity: String) -> String

1. **Omitting Argument Labels**

If you don’t want an argument label for a parameter, write an **underscore** (\_) instead of an explicit argument label for that parameter.

1. func someFunction(\_ firstParameterName: Int, secondParameterName: Int) {
2. // In the function body, firstParameterName and secondParameterName
3. // refer to the argument values for the first and second parameters.
4. }
5. someFunction(1, secondParameterName: 2)

If a parameter has an argument label, the argument *must* be labeled when you call the function.

1. **Default Parameter Values**

You can define a *default value* for any parameter in a function by assigning a value to the parameter after that parameter’s type. If a default value is defined, you can omit that parameter when calling the function.

//Default Parameter Values

func defaultParameterValue(parameterWithoutDefault: String, parameterWithDefault: String = "Karnal") -> String {

return ("Hello \(parameterWithoutDefault) which city you are \(parameterWithDefault)")

}

print(defaultParameterValue(parameterWithoutDefault: "Vipin"))

### Variadic Parameters (Don’t understand concept know)

A variadic parameter accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called. Write variadic parameters by inserting three period characters (...) after the parameter’s type name.

// veriodic parameter

func veriodicParameter(names: String...) {

for name in names{

print("Name \(name)")

}

}

print(veriodicParameter(names: "Jony","Vikrant","Vipin","Ankur"))

### In-Out Parameters(Call by reference – change original value):

Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. This means that you can’t change the value of a parameter by mistake. If you want a function to modify a parameter’s value, and you want those changes to persist after the function call has ended, define that parameter as an in-out parameter instead.

You write an in-out parameter by placing the inout keyword right before a parameter’s type. An in-out parameter has a value that is passed in to the function, is modified by the function, and is passed back out of the function to replace the original value. For a detailed discussion of the behavior of in-out parameters and associated compiler optimizations, see [In-Out Parameters](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Declarations.html#//apple_ref/doc/uid/TP40014097-CH34-ID545).

You can only pass a variable as the argument for an in-out parameter. You cannot pass a constant or a literal value as the argument, because constants and literals cannot be modified. You place an ampersand (&) directly before a variable’s name when you pass it as an argument to an in-out parameter, to indicate that it can be modified by the function.

**NOTE**

In-out parameters cannot have default values, and variadic parameters cannot be marked as inout.

You can call the swapTwoInts(\_:\_:) function with two variables of type Int to swap their values. Note that the names of someInt and anotherInt are prefixed with an ampersand (&) when they are passed to the swapTwoInts(\_:\_:) function:

1. var someInt = 3
2. var anotherInt = 107
3. swapTwoInts(&someInt, &anotherInt)
4. print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
5. // Prints "someInt is now 107, and anotherInt is now 3"

The example above shows that the original values of someInt and anotherInt are modified by the swapTwoInts(\_:\_:) function, even though they were originally defined outside of the function.

**NOTE**

In-out parameters are not the same as returning a value from a function. The swapTwoInts example above does not define a return type or return a value, but it still modifies the values of someInt and anotherInt. In-out parameters are an alternative way for a function to have an effect outside of the scope of its function body.

### Function Types

Every function has a specific function type, made up of the parameter types and the return type of the function.

For example:

1. func addTwoInts(\_ a: Int, \_ b: Int) -> Int {
2. return a + b
3. }
4. func multiplyTwoInts(\_ a: Int, \_ b: Int) -> Int {
5. return a \* b
6. }

The type of both of these functions is (Int, Int) -> Int. This can be read as:

“A function that has two parameters, both of type Int, and that returns a value of type Int.”

### Using Function Types

You use function types just like any other types in Swift. For example, you can define a constant or variable to be of a function type and assign an appropriate function to that variable:

var mathFunction: (Int, Int) -> Int = addTwoInts

This can be read as:

“Define a variable called mathFunction, which has a type of ‘a function that takes two Int values, and returns an Int value.’ Set this new variable to refer to the function called addTwoInts.”

The addTwoInts(\_:\_:) function has the same type as the mathFunction variable, and so this assignment is allowed by Swift’s type-checker.

You can now call the assigned function with the name mathFunction:

1. print("Result: \(mathFunction(2, 3))")
2. // Prints "Result: 5"

### Function Types as Parameter Types

You can use a function type such as (Int, Int) -> Int as a parameter type for another function. This enables you to leave some aspects of a function’s implementation for the function’s caller to provide when the function is called.

Here’s an example to print the results of the math functions from above:

1. func printMathResult(\_ mathFunction: (Int, Int) -> Int, \_ a: Int, \_ b: Int) {
2. print("Result: \(mathFunction(a, b))")
3. }
4. printMathResult(addTwoInts, 3, 5)
5. // Prints "Result: 8"

This example defines a function called printMathResult(\_:\_:\_:), which has three parameters. The first parameter is called mathFunction, and is of type (Int, Int) -> Int. You can pass any function of that type as the argument for this first parameter. The second and third parameters are called a and b, and are both of type Int. These are used as the two input values for the provided math function.

### Function Types as Return Types

You can use a function type as the return type of another function. You do this by writing a complete function type immediately after the return arrow (->) of the returning function.

### Nested Functions

All of the functions you have encountered so far in this chapter have been examples of global functions, which are defined at a global scope. You can also define functions inside the bodies of other functions, known as nested functions.