Funciones, parte uno

Las funciones nos permiten envolver piezas de código para que se puedan usar en muchos lugares. Podemos enviar datos a funciones para personalizar cómo funcionan y recuperar datos que nos indiquen el resultado que se calculó.

Lo creas o no, las llamadas a las funciones solían ser muy lentas. Steve Johnson, el autor de muchas de las primeras herramientas de codificación para el sistema operativo Unix, dijo esto:

> "Dennis Ritchie (el creador del lenguaje de programación C) alentó la modularidad diciéndole a todos y varios que las llamadas de función eran muy, muy baratas en C. Todo el mundo empezó a escribir pequeñas funciones y a modularizar. Años más tarde descubrimos que las llamadas a funciones todavía eran caras, y nuestro código a menudo pasaba el 50 % de su tiempo solo llamándolas. ¡Dennis nos había mentido! Pero era demasiado tarde; todos estábamos enganchados..."

¿Por qué estarían "enganchados" a las llamadas a funciones? Porque hacen mucho para ayudar a simplificar nuestro código: en lugar de copiar y pegar las mismas 10 líneas de código en una docena de lugares, podemos envolverlas en una función y usarla en su lugar. Eso significa menos duplicación de código, pero también significa que si cambias esa función, tal vez agregando más trabajo, entonces en todas partes que la uses obtendrás automáticamente el nuevo comportamiento, y no hay riesgo de que te olvides de actualizar uno de los lugares en los que lo pegó.

Hoy tienes cuatro tutoriales a seguir, y aprenderás a escribir tus propias funciones, a aceptar parámetros y a devolver datos. Una vez que hayas completado cada vídeo, puedes repasar cualquier lectura opcional si lo necesitas, y luego hacer una breve prueba para asegurarte de que has entendido lo que se enseñó.

Cómo reutilizar el código con funciones

Cuando hayas escrito algún código que realmente te guste y quieras usarlo una y otra vez, deberías considerar ponerlo en una función. Las funciones son solo trozos de código que has separado del resto de tu programa y que se te ha dado un nombre para que puedas hacer referencia a ellos fácilmente.

Por ejemplo, digamos que teníamos este código bonito y sencillo:

```
print("Welcome to my app!")
print("By default This prints out a conversion")
print("chart from centimeters to inches, but you")
print("can also set a custom range if you want.")
```

Ese es un mensaje de bienvenida para una aplicación, y es posible que desee que se imprima cuando se inicie la aplicación, o tal vez cuando el usuario pida ayuda.

Pero, ¿y si quisieras que fuera en ambos lugares? Sí, podrías copiar esas cuatro líneas de print() y ponerlas en ambos lugares, pero ¿y si quisieras ese texto en diez lugares? ¿O qué pasaría si quisieras cambiar la redacción más tarde? ¿Realmente te acordarías de cambiarla dondequiera que apareciera en tu código?

Aquí es donde entran las funciones: podemos extraer ese código, darle un nombre y ejecutarlo cuando y donde lo necesitemos. Esto significa que todas las líneas print() permanecen en un solo lugar y se reutilizan en otro lugar.

Así es como se ve eso:

```
func showWelcome() {
    print("Welcome to my app!")
    print("By default This prints out a conversion")
    print("chart from centimeters to inches, but you")
    print("can also set a custom range if you want.")
}
```

Vamos a desglosar eso...

1. Comienza con la palabra clave func, que marca el inicio de una función.

2. Estamos nombrando la función showWelcome. Este puede ser cualquier nombre que quieras, pero trata de hacerlo memorable: printInstructions(), displayHelp(), etc. son buenas opciones.

3. El cuerpo de la función está contenido dentro de las llaves abiertas y cerradas, al igual que el cuerpo de bucles y el cuerpo de condiciones.

Hay una cosa extra allí, y es posible que lo reconozcas por nuestro trabajo hasta ahora: el () directamente después del espectáculo. Bienvenido. Nos conocimos por primera vez hace mucho tiempo cuando miramos las cadenas, cuando dije que el conteo no tiene () después, pero en mayúsculas() sí.

Bueno, ahora estás aprendiendo por qué: esos () se usan con funciones. Se utilizan cuando creas la función, como puedes ver arriba, pero también cuando llamas a la función, cuando le pides a Swift que ejecute su código. En nuestro caso, podemos llamar a nuestra función así:

```
showWelcome()
```

Eso se conoce como el sitio de llamada de la función, que es un nombre elegante que significa "un lugar donde se llama a una función".

Entonces, ¿qué hacen realmente los paréntesis? Bueno, ahí es donde añadimos opciones de configuración para nuestras funciones: podemos pasar datos que personalizan la forma en que funciona la función, para que la función se vuelva más flexible.

