## Lesson 18. Repeating tasks while conditions hold

After reading lesson 18, you'll be able to

- Understand the syntax of another way to write a loop in a program
- Repeat actions while a certain condition is true
- Exit out of loops early
- Skip statements in a loop

In the previous lessons, you assumed that you knew the number of times you wanted to repeat a block of code. But suppose, for example, that you're playing a game with your friend. Your friend is trying to guess a number that you have in mind. Do you know in advance how many times your friend will guess? Not really. You want to keep asking them to try again until they get it right. In this game, you don't know how many times you want to repeat the task. Because you don't know, you can't use a `for` loop. Python has another type of loop that's useful in these kinds of situations: a `while` loop.

**Consider this**

Using only the information given in the following scenarios, do you know the maximum number of times you want to repeat the task?

- You have five TV channels and you cycle though using the Up button until you've checked out what's on every channel.
- Eat a cookie until there are no more cookies in the box.
- Say "punch buggy" every time you see a VW Beetle.
- Click Next on your jogging song playlist until you've sampled 20 songs.

Answer:

- Yes
- Yes
- No
- Yes

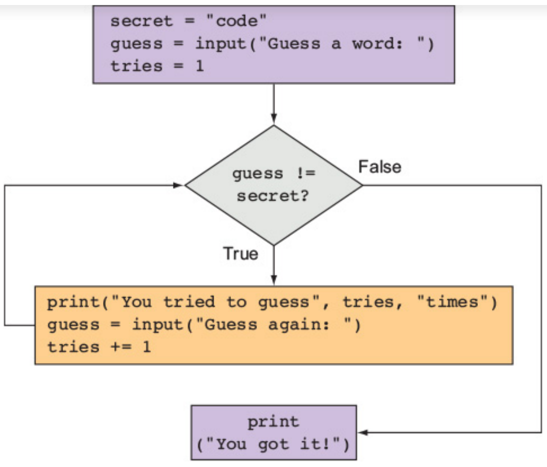
### 18.1. LOOPING WHILE A CONDITION IS TRUE

If you have a task that must be repeated an uncertain number of times, a `for` loop won't be appropriate because it won't work.

#### 18.1.1. Looping to make a guess

Start with a guessing game. You think of a secret word and ask a player to guess your word. Every time the player makes a guess, tell them whether they're right. If they're wrong, ask again. Keep track of the number of guesses a player makes until they get it right. Figure 18.1 shows a flowchart of this game.

**Figure 18.1. Flowchart of the guessing game. A user guesses a word. The guessing loop is represented by the gray diamond, which checks whether the user guess is equal to the secret word. If it is, the game finishes. If it isn't equal, you tell the player that they're wrong, ask them to guess again, and add 1 to the number of times that they made a guess.**


secret = "code"
guess = input("Guess a word: ")
tries = 1
guess != secret?
False
True
print("You tried to guess", tries, "times")
guess = input("Guess again: ")
tries += 1
print ("You got it!")

Listing 18.1 shows an implementation of the game in code. The user is trying to guess a secret word chosen by the programmer. The user is first prompted to enter a word. The first time you reach the `while` loop, you compare the user guess with the secret word. If the guess isn't correct, you enter the `while` loop code block, consisting of three lines. You first print the number of times the user guessed so far. Then you ask the user for another guess; notice that the user's guess is assigned to the variable `guess`, which is used in the `while` loop's condition to check the guess against the secret word. Lastly, you increment the number of tries to keep an accurate count of the number of times the user tried to guess the word.

After these three lines are executed, you check the condition of the `while` loop again, this time with the updated guess. If the user continues to guess the secret word incorrectly, the program doesn't go past the `while` loop and its code block. When the user gets the secret word correct, the `while` loop condition becomes false, and the `while` loop code block isn't executed. Instead, you skip the `while` code block, move to the statement immediately following the `while` loop and its code block, and print a congratulatory message. In this game, you must use a `while` loop because you don't know the number of wrong guesses the user will give.

**Listing 18.1. A `while` loop example of a guessing game**

```
secret = "code"
guess = input("Guess a word: ")
tries = 1
while guess != secret:
    print("You tried to guess", tries, "times")
    guess = input("Guess again: ")
    tries += 1
print("You got it!")
```

- ***1*** **Checks whether the guess is different than the secret word**
- ***2*** **Asks user again**
- ***3*** **Reached when guess is correct**

At this point, you should notice that the code block has to include a statement to change something about the condition itself. If the code block is independent of the condition, you enter an infinite loop. In listing 18.1, the guess was updated by asking the user to enter another word.

