Loops

Los bucles son una de las cosas que hacen que los ordenadores sean tan brillantes: la capacidad de repetir algunas tareas simples miles de millones de veces cada segundo.

Por supuesto, lo que va en esos bucles depende de ti: podrías estar agregando algunos números, podrías estar leyendo algunos datos de sensores o podrías estar redibujando la pantalla 120 veces por segundo. Como dijo Craig Bruce, "es el hardware lo que hace que una máquina sea rápida, pero es el software lo que hace que una máquina rápida sea lenta".

Hoy tienes tres tutoriales para trabajar, además de un resumen y un punto de control. Una vez que hayas completado cada vídeo, puedes leer la sección adicional opcional, y hay una breve prueba para asegurarte de que has entendido lo que se enseñó.

Cómo usar un bucle for para repetir el trabajo

Los ordenadores son realmente geniales para hacer trabajos repetitivos, y Swift hace que sea fácil repetir algún código un número fijo de veces, o una vez por cada elemento en una matriz, diccionario o conjunto.

Comencemos con algo simple: si tenemos una matriz de cadenas, podemos imprimir cada cadena de esta manera:

```
let platforms = ["iOS", "macOS", "tvOS", "watchOS"]

for os in platforms {
    print("Swift works great on \(os).")
}
```

Eso pasa por encima de todos los elementos en las plataformas, poniéndolos uno por uno en os. No hemos creado os en ningún otro lugar; se ha creado para nosotros como parte del bucle y solo está disponible dentro de los corséntesis de apertura y cierre.

Dentro de las llaves está el código que queremos ejecutar para cada elemento de la matriz, por lo que el código anterior imprimirá cuatro líneas, una para cada elemento de bucle. Primero pone "iOS" allí, luego llama a print(), luego pone "macOS" allí y llama a print(), luego a "tvOS", luego a "watchOS".

Para que las cosas sean más fáciles de entender, le damos a estas cosas nombres comunes:

- Llamamos al código dentro de los brackets el cuerpo del bucle
- Llamamos a un ciclo a través del cuerpo del bucle una iteración de bucle.
- Llamamos a la variable de bucle. Esto solo existe dentro del cuerpo del bucle, y cambiará a un nuevo valor en la siguiente iteración del bucle.

Debo decir que el nombre os no es especial, podríamos haber escrito esto en su lugar:

```
for name in platforms {
    print("Swift works great on \(name).")
}
```

O incluso esto:

```
for rubberChicken in platforms {
    print("Swift works great on \(rubberChicken).")
}
```

El código seguirá funcionando exactamente igual.

De hecho, Xcode es realmente inteligente aquí: si escribes para la plataforma, reconocerá que hay una matriz llamada plataformas, y ofrecerá completar automáticamente todo para la plataforma en plataformas: reconoce que las plataformas son plurales y sugiere el nombre singular para la variable de bucle. Cuando veas aparecer la sugerencia de Xcode, pulsa Retorno para seleccionarla.

En lugar de hacer un bucle sobre una matriz (o conjunto, o diccionario, ¡la sintaxis es la misma!), también puedes hacer un bucle sobre un rango fijo de números. Por ejemplo, podríamos imprimir la tabla de 5 veces del 1 al 12 de la siguiente manera:

```
for i in 1...12 {
    print("5 x \(i) is \(5 * i)")
}
```

Un par de cosas son nuevas allí, así que vamos a hacer una pausa y examinarlas:

- Usé la variable de bucle i, que es una convención de codificación común para "número con el que estás contando". Si estás contando un segundo número, usarías j, y si estás contando un tercio, usarías k, pero si estás contando un cuarto, tal vez deberías elegir mejores nombres de variables.
- La parte 1...12 es un rango, y significa "todos los números enteros entre 1 y 12, así como 1 y 12 en sí". Los rangos son su propio tipo de datos único en Swift.

Entonces, cuando ese bucle se ejecute por primera vez, seré 1, luego será 2, luego 3, etc., hasta 12, después de lo cual el bucle termina.

