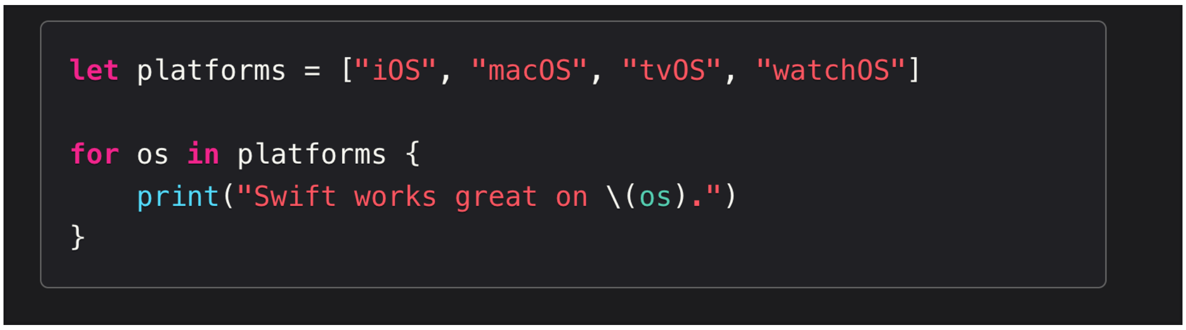
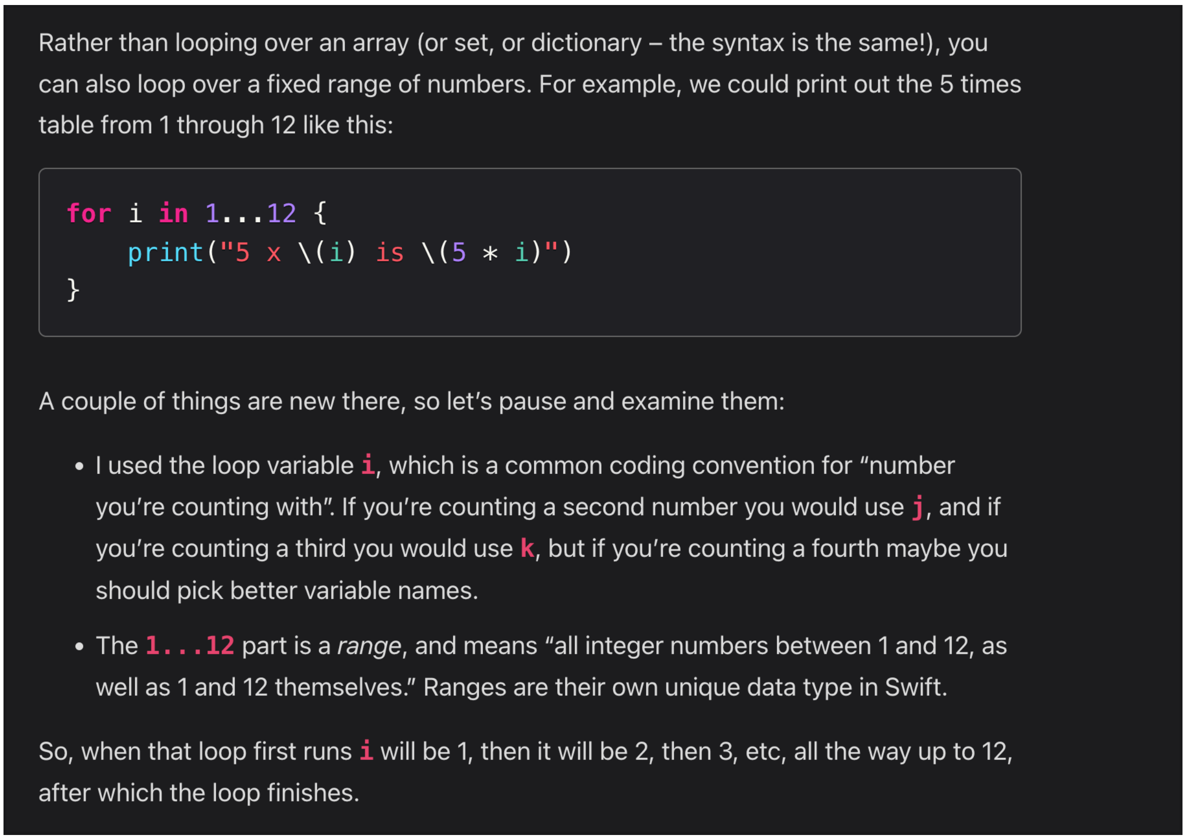
Day 6



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

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



Rather than looping over an array (or set, or dictionary – the syntax is the same!), you can also loop over a fixed range of numbers. For example, we could print out the 5 times table from 1 through 12 like this:

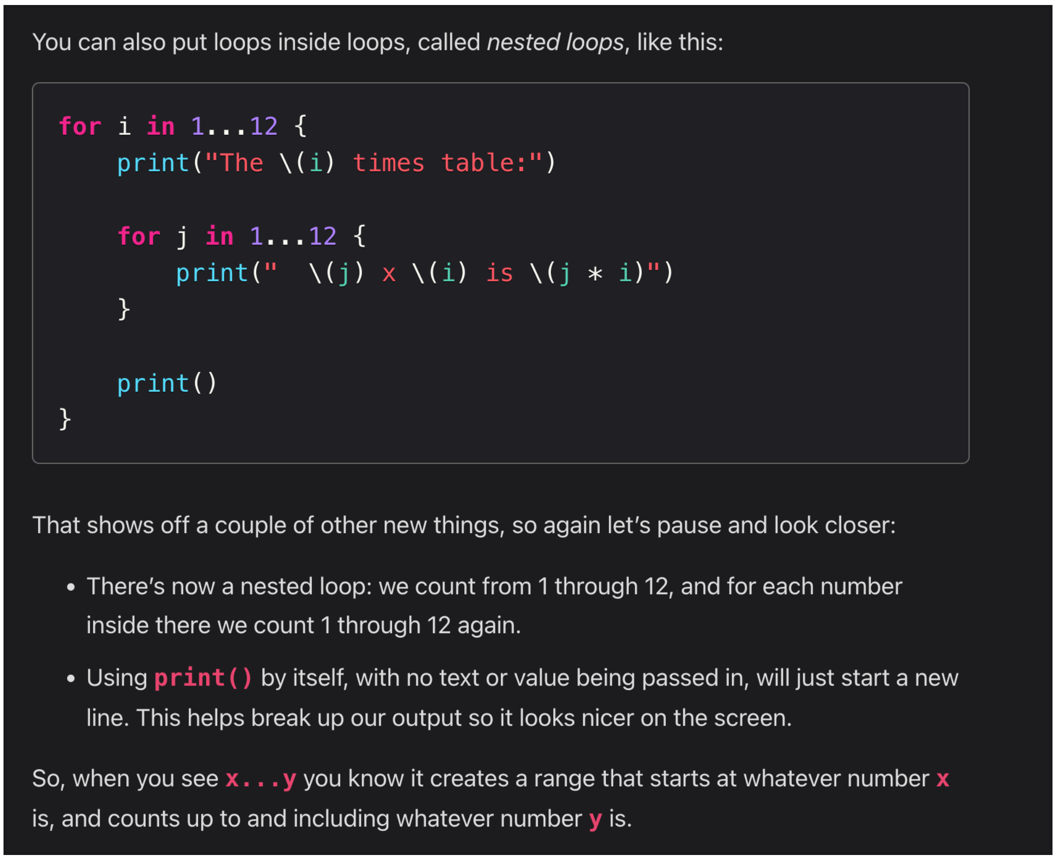
```swift
for i in 1...12 {
    print("5 x \(i) is \(5 * i)")
}
```