### 18.1.2. while loops

In Python, the keyword that begins a `while` loop is, not surprisingly, `while`. The following listing shows a general way of writing a `while` loop.

**Listing 18.2. A general way to write a `while` loop**

```
while <condition>:                1
    <do something>
```

- ***1*** **Indicates beginning of loop**

When Python first encounters the `while` loop, it checks whether the condition is true. If it is, it enters the `while` loop code block and executes the statements as part of that block. After it finishes with the code block, it checks the condition again. It executes the code block inside the `while` loop as long as the condition is true.

**Quick check 18.1**

**Q1:**

Write a piece of code that asks the user for a number between 1 and 14. If the user guesses right, print `You guessed right, my number`

### 18.1.3. Infinite loop

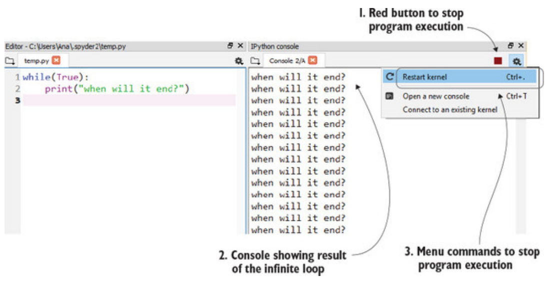
With `while` loops, it's possible to write code that will never finish. For example, this piece of code infinitely prints `when will it end?!`

```
while True:
    print("when will it end?!")
```

Letting a program like this run for a long time will slow your computer. But if this happens, don't panic! There are a few ways to manually stop a program that entered an infinite loop, as shown in figure 18.2. You can do one of the following:

- Click the red square at the top of the console.
- Click into the console and then hit Ctrl-C (press and hold the Ctrl key and then press the C key).
- Click the menu in the console (beside the red square) and choose Restart Kernel.

**Figure 18.2. To manually exit out of an infinite loop, you can click the red square, or press Ctrl-C, or choose Restart Kernel from the console menu beside the red square.**


1. Red button to stop program execution
Editor - C:\Users\Ana\.spyder2\temp.py
temp.py
IPython console
Console 2/A
Restart kernel Ctrl+.
Open a new console Ctrl+T
Connect to an existing kernel
2. Console showing result of the infinite loop
3. Menu commands to stop program execution

```
while(True):
    print("when will it end?")
```

```
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
when will it end?
```

## 18.2. USING FOR LOOPS VS. WHILE LOOPS

Any `for` loop can be converted into a `while` loop. A `for` loop iterates a set number of times. To convert this to a `while` loop, you need to add a variable whose value is checked in the `while` condition. The variable is changed every time through the `while` loop. The following listing shows a `for` loop and `while` loop side by side. The while loop case is more verbose. You must initialize a loop variable yourself; otherwise, Python doesn't know to what variable x you're referring inside the loop. You must also increment the loop variable. In the `for` loop case, Python does these two steps automatically for you.

**Listing 18.3. A `for` loop rewritten as a `while` loop**

**With a for loop**

```
for x in range(3):              1
    print("var x is", x)
```

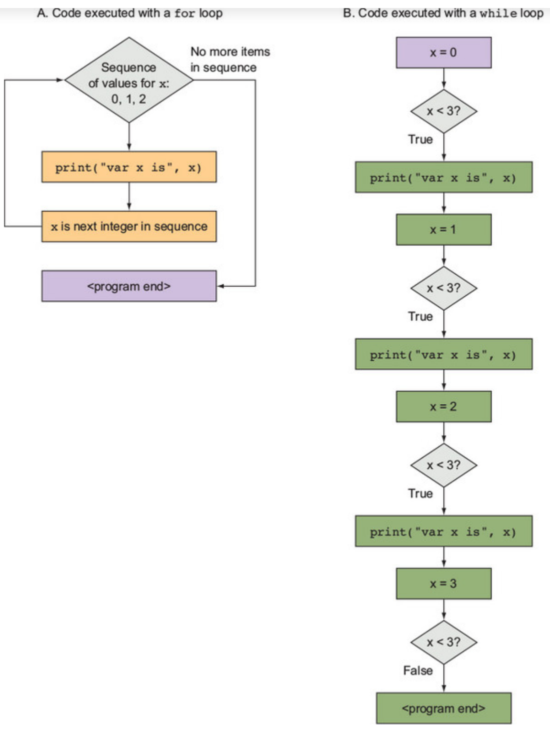
**With a while loop**

```
x = 0                      2
while x < 3:
   print("var x is", x)
   x += 1                 3
```

- ***1*** **Indicates beginning of loop**
- ***2*** **Initializes loop variable**
- ***3*** **Increments loop variable**

In listing 18.3, you have to create another variable. You must manually increment its value inside the `while` loop; remember that the `for` loop increments the value of the loop variable automatically. Figure 18.3 shows how to visualize the code in listing 18.3.