También puedes poner bucles dentro de bucles, llamados bucles anidados, de esta manera:

```
for i in 1...12 {
    print("The \(i) times table:")

    for j in 1...12 {
        print("  \(j) x \(i) is \(j * i)")
    }

    print()
}
```

Eso muestra un par de cosas nuevas, así que de nuevo hagamos una pausa y miremos más de cerca:

- Ahora hay un bucle anidado: contamos del 1 al 12, y para cada número dentro de allí contamos del 1 al 12 de nuevo.
- El uso de print() por sí solo, sin pasar texto ni valor, solo iniciará una nueva línea. Esto ayuda a romper nuestra salida para que se vea mejor en la pantalla.

Entonces, cuando ves x... y sabes que crea un rango que comienza en cualquier número x, y cuenta hasta e incluye cualquier número y.

Swift tiene un tipo de rango similar pero diferente que cuenta hasta pero excluyendo el número final: .. <. Esto se ve mejor en el código:

```
for i in 1...5 {
    print("Counting from 1 through 5: \(i)")
}

print()

for i in 1..<5 {
    print("Counting 1 up to 5: \(i)")
}
```

Cuando eso se ejecute, se imprimirá para los números 1, 2, 3, 4, 5 en el primer bucle, pero solo los números 1, 2, 3 y 4 en el segundo. Pronuncio 1...5 como "de uno a cinco", y 1..<5 como "de uno a cinco", y verás una redacción similar en otra parte de Swift.

Consejo: ..< es muy útil para trabajar con matrices, donde contamos desde 0 y a menudo queremos contar hasta, pero excluyendo el número de elementos en la matriz.

Antes de que terminemos con los bucles for, hay una cosa más que quiero mencionar: a veces quieres ejecutar algo de código un cierto número de veces usando un rango, pero en realidad no quieres la variable de bucle, no quieres la i o la j, porque no la usas.

En esta situación, puede reemplazar la variable de bucle con un guión bajo, como este:

```
var lyric = "Haters gonna"

for _ in 1...5 {
    lyric += " hate"
}

print(lyric)
```

(Sí, esa es una letra de Taylor Swift de Shake It Off, escrita en Swift).

¿Por qué Swift usa guiones bajos con bucles?

Si desea hacer un bucle sobre los elementos de una matriz, puede escribir código como este:

```
let names = ["Sterling", "Cyril", "Lana", "Ray", "Pam"]

for name in names {
    print("\(name) is a secret agent")
}
```

Cada vez que el bucle da la vuelta, Swift tomará un elemento de la matriz de nombres, lo pondrá en la constante de nombre y luego ejecutará el cuerpo de nuestro bucle, ese es el método print().

Sin embargo, a veces en realidad no necesitas el valor que se está leyendo actualmente, que es donde entra el guión bajo: Swift reconocerá que en realidad no necesitas la variable y no hará la constante temporal para ti.

Por supuesto, Swift realmente puede ver eso de todos modos: puede ver si estás usando o no el nombre dentro del bucle, por lo que puede hacer el mismo trabajo sin el guión bajo. Sin embargo, el uso de un guión bajo hace algo muy similar para nuestro cerebro: podemos mirar el código e inmediatamente ver que la variable de bucle no se está utilizando, sin importar cuántas líneas de código haya dentro del cuerpo del bucle.

Por lo tanto, si no usas una variable de bucle dentro del cuerpo, Swift te advertirá que la reescribas de esta manera:

```
let names = ["Sterling", "Cyril", "Lana", "Ray", "Pam"]

for _ in names {
    print("[CENSORED] is a secret agent!")
}
```

¿Por qué Swift tiene dos operadores de rango?

Cuando pensamos en rangos de valores, el inglés es bastante confuso. Si digo "dame las cifras de ventas hasta ayer", ¿significa eso incluir ayer o excluir ayer? Ambos son útiles por derecho propio, por lo que Swift nos da una forma de representarlos a ambos: ..< es el rango medio abierto que especifica "hasta pero excluyendo" y ... es el operador de rango cerrado que especifica "hasta e incluyendo".