A couple of things are new there, so let's pause and examine them:

- I used the loop variable `i`, which is a common coding convention for "number you're counting with". If you're counting a second number you would use `j`, and if you're counting a third you would use `k`, but if you're counting a fourth maybe you should pick better variable names.
- The `1...12` part is a *range*, and means "all integer numbers between 1 and 12, as well as 1 and 12 themselves." Ranges are their own unique data type in Swift.

So, when that loop first runs `i` will be 1, then it will be 2, then 3, etc, all the way up to 12, after which the loop finishes.

Range in swift is same as for(i=0 ,i<= k ,I++) in C++ .



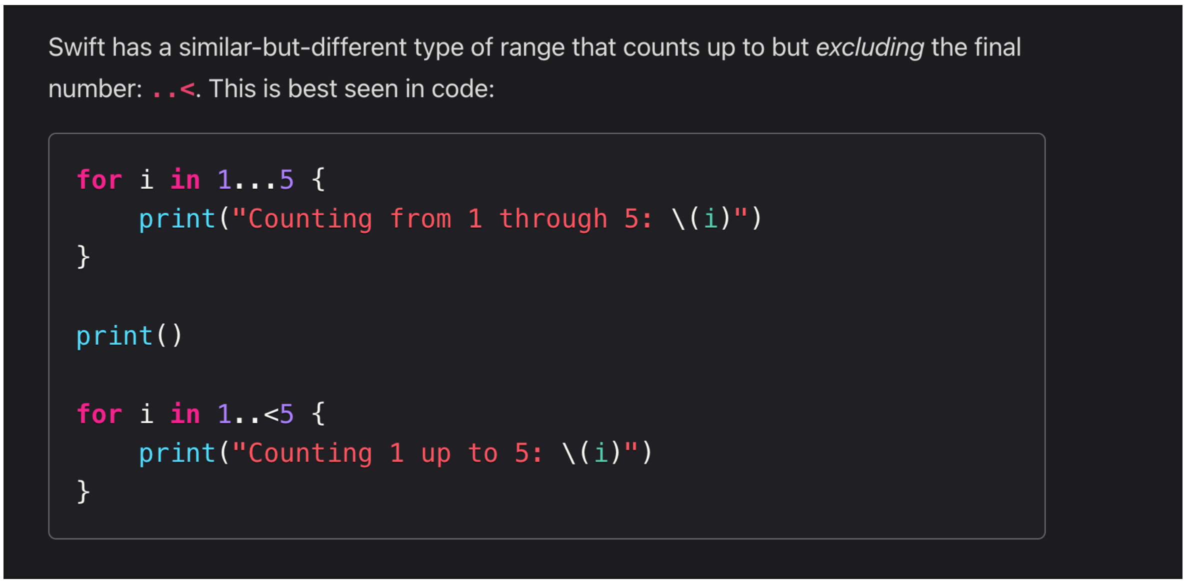
You can also put loops inside loops, called *nested loops*, like this:

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

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

    print()
}
```

That shows off a couple of other new things, so again let's pause and look closer:

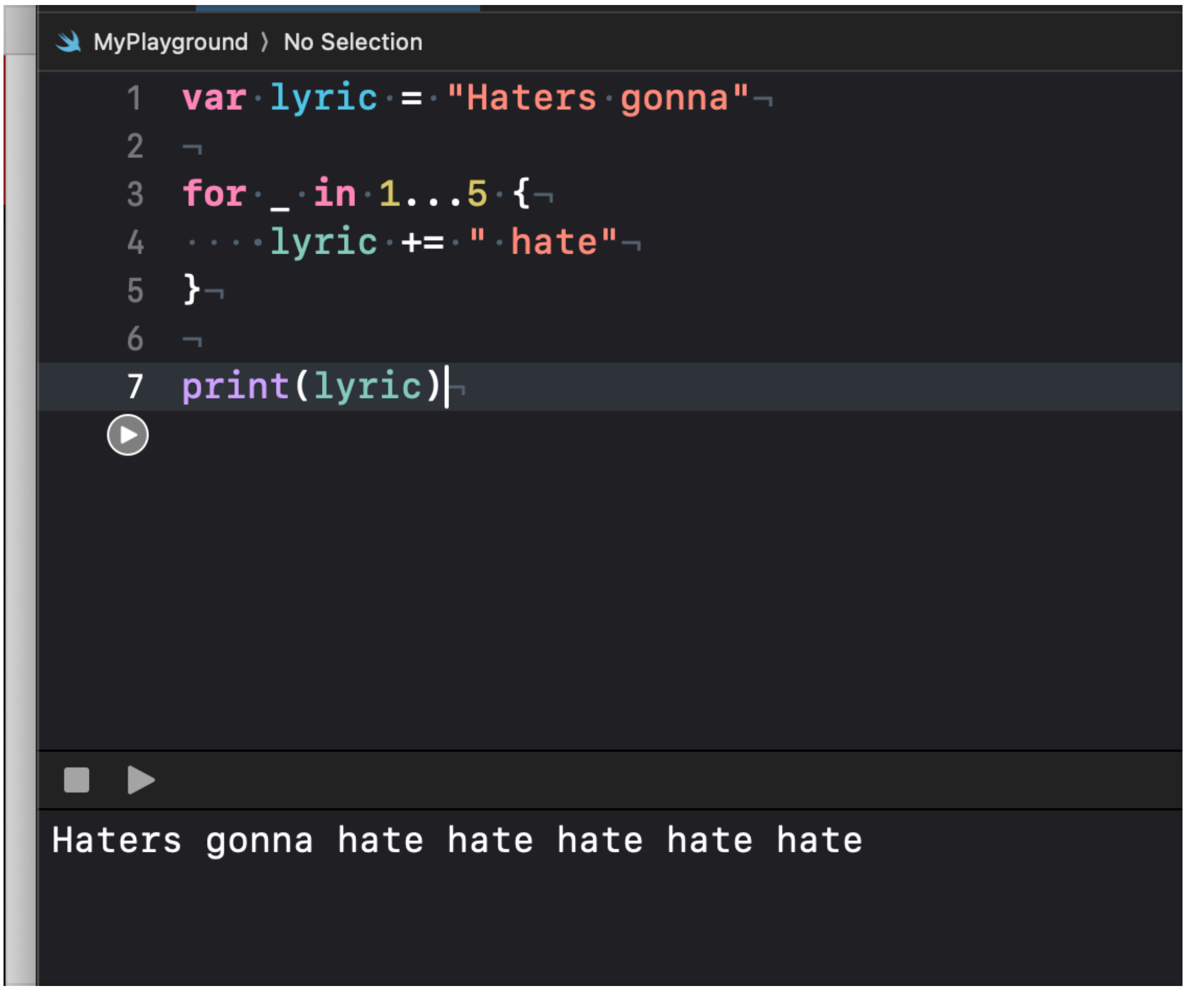
- There's now a nested loop: we count from 1 through 12, and for each number inside there we count 1 through 12 again.
- Using `print()` by itself, with no text or value being passed in, will just start a new line. This helps break up our output so it looks nicer on the screen.

So, when you see `x...y` you know it creates a range that starts at whatever number `x` is, and counts up to and including whatever number `y` is.



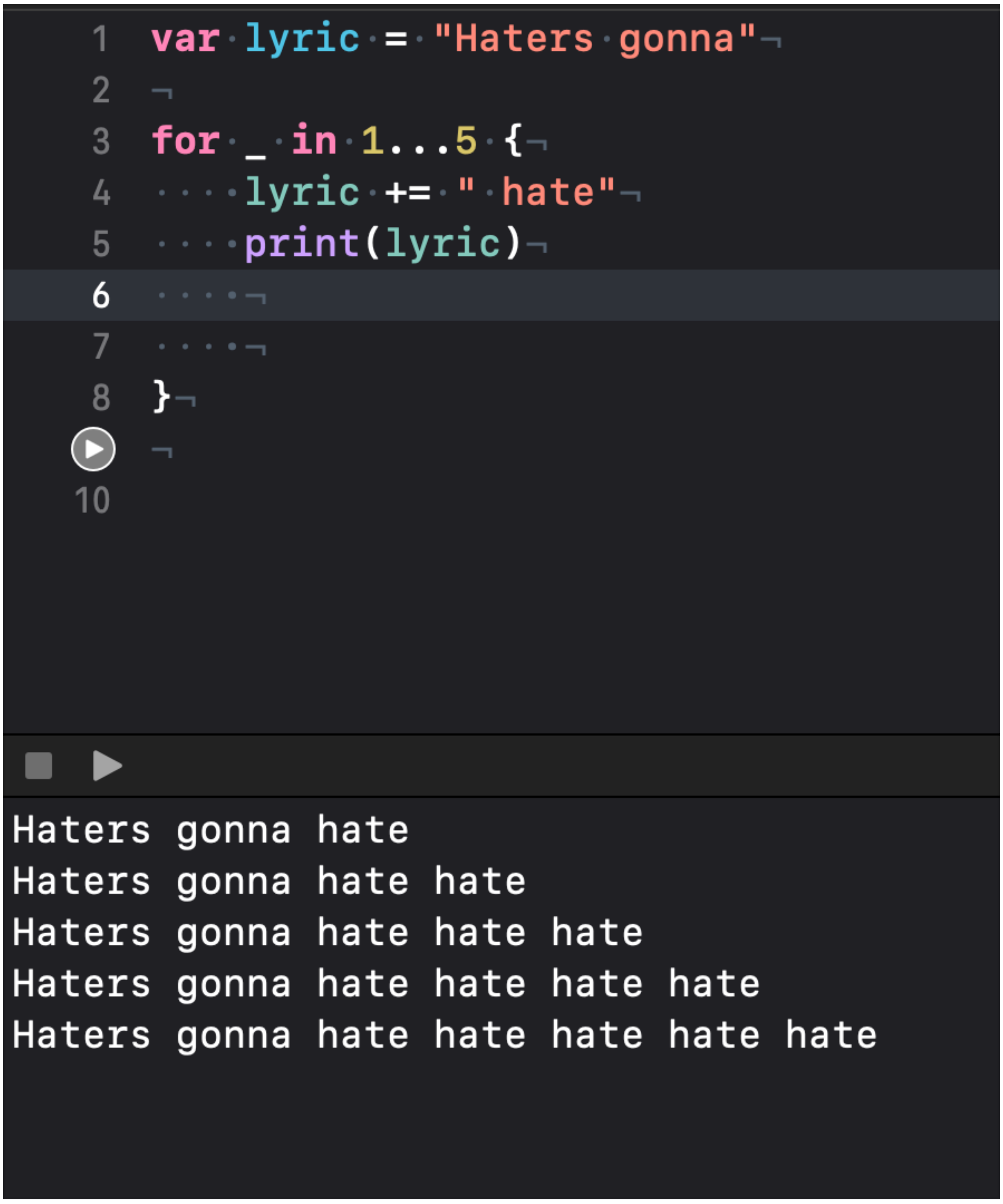
Swift has a similar-but-different type of range that counts up to but *excluding* the final number: `..<`. This is best seen in code:

```
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)")
}
```



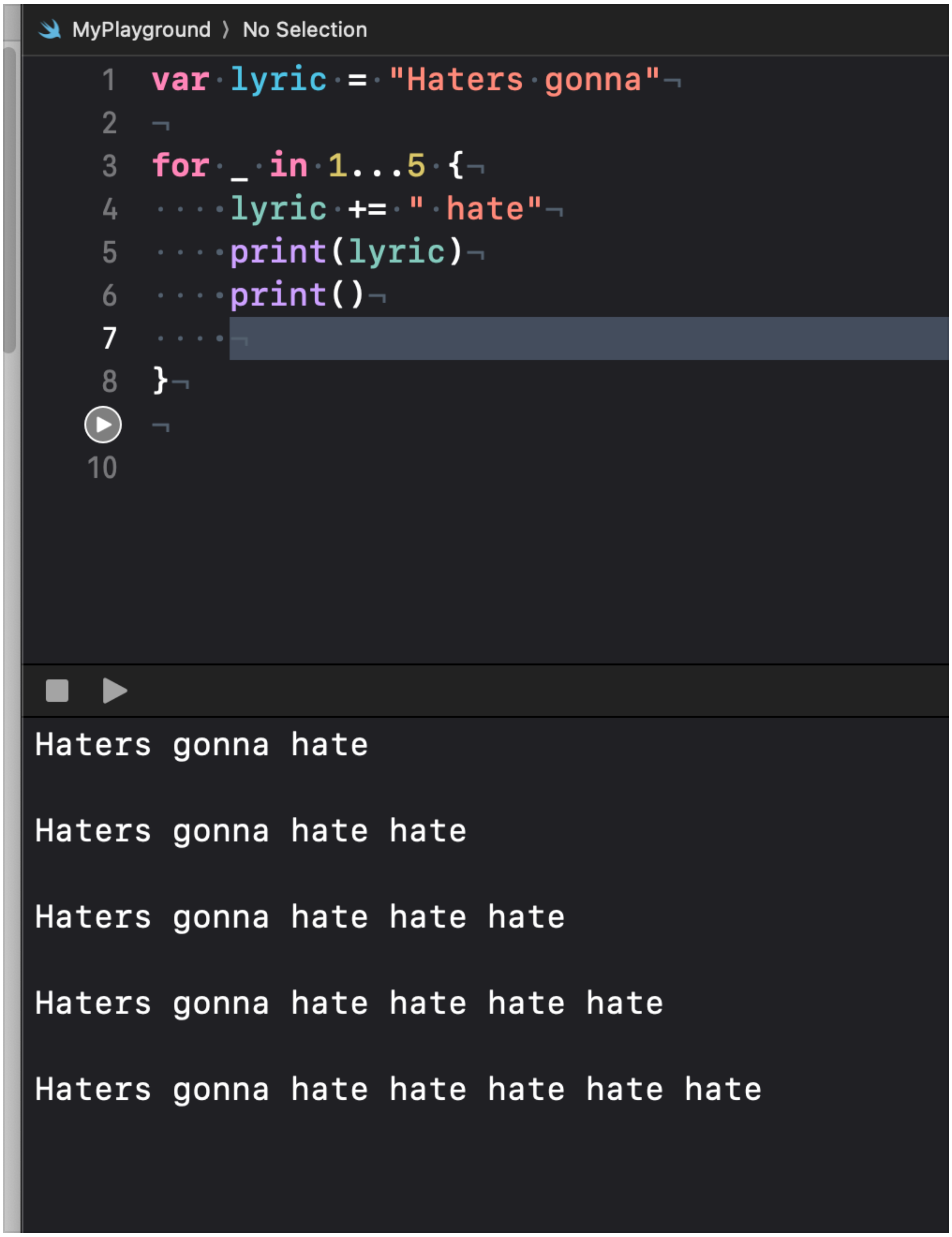
```swift
var lyric = "Haters gonna"

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

print(lyric)
```

```
Haters gonna hate hate hate hate hate
```



```swift
var lyric = "Haters gonna"

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

```

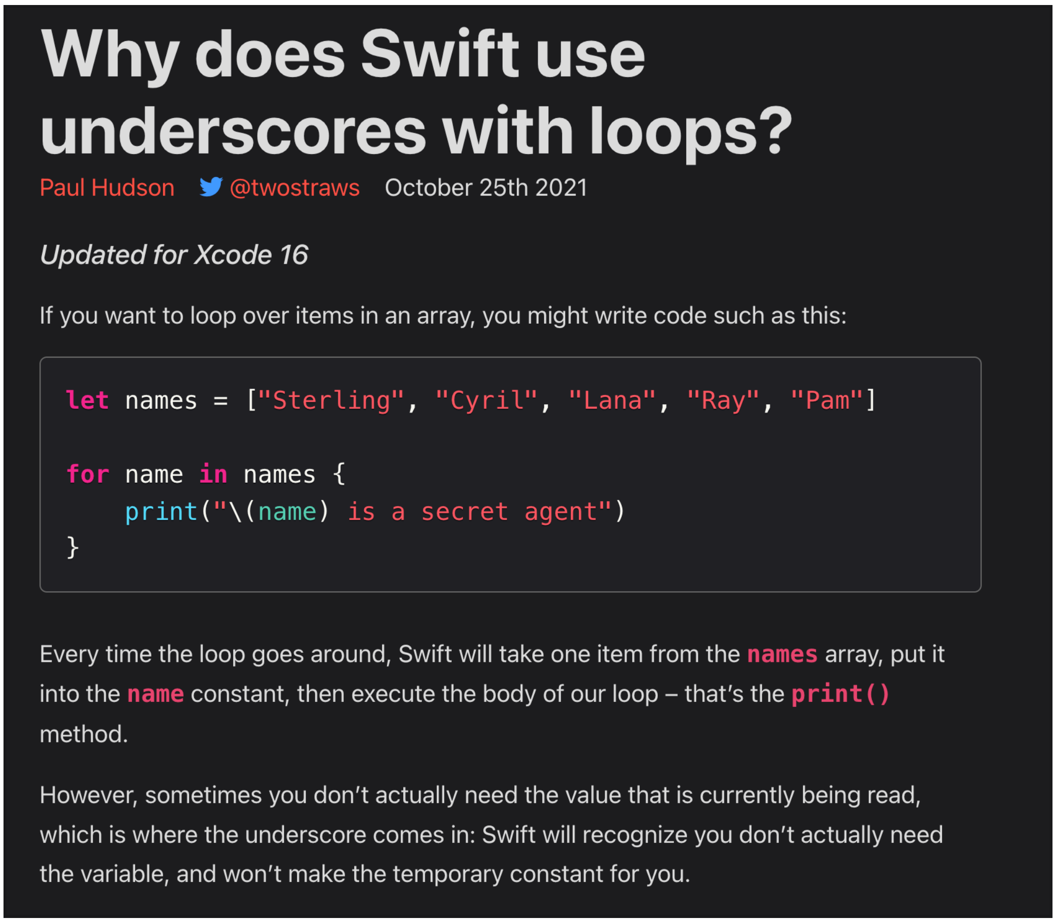
```
Haters gonna hate
Haters gonna hate hate
Haters gonna hate hate hate
Haters gonna hate hate hate hate
Haters gonna hate hate hate hate hate
```


MyPlayground 〉 No Selection