**Figure 18.3. (A) shows the `for` loop code that prints the value of the loop variable with each iteration of the loop. (B) shows the `while` loop code and how the variables in change values when the `for` loop is converted into a `while` loop. You have to create your own variable and increment it yourself inside the body of the `while` loop. Additionally, you have to write a conditional as a function of your variable that will cause the body of the `while` loop to repeat three times.**


A. Code executed with a for loop
B. Code executed with a while loop
Sequence of values for x: 0, 1, 2
No more items in sequence
print("var x is", x)
x is next integer in sequence
<program end>
x = 0
x < 3?
True
x = 1
x = 2
x = 3
False

Any `for` loop can be converted into a `while` loop. But not all `while` loops can be converted into `for` loops, because some `while` loops don't have a set number of times to iterate. For example, when you ask the user to guess a number, you don't know how many tries the user will take to guess the number, so you don't know what sequence of items to use with a `for` loop.

**Quick check 18.2**

**Q1:**

Rewrite the following code with a `for` loop:

```
password = "robot fort flower graph"
space_count = 0
i= 0
while i < len(password):
    if password[i] == " ":
        space_count += 1
    i += 1
print(space_count)
```

**Thinking like a programmer**

In programming, there's usually more than one way to do something. Some ways are terse, and some are verbose. A good Python programmer should find a way to write code that's as simple and short as possible, while being easy to understand.

## 18.3. MANIPULATING LOOPS

You've seen the basic structure of `for` and `while` loops. Their behavior is straightforward, but they're a bit limiting. When you use a `for` loop, you go through the loop as many times as you still have items in your sequence of values (either integers or string characters). With a `while` loop, the only way to stop repeating it is for the `while` loop condition to become false.

But to write more-flexible code, you may want the option to exit out of a `for` loop early. Or you may want to exit out of a `while` loop early if an event happens inside the code block that's independent of the `while` loop condition.

### 18.3.1. Exiting early out of a loop

The only way that you know how to exit out of a `while` loop is for the `while` condition to become false. But you'll often want to exit out of a loop early (either a `for` or `while` loop). A keyword in Python, `break`, makes it possible to exit out of a loop whenever Python executes that keyword, even if the `while` loop condition is still true. Listing 18.4 shows example code using the `break` keyword.

In listing 18.4, you see the addition of an extra condition within the loop to counter the break statement, the loop immediately terminates; nothing after the break statement, but within the loop, is executed. Because there's now a possibility of exiting the loop for causes other than the user getting the word right, you have to add a conditional after the loop to check why the loop terminated.

**Listing 18.4. Using the `break` keyword**

```
secret = "code"
max_tries = 100
guess = input("Guess a word: ")
tries = 1
while guess != secret:
    print("You tried to guess", tries, "times")
    if tries == max_tries:
        print("You ran out of tries.")
        break                                         1
    guess = input("Guess again: ")
    tries += 1
if tries <= max_tries and guess == secret:            2
    print("You got it!")                              2
```

- *1* **Breaks out of loop when exceed max_tries**
- *2* **Checks why exited out of loop**

Why does this code have an extra `if` statement after the `while` loop? Think about what happens in two cases: the user guesses the secret word, or the user runs out of tries. In either case, you stop executing the `while` loop and execute any statements right after the `while` loop block. You want to print a congratulatory message only if the exit from the loop was due to the correct guess. The congratulatory message has to come after the `while` loop terminates, but you can't just print the message outright. The `while` loop may have terminated because the user ran out of tries; you just don't know why you exited the `while` loop.

You need to add a conditional that checks whether the user still has tries left and whether the user's guess matched the secret. The conditional ensures that if the user still has tries left, you exited the `while` loop because the user guessed the secret word and not because the user ran out of tries.

**Thinking like a programmer**

It's always a good idea to create variables to store values that you're going to reuse many times in your code. In listing 18.4, you created a variable named `max_tries` to hold the number of times to ask the user to guess. If you ever decide to change the value, you have to change it in only one place (where it's initialized) instead of trying to remember everywhere you used it.

The `break` statement works with `for` and `while` loops. It can be useful in many situations, but you have to use it with caution. If you have loops within loops, only the loop that the `break` statement is a part of terminates.

**Quick check 18.3**

Write a program that uses a `while` loop to ask the user to guess a secret word of your choosing. The user gets 21 tries. When the user gets it right, end the program. If the user uses up all 21 tries, exit the loop and print an appropriate message.

### 18.3.2. Going to the beginning of a loop

The `break` statement in the previous section caused any remaining statements in a loop to be skipped, and the next statement executed was the one right after the loop.

Another situation you might find yourself in is that you want to skip any remaining statements inside a loop and go to the beginning of the loop to check the conditional again. To do this, you use the `continue` keyword, which is often used to make code look cleaner. Consider listing 18.5. Both versions of the code do the same thing. In the first version, you use nested conditionals to make sure all conditions are satisfied. In the second version, the `continue` keyword skips all subsequent statements inside the loop and fast-forwards to the beginning of the loop with the next x in the sequence.

**Listing 18.5. Comparing code that does and doesn't use the `continue` keyword**

```
### Version 1 of some code ###
x = 0
for x in range(100):
        if x%10 != 0:
            print("x is not divisible by 10")
            if x==2 or x==4 or x==16 or x==32 or x==6
                print("x is a power of 2")
                # perhaps more code

### Version 2 of some code ###
x = 0
for x in range(100):
    print("x is", x)
    if x <= 5:
        continue
    print("x is greater than 5")
    if x%10 == 0:
        continue
    print("x is not divisible by 10")
    if x!=2 and x!=4 and x!=16 and x!=32 and x!=64:
        continue
    print("x is a power of 2")
    # perhaps more code
```

- ***1* Skips remaining loop statements**
- ***2* Get here when x > 5**
- ***3* Get here when x%10 ! = 0**
- ***4* Get here when x is 2, 4, 16, 32, or 64**

In listing 18.5, you can write two versions of the same code: with and without the `continue` keyword. In the version that contains the `continue` keyword, when the conditionals evaluate to true, all remaining statements in the loop are skipped. You go to the beginning of the loop and assign the next value for `x`, as if the loop statements are executed. But the code that doesn't use the `continue` keyword ends up being a lot more convoluted than the one that does. In this situation, it's useful to use the `continue` keyword when you have a lot of code that you want to execute when a bunch of nested conditions hold.

## SUMMARY

In this lesson, my objective was for you to write programs that repeat certain tasks with a `while` loop. `while` loops repeat tasks while a certain condition holds.

Whenever you write a `for` loop, you can convert it into a `while` loop. The opposite isn't always possible. This is because `for` loops repeat a certain number of times, but the condition for entering inside a `while` loop might not have a known, set number of times that it can happen. In the examples, you saw that you can ask the user to enter a value; you don't know how many times the user will enter the wrong value, which is why a `while` loop is useful in that situation.

You also saw how to use the `break` and `continue` statements within loops. The `break` statement is used to stop executing all remaining statements inside the innermost loop. The `continue` statement is used to skip all remaining statements inside the innermost loop and continue from the beginning of the innermost loop.

Here are key takeaways from this lesson:

- `while` loops repeat statements as long as a certain condition holds.
- A `for` loop can be written as a `while` loop, but the opposite may not be true.
- A `break` statement can be used to exit a loop prematurely.
- A `continue` statement can be used to skip remaining statements in the loop and check the `while` loop conditional again or go to the next item in the `for` loop sequence.
- There's no penalty for trying one kind of loop (`for` or `while`) and finding it's not working out in your program. Try out a couple of things and think of it as trying to put together a coding puzzle.

Let's see if you got this...

**Q18.1**

This program has a bug. Change one line to avoid the infinite loop. For a few pairs of inputs, write what the program does and what it's supposed to do.

```
num = 8
guess = int(input("Guess my number: "))
while guess != num:
    guess = input("Guess again: ")
print("Right!")
```

**Q18.2**

tween 1 and 10 and ask the user to guess the number. Your program should continue asking the user to guess the number until they get it right. If they get it right, print a congratulatory message and then ask if they want to play again. This process should be repeated as long as the user enters `y` or `yes`.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Lesson 17. Customizing loops

NEXT
Lesson 19. Capstone project: Scrabble, Art Edition