Functions and Closures

Use `func` to declare a function. Call a function by following its name with a list of arguments in parentheses. Use `->` to separate the parameter names and types from the function's return type.

```
func greet(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet(person: "Bob", day: "Tuesday")
```

EXPERIMENT

Remove the `day` parameter. Add a parameter to include today's lunch special in the greeting.

By default, functions use their parameter names as labels for their arguments. Write a custom argument label before the parameter name, or write `_` to use no argument label.

```
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

Use a tuple to make a compound value—for example, to return multiple values from a function. The elements of a tuple can be 참조하다 referred to either by name or by number.

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}
let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics.sum)
// Prints "120"
print(statistics.2)
// Prints "120"
```

Functions can be 중첩하다 nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that's long or complex.

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

Functions are a first-class type. This means that a function can return another function as its value.

```swift
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

A function can take another function as one of its arguments.

```swift
func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(list: numbers, condition: lessThanTen)
```

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it's executed—you saw an example of this already with nested functions. You can write a closure without a name by surrounding code with braces (`{}`). Use `in` to separate the arguments and return type from the body.

```swift
numbers.map({ (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

EXPERIMENT

Rewrite the closure to return zero for all odd numbers.

You have several options for writing closures more 간결하게 concisely. When a closure's type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both. Single statement closures implicitly return the value of their only statement.

```swift
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
// Prints "[60, 57, 21, 36]"
```

You can refer to parameters by number instead of by name—this approach is especially useful in very short closures. A closure that is passed as the last argument to a function can appear immediately after the parentheses. When a closure is the only argument to a function, you can omit the parentheses entirely.

```swift
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
// Prints "[20, 19, 12, 7]"
```