Por ejemplo, ya usamos código como este:

```
let number = 139

if number.isMultiple(of: 2) {
    print("Even")
} else {
    print("Odd")
}
```

isMultiple(of:) es una función que pertenece a enteros. Si no permitiera ningún tipo de personalización, simplemente no tendría sentido, ¿es un múltiplo de qué? Claro, Apple podría haber hecho que esto fuera algo así como isOdd() o isEven(), por lo que nunca necesitó tener opciones de configuración, pero al poder escribir (de: 2) de repente la función se vuelve más poderosa, porque ahora podemos comprobar si hay múltiplos de 2, 3, 4, 5, 50 o cualquier otro número.

Del mismo modo, cuando estábamos tirando dados virtuales antes, usamos código como este:

```
let roll = Int.random(in: 1...20)
```

Una vez más, random() es una función, y la parte (en: 1...20) marca las opciones de configuración; sin eso no tendríamos control sobre el rango de nuestros números aleatorios, lo que sería significativamente menos útil.

Podemos hacer nuestras propias funciones que estén abiertas a la configuración, todo poniendo código adicional entre los paréntesis cuando creamos nuestra función. A esto se le debe dar un solo número entero, como 8, y calcular las tablas de multiplicación para el del 1 al 12.

Aquí está el código:

```
func printTimesTables(number: Int) {
    for i in 1...12 {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(number: 5)
```

¿Te das cuenta de cómo he colocado el número: Int dentro de los paréntesis? Eso se llama parámetro, y es nuestro punto de personalización. Estamos diciendo que quienquiera que llame a esta función debe pasar un entero aquí, y Swift lo hará cumplir. Dentro de la función, el número está disponible para usar como cualquier otra constante, por lo que aparece dentro de la llamada print().

Como puede ver, cuando se llama a printTimesTables(), necesitamos escribir explícitamente el número: 5 - necesitamos escribir el nombre del parámetro como parte de la llamada a la función. Esto no es común en otros idiomas, pero creo que es muy útil en Swift como recordatorio de lo que hace cada parámetro.

Esta denominación de parámetros se vuelve aún más importante cuando tienes varios parámetros. Por ejemplo, si quisiéramos personalizar lo alto que fueron nuestras tablas de multiplicar, podríamos hacer que el final de nuestro rango se establezca utilizando un segundo parámetro, como este:

```
func printTimesTables(number: Int, end: Int) {
    for i in 1...end {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(number: 5, end: 20)
```

Ahora eso toma dos parámetros: un entero llamado número y un punto final llamado final. Ambos deben nombrarse específicamente cuando se ejecuta printTimesTables(), y espero que puedas ver ahora por qué son importantes. Imagínate si nuestro código fuera este en su lugar:

```
printTimesTables(5, 20)
```

¿Recuerdas qué número era cuál? Tal vez. Pero, ¿te acuerdas dentro de seis meses? Probablemente no.

Ahora, técnicamente le damos nombres ligeramente diferentes al envío de datos y a la recepción de datos, y aunque muchas personas (yo incluido) ignoran esta distinción, al menos voy a hacerte consciente de ello para que no te pillen desprevenido más tarde.

Echa un vistazo a este código:

```
printTimesTables(number: 5, end: 20)
```

Allí los argumentos 5 y 20 son: son los valores reales que se envían a la función para trabajar, que se utilizan para rellenar el número y el final.

Por lo tanto, tenemos tanto parámetros como argumentos: uno es un marcador de posición, el otro es un valor real, por lo que si alguna vez olvidas cuál es cuál, solo recuerda el parámetro/marcador de posición, argumento/valor real.

¿Importa esta distinción de nombre? En realidad no: uso "parámetro" para ambos, y he sabido que otras personas usan "argumento" para ambos, y honestamente, ni una sola vez en mi carrera ha causado el más mínimo problema. De hecho, como verás en breve en Swift, la distinción es muy confusa, por lo que no vale la pena pensar en ello.

Importante: Si prefieres usar "argumento" para los datos que se pasan y "parámetro" para los datos que se reciben, eso depende de ti, pero realmente uso "parámetro" para ambos, y lo haré a lo largo de este libro y más allá.

Independientemente de si los está llamando "argumentos" o "parámetros", cuando le pide a Swift que llame a la función, siempre debe pasar los parámetros en el orden en que se enumeraron cuando creó la función.

Así que, para este código:

```
func printTimesTables(number: Int, end: Int) {
```

Este no es un código válido, porque termina antes del número:

```
printTimesTables(end: 20, number: 5)
```

Consejo: Cualquier dato que crees dentro de una función se destruye automáticamente cuando se termina la función.