Para facilitar la distinción al hablar, Swift utiliza regularmente un lenguaje muy específico: "1 a 5" significa 1, 2, 3 y 4, pero "1 a 5" significa 1, 2, 3, 4 y 5. Si recuerdas, las matrices de Swift comienzan en el índice 0, lo que significa que una matriz que contiene tres elementos tiene elementos en los índices 0, 1 y 2, un caso de uso perfecto para el operador de rango medio abierto.

Las cosas se ponen más interesantes cuando solo quieres una parte de un rango, como "cualquier cosa desde 0 hacia arriba" o "índice 5 hasta el final de la matriz". Verás, estos son bastante útiles en la programación, por lo que Swift los hace más fáciles de crear al permitirnos especificar solo una parte de un rango.

Por ejemplo, si tuviéramos una variedad de nombres como este:

```
let names = ["Piper", "Alex", "Suzanne", "Gloria"]
```

Podríamos leer un nombre individual como este:

```
print(names[0])
```

Con los rangos, también podemos imprimir un rango de valores como este:

```
print(names[1...3])
```

Sin embargo, eso conlleva un pequeño riesgo: si nuestra matriz no contudiera al menos cuatro elementos, entonces 1...3 fallaría. Afortunadamente, podemos usar un rango unilateral para decir "dame 1 hasta el final de la matriz", así:

```
print(names[1...])
```

Por lo tanto, los rangos son excelentes para contar a través de valores específicos, pero también son útiles para leer grupos de elementos de matrices.

Si quieres seguir aprendiendo más sobre los rangos en Swift, deberías echar un vistazo al artículo de Antoine van der Lee sobre el tema: https://www.avanderlee.com/swift/ranges-explained/

Cómo usar un bucle while para repetir el trabajo

Swift tiene un segundo tipo de bucle llamado while: proporcionarle una condición, y un bucle while ejecutará continuamente el cuerpo del bucle hasta que la condición sea falsa.

Aunque todavía verás bucles while de vez en cuando, no son tan comunes como para los bucles. Como resultado, quiero cubrirlos para que sepas que existen, pero no nos detengamos en ellos demasiado tiempo, ¿de acuerdo?

Aquí hay un bucle básico para empezar

```
var countdown = 10

while countdown > 0 {
    print("\(countdown)…")
    countdown -= 1
}

print("Blast off!")
```

Eso crea un contador de enteros a partir de 10, luego comienza un bucle while con la condición de cuenta atrás > 0. Por lo tanto, el cuerpo del bucle, imprimiendo el número y restando 1, se ejecutará continuamente hasta que la cuenta atrás sea igual o inferior a 0, momento en el que el bucle terminará y se imprimirá el mensaje final.

Mientras que los bucles son realmente útiles cuando simplemente no sabes cuántas veces se dará la vuelta al bucle. Para demostrar esto, quiero presentarte una funcionalidad realmente útil que tanto Int como Double tienen: random(in:). Dale un rango de números con los que trabajar, y te devolverá un Int o Double aleatorio en algún lugar dentro de ese rango.

Por ejemplo, esto crea un nuevo entero entre 1 y 1000

```
let id = Int.random(in: 1...1000)
```

Y esto crea un decimal aleatorio entre 0 y 1:

```
let amount = Double.random(in: 0...1)
```

Podemos usar esta funcionalidad con un bucle de tiempo para lanzar algunos dados virtuales de 20 lados una y otra vez, terminando el bucle solo cuando se lanza un 20, un éxito crítico para todos los jugadores de Dungeons & Dragons que hay por ahí.

Aquí está el código para que eso suceda:

```swift
// create an integer to store our roll
var roll = 0

// carry on looping until we reach 20
while roll != 20 {
    // roll a new dice and print what it was
    roll = Int.random(in: 1...20)
    print("I rolled a \(roll)")
}

// if we're here it means the loop ended – we got a 20!
print("Critical hit!")
```

Te encontrarás usando bucles for y while en tu propio código: los bucles for son más comunes cuando tienes una cantidad finita de datos por los que pasar, como un rango o una matriz, pero mientras que los bucles son realmente útiles cuando necesitas una condición personalizada.

¿Cuándo deberías usar un bucle while?