```swift
var lyric = "Haters gonna"

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

```

```
Haters gonna hate

Haters gonna hate hate

Haters gonna hate hate hate

Haters gonna hate hate hate hate

Haters gonna hate hate hate hate hate
```

See what does print() does here



Why does Swift use underscores with loops?

Paul Hudson @twostraws October 25th 2021

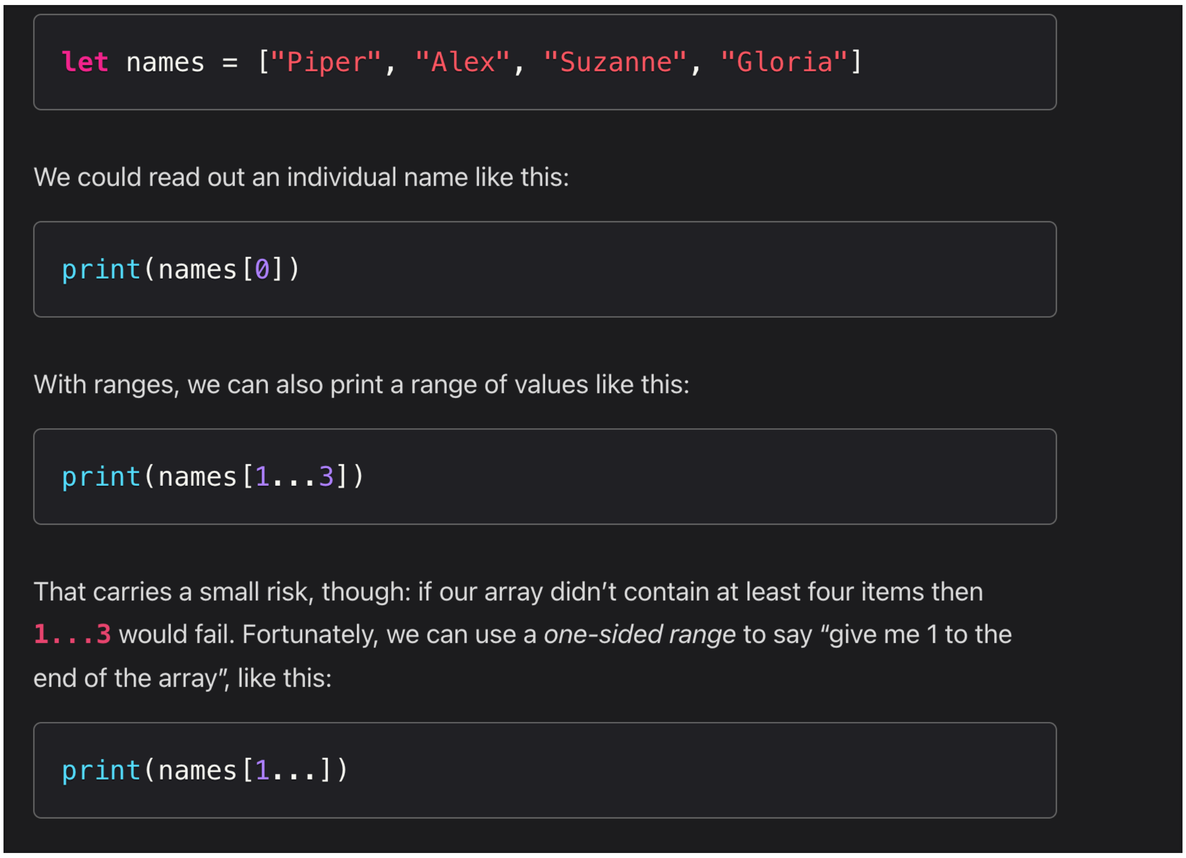
Updated for Xcode 16

If you want to loop over items in an array, you might write code such as this:

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

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

Every time the loop goes around, Swift will take one item from the `names` array, put it into the `name` constant, then execute the body of our loop – that's the `print()` method.

However, sometimes you don't actually need the value that is currently being read, which is where the underscore comes in: Swift will recognize you don't actually need the variable, and won't make the temporary constant for you.



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

We could read out an individual name like this:

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

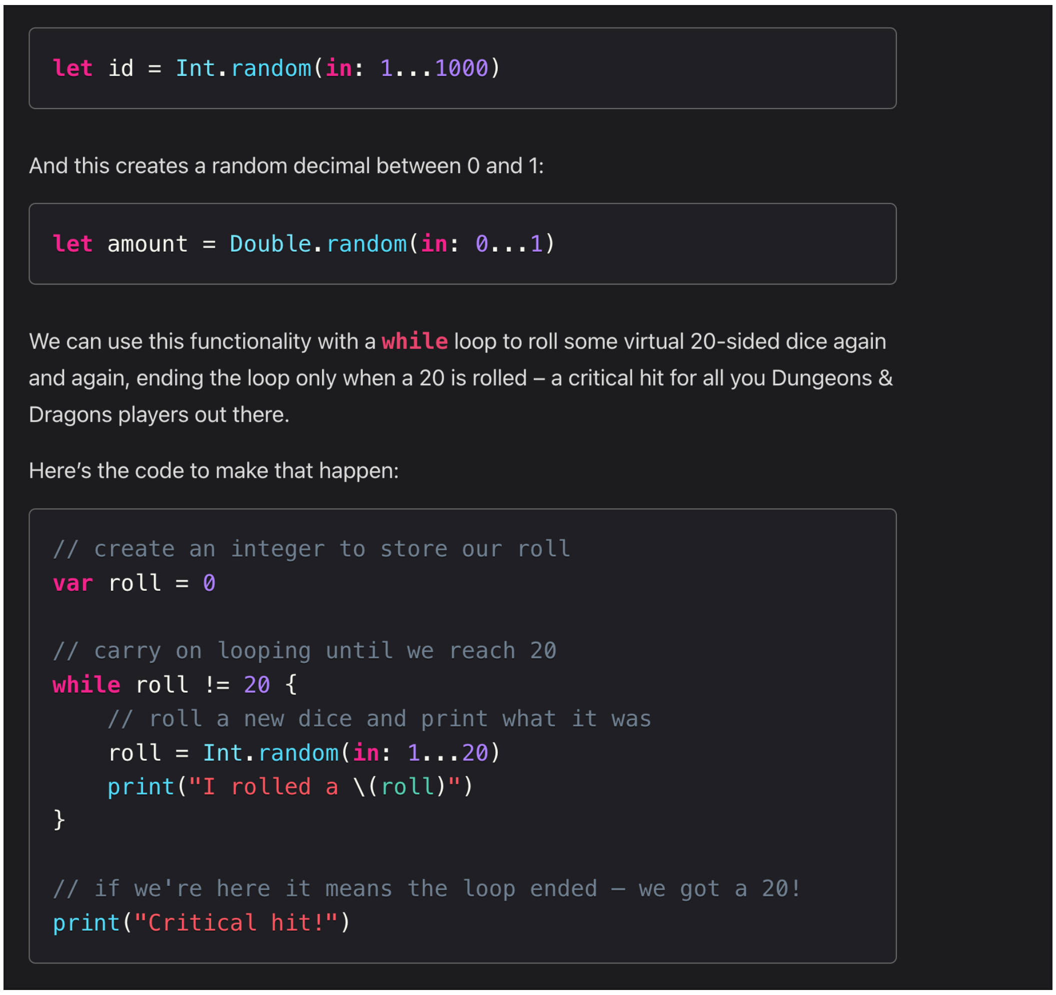
With ranges, we can also print a range of values like this:

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

That carries a small risk, though: if our array didn't contain at least four items then `1...3` would fail. Fortunately, we can use a *one-sided range* to say "give me 1 to the end of the array", like this:

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

While Loop —>



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

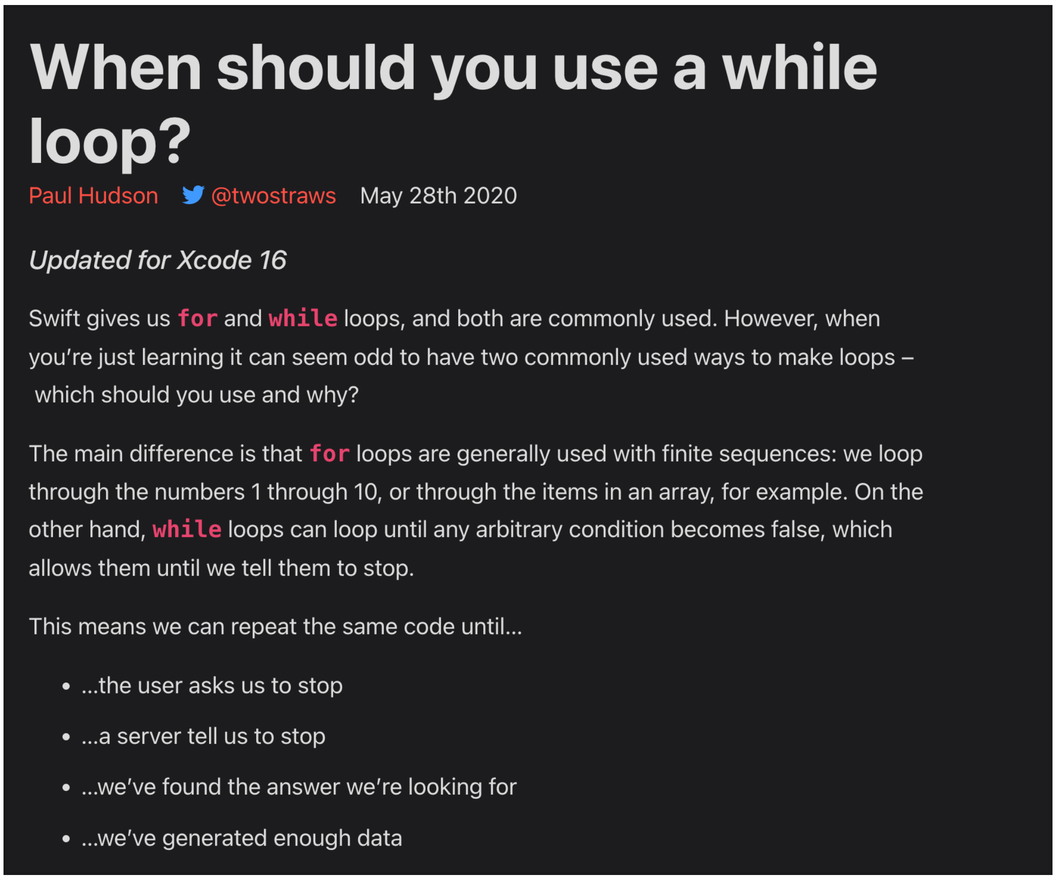
And this creates a random decimal between 0 and 1:

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

We can use this functionality with a `while` loop to roll some virtual 20-sided dice again and again, ending the loop only when a 20 is rolled – a critical hit for all you Dungeons & Dragons players out there.

Here’s the code to make that happen:

```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!")
```



When should you use a while loop?

Paul Hudson @twostraws May 28th 2020

Updated for Xcode 16

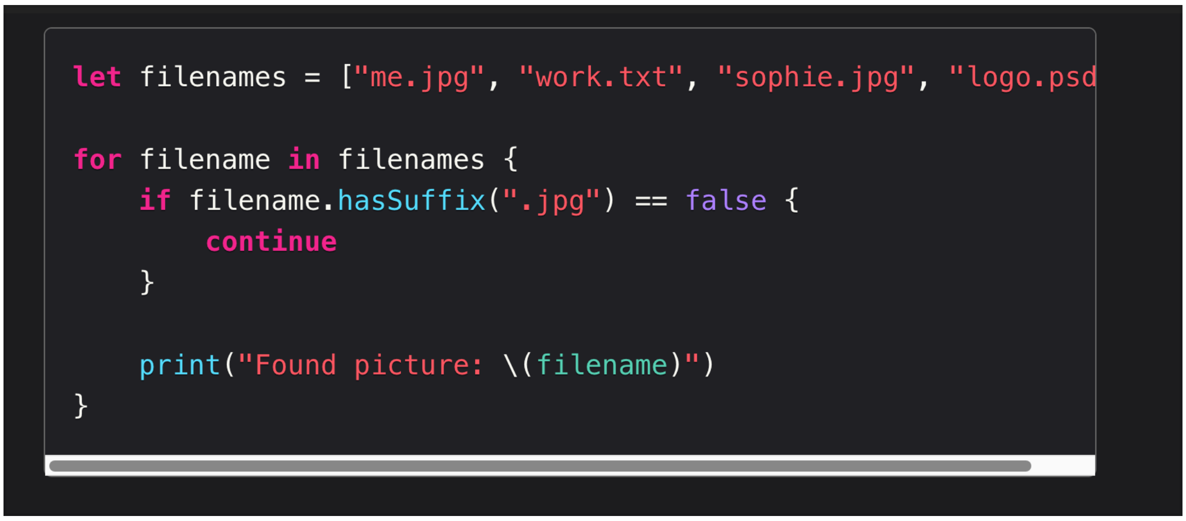
Swift gives us `for` and `while` loops, and both are commonly used. However, when you're just learning it can seem odd to have two commonly used ways to make loops – which should you use and why?

The main difference is that `for` loops are generally used with finite sequences: we loop through the numbers 1 through 10, or through the items in an array, for example. On the other hand, `while` loops can loop until any arbitrary condition becomes false, which allows them until we tell them to stop.