¿Qué código se debe poner en una función?

Las funciones están diseñadas para que podamos reutilizar el código fácilmente, lo que significa que no tenemos que copiar y pegar código para obtener comportamientos comunes. Podrías usarlos muy raramente si quisieras, pero honestamente no tiene sentido: son herramientas maravillosas para ayudarnos a escribir un código más claro y flexible.

Hay tres veces que querrás crear tus propias funciones:

- El momento más común es cuando quieres la misma funcionalidad en muchos lugares. El uso de una función aquí significa que puede cambiar una pieza de código y tener en todas partes que use su función se actualice.

- Las funciones también son útiles para romper el código. Si tienes una función larga, puede ser difícil seguir todo lo que está sucediendo, pero si la divides en tres o cuatro funciones más pequeñas, entonces se vuelve más fácil de seguir.

- La última razón es más avanzada: Swift nos permite construir nuevas funciones a función de las funciones existentes, que es una técnica llamada composición de funciones. Al dividir su trabajo en múltiples funciones pequeñas, la composición de funciones nos permite construir grandes funciones combinando esas pequeñas funciones de varias maneras, un poco como los ladrillos de Lego.

¿Cuántos parámetros debe aceptar una función?

A primera vista, esta pregunta parece "¿cuánto dura un trozo de cuerda?" Es decir, es algo en lo que no hay una respuesta real y difícil: una función podría tomar ningún parámetro o tomar 20 de ellos.

Eso es ciertamente cierto, pero cuando una función toma muchos parámetros, tal vez seis o más, pero esto es extremadamente subjetivo, tienes que empezar a preguntarte si esa función tal vez esté haciendo demasiado trabajo.

- ¿Necesita esos seis parámetros?

- ¿Podría dividirse esa función en funciones más pequeñas que tomen menos parámetros?

- ¿Deberían agruparse esos parámetros de alguna manera?

Examinaremos algunas técnicas para resolver esto más adelante, pero hay una lección importante que aprender aquí: esto se llama un "olor a código", algo sobre nuestro código que sugiere un problema subyacente en la forma en que hemos estructurado nuestro programa.

Cómo devolver valores de funciones

Has visto cómo crear funciones y cómo agregarles parámetros, pero las funciones a menudo también envían datos de vuelta: realizan algunos cálculos y luego devuelven el resultado de ese trabajo al sitio de llamada.

Swift tiene muchas de estas funciones integradas, y hay decenas de miles más en los marcos de Apple. Por ejemplo, nuestro patio de recreo siempre ha tenido Cocoa de importación en la parte superior, y eso incluye una variedad de funciones matemáticas como sqrt() para calcular la raíz cuadrada de un número.

La función sqrt() acepta un parámetro, que es el número del que queremos calcular la raíz cuadrada, y seguirá adelante y hará el trabajo y luego devolverá la raíz cuadrada.

Por ejemplo, podríamos escribir esto:

```
let root = sqrt(169)
print(root)
```

Si quieres devolver tu propio valor de una función, tienes que hacer dos cosas:

- Escriba una flecha y luego un tipo de datos antes de la llave de apertura de su función, que le dice a Swift qué tipo de datos se devolverán.
- Utilice la palabra clave return para devolver sus datos.

Por ejemplo, tal vez quieras tirar un dado en varias partes de tu programa, pero en lugar de forzar siempre la tirada de dados a usar un dado de 6 lados, podrías convertirlo en una función:

```
func rollDice() -> Int {
    return Int.random(in: 1...6)
}

let result = rollDice()
print(result)
```

Por lo tanto, eso dice que la función debe devolver un entero, y el valor real se envía de vuelta con la palabra clave return.

Usando este enfoque, puedes llamar a rollDice() en muchos lugares de tu programa, y todos usarán dados de 6 caras. Pero si en el futuro decides que quieres usar un dado de 20 lados, solo tienes que cambiar esa función para actualizar el resto de tu programa.

Importante: Cuando digas que tu función devolverá un Int, Swift se asegurará de que siempre devuelva un Int. No puedes olvidarte de devolver un valor, porque tu código no se construirá.

Vamos a probar un ejemplo más complejo: ¿dos cadenas contienen las mismas letras, independientemente de su orden? Esta función debe aceptar dos parámetros de cadena, y devolver true si sus letras son las mismas, por lo que "abc" y "cab" deben devolver true porque ambos contienen una "a", una "b" y una "c".

En realidad, ya sabes lo suficiente como para resolver este problema tú mismo, pero ya has aprendido tanto que probablemente hayas olvidado la única cosa que hace que esta tarea sea tan fácil: si llamas a sorted() en cualquier cadena, obtienes una nueva cadena con todas las letras en orden alfabético. Por lo tanto, si haces eso para ambas cadenas, puedes usar == para compararlas y ver si sus letras son iguales.