Swift nos da bucles para y while, y ambos se usan comúnmente. Sin embargo, cuando estás aprendiendo, puede parecer extraño tener dos formas comúnmente utilizadas de hacer bucles: ¿cuál deberías usar y por qué?

La principal diferencia es que los bucles se utilizan generalmente con secuencias finitas: hacemos un bucle a través de los números del 1 al 10, o a través de los elementos de una matriz, por ejemplo. Por otro lado, mientras que los bucles pueden hacer un bucle hasta que cualquier condición arbitraria se vuelva falsa, lo que les permite hasta que les digamos que se detengan.

Esto significa que podemos repetir el mismo código hasta…

- ...El usuario nos pide que nos detengamos
- ...Un servidor nos dice que nos detengamos
- ...Hemos encontrado la respuesta que estamos buscando
- ...Hemos generado suficientes datos

Y esos son solo un puñado de ejemplos. Piénsalo: si te pidiera que tiraras 10 dados e imprimieras sus resultados, podrías hacerlo con un simple bucle for contando del 1 al 10. Pero si te pedí que tiraras 10 dados e imprimieras los resultados, mientras que también lanzas automáticamente otros dados si los dados anteriores tuvieron el mismo resultado, entonces no sabes de antemano cuántos dados tendrás que tirar. Tal vez tengas suerte y solo necesites 10 rollos, pero tal vez consigas algunos rollos duplicados y necesites 15 rollos. O tal vez obtengas muchos rollos duplicados y necesites 30, ¿quién sabe?

Ahí es donde un bucle de tiempo es útil: podemos mantener el bucle hasta que estemos listos para salir.

Cómo omitir elementos de bucle con break y continuar

Swift nos da dos formas de omitir uno o más elementos en un bucle: continuar omite la iteración de bucle actual y break omite todas las iteraciones restantes. Como los bucles while, estos a veces se usan, pero en la práctica mucho menos de lo que podrías pensar.

Echemos un vistazo individualmente, empezando por continuar. Cuando estés haciendo un bucle sobre una matriz de datos, Swift sacará un elemento de la matriz y ejecutará el cuerpo del bucle usándolo. Si llama a continuar dentro de ese cuerpo de bucle, Swift dejará de ejecutar inmediatamente la iteración de bucle actual y saltará al siguiente elemento del bucle, donde continuará como de costumbre. Esto se usa comúnmente cerca del inicio de los bucles, donde eliminas las variables de bucle que no pasan una prueba de tu elección.

He aquí un ejemplo:

```
let filenames = ["me.jpg", "work.txt", "sophie.jpg", "logo.psd"]

for filename in filenames {
    if filename.hasSuffix(".jpg") == false {
        continue
    }

    print("Found picture: \(filename)")
}
```

Eso crea una matriz de cadenas de nombre de archivo, luego pasa por encima de cada una y comprueba para asegurarse de que tiene el sufijo ".jpg" - que es una imagen. continue se utiliza con todos los nombres de archivo que no fallan en esa prueba, de modo que se omite el resto del cuerpo del bucle.

En cuanto al descanso, eso sale de un bucle inmediatamente y omite todas las iteraciones restantes. Para demostrar esto, podríamos escribir algún código para calcular 10 múltiplos comunes para dos números:

```
let number1 = 4
let number2 = 14
var multiples = [Int]()

for i in 1...100_000 {
    if i.isMultiple(of: number1) && i.isMultiple(of: number2) {
        multiples.append(i)

        if multiples.count == 10 {
            break
        }
    }
}

print(multiples)
```

Eso hace mucho:

1. Crea dos constantes para contener dos números.
2. Crea una variable de matriz entera que almacenará múltiplos comunes de nuestros dos números.
3. Cuenta de 1 a 100.000, asignando cada variable de bucle a i.
4. Si i es un múltiplo del primer y segundo número, añádelo a la matriz entera.
5. Una vez que lleguemos a 10 múltiplos, llame a la interrupción para salir del bucle.
6. Imprima la matriz resultante.

Por lo tanto, use continuar cuando desee omitir el resto de la iteración de bucle actual, y use break cuando desee omitir todas las iteraciones de bucle restantes.

¿Por qué querrías salir de un bucle?

