ON THIS PAGE

Swift

Functions

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to “call” the function to perform its task when needed.

Swift’s unified function syntax is flexible enough to express anything from a simple C-style function with no parameter names to a complex Objective-C-style method with names and argument labels for each parameter. Parameters can provide default values to simplify function calls and can be passed as in-out parameters, which modify a passed variable once the function has completed its execution.

Every function in Swift has a type, consisting of the function’s parameter types and return type. You can use this type like any other type in Swift, which makes it easy to pass functions as parameters to other functions, and to return functions from functions. Functions can also be written within other functions to encapsulate useful functionality within a nested function scope.

Defining and Calling Functions

When you define a function, you can optionally define one or more named, typed values that the function takes as input, known as *parameters*. You can also optionally define a type of value that the function will pass back as output when it is done, known as its *return type*.

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 it input values (known as *arguments*) that match the types of the function's parameters. A function's arguments must always be provided in the same order as the function's parameter list.

The function in the example below is called `greet(person:)`, because that's what it does—it takes a person's name as input and returns a greeting for that person. To accomplish this, you define one input parameter—a `String` value called `person`—and a return type of `String`, which will contain a greeting for that person:

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

All of this information is rolled up into the function's *definition*, which is prefixed with the `func` keyword. You indicate the function's return type with the *return arrow* `->` (a hyphen followed by a right angle bracket), which is followed by the name of the type to return.

The definition describes what the function does, what it expects to receive, and what it returns when it is done. The definition makes it easy for the function to be called unambiguously from elsewhere in your code:

```
print(greet(person: "Anna"))
// Prints "Hello, Anna!"
print(greet(person: "Brian"))
// Prints "Hello, Brian!"
```

You call the `greet(person:)` function by passing it a `String` value after the `person` argument label, such as `greet(person: "Anna")`. Because the function returns a `String` value, `greet(person:)` can be wrapped in a call to the `print(_:separator:terminator:)` function to print that string and see its return value, as shown above.

> NOTE

> The `print(_:separator:terminator:)` function doesn't have a label for its first argument, and its other arguments are optional because they have a default value. These variations on function syntax are discussed below in Function Argument Labels and Parameter Names and Default Parameter Values.

The body of the `greet(person:)` function starts by defining a new `String` constant called `greeting` and setting it to a simple greeting message. This greeting is then passed back out of the function using the `return` keyword. In the line of code that says `return greeting`, the function finishes its execution and returns the current value of `greeting`.

You can call the `greet(person:)` function multiple times with different input values. The example above shows what happens if it is called with an input value of `"Anna"`, and an input value of `"Brian"`. The function returns a tailored greeting in each case.

To make the body of this function shorter, you can combine the message creation and the return statement into one line:

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
```

Function Parameters and Return Values

Function parameters and return values are extremely flexible in Swift. You can define anything from a simple utility function with a single unnamed parameter to a complex function with expressive parameter names and different parameter options.

Functions Without Parameters

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:

```
func sayHelloWorld() -> String {
```

```
    return "hello, world"
}
print(sayHelloWorld())
// 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.

Functions With Multiple Parameters

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

This function takes a person's name and whether they have already been greeted as input, and returns an appropriate greeting for that person:

```
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!"
```

You call the `greet(person:alreadyGreeted:)` function by passing it both a `String` argument value labeled `person` and a `Bool` argument value labeled `alreadyGreeted` in parentheses, separated by commas. Note that this function is distinct from the `greet(person:)` function shown in an earlier section. Although both functions have names that begin with `greet`, the `greet(person:alreadyGreeted:)` function takes two arguments but the `greet(person:)` function takes only one.

Functions Without Return Values

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

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 `()`.

The return value of a function can be ignored when it is called:

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting(string: "hello, world")
// prints "hello, world" but does not return a value
```

The first function, `printAndCount(string:)`, prints a string, and then returns its character count as an `Int`. The second function, `printWithoutCounting(string:)`, calls the first function, but ignores its return value. When the second function is called, the message is still printed by the first function, but the returned value is not used.

> NOTE
>
> Return values can be ignored, but a function that says it will return a value must always do so. A function with a defined return type cannot allow control to fall out of the bottom of the function without returning a value, and attempting to do so will result in a compile-time error.

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.

The example below defines a function called `minMax(array:)`, which finds the smallest and largest numbers in an array of `Int` values:

```
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)
}
```

The `minMax(array:)` function returns a tuple containing two `Int` values. These values are labeled `min` and `max` so that they can be accessed by name when querying the function's return value.

The body of the `minMax(array:)` function starts by setting two working variables called `currentMin` and `currentMax` to the value of the first integer in the array. The function then iterates over the remaining values in the array and checks each value to see if it is smaller or larger than the values of `currentMin` and `currentMax` respectively. Finally, the overall minimum and maximum values are returned as a tuple of two `Int` values.

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"
```

Note that the tuple's members do not need to be named at the point that the tuple is returned from the function, because their names are already specified as part of the function's return type.

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)?`.

> 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.

The `minMax(array:)` function above returns a tuple containing two `Int` values. However, the function does not perform any safety checks on the array it is passed. If the `array` argument contains an empty array, the `minMax(array:)` function, as defined above, will trigger a runtime error when attempting to access `array[0]`.

