Functions

Swift

Functions

Functions are self-contained chunks of code that perform a specific task.

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

Function Parameters and Return Values

Functions without parameters

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Prints "hello, world"
```

Functions with multiple parameters

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

```
func greet(person: String, alreadyGreeted: Bool) -> String {
   if alreadyGreeted {
      return greetAgain(person: person)
   } else {
      return greet(person: person)
   }
}
print(greet(person: "Tim", alreadyGreeted: true))
   // Prints "Hello again, Tim!"
```

Functions Without Return Values

Functions are not required to define a return type.

```
func greet(person: String) {
 print("Hello, \(person)!")
}
greet(person: "Dave")
// Prints "Hello, Dave!"
```

Functions With Multiple Return Values

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 minMax(array: [Int]) -> (min: Int, max: Int) {
    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)
}
```

Because the tuple's member values are named as part of the function's return type, they can be accessed with dot syntax to retrieve the minimum and maximum found values:

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

Optional Return Values

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`.

```
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)
}
```

Function Argument Labels and Parameter Names

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.

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

Specifying Argument Labels

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

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

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.

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

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.

```
func someFunction(parameterWithoutDefault: Int,
   parameterWithDefault: Int = 12) {
// If you omit the second argument when calling this
   function, then
// the value of parameterWithDefault is 12 inside the
   function body.
 }
someFunction(parameterWithoutDefault: 3,
   parameterWithDefault: 6) // parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4) //
   parameterWithDefault is 12
```

Variadic Parameters

A *variadic parameter* accepts zero or more values of a specified type.

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
arithmeticMean(3, 8.25, 18.75)
```

Int Out Parameters

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.

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
  let temporaryA = a
  a = b
  b = temporaryA
}
```

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

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \
   (anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

Function Types

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

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

The type of both of these functions is `(Int, Int) -> Int`.

Using Function Type

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

mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
```

Function Types as Parameter Types

You can use a function type such as `(Int, Int) -> Int` as a parameter type for another function.

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

Function Types as Return Type

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

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

Nested Functions

You can also define functions inside the bodies of other functions, known as *nested functions*.

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#/