La palabra clave break de Swift nos permite salir de un bucle inmediatamente, independientemente del tipo de bucle del que estemos hablando. La mayor parte del tiempo no necesitarás esto, porque estás revisando los elementos de una matriz y quieres procesarlos todos, o porque estás contando del 1 al 10 y quieres manejar todos esos valores.

Sin embargo, a veces quieres terminar tu bucle prematuramente. Por ejemplo, si tenías una serie de puntuaciones y quieres averiguar cuántas de ellas logró el jugador sin obtener un 0, podrías escribir esto:

```
let scores = [1, 8, 4, 3, 0, 5, 2]
var count = 0

for score in scores {
    if score == 0 {
        break
    }

    count += 1
}

print("You had \(count) scores before you got 0.")
```

Sin descanso, tendríamos que seguir revisando las puntuaciones incluso después de encontrar el primer 0, lo cual es un desperdicio.

¿Por qué Swift tiene declaraciones etiquetadas?

Las declaraciones etiquetadas de Swift nos permiten nombrar ciertas partes de nuestro código, y se usa más comúnmente para romper los bucles anidados.

Para demostrarlos, echemos un vistazo a un ejemplo en el que estamos tratando de averiguar la combinación correcta de movimientos para desbloquear una caja fuerte. Podríamos comenzar definiendo una serie de todos los movimientos posibles:

```
let options = ["up", "down", "left", "right"]
```

Para fines de prueba, esta es la combinación secreta que estamos tratando de adivinar:

```
let secretCombination = ["up", "up", "right"]
```

Para encontrar esa combinación, necesitamos hacer matrices que contengan todos los movimientos posibles de tres colores:

- Arriba, arriba, arriba
- Arriba, arriba, abajo
- Arriba, arriba, a la izquierda
- Arriba, arriba, correcto
- Arriba, abajo, izquierda
- Arriba, abajo, a la derecha

...Entiendes la idea.

Para que eso suceda, podemos escribir tres bucles, uno anidado dentro del otro, así:

```
for option1 in options {
    for option2 in options {
        for option3 in options {
            print("In loop")
            let attempt = [option1, option2, option3]

            if attempt == secretCombination {
                print("The combination is \(attempt)!")
            }
        }
    }
}
```

Eso repasa los mismos elementos varias veces para crear una matriz de intento, e imprime un mensaje si su intento coincide con la combinación secreta.

Pero ese código tiene un problema: tan pronto como encontramos la combinación, terminamos con los bucles, así que ¿por qué siguen funcionando? Lo que realmente queremos decir es "tan pronto como se encuentre la combinación, salga de todos los bucles a la vez", y ahí es donde entran en trada las declaraciones etiquetadas. Nos dejaron escribir esto:

```
outerLoop: for option1 in options {
    for option2 in options {
        for option3 in options {
            print("In loop")
            let attempt = [option1, option2, option3]

            if attempt == secretCombination {
                print("The combination is \(attempt)!")
                break outerLoop
            }
        }
    }
}
```

Con ese pequeño cambio, esos tres bucles dejan de funcionar tan pronto como se encuentra la combinación. En este caso trivial, es poco probable que haya una diferencia en el rendimiento, pero ¿qué pasaría si sus artículos tuvieran cientos o incluso miles de artículos? Guardar el trabajo como este es una buena idea, y vale la pena recordarlo por tu propio código.

Cuándo usar el descanso y cuándo usar continuar

A veces, a los estudiantes de Swift les resulta difícil entender cuándo la palabra clave break es correcta y cuándo la palabra clave continue es correcta, porque ambos alteran el flujo de nuestros bucles.

Cuando usamos continuar, estamos diciendo "He terminado con la ejecución actual de este bucle" - Swift se saltará el resto del cuerpo del bucle e irá al siguiente elemento del bucle. Pero cuando decimos descanso, estamos diciendo "He terminado con este bucle por completo, así que sal por completo". Eso significa que Swift omitirá el resto del bucle del cuerpo, pero también omitirá cualquier otro elemento del bucle que aún esté por venir.

