# Lecture 09

# The Python `for` -Loop

## 5 Ventôse CCXXX

***Song of the day***: ***Sonny Boy Rhapsody*** *by toe (2021).*

## Part 1: *Control Flow Structures (so far)*

Recall the definition of **control flow structures**:

> **Control flow structures**: language features that affect the order (flow) in which instructions are executed

Python executes from top to bottom. That is, if your program looks like this:

```
author_name = "David Hoon Kim"  # Line a
book_name = "Paris Is A Party, Paris Is A Ghost"  # Line b
print("My favorite book from 2021 was", book_name, "by", author_name)  # Line c
```

Python will ***always*** "execute" in the line `a`, line `b`, line `c` order. We can affect this flow by using some of the control flow structures that we already know:

```
genre = input("Enter a music genre: ")  # Line a

if genre == "pop":
    print("The 1975")  # Line b
elif genre == "jazz":
    print("Roy Hargrove")  # Line c
else:
    print("Too many to choose from")  # Line d
```

Here, the order of "execution" could be any of the three below depending on the user's input:

- Line `a`, line `b`
- Line `a`, line `c`

- Line `a`, line `d`

Similarly, in the case of `while`-loops:

```
turns = int(input("How many times do you want the while-loop to execute? "))  # li

current_turn = 1  # line b

while current_turn <= turns:
    print("Turn number", current_turn)  # line c
    current_turn += 1  # line d
```

**Code block 1**: Counting from 1 to whatever integer the user enters.

The order will be line `a`, line `b`, (line `c` and `d`) * `n`, where `n` is depends completely on the user input. For example, if they enter `3` as a value for `turns`, the order of execution is:

> Line `a`, line `b`, line `c`, line `d`, line `c`, line `d`, line `c`, line `d`

We have learned that what makes a machine an actual computer is its ability to make decision depending on its current situation (e.g. depending on whether the value of `genre` equals the string `"pop"`, or whether the value of `current_turn` is less than or equal to the value of `turns`).

`while`-loops are most useful for situations where ***we don't know at which point the loop will end***. For example, let's say we ask the user to enter character at random. If they enter the number 0, however, the program will end:

```
current_character = input("Enter a character to start: ")

while current_character != "0":
    current_character = input("Enter another character, or '0' to quit: ")

print("End of program.")
```

**Code block 2**: We don't know how many times the `while`-loop will iterate—it depends entirely on outside input.

In both code block 1 and code block 2, how many times our `while`-loop will run is depends on what the user inputs, sure. But there's a critical difference between the two of them.

In code block 1, the user inputs a single number at the beginning of the program, and the program takes care of the rest; it only needs to know until which number to count **once**. In code

block 2, however, the code's execution is ***dependent on the user from start to finish***; we simply don't know when the user will decide to enter the character `"0"`. These are the types of situations that the `while`-loop excels at—when we can't reasonably predict at which point the loop will stop.

So, is code block 1 bad practice? No, not really. It's a perfectly fine execution of the task. There is, however, a control flow structure that is even better fitted for situations where we know exactly when the program will end:

The `for`-loop.

## Part 2: *The* `for` *-Loop*

The "inefficiency" in code block 1 is that we **have to manually increase the value of** `current_turn`. This may not sound like a big deal, but imagine a situation where changing the value of `current_turn` was an expensive operation— that is, imagine it took a good chunk of your computer's RAM to change the value of `current_turn`. In this case, we definitely would not want to do this and, preferably, leave it up to the program to do it in the most efficient way possible. That is exactly what the `for`-loop does.

The general structure of the `for`-loop is as follows (in pseudocode):

```
for loop_var in sequence:
    # perform these instructions
```

In English, this would read like this:

> **For** every value that our loop variable `loop_var` takes from `sequence`, perform these instructions.

This sounds a little abstract, so let's apply it to our counting program from code block 1 (pseudocode):

```
turns = int(input("How many times do you want the for-loop to execute? "))

for current_turn in range(turns):
    print("Turn number", current_turn)
```

**Code Block 3**: The `for`-loop equivalent of code block 1.

We see here a couple of things we haven't seen before, but let's first turn this code into English:

> For every value that our loop variable `current_turn` takes from a ***range of numbers*** of `0` to `turns`, execute the line `print("Turn number", current_turn)`.