Adelante, intenta escribir la función tú mismo. Una vez más, no te preocupes si tienes problemas, todo es muy nuevo para ti, y luchar por recordar nuevos conocimientos es parte del proceso de aprendizaje. Te mostraré la solución en un momento, pero por favor, pruébalo tú mismo primero.

¿Sigues aquí? Vale, aquí hay un ejemplo de solución:

```
func areLettersIdentical(string1: String, string2: String) -> Bool {
    let first = string1.sorted()
    let second = string2.sorted()
    return first == second
}
```

Vamos a desglosar eso:

- Crea una nueva función llamada areLettersIdentical().
- La función acepta dos parámetros de cadena, string1 y string2.
- La función dice que devuelve un Bool, por lo que en algún momento siempre debemos devolver verdadero o falso.

Dentro del cuerpo de la función, clasificamos ambas cadenas y luego usamos == para comparar las cadenas; si son las mismas, devolverá true, de lo contrario, devolverá false.

Ese código ordena tanto la cadena 1 como la cadena2, asignando sus valores ordenados a las nuevas constantes, primera y segunda. Sin embargo, eso no es necesario: podemos omitir esas constantes temporales y simplemente comparar los resultados de sorted() directamente, así:

```
func areLettersIdentical(string1: String, string2: String) -> Bool {
    return string1.sorted() == string2.sorted()
}
```

Eso es menos código, pero podemos hacerlo aún mejor. Verás, le hemos dicho a Swift que esta función debe devolver un booleano, y debido a que solo hay una línea de código en la función, Swift sabe que es la que debe devolver datos. Debido a esto, cuando una función tiene solo una línea de código, podemos eliminar la palabra clave de retorno por completo, así:

```
func areLettersIdentical(string1: String, string2: String) -> Bool {
    string1.sorted() == string2.sorted()
}
```

También podemos volver atrás y hacer lo mismo con la función rollDice():

```
func rollDice() -> Int {
    Int.random(in: 1...6)
}
```

Recuerde, esto solo funciona cuando su función contiene una sola línea de código y, en particular, esa línea de código debe devolver los datos que prometió devolver.

Vamos a probar un tercer ejemplo. ¿Recuerdas el teorema de Pitágoras de la escuela? Afirma que si tienes un triángulo con un ángulo recto en el interior, puedes calcular la longitud de la hipotenusa cuadrando ambos lados, sumándolos y luego calculando la raíz cuadrada del resultado

Ya has aprendido a usar sqrt(), para que podamos construir una función pythagoras() que acepte dos números decimales y devuelva otro:

```swift
func pythagoras(a: Double, b: Double) -> Double {
    let input = a * a + b * b
    let root = sqrt(input)
    return root
}

let c = pythagoras(a: 3, b: 4)
print(c)
```

Por lo tanto, esa es una función llamada pythagoras(), que acepta dos parámetros de doble y devuelve otro doble. En su interior, cuadra a y b, los suma, luego los pasa a sqrt() y devuelve el resultado.

Esa función también se puede reducir a una sola línea y eliminar su palabra clave de retorno, inténtalo. Como de costumbre, te mostraré mi solución después, pero es importante que lo intentes.

¿Sigues aquí? Vale, aquí está mi solución:

```swift
func pythagoras(a: Double, b: Double) -> Double {
    sqrt(a * a + b * b)
}
```

Hay una última cosa que quiero mencionar antes de seguir adelante: si su función no devuelve un valor, aún puede usar return por sí misma para forzar que la función salga temprano. Por ejemplo, tal vez tenga una comprobación de que la entrada coincide con lo que esperaba, y si no es así, desea salir de la función inmediatamente antes de continuar.

¿Cuándo no se necesita la palabra clave return en una función Swift?

Utilizamos la palabra clave return para devolver valores de las funciones en Swift, pero hay un caso específico en el que no es necesario: cuando nuestra función contiene una sola expresión.

Ahora, "expresión" no es una palabra que uso a menudo, pero es importante entenderla aquí. Cuando escribimos programas hacemos cosas como esta:

```swift
5 + 8
```

O esto:

```swift
greet("Paul")
```

Estas líneas de código se resuelven a un solo valor: 5 + 8 se resuelve a 13, y greet ("Paul") podría devolver una cadena "¡Hola, Paul!"