Al igual que el descanso, puedes usar continuar con declaraciones etiquetadas si quieres, ¡pero honestamente no recuerdo haberlo visto hecho!

Resumen: Condiciones y bucles

Hemos cubierto mucho sobre las condiciones y los bucles en los capítulos anteriores, así que recapitulemos:

- Utilizamos sentencias if para comprobar que una condición es verdadera. Puedes pasar en cualquier condición que quieras, pero en última instancia debe reducirse a un booleano.
- Si lo desea, puede agregar otro bloque y/o varios otros bloques para comprobar otras condiciones. Swift los ejecuta en orden.
- Puede combinar condiciones usando ||, lo que significa que toda la condición es verdadera si cualquiera de las subcondiciones es verdadera, o &&, lo que significa que toda la condición es verdadera si ambas subcondiciones son verdaderas.
- Si estás repitiendo mucho los mismos tipos de comprobación, puedes usar una instrucción switch en su lugar. Estos siempre deben ser exhaustivos, lo que podría significar agregar un caso predeterminado.
- Si uno de sus casos de cambio utiliza fallthrough, significa que Swift ejecutará el siguiente caso después. Esto no se usa comúnmente.
- El operador condicional ternario nos permite comprobar WTF: Qué, Verdadero, Falso. Aunque es un poco difícil de leer al principio, verás que esto se usa mucho en SwiftUI.
- Para los bucles, vamos a hacer un bucle sobre matrices, conjuntos, diccionarios y rangos. Puede asignar elementos a una variable de bucle y usarla dentro del bucle, o puede usar el guión bajo, _, para ignorar la variable de bucle.
- Mientras que los bucles nos permiten crear bucles personalizados que continuarán ejecutándose hasta que una condición se vuelva falsa.
- Podemos omitir algunos o todos los elementos de bucle usando continuar o romper, respectivamente.

Esa es otra gran cantidad de material nuevo, pero con condiciones y bucles que ahora sabes lo suficiente como para construir un software realmente útil, ¡pruébalo!

Punto de control 3

Ahora que puedes usar condiciones y bucles, me gustaría que probaras un problema clásico de informática. No es difícil de entender, ¡pero podría llevarte un poco de tiempo resolverlo dependiendo de tu experiencia previa!

El problema se llama fizz buzz, y se ha utilizado en entrevistas de trabajo, pruebas de ingreso a la universidad y más desde que puedo recordar. Tu objetivo es hacer un bucle del 1 al 100, y para cada número:

- Si es un múltiplo de 3, imprime "Fizz"
- Si es un múltiplo de 5, imprime "Buzz"
- Si es un múltiplo de 3 y 5, imprime "FizzBuzz"
- De lo contrario, solo imprime el número.

Por lo tanto, aquí hay algunos valores de ejemplo que debe tener cuando se ejecute su código:

- 1 debería imprimir "1"
- 2 deberían imprimir "2"
- 3 deberían imprimir "Fizz"
- 4 debería imprimir "4"
- 5 deberían imprimir "Buzz"
- 6 deberían imprimir "Fizz"
- 7 debería imprimir "7"
- ...
- 15 debería imprimir "FizzBuzz"
- ...
- 100 debería imprimir "Buzz"

Antes de empezar: este problema parece extremadamente simple, pero muchos, muchos desarrolladores luchan por resolverlo. Lo he visto suceder personalmente, así que no te estreses por ello, solo tratar de resolver el problema ya te enseña al respecto.

Ya sabes todo lo que necesitas para resolver ese problema, pero si quieres algunas pistas, añadiré algunas a continuación.

Por favor, adelante e intenta construir el patio de recreo ahora.

¿Sigues aquí? Vale, aquí hay algunas pistas:

Puedes comprobar si un número es un múltiplo de otro usando .isMultiple(of:). Por ejemplo, i.isMultiple(de: 3).

En este caso, primero debes comprobar 3 y 5 porque es el más específico, luego 3, luego 5, y finalmente tener un bloque de otro para manejar todos los demás números.

Puede usar && para comprobar si hay números que son múltiplos de 3 y 5, o tener una declaración if anidada.

Tienes que contar del 1 al 100, así que usa... en lugar de... <.