Functions

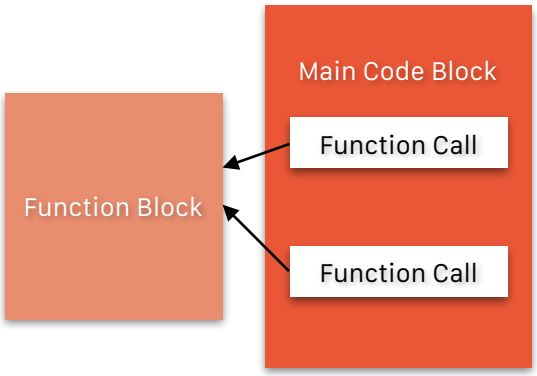
Avoid Repeating Code

When you repeat code of any kind, you are very liable to make a mistake, even, and probably especially, when you are copying and pasting code. You are likely to forget to change something important in your repeating code, causing a bug in your code. You are likely to change some part of one instance of your repeating code, but not in other instances where it needs to be changed. You are likely to make many mistakes and not be able to find them easily in repeating code.

Previously, we used constants, variables, while loops, and for loops to avoid repeating code. However, those don't help with repeating code that needs to be used at different times. So how can we create code that can be used again and again at different points in our code without having to copy and paste it?

Functions

With functions! Functions are blocks of code that can be "called" as many times as needed. When you call a function, you run all the statements in the code block.



Example:

```
func print_cat() {
      print("  /\\-/\\")
      print(" (=^Y^=)")
      print("  (>o<)")
}

print_cat()
print()
print_cat()
```

Output

```
 /\-/\
(=^Y^=)
 (>o<)

 /\-/\
(=^Y^=)
 (>o<)
```

print()

Because print() always prints a new line after the value in its parentheses, you can just write `print()` like in the example to the left and that will print just a new line.

func denotes that the code in curly braces { } will be a function. print_cat() is the name of the function and can be anything just the name of a variable / constant, although the parentheses are necessary (you'll see why later on.)

To actually use the code inside of the function, simply type the function's name and parentheses. A function's code will *only* be run at the point or points in the code where the function's name is called. It will not run by simply being defined.

Communicating with Functions

Just like in while and for loops, sometimes you want a function to do something different based on input. However, unlike while and for loops, code that calls the function cannot use the function's variables / constants, so there needs to be some way of returning values to the main code.

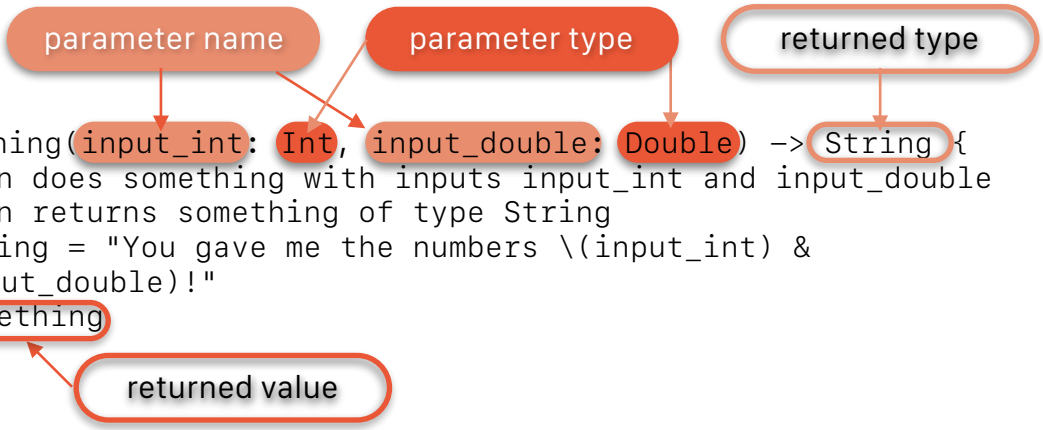
Example:

```
func greeting(forPerson: String) -> String {
      let message = "Hello, \(forPerson)!"
      return message
}

print(greeting(forPerson: "Ash"))
print(greeting(forPerson: "Donald"))
```

Output
```
Hello, Ash!
Hello, Donald!
```

A function broken down:



Parameters

The parameters of a function are the function's inputs and are, by default, constants (even if they are variables in the calling of the function) that the function can use to complete its code.

