```
import UIKit
//: # Functions
//: ### A function is a group of statements that defines an action to be performed
//: ### The main use of a function is to make the code reusable
/*:
 ## Types of Functions
 1. Library functions - Functions that are defined already in Swift Framework.
 2. User-defined functions - Functions created by the programmer themselves.
*/
//: ### Defining a Function
//func function_name(args...) -> ReturnType {
//    //statements
//    return value
//}
//: ### Ex:
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
// or
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
//: ## Calling a function
// greet(person: "Anna")
/*:
 ### Things to remember before declaring a function
 * Give a function name that reflects the purpose of the function.
 * A function should accomplish only one task. If a function does more than one task, break down it into multiple functions.
 * Try to think early and group statements inside a function which makes the code reusable and modular.
*/
//: ## Function Parameters and Return Values
//: ## Functions Without Parameters
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Prints "hello, world"
//: ### Functions With Multiple Parameters
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!"
```

```
//: The two greet function have the same name --> overloading
//: ## function overloading
//: * same name for the functions
//: * having different number or types of parameters
//: * Ex
func output(value text:String) {
    print(text)
}

func output(value num:Int) {
    print(num)
}

output(value: 2)
output(value: "Hello")
//: ### Functions Without Return Values
func greet2(person: String) {
    print("Hello, \(person)!")
}
greet2(person: "Dave")
// Prints "Hello, Dave!"
//: ## Functions with Multiple Return 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)
}
//: 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 Tuple Return Types
//: * handle an empty array safely in the example above
func minMax2(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)
}
//: retrieve the minimum and maximum found values
if let bounds = minMax2(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"
//: ## Function Argument Labels and Parameter Names
//: * Each function parameter has both an argument label and a parameter name
//: * write an argument label before the parameter name, separated by a space
//: * The argument label is used when calling the function
//: * The parameter name is used in the implementation of the function
//: ### The use of argument labels allow a function to be called in an expressive way, sentence-like manner, while still providing a function body that is readable and clear in intent
/*:
 ## Function Syntax with argument labels
 **To define a function in Swift, the following syntaxes can be used:**

 *func funcname(argumentLabel parameterName:Type)-> Return Type {*

 *W body of the function*

 *}*
 */
//: ### another variation of the greet(person:) function
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
//: * Use an underscore (_) instead of an explicit argument label for that parameter
/*:
 ### Syntax:
 *func someFunction(_ firstParameterName: Int, secondParameterName: Int) {*

 *W In the function body, firstParameterName and secondParameterName*

 *W refer to the argument values for the first and second parameters*

 *}*
 *someFunction(1, secondParameterName: 2)*
*/
//: ### Example with argument labels
func sum(of a :Int, and b:Int) -> Int {

    return a + b
}
let result = sum(of: 2, and: 3)
print("The sum is \(result)")
//: ## omitting the argument label
```

```
func sum(_ a :Int, _ b:Int) -> Int {

    return a + b
}
let resultNew = sum(2, 3)
print("The sum is \(result)")
//: ### Default Parameter Values
/*:
 ### Syntax:

 *func funcname(parameterName:Type = value) -> Return Type {*

 *\Wstatements*

 *}*
*/
func sumNew(of a :Int = 7, and b:Int = 8) -> Int {

    return a + b
}
let result1 = sumNew(of: 2, and: 3)
print("The sum is \(result1)")

let result2 = sumNew(of: 2)
print("The sum is \(result2)")

let result3 = sumNew(and: 2)
print("The sum is \(result3)")

let result4 = sumNew()
print("The sum is \(result4)")
//: ## Variadic Parameters
//: * A variadic parameter can accept zero or more values of a specific type
//: * The values passed to a variadic parameter are made available within the function’s body as an array of the appropriate type
//: * A function may have at most one variadic parameter
//: * In the example below, the values 1, 2, 3, 4, 5, 6, 7, 8 passed as a variadic parameter are made available within the function's body as a constant array of the Int type
/*:
 ### Syntax:

 *func funcname(parameterName:Type...) -> Return Type {*

 *\Wstatements*

 *}*
 */
//: ### Example:
func sum(of numbers:Int...) {
    var result = 0
    for num in numbers {
```

```
        result += num
    }
    print("The sum of numbers is \(result)")
}
sum(of: 1, 2, 3, 4, 5, 6, 7, 8)
//: ## In-Out Parameters
//: * Function parameters are constants by default and so you cannot change the value of a parameter
//: * *write an in-out parameter by placing the* inout* keyword right before a parameter's type to modify a parameter's value
//: * In-out parameters cannot have default values, and variadic parameters cannot be marked as inout
//: * place an ampersand (&) directly before a variable’s name when you pass it as an argument to an in-out parameter
//: * the value passed in the function call cannot be a constant, it must be a variable
/*:
 ### Syntax:

 *func funcname(parameterName:inout Type) -> Return Type {*

 *\Wstatements*

 *}*
 */
//: ### Example 1:
func process(name:inout String) {
    if name == ""{
        name = "guest"
    }
}
var userName = ""
process(name: &userName)
print(userName)
//: ### Example 2:
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

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 --> parameter types + the return type
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 Types
var mathFunction: (Int, Int) -> Int = addTwoInts
//: ### this new variable refers to the function called addTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"

mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
//: ### function type can be infered with swift
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
//: ## Function Types as Parameter Types
//: * Our example function type: (Int, Int) -> Int
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8"
//: ### mathFunction is of type (Int, Int) -> Int
//: ## Function Types as Return Types
//: ### Example:
func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}

func stepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
var currentValue = 3
let moveToZero = stepFunction(backward: currentValue > 0)
// moveToZero now refers to the stepBackward() function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
//: ## Nested Functions
//: * Nested functions are hidden from the outside world
//: * An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope
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
}
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!

//: ## Swift Input
/*: ### readLine() is used to read the input from the user. It has two forms:
 * readLine() : The default way.
 * readLine(strippingNewLine: Bool) : This is default set to true. Swift always assumes that the newline is not a part of the input
*/
//: ### readLine() function always returns an Optional String by default.
let str = readLine() //assume you enter your Name
//: print(str) //prints Optional(name)
//: ### To unwrap the optional we can use the following code:
if let str = readLine(){
    print(str)
}
//: ### Reading an Int or a Float
if let input = readLine()
{
    if let int = Int(input)
    {
        print("Entered input is \(int) of the type:\(type(of: int))")
    }
    else{
        print("Entered input is \(input) of the type:\(type(of: input))")
    }
}
//: ### Reading Multiple Inputs
```