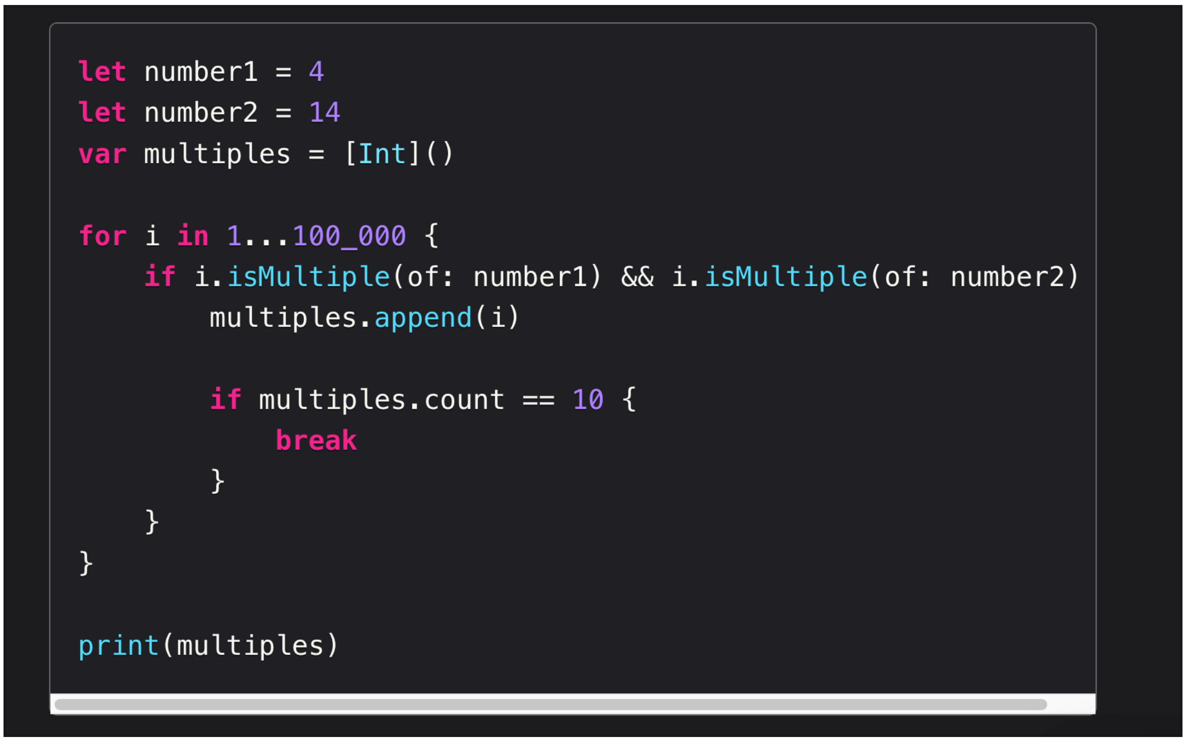
This means we can repeat the same code until…

- …the user asks us to stop
- …a server tell us to stop
- …we've found the answer we're looking for
- …we've generated enough data

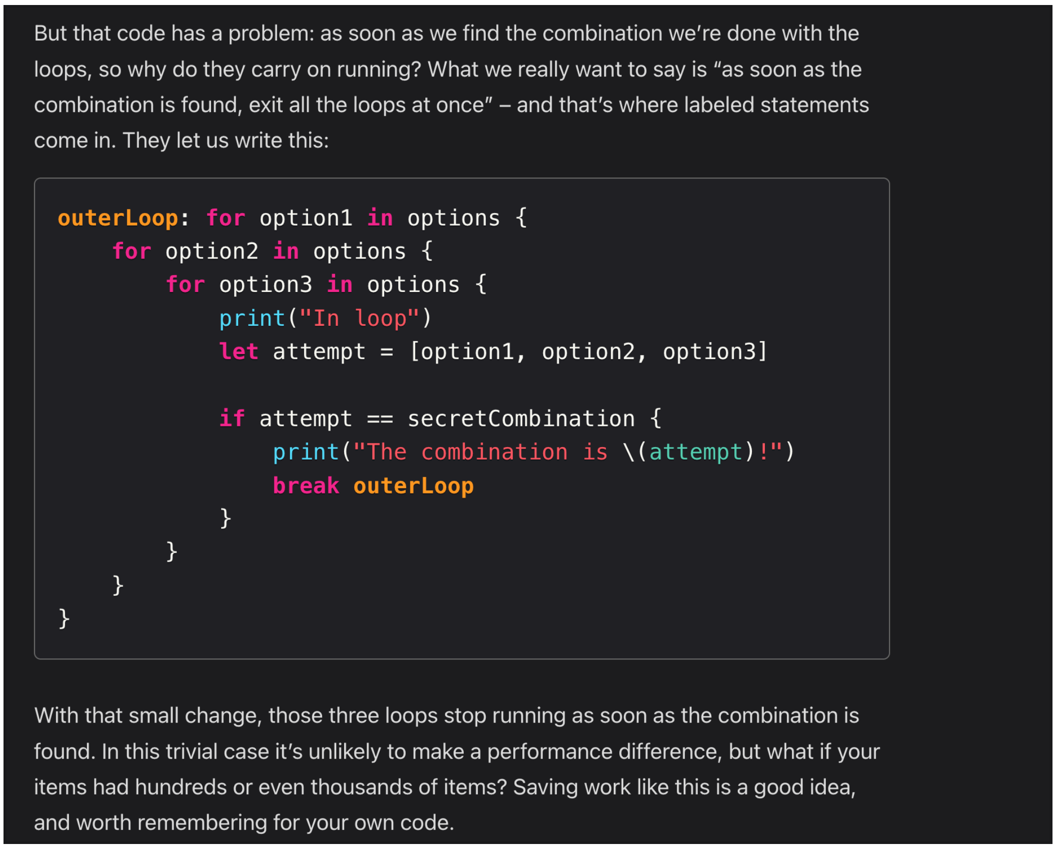
Continue and Break. —>



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

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

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



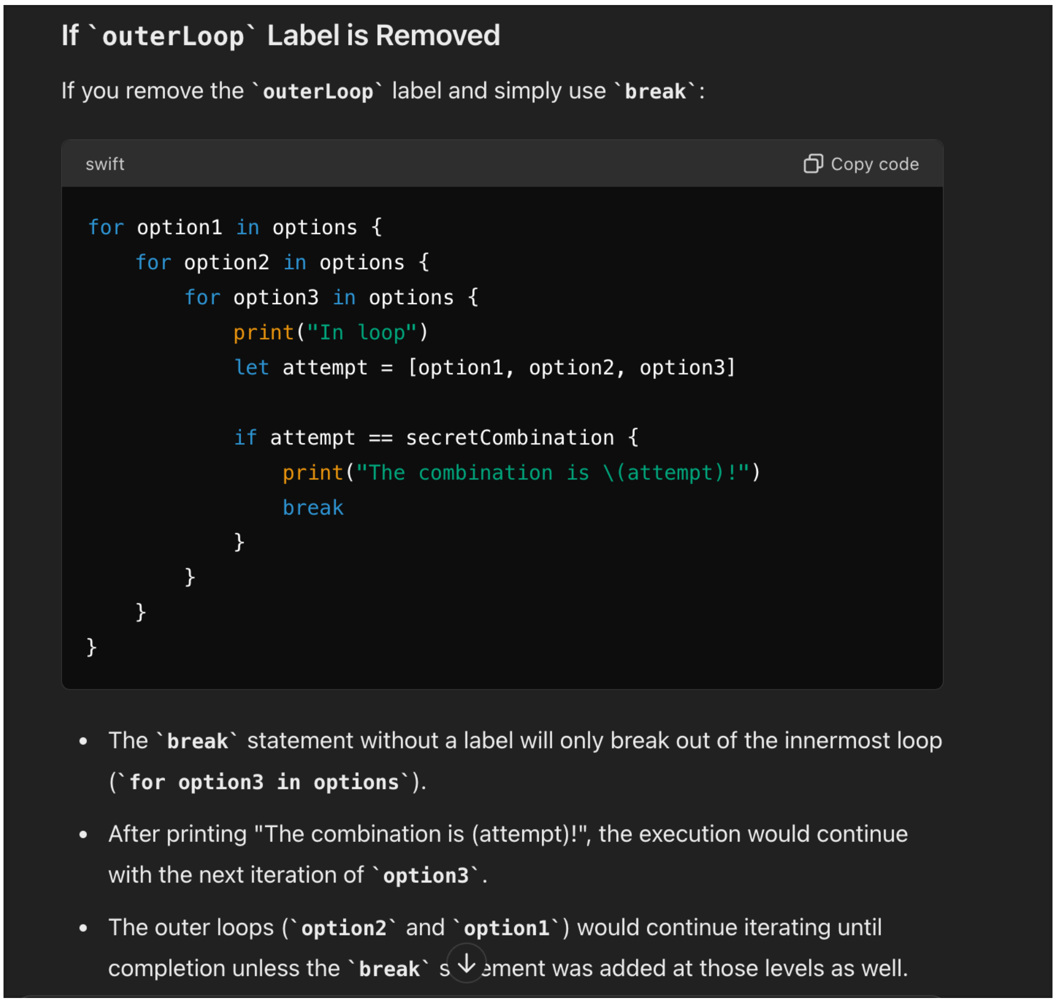
```swift
let number1 = 4
let number2 = 14
var multiples = [Int]()