It might be helpful to think of functions in math that you already know. For example, x is a parameter for f(x). When you get into later years of math, you'll see functions like f(x, y) that take in multiple parameters separated by commas. That's the same with Swift functions, where multiple parameters are separated by commas. Also, just like functions in math, to use parameters given to Swift functions, you simply call the parameters by their name.

A parameter's name is what is in the parentheses of the function. By default, you must write the parameter name when calling the function. For example,

```
let some_double = 56.12
print(does_something(input_int: 36, input_double: some_double))
```

Output
```
You gave me the numbers 36 and 56.12!
```

When calling a function, you put its parameters in the parentheses after the function's name in the call, like above. You can use literal and constant / variable values in the function call, as long as their type matches the parameter's type.

The parameter types in the creation of the function is necessary so that Swift knows what the function is supposed to be using in the function. You must then, in the function call, list the inputs in the right order. So, if the function asks for a String that is someone's name, another String for their nickname, a Double for their height, and an Int for their age, then you have to call the function with the parameters in that order or you will have a bug in your code.

Returning Values

The `return` keyword denotes that whatever comes after it on its line will become the value of the function call. For instance, if you have

```
let does_something_value = does_something(input_int: 12, input_double: 12)
```

`does_something_value` becomes the value of `something` from the function body, because the function says `return something`. The code outside of a function doesn't know that `something` exists, so if you want to use the value of `something`, you must return it and assign the function call to a constant, like above.

Like with parameters, Swift has to know what types its dealing with, so, in this case, it has to know what type of value it will be returning. That's why we write `-> Type`, or in the above example, `-> String`, where Type is the type of value the function will be returning.

Same Name!

You can have separate function with the same name as long as they take in different types of inputs. For example,

```
func greet(person: String) -> String {
     return "Hello, \(person)!"
}

func greet(person: String, alreadyGreeted: Bool) -> String {
     if alreadyGreeted {
          return "Hello again, \(person)!"
     } else {
          return "Hello, \(person)!"
     }
}
```

In this example, greet(person:) has the same name as greet(person:alreadyGreeted:), however, they have different inputs, so they can have the same name.

Argument Labels & Parameter Names

Each function parameter has both an argument label and a parameter name. However, so far, you've only seen one label for each parameter. That's because, by default, parameter names act as argument labels.

```
func someFunction(firstArgumentLabel firstParameterName: Int,
                  secondArgumentLabel secondParameterName: Int) {
     let some = firstParameterName + secondParameterName
     print(sum)
}

someFunction(firstArgumentLabel: 1, secondArgumentLabel: 2)
```

When you actually specify an argument label, it is what is typed in the function call, and the parameter name, in that case, is only used inside of the function. Because parameter names are used in the actual function's code, they must all have distinct names from each other. Argument labels do not have this rule. However, unique argument labels help make your code more readable.

Default Parameter Values

You can define a default value for any parameter in a function by assigning a value to the parameter in the function's definition. If a default value is defined, you can omit that parameter when calling the function. For example,

```
func someFunction(parameterWithoutDefault: Int,
                  parameterWithDefault: Int = 12) {
    print("Parameter without default value:
        \(parameterWithoutDefault)")
    print("Parameter with default value: \(parameterWithDefault)")
}

someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6)
print()
someFunction(parameterWithoutDefault: 4)
```

Output

```
Parameter without default
value: 3
Parameter with default value:
6

Parameter without default
value: 4
Parameter with default value:
12
```

To declare a default parameter, you give it a value in the function's definition, like above, where `parameterWithDefault` is given the value 12. Note that Int there is still needed and must match the default value's type.

If you don't give `parameterWithDefault` a value in calling the function, it is then assigned its default value, which is 12 in this case. If you do give `parameterWithDefault` a value in calling the function, it is assigned the value in the function call, not its default value.

Note that you can put parameters with default values in front of parameters without default values and omit the parameters with default values. Swift can still tell what parameters you're using if you write unique argument labels because the argument label says what parameter the value should be assigned to.

However, it is better and clearer to write default parameters at the end of your parameter list in the function definition. By doing so, you can still tell that you are using the same function.

Multiple Return Values

Sometimes, you'll want to return multiple values from a function, like producing coordinates (x, y). Sometimes, your function will be doing a time / resource-consuming task, like iterating over an array, and you want to get multiple values from that task. Instead of doing the time / resource-consuming task to find the minimum value of an array and then redoing the task to find the maximum value of an array.