Incluso un código más largo se resolverá en un solo valor. Por ejemplo, si tuviéramos tres constantes booleanas como esta:

```swift
let isAdmin = true
let isOwner = false
let isEditingEnabled = false
```

Entonces esta línea de código se resolvería en un solo valor:

```
isOwner == true && isEditingEnabled || isAdmin == true
```

Eso se convertiría en "verdadero", porque a pesar de que isOwner es falso, isAdmin es verdadero, por lo que toda la expresión se vuelve verdadera.

Por lo tanto, mucho código que escribimos se puede resolver en un solo valor. Pero también hay mucho código que no se puede resolver en un solo valor. Por ejemplo, ¿cuál es el valor aquí?

```
let name = "Otis"
```

Sí, eso crea una constante, pero no se convierte en un valor por derecho propio: no podríamos escribir return let name = "Otis".

Del mismo modo, escribimos que podríamos tomar acciones como esta:

```
if name == "Maeve" {
    print("Hello, Maeve!")
    print("How are you?")
}
```

Eso tampoco puede convertirse en un solo valor, porque tiene dos llamadas a funciones.

Ahora, todo esto importa porque estas divisiones tienen nombres: cuando nuestro código se puede reducir a un solo valor, como verdadero, falso, "Hola" o 19, lo llamamos una expresión. Las expresiones son cosas que se pueden asignar a una variable o imprimir usando print(). Por otro lado, cuando estamos realizando acciones como crear variables, iniciar un bucle o comprobar una condición, entonces lo llamamos una declaración.

Todo esto importa porque Swift nos permite omitir el uso de la palabra clave return cuando solo tenemos una expresión en nuestra función. Por lo tanto, estas dos funciones hacen lo mismo:

```
func doMath() -> Int {
    return 5 + 5
}

func doMoreMath() -> Int {
    5 + 5
}
```

Recuerda, la expresión que hay dentro puede ser todo el tiempo que quieras, pero no puede contener ninguna declaración, no hay nuevas variables, y así sucede.

Swift es muy inteligente aquí, y nos permitirá automáticamente usar si simples y cambiar valores de la misma manera, siempre y cuando devuelvan valores directamente en lugar de intentar crear nuevas variables, etc.

Por ejemplo, esto está permitido:

```
func greet(name: String) -> String {
    if name == "Taylor Swift" {
        "Oh wow!"
    } else {
        "Hello, \(name)"
    }
}
```

Se devuelve una cadena directamente de ambas partes de esa condición, por lo que está permitida. Sin embargo, esto no está permitido:

```
func greet(name: String) -> String {
    if name == "Taylor Swift" {
        "Oh wow!"
    } else {
        let greeting = "Hello, \(name)"
        return greeting
    }
}
```

Eso intenta crear una nueva constante de saludo antes de devolverla.

Internamente, lo que está sucediendo aquí es que si es capaz de convertirse en una expresión por derecho propio, siempre y cuando cada rama del if, cada posible resultado que puede producir, sea en sí misma una sola expresión.

Esto nos permite asignar el resultado de una condición directamente a un nuevo valor, que parece un poco extraño al principio:

```
func greet(name: String) -> String {
    let response = if name == "Taylor Swift" {
        "Oh wow!"
    } else {
        "Hello, \(name)"
    }

    return response
}
```

Eso suele ser un poco difícil de entender al principio, pero trata de pensar en ello como un operador condicional ternario:

```
func greet(name: String) -> String {
    let response = name == "Taylor Swift" ? "Oh wow!" : "Hello, \(name)"
    return response
}
```

Cómo devolver varios valores de las funciones

Cuando desea devolver un solo valor de una función, escribe una flecha y un tipo de datos antes de la llave de apertura de su función, de la siguiente manera:

```
func isUppercase(string: String) -> Bool {
    string == string.uppercased()
}
```

Eso compara una cadena con la versión en mayúsculas de sí misma. Si la cadena ya estaba completamente en mayúsculas, entonces nada habrá cambiado y las dos cadenas serán idénticas, de lo contrario serán diferentes y == devolverán false.

Si desea devolver dos o más valores de una función, podría usar una matriz. Por ejemplo, aquí hay uno que devuelve los detalles de un usuario:

```
func getUser() -> [String] {
    ["Taylor", "Swift"]
}

let user = getUser()
print("Name: \(user[0]) \(user[1])")
```

Eso es problemático, porque es difícil recordar qué son el usuario[0] y el usuario[1], y si alguna vez ajustamos los datos de esa matriz, entonces el usuario[0] y el usuario[1] podrían terminar siendo otra cosa o tal vez no existir en absoluto.