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)
```



But that code has a problem: as soon as we find the combination we're done with the loops, so why do they carry on running? What we really want to say is "as soon as the combination is found, exit all the loops at once" – and that's where labeled statements come in. They let us write this:

```
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
            }
        }
    }
}
```

With that small change, those three loops stop running as soon as the combination is found. In this trivial case it's unlikely to make a performance difference, but what if your items had hundreds or even thousands of items? Saving work like this is a good idea, and worth remembering for your own code.

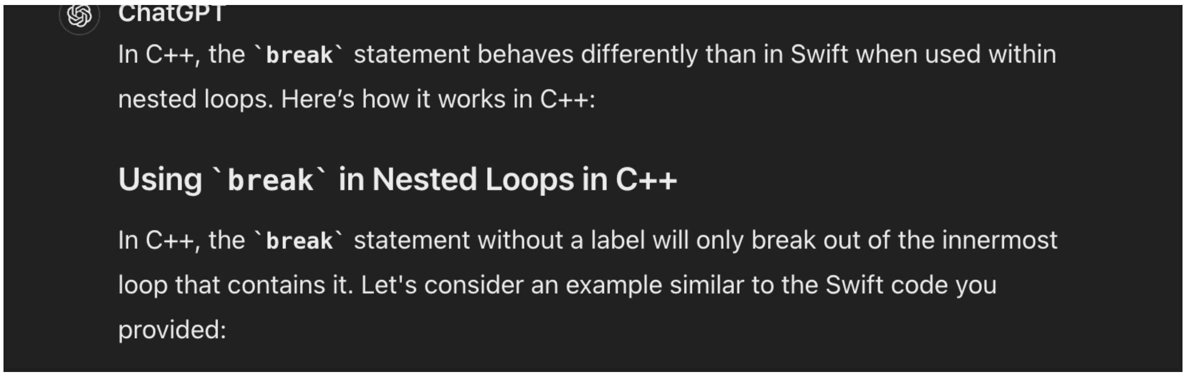


If `outerLoop` Label is Removed

If you remove the `outerLoop` label and simply use `break`:

```swift
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
            }
        }
    }
}
```

- The `break` statement without a label will only break out of the innermost loop (`for option3 in options`).
- After printing "The combination is (attempt)!", the execution would continue with the next iteration of `option3`.
- The outer loops (`option2` and `option1`) would continue iterating until completion unless the `break` s ement was added at those levels as well.



In C++, the `break` statement behaves differently than in Swift when used within nested loops. Here's how it works in C++:

Using `break` in Nested Loops in C++

In C++, the `break` statement without a label will only break out of the innermost loop that contains it. Let's consider an example similar to the Swift code you provided: