# Chapter 2 : Branching and Iteration

Aviral Janveja

## 1 Branching

Branching refers to making decisions in your code based on certain conditions. Conditionals allow us to write programs that are more interesting than straight line programs. The **if** statement is a fundamental branching statement that allows you to execute a block of code only if a specified condition is true.

### 1.1 if

The basic syntax of the **if** statement is as follows :

```
if <condition>:
    # Code to be executed
    # if the condition is true
```

The **condition** is a Boolean expression that evaluates to either true or false. The code inside the **if** block will only be executed if the condition is true. If the condition is false, the code inside the block will be skipped. For example:

```
x = 3
if x > 5:
    print("x is greater than 5")
```

You can use an **else** block to specify what should happen if the condition in the if statement is not true. For example:

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

You can use **elif** (short for "else if") to check for multiple conditions in sequence. For example:

```
x = 5
if x > 5:
    print("x is greater than 5")
elif x < 5:
    print("x is less than 5")
else:
    print("x is equal to 5")
```

Here, the program will check the conditions one by one and execute the block associated with the first true condition. You can run the branching_if.py program on GitHub and try for yourself. Also checkout the nested_if.py program on GitHub for an example of how **nested-if** statements work.

### 1.2 Indentation

Notice, how the **code blocks** following the if-statement are always indented. In Python, indentation is a fundamental aspect of the language's syntax. Unlike many other programming languages that use braces or other explicit markers to indicate blocks of code, Python uses indentation to define the structure and scope of code blocks.

Blocks of code that are at the same level of indentation are considered part of the same block or scope. Indentation is typically done using four spaces for each level and incorrect indentation can lead to syntax errors or affect the logical structure of your code. An advantage of the Python approach is that the visual structure of the program accurately reflects the semantic structure of the program.

## 2 Iteration

Iteration in Python involves repeatedly executing a block of code. One of the primary tools for iteration is the while loop, which allows you to execute a block of code as long as a specified condition is true.

### 2.1 While

The basic syntax of the while-loop is as follows :

```
while <condition>:
    # Code to be executed as long
    # as the condition is true
```

Like the conditional statement, it begins with a test. If the test evaluates to True, the program executes the loop body once, and then goes back to re-evaluate the test. This process is repeated until the test evaluates to False. After which, control passes to the code following the loop body. For example :

```
count = 0
while count <= 5:
    print("Iteration", count)
    count += 1
```

In this example, the while loop will continue executing the indented block of code as long as the condition **count** $<=$ **5** is true. The **count += 1** statement increments the value of count in each iteration, thus preventing an infinite loop.

**Note** : Be sure to avoid unintentional infinite loops by ensuring that the condition in the while loop will eventually become false. While loops are powerful constructs for repeated execution of code, especially when the exact number of iterations is unknown beforehand or dependent on a specific condition. For instance, while asking for user-input.

Checkout the iteration_while.py program on GitHub for example.

## 2.2 For

In Python, the for loop is used for iterating over a sequence, such as a range of numbers or characters in a string. It allows you to execute a block of code for each element in the sequence. The basic syntax of a for loop is as follows :

```
for <variable> in <sequence>:
    # Code to be executed for each
    # element in the sequence
```

The loop variable takes on the value of each element in the specified sequence with each iteration of the loop. The **range** function is often used to generate a sequence of numbers. It can take one, two or three arguments to specify the start, stop and step values for the sequence. For example :

```
for number in range(6):
    print(number)
```

The above code produces the following output :

```
0
1
2
3
4
5
```

Here is another basic example :

```
word = "python"
for char in word:
    print(char)
```

In this example, we have a string "Python". The for loop iterates over each character in the string and prints it out, producing the following output :

```
p
y
t
h
o
n
```

These examples illustrate how a for loop can be used to iterate over a sequence, one element at a time. The loop executes the indented block of code for each element in the sequence. It is particularly useful when you know the number of iterations beforehand. Checkout the iteration_for.py program on GitHub for example.

### 2.3 Break

The break and continue statements are control-flow statements in Python that are used within loops to modify the normal execution of the loop.

The break statement is used to exit a loop prematurely, regardless of whether the loop's condition is still true. When the break statement is encountered, the loop is terminated immediately, and the program continues with the next statement after the loop. For example :

```
for number in range(6):
    if number == 3:
        break
    print(number)
```

In this example, the loop is exited as soon as the value 3 is encountered, and the program moves on to the next statement after the loop.

### 2.4 Continue

The continue statement is used to skip the rest of the loop body for the current iteration and move on to the next iteration. It does not exit the loop entirely, instead it jumps to the next iteration, bypassing the remaining code within the loop for the current iteration. For example :

```
for number in range(6):
    if number == 3:
        continue
    print(number)
```

In this example, when the value 3 is encountered, the continue statement causes the loop to skip the print statement for that particular iteration and move on to the next iteration.

These statements provide flexibility in controlling the flow of iteration, allowing you to tailor the loop's behavior based on specific conditions or requirements. You can check the code on Github here for_break_continue.py

## 3 String Manipulation

Strings in Python are sequences of characters and are one of the fundamental data types in the language. You can create strings using single (') or double (") quotes.

### 3.1 Concatenation

Combining two or more strings using the + operator. For example :

```
a = "Hello"
b = "World"
result = a + " " + b
print(result)
# Output : Hello World
```

## 3.2 Repetition

Repeating a string using the $*$ operator.

```
original = "Python "
repeated = original * 3
print(repeated)
# Output : Python Python Python
```

## 3.3 Length Function

This function is used to retrieve the length of the string in the parenthesis. For example :

```
s = "abc"
L = len(s)
print(L)
# Output = 3
```

## 3.4 Indexing

Indexing in Python allows you to access individual characters in a string by their position (index) within the string. The index **starts from 0** for the first character, 1 for the second character, and so on. Here is an example :

```
word = "Hello"
print(word[0])
# Output = H
```

## 3.5 Slicing

String slicing in Python allows you to extract a portion of a string by specifying a range of indices. The basic syntax for string slicing is as follows :

```
string[start:stop:step]
```

**start** is the index from which the slicing begins, **including** the specified index. **stop** is the index at which the slicing ends, **excluding** the specified index. **step** indicates how many characters to skip. It is optional and defaults to 1. Here is an example to illustrate string slicing :

```
word = "Hello"
slice1 = word[1:3:1]
slice2 = word[::]
slice3 = word[::-1]
print(slice1)
print(slice2)
print(slice3)
```

The above code produces the following output :

```
el
Hello
olleH
```

**Note** : Strings in Python are **immutable**. This means that once a string is created, you cannot change its individual characters or the length of the string. If you want to make modifications to a string, you will have to create a new string. You can checkout the code for the above section here : string_examples.py