Podríamos usar un diccionario en su lugar, pero eso tiene sus propios problemas:

```
func getUser() -> [String: String] {
    [
        "firstName": "Taylor",
        "lastName": "Swift"
    ]
}

let user = getUser()
print("Name: \(user["firstName", default: "Anonymous"]) \(user["lastName",
```

Sí, ahora hemos dado nombres significativos a las diversas partes de nuestros datos de usuario, pero mira esa llamada a print() - a pesar de que sabemos que tanto el nombre como el apellido existirán, todavía tenemos que proporcionar valores predeterminados en caso de que las cosas no sean lo que esperamos.

Ambas soluciones son bastante malas, pero Swift tiene una solución en forma de tuplas. Al igual que las matrices, los diccionarios y los conjuntos, las tuplas nos permiten poner múltiples piezas de datos en una sola variable, pero a diferencia de esas otras opciones, las tuplas tienen un tamaño fijo y pueden tener una variedad de tipos de datos.

Así es como se ve nuestra función cuando devuelve una tupla:

```
func getUser() -> (firstName: String, lastName: String) {
    (firstName: "Taylor", lastName: "Swift")
}

let user = getUser()
print("Name: \(user.firstName) \(user.lastName)")
```

Vamos a desglosar eso...

- Nuestro tipo de retorno es ahora (firstName: String, lastName: String), que es una tupla que contiene dos cadenas.
- Cada cadena de nuestra tupla tiene un nombre. Estos no están entre comillas: son nombres específicos para cada elemento de nuestra tupla, a diferencia de los tipos de claves arbitrarias que teníamos en los diccionarios.
- Dentro de la función, enviamos de vuelta una tupla que contiene todos los elementos que prometimos, adjunta a los nombres: firstName se establece en "Taylor", etc.
- Cuando llamamos a getUser(), podemos leer los valores de la tupla usando los nombres de clave: nombre, apellido, etc.

Sé que las tuplas parecen muy similares a los diccionarios, pero son diferentes:

- Cuando accedes a los valores de un diccionario, Swift no puede saber de antemano si están presentes o no. Sí, sabíamos que el usuario ["nombre"] iba a estar allí, pero Swift no puede estar seguro, por lo que tenemos que proporcionar un valor predeterminado.
- Cuando accedes a valores en una tupla, Swift sabe de antemano que está disponible porque la tupla dice que estará disponible.
- Accedemos a los valores usando user.firstName: no es una cadena, por lo que tampoco hay posibilidad de errores tipográficos.
- Nuestro diccionario puede contener cientos de otros valores junto con "nombre", pero nuestra tupla no puede. Debemos enumerar todos los valores que contendrá y, como resultado, se garantiza que los contendrá todos y nada más.

Por lo tanto, las tuplas tienen una ventaja clave sobre los diccionarios: especificamos exactamente qué valores existirán y qué tipos tienen, mientras que los diccionarios pueden o no contener los valores que estamos pidiendo.

Hay otras tres cosas que es importante saber cuando se usan tuplas.

En primer lugar, si devuelves una tupla de una función, Swift ya sabe los nombres que le das a cada elemento de la tupla, por lo que no necesitas repetirlos al usar return. Por lo tanto, este código hace lo mismo que nuestra tupla anterior:

```
func getUser() -> (firstName: String, lastName: String) {
    ("Taylor", "Swift")
}
```

En segundo lugar, a veces te darás tuplas donde los elementos no tienen nombres. Cuando esto sucede, puedes acceder a los elementos de la tupla usando índices numéricos a partir de 0, así:

```
func getUser() -> (String, String) {
    ("Taylor", "Swift")
}

let user = getUser()
print("Name: \(user.0) \(user.1)")
```

Estos índices numéricos también están disponibles con tuplas que tienen elementos nombrados, pero siempre me ha parecido preferible usar nombres.

Finalmente, si una función devuelve una tupla, puedes separar la tupla en valores individuales si lo deseas.

Para entender lo que quiero decir con eso, primero eche un vistazo a este código:

```
func getUser() -> (firstName: String, lastName: String) {
    (firstName: "Taylor", lastName: "Swift")
}

let user = getUser()
let firstName = user.firstName
let lastName = user.lastName

print("Name: \(firstName) \(lastName)")
```

Eso se remonta a la versión nombrada de getUser(), y cuando sale la tupla, estamos copiando los elementos de allí en contenidos individuales antes de usarlos. No hay nada nuevo aquí; solo estamos moviendo un poco los datos.

Sin embargo, en lugar de asignar la tupla al usuario y luego copiar valores individuales de allí, podemos omitir el primer paso: podemos separar el valor de retorno de getUser() en dos constantes separadas, como esta:

```
let (firstName, lastName) = getUser()
print("Name: \(firstName) \(lastName)")
```

Esa sintaxis podría lastimarte la cabeza al principio, pero en realidad es solo una abreviatura de lo que teníamos antes: convertir la tupla de dos elementos que obtenemos de getUser() en dos constantes separadas.

De hecho, si no necesitas todos los valores de la tupla, puedes ir un paso más allá usando _ para decirle a Swift que ignore esa parte de la tupla:

¿Cuándo deberías usar una matriz, un conjunto o una tupla en Swift?

Debido a que las matrices, los conjuntos y las tuplas funcionan de maneras ligeramente diferentes, es importante asegurarse de elegir la correcta para que sus datos se almacenen de manera correcta y eficiente.

Recuerde: las matrices mantienen el orden y pueden tener duplicados, los conjuntos no están ordenados y no pueden tener duplicados, y las tuplas tienen un número fijo de valores de tipos fijos dentro de ellas.

Así que:

- Si quieres almacenar una lista de todas las palabras en un diccionario para un juego, que no tiene duplicados y el orden no importa, así que irías por un set.
- Si desea almacenar todos los artículos leídos por un usuario, usaría un conjunto si el pedido no importaba (si todo lo que le importaba era si lo habían leído o no), o usaría una matriz si el orden sí importaba.
- Si quieres almacenar una lista de puntuaciones altas para un videojuego, que tiene un orden que importa y podría contener duplicados (si dos jugadores obtienen la misma puntuación), por lo que usarías una matriz.
- Si desea almacenar artículos para una lista de tareas pendientes, eso funciona mejor cuando el pedido es predecible, por lo que debe usar una matriz.
- Si quieres tener con precisión dos cadenas, o precisamente dos cadenas y un entero, o precisamente tres booleanos, o similar, debes usar una tupla.

Cómo personalizar las etiquetas de parámetros

Has visto cómo a los desarrolladores de Swift les gusta nombrar sus parámetros de función, porque hace que sea más fácil recordar lo que hacen cuando se llama a la función. Por ejemplo, podríamos escribir una función para tirar un dado un cierto número de veces, utilizando parámetros para controlar el número de lados de los dados y el número de tiradas:

```
func rollDice(sides: Int, count: Int) -> [Int] {
    // Start with an empty array
    var rolls = [Int]()

    // Roll as many dice as needed
    for _ in 1...count {
        // Add each result to our array
        let roll = Int.random(in: 1...sides)
        rolls.append(roll)
    }

    // Send back all the rolls
    return rolls
}

let rolls = rollDice(sides: 6, count: 4)
```

Incluso si vuelves a este código seis meses después, estoy seguro de que rollDice (lados: 6, conteo: 4) se explica por sí mismo.

Este método de nombrar parámetros para uso externo es tan importante para Swift que en realidad utiliza los nombres cuando se trata de averiguar a qué método llamar. Esto es bastante diferente a muchos otros idiomas, pero esto es perfectamente válido en Swift:

```
func hireEmployee(name: String) { }
func hireEmployee(title: String) { }
func hireEmployee(location: String) { }
```

Sí, todas esas son funciones llamadas hireEmployee(), pero cuando las llamas Swift sabe a cuál te refieres en función de los nombres de los parámetros que proporcionas. Para distinguir entre las diversas opciones, es muy común ver que la documentación se refiere a cada función, incluidos sus parámetros, como este: hireEmployee(name:) o hireEmployee(title:).

A veces, sin embargo, estos nombres de parámetros son menos útiles, y hay dos formas que quiero ver.

Primero, piensa en la función hasPrefix() que aprendiste antes:

```
let lyric = "I see a red door and I want it painted black"
print(lyric.hasPrefix("I see"))
```

Cuando llamamos a hasPrefix() pasamos el prefijo para comprobarlo directamente - no decimos hasPrefix(string:) o, peor aún, hasPrefix(prefix:). ¿Cómo es que?

Bueno, cuando estamos definiendo los parámetros para una función, en realidad podemos agregar dos nombres: uno para usar donde se llama la función, y otro para usar dentro de la función en sí. hasPrefix() usa esto para especificar _ como el nombre externo de su parámetro, que es la forma de Swift de decir "ignorar esto" y hace que no haya una etiqueta externa para ese parámetro.

Podemos usar la misma técnica en nuestras propias funciones, si crees que se lee mejor. Por ejemplo, anteriormente teníamos esta función:

```
func isUppercase(string: String) -> Bool {
    string == string.uppercased()
}

let string = "HELLO, WORLD"
let result = isUppercase(string: string)
```

Podrías mirar eso y pensar que es exactamente correcto, pero también podrías mirar la cadena: cadena y ver una molesta duplicación. Después de todo, ¿qué más vamos a pasar allí aparte de una cuerda?

Si añadimos un guión bajo antes del nombre del parámetro, podemos eliminar la etiqueta del parámetro externo de la siguiente manera:

```swift
func isUppercase(_ string: String) -> Bool {
    string == string.uppercased()
}

let string = "HELLO, WORLD"
let result = isUppercase(string)
```

Esto se usa mucho en Swift, como con append() para agregar elementos a una matriz, o contains() para comprobar si un elemento está dentro de una matriz; en ambos lugares es bastante evidente cuál es el parámetro sin tener una etiqueta también.

El segundo problema con los nombres de parámetros externos es cuando no son del todo correctos: quieres tenerlos, así que _ no es una buena idea, pero simplemente no se leen de forma natural en el sitio de llamada de la función.

A modo de ejemplo, aquí hay otra función que vimos anteriormente:

```swift
func printTimesTables(number: Int) {
    for i in 1...12 {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(number: 5)
```

El código de sombrero es Swift válido, y podríamos dejarlo en paz como está. Pero el sitio de llamadas no se lee bien: printTimesTables (número: 5). Sería mucho mejor así:

```swift
func printTimesTables(for: Int) {
    for i in 1...12 {
        print("\(i) x \(for) is \(i * for)")
    }
}

printTimesTables(for: 5)
```

Eso se lee mucho mejor en el sitio de la llamada: literalmente puedes decir "tabla de tiempos de impresión para 5" en voz alta, y tiene sentido. Pero ahora tenemos Swift no válido: aunque se permite y se lee muy bien en el sitio de llamadas, no se permite dentro de la función.

Ya has visto cómo podemos poner _ antes del nombre del parámetro para que no tengamos que escribir un nombre de parámetro externo. Bueno, la otra opción es escribir un segundo nombre allí: uno para usar externamente y otro para usar internamente.

Así es como se ve eso:

```swift
func printTimesTables(for number: Int) {
    for i in 1...12 {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(for: 5)
```

Hay tres cosas que debes mirar de cerca:

- Escribimos para el número: Int: el nombre externo es para, el nombre interno es número y es de tipo Int.
- Cuando llamamos a la función, usamos el nombre externo para el parámetro: printTimesTables (para: 5).
- Dentro de la función usamos el nombre interno para el parámetro: print("\(i) x \(number) is \(i * number)").

Por lo tanto, Swift nos da dos formas importantes de controlar los nombres de los parámetros: podemos usar _ para el nombre del parámetro externo para que no se use, o agregar un segundo nombre allí para que tengamos nombres de parámetros externos e internos.

Consejo: Anteriormente mencioné que técnicamente los valores que pasas a una función se llaman "argumentos", y los valores que recibes dentro de la función se llaman parámetros. Aquí es donde las cosas se confunden un poco, porque ahora tenemos etiquetas de argumentos y nombres de parámetros uno al lado del otro, ambos en la definición de la función. Como dije, usaré el término "parámetro" para ambos, y cuando la distinción sea importante, verás que los distinguo usando "nombre de parámetro externo" y "nombre de parámetro interno".

¿Cuándo deberías omitir una etiqueta de parámetro?

Si usamos un guión bajo para la etiqueta externa de un parámetro de función, Swift no nos permite usar ningún nombre para ese parámetro. Esta es una práctica muy común en algunas partes del desarrollo de Swift, particularmente cuando se crean aplicaciones que no usan SwiftUI, pero hay muchas otras ocasiones en las que también querrás usar esto.

La razón principal para omitir el nombre de un parámetro es cuando el nombre de su función es un verbo y el primer parámetro es un sustantivo sobre el que actúa el verbo. Por ejemplo:

Saludar a una persona sería saludar (taylor) en lugar de saludar (persona: taylor)

Comprar un producto sería comprar (cepillo de dientes) en lugar de comprar (artículo: cepillo de dientes)

Encontrar un cliente sería encontrar (cliente) en lugar de encontrar (usuario: cliente)

Esto es particularmente importante cuando es probable que la etiqueta de parámetros sea la misma que el nombre de lo que sea que estés pasando:

Cantar una canción sería cantar (canción) en lugar de cantar (canción: canción)

Habilitar una alarma sería habilitar (alarma) en lugar de habilitar (alarma: alarma)

Leer un libro se leería (libro) en lugar de leer (libro: libro)

Antes de que llegara SwiftUI, las aplicaciones se creaban utilizando los marcos UIKit, AppKit y WatchKit de Apple, que se diseñaron utilizando un lenguaje más antiguo llamado Objective-C. En ese lenguaje, el primer parámetro de una función siempre se dejó sin nombre, por lo que cuando uses esos marcos en

Swift verás muchas funciones que tienen guiones bajos para su primera etiqueta de parámetro para preservar la interoperabilidad con Objective-C.