

# Loops and Flags

---

## While Loops

A common task for programmers is repeating the same code over and over, also known as **iteration**. There are a couple different ways to do this in Python, but the simplest way is to use a **while loop**.

```
while True:
    print("So many loops!")
```

While loops will continue running as long as their condition stays **True**. For example, the loop above will never end, since its condition never changes to **False**. This type of behavior is called an **infinite loop**, and should be avoided to prevent programs from getting stuck or running forever. If a loop doesn't have an exit condition or the exit condition can never be reached, the code inside the loop will run forever (or at least until you force-exit the program – in IDLE you can use [Ctrl] + [C] to exit whatever's running). To get the loop to exit, the condition needs to be a variable that changes throughout the flow of the loop.

```
x = 3
while x > 0:
    print("The value of x is", x)
    x = x - 1
print("The final value of x is", x)
```

The output of the loop above looks like

```
The value of x is 3
The value of x is 2
The value of x is 1
The final value of x is 0
```

Python checks the value of *x* each time the loop runs, and exits when the condition fails.

If the condition is **False** the first time through, Python will skip over the loop entirely. If *x* was set to 0 instead of 3 in the last example, the output would look like

```
The final value of x is 0
```

---

This type of strategy is called **decrementing**, where a variable is decreased every time the loop runs. Another common technique is **incrementing**, where the variable is increased every time.

```
x = 0
while x < 3:
    print("The value of x is", x)
    x = x + 1
print("The final value of x is", x)
```

```
The value of x is 0
The value of x is 1
The value of x is 2
The final value of x is 3
```

While loops are also useful for user input, such as waiting for a user to enter a command during a program. The follow snippet of code asks the user to enter the word "red", then keeps running until they enter that word.

```
userInput = ""
while userInput != "red":
    userInput = input("Enter the word red! ")

print("The user entered red!")
```

This then outputs:

```
Enter the word red! blue
Enter the word red! orange
Enter the word red! red
The user entered red!
```

**Try it yourself:** print out your name 100 times.

## Using Flags With Loops

One problem with while loops is that they only have one conditional. Conditionals can be combined with operators, but that can create messy code that's hard to read and write. Especially as code gets more complicated, it can become very handy to use **flags** to check conditions throughout the loop.

If a programmer is writing a loop to run diagnostics on a car, the loop might need to run until it finds a broken component. Writing the loop so that it checks every single component inside its conditional would be cumbersome and impossible to read through.

---

To avoid this, the programmer can add a flag to signal that a broken component was found.

```
maintenance = False
while maintenance != True:
    if tirePressure < 25:
        print("Tire pressure too low!")
        maintenance = True
    if idleRpm < 600:
        print("Idle RPM too low!")
        maintenance = True

    # Run checks on the rest of the car

print("This car needs maintenance!")
```

If the tire pressure was 35 but the IDLE RPM was only 500, the maintenance check would fail, and the program would output:

```
Idle RPM too low!
This car needs maintenance!
```

**Try it yourself:** write a program that continuously takes user input until the input either has a "r", "b", or "g" in it. It may be easier to do this using a flag.

## For Loops

An easier way to decrement or increment in a loop is to use the second type, the **for loop**. To write a while loop that executes ten times, we would write

```
counter = 0
while(counter < 10):
    print("The Loop is running")
    counter = counter + 1
print("The loop has finished")
```

Using a for loop instead, we can do the same thing using the following code

```
for counter in range(10):
    print("The loop is running")
print("The loop has finished")
```

Both of these programs will do exactly the same thing; they are semantically equivalent. The only difference is the way the code runs in the background. The variable **counter** is accessible from inside the loop on either one, so we could have them print out the counter

---

with a statement such as `print(counter)`, and both loops would still have the same output.

For loops use a different kind of approach to incrementing. The loop variable is set to an element in a list of numbers, which we create using `range()`, which provides a list of numbers from the start point to the end point minus 1. You can set this point using `range(max)` or `range(min, max)`, where *min* and *max* can be literal integers or a variable. In the code above, the variable `counter` is set to the numbers 0 to 9, increasing by one on each pass of the loop. It's fine to simply know that for loops have a counter variable, a start point (0 by default), and an end point. This list reading allows some more advanced functionality that you may want to explore on your own later.

A common use of **for** loops is repeating a chunk of code with different values each time.

```
for i in range(5, 11):
    print(i, "to the second power is", i**2)
```

The loop above will run through six times, using the values 5, 6, 7, 8, 9, and 10 for *i* on each pass. The output looks like

```
5 to the second power is 25
6 to the second power is 36
7 to the second power is 49
8 to the second power is 64
9 to the second power is 81
10 to the second power is 100
```

**While** loops are best for code that needs to wait for something to happen. **For** loops are best for code that needs to repeat a certain number of times.

**Try it yourself:** write a program that counts from 100 to 0, then counts back up from 0 to 100.

## Nested Loops

Sometimes a task needs to include a loop, but that task needs to be in a loop too. Just as **if** statements can be nested inside each other, loops can contain other loops. **For** loops can be put inside of **while** loops, and **while** loops can be put inside of **for** loops, but most of the time it's as simple as a **for** loop inside another **for** loop.

---

```
for i in range(2):
    print("<", end='')
    for n in range(5):
        print("-", end='')
    print(">")
```

Python runs through the outer loop, prints out an arrow, and starts the inner loop. The inner loop runs five times and prints a line of dashes. Once it ends, it dumps back to the outer loop, and moves on to the second pass. The output looks like

```
<----->
<----->
```

**print** functions formatted with an extra tag, `end=' '`, stop Python from moving to the next line after printing the string.

**Try it yourself:** write a program that counts from 1 to 10, 10 times.

## Exercises

1. Write a program that prints out the lyrics to *99 Bottles of Beer on the Wall*.
2. Write a program that asks for your name until you get it right.
3. Write a program that plays "Guess a number between 1 and 20" with you. You can use `random.randint(a, b)` to generate a random number between `a` and `b`.