To handle an empty array safely, write the `minMax(array:)` function with an optional tuple return type and return a value of `nil` when the array is empty:

```
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)
}
```

You can use optional binding to check whether this version of the `minMax(array:)` function returns an actual tuple value or `nil`:

```
if 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"
```

Functions With an Implicit Return

If the entire body of the function is a single expression, the function implicitly returns that expression. For example, both functions below have the same behavior:

```
func greeting(for person: String) -> String {
    "Hello, " + person + "!"
}
print(greeting(for: "Dave"))
// Prints "Hello, Dave!"

func anotherGreeting(for person: String) -> String {
    return "Hello, " + person + "!"
}
print(anotherGreeting(for: "Dave"))
// Prints "Hello, Dave!"
```

The entire definition of the `greeting(for:)` function is the greeting message that it returns, which means it can use this shorter form. The `anotherGreeting(for:)` function returns the same greeting message, using the `return` keyword like a longer function. Any function that you write as just one `return` line can omit the `return`.

As you'll see in Shorthand Getter Declaration, property getters can also use an implicit return.

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.

```
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)
```

All parameters must have unique names. Although it's possible for multiple parameters to have the same argument label, unique argument labels help make your code more readable.

Specifying Argument Labels

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

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

Here's a variation of the `greet(person:)` function that takes a person's name and hometown and returns a greeting:

```
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."
```

The use of argument labels can allow a function to be called in an expressive, sentence-like manner, while still providing a function body that is readable and clear in intent.

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)
```

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

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
```

Place parameters that don't have default values at the beginning of a function's parameter list, before the parameters that have default values. Parameters that don't have default values are usually more important to the function's meaning—writing them first makes it easier to recognize that the same function is being called, regardless of whether any default parameters are omitted.

Variadic Parameters

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.

The values passed to a variadic parameter are made available within the function's body as an array of the appropriate type. For example, a variadic parameter with a name of `numbers` and a type of `Double...` is made available within the function's body as a constant array called `numbers` of type `[Double]`.

The example below calculates the *arithmetic mean* (also known as the *average*) for a list of numbers of any length:

```
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)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

> NOTE
>
> A function may have at most one variadic parameter.

In-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. 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.

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`.

Here's an example of a function called `swapTwoInts(_:_:)`, which has two in-out integer parameters called `a` and `b`:

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

The `swapTwoInts(_:_:)` function simply swaps the value of `b` into `a`, and the value of `a` into `b`. The function performs this swap by storing the value of `a` in a temporary constant called `temporaryA`, assigning the value of `b` to `a`, and then assigning `temporaryA` to `b`.

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:

```
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"
```

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:

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

This example defines two simple mathematical functions called `addTwoInts` and `multiplyTwoInts`. These functions each take two `Int` values, and return an `Int` value, which is the result of performing an appropriate mathematical operation.

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`."

Here's another example, for a function with no parameters or return value:

```
func printHelloWorld() {
    print("hello, world")
}
```

The type of this function is `() -> Void`, or "a function that has no parameters, and returns `Void`."

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`:

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

A different function with the same matching type can be assigned to the same variable, in the same way as for nonfunction types:

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

```
// Prints "Result: 6"
```

As with any other type, you can leave it to Swift to infer the function type when you assign a function to a constant or variable:

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int)
  -> Int
```

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:

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

When `printMathResult(_:_:_:)` is called, it is passed the `addTwoInts(_:_:)` function, and the integer values `3` and `5`. It calls the provided function with the values `3` and `5`, and prints the result of `8`.

The role of `printMathResult(_:_:_:)` is to print the result of a call to a math function of an appropriate type. It doesn't matter what that function's implementation actually does—it matters only that the function is of the correct type. This enables `printMathResult(_:_:_:)` to hand off some of its functionality to the caller of the function in a type-safe way.

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.

The next example defines two simple functions called `stepForward(_:)` and `stepBackward(_:)`. The `stepForward(_:)` function returns a value one more than its input value, and the `stepBackward(_:)` function returns a value one less than its input value. Both functions have a type of `(Int) -> Int`:

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

Here's a function called `chooseStepFunction(backward:)`, whose return type is `(Int) -> Int`. The `chooseStepFunction(backward:)` function returns the `stepForward(_:)` function or the `stepBackward(_:)` function based on a Boolean parameter called `backward`:

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

You can now use `chooseStepFunction(backward:)` to obtain a function that will step in one direction or the other:

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward:
  currentValue > 0)
// moveNearerToZero now refers to the stepBackward()
  function
```

The example above determines whether a positive or negative step is needed to move a variable called `currentValue` progressively closer to zero. `currentValue` has an initial value of `3`, which means that `currentValue > 0` returns `true`, causing `chooseStepFunction(backward:)` to return the `stepBackward(_:)` function. A reference to the returned function is stored in a constant called `moveNearerToZero`.

Now that `moveNearerToZero` refers to the correct function, it can be used to count to zero:

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

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*.

Nested functions are hidden from the outside world by default, but can still be called and used by their enclosing function. An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope.

You can rewrite the `chooseStepFunction(backward:)` example above to use and return 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!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

Control Flow Closures

BETA SOFTWARE

This documentation contains preliminary information about an API or technology in development. This information is subject to change, and software implemented according to this documentation should be tested with final operating system software.

Learn more about using Apple's beta software