How would you do this? With tuples! Tuples group multiple values into a single compound value, like arrays. However, unlike arrays, values in a tuple do not have to be of the same type. So, for example, you could have

```
let http404Error = (code: 404, description: "Not found")
```

Here, `http404Error` is a tuple of type `(Int, String)`, and equals `(404, "Not found")`. As you can see, tuples are denoted by parentheses ( ), just like how arrays are denoted by square

brackets [ ].

Tuples can contain as many values as you'd like, however, unlike arrays, tuples cannot grow or decrease in size.

You do not actually need to write labels for tuples as tuples, just like arrays, are numbered, starting at 0. For example, you could just write

```
let http404Error = (404, "Not found")
```

However, labels make it clearer what you accessing in the tuple. If you wanted to access the Int 404, you could type `http404Error.code` instead of the unclear `http404Error.0`.

So how do you actually return a tuple? Here's an example:

```
func min_max(array: [Int]) -> (min: Int, max: Int) {
     var current_min = array[0]
     var current_max = array[0]
     for value in array[1..<array.count] {
          if value < current_min {
               current_min = value
          } else if value > current_max {
               current_max = value
          }
     }
     return (current_min, current_max)
}
```

On the first line of the function, you can see that min_max(array:) returns a tuple of type (Int, Int). The labels min and max are the labels of the returned tuple and *can* be used outside of the function. (current_min, current_max) is the value of the returned tuple, but these names *cannot* be used outside of the function.

So to access a value from a tuple, you could write,

```
let data = [8, -6, 2, 109, 3, 71]
let bounds = min_max(array: data)
// bounds is a tuple of type (Int, Int) because the value being
// returned from min_max(array:) is a tuple of type (Int, Int)
print("min is \(bounds.min) and max is \(bounds.max)")
```

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

To access the value from the tuple, like above, you just write the tuple that you assigned your function's return value, a period, and the label of the value you want. The label, as said before, is defined in the function's first line.

Miscellaneous

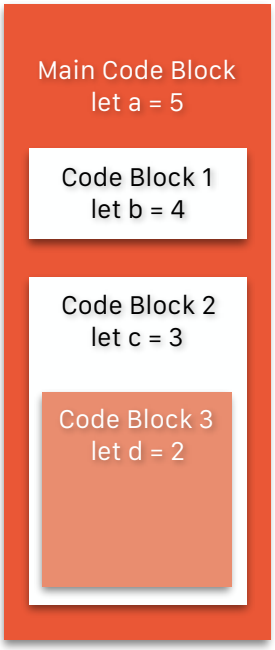
You don't actually have to use a function's return value or store it in a constant when the function ends. You can simply call the function on one line, like a function that doesn't return anything. However, you have to return a value in the actual function if it says it will return a value. As well, the return value must be the same type as the type the function says it will return. You can use the `return` keyword in a function that doesn't return anything to end the function, but there can't be anything after it, as the function said it wouldn't return anything.

Scope

Scope is basically the area where a variable / constant lives. The main code block is where the code starts running. The other code blocks can be anything with curly braces { }, like if statements, if-else statements, functions, for loops, while loops, etc.

In programming, each code block has its own scope where all of its variables / constants live, and code blocks in other code blocks can access the variables / constants of the code blocks they are in. However, these code blocks cannot access any other variables / constants.

That probably sounds pretty confusing, so let's break it down into boxes. Think of each code block as its own box with its variables / constants inside. Now think of these boxes as closed that can only be opened from the inside. A box can open itself and access anything in itself. Now think of some of these boxes as inside other boxes, just like how code blocks can be in other code blocks, so that they don't need to worry about opening the box they are in, because they are already on the inside.



Modifying Variables in Outer Boxes

If you can use a variable in another code block, you can modify it as well. When you exit the inner code block, and the variable still exists, then the variable's last value is the variable's value, no matter if the code block that modified it last has exited.

Let's put some actual numbers to this example now. In the right figure, the main code block is like our main box with everything in it. This means, as we said before, that all boxes in this main box can access the main box's variables / constants, which, in this example, is the constant a. Code block 1 can use b and a, code block 2 can use c and a, and code block 3 can use d and a.

Great. Now what about code block 1's constant b? Well, the main code box / block cannot access b because code block 1 is closed and the main code box is not inside of code block 1. The same goes for code block 2 and 3. The only box that can access the constant b is code block 1.

What about code block 2's constant c? Like before, the main code block and code block 1 cannot access c because they are not inside code block 2. However, code block 3 *can* because it is inside of code block 2, so code block 3 can use a, c, and d.

Lastly, no other box except code block 3 can access code block 3's constant d, because there are no other code blocks in it.