So what exactly is Python doing here? Let's hard-code the value of `turns` to `3`, and go line by line:

```
for current_turn in range(3):
    print("Turn number", current_turn)
```

1. The `for`-loop starts, and **automatically** assigns `current_turn` a value of `0`.
2. The `for`-loop checks if the value of `current_value` (`0`) is less than `3`. In this case, it is.
3. Because it is less than `3`, it executes the code inside the `for`-loop; the line `"Turn number 0"` is printed.
4. The `for`-loop **automatically** gives `current_value` a value of `1`.
5. The `for`-loop checks if the value of `current_value` (`1`) is less than `3`. In this case, it is.
6. Because it is less than `3`, it executes the code inside the `for`-loop; the line `"Turn number 1"` is printed.
7. The `for`-loop **automatically** gives `current_value` a value of `2`.
8. The `for`-loop checks if the value of `current_value` (`2`) is less than `3`. In this case, it is.
9. Because it is less than `3`, it executes the code inside the `for`-loop; the line `"Turn number 2"` is printed.
10. The `for`-loop **automatically** gives `current_value` a value of `3`.
11. The `for`-loop checks if the value of `current_value` (`3`) is less than `3`. In this case, ***it is not***.
12. The `for`-loop ends.

So, in this case, the `for`-loop is doing two things for you that you had to manually do in code block 1:

1. **It creates a loop variable for you and assigns it an initial value.**
2. **It changes the value of the loop variable for you every single turn of the loop (iteration).**

So what is this `range()` function? `range()` is a **built-in Python function that creates a sequence of integers**.

Here's a couple of examples:

```
# If we wanted the sequence 0, 1, 2
zero_to_two = range(3)


# If we wanted the sequence 0, 1, 2, 3
zero_to_four = range(4)


# If we wanted a sequence based on user-input
upper_limit = int(input("Enter an upper limit: "))
zero_to_limit = range(upper_limit + 1)
```

There are two other forms that the `range()` function can have. If we enter **two** integers inside its parentheses, it will accept the first one as its ***start***, and the second one as its ***stop***:

```
# If we wanted the sequence 2, 3, 4
two_to_five = range(2, 5)


# If we wanted the sequence 4, 5, 6, 7, 8, 9
four_to_ten = range(4, 10)
```

The last form of the `range()` function allows us to specify the **step**—by how many numbers our values change—ever inside the sequence:

```
# If we wanted the sequence 2, 4, 6, 8
two_to_eight_skip_two = range(2, 9, 2)

# If we wanted the sequence 4, 7, 10
four_to_ten_skip_three = range(4, 11, 3)

# If we wanted the sequence 0, 5, 10, 15
zero_to_fifteen_skip_five = range(0, 16, 5)

# If we wanted the sequence 7, 6, 5, 4
seven_to_four = range(7, 3, -1)
```

So now our `range()` function in code block 3 makes a lot more sense. `range(turns)` would read in English as:

> The sequence of number starting from `0` (inclusive; the default starting value) to the value of `turns` (non-inclusive), with a step of `1` (the default step value).

## Part 4: *When should I use a* `while` *-loop and when should I use a* `for` *-loop?*

Invariably, the decision of whether to use a `while` - or a `for` -loop is something that programming newcomers always struggle with.

As we said before:

> A `while` -loop is better suited to situations when ***we can't faithfully predict at which point the loop will stop***.

Examples of this include:

- A game running until the user selects "quit". (i.e. the game runs *while* the user doesn't select quit)
- A light sensor leaving the lights on until the room is empty. (i.e. the lights stay on *while* there are people inside)
- Instructing Turtle to move forward until the user quits the window, if they ever do. (i.e. Turtle will move *while* the window is open)

On the other hand:

> A `for` -loop is better suited to situations when ***we are able to know at which points the loop will start and stop***.

Examples of this include:

- A program that changes the names of several selected files. (i.e. *for* every *file* in a selection, perform a name change)
- Excel applying the same formula to every cell if you drag it across a row of cells. (i.e. *for* every *cell* in a group of highlighted cells, apply the same formula)
- A shuffle program playing every single song from an album in a random order. (i.e. *for* every *song* in a collection of songs, perform some sort of play operation. Here, our start, stop, and step values are randomized).

Interestingly, while `for` -loops are *far* more common in professional programming situations, beginner programmers tend to gravitate towards using the `while` -loop for everything. I have no idea why this is the case, to be honest, but don't feel like you're doing something wrong by having a hard time choosing one. An inelegant solution is better than no solution at all.

And after a while, this decision becomes such second nature to you that choosing one comes as quickly to you as knowing that you have to print something, or ask the user for input.

Oh and, to reiterate: no, the `break` keyword is absolutely not allowed in this class.