IIT Madras
BSc Degree

Home Lesson-2.4

Lesson-2.3

Conditional Statements

if

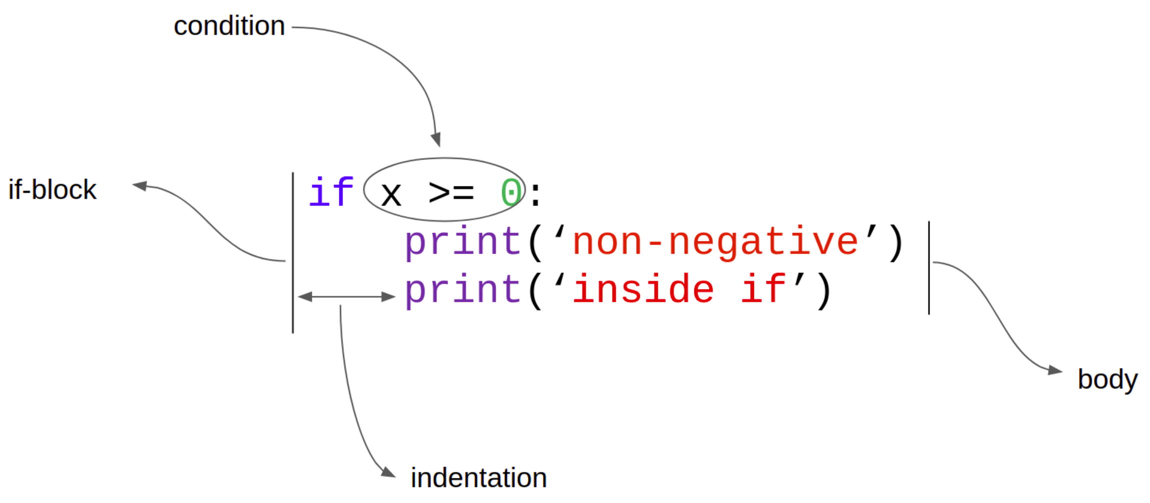
Let us explore the idea of conditional statements by solving a simple problem:

> **Problem**: Accept an integer as input from the user. If the number is greater than or equal to zero, print: `non-negative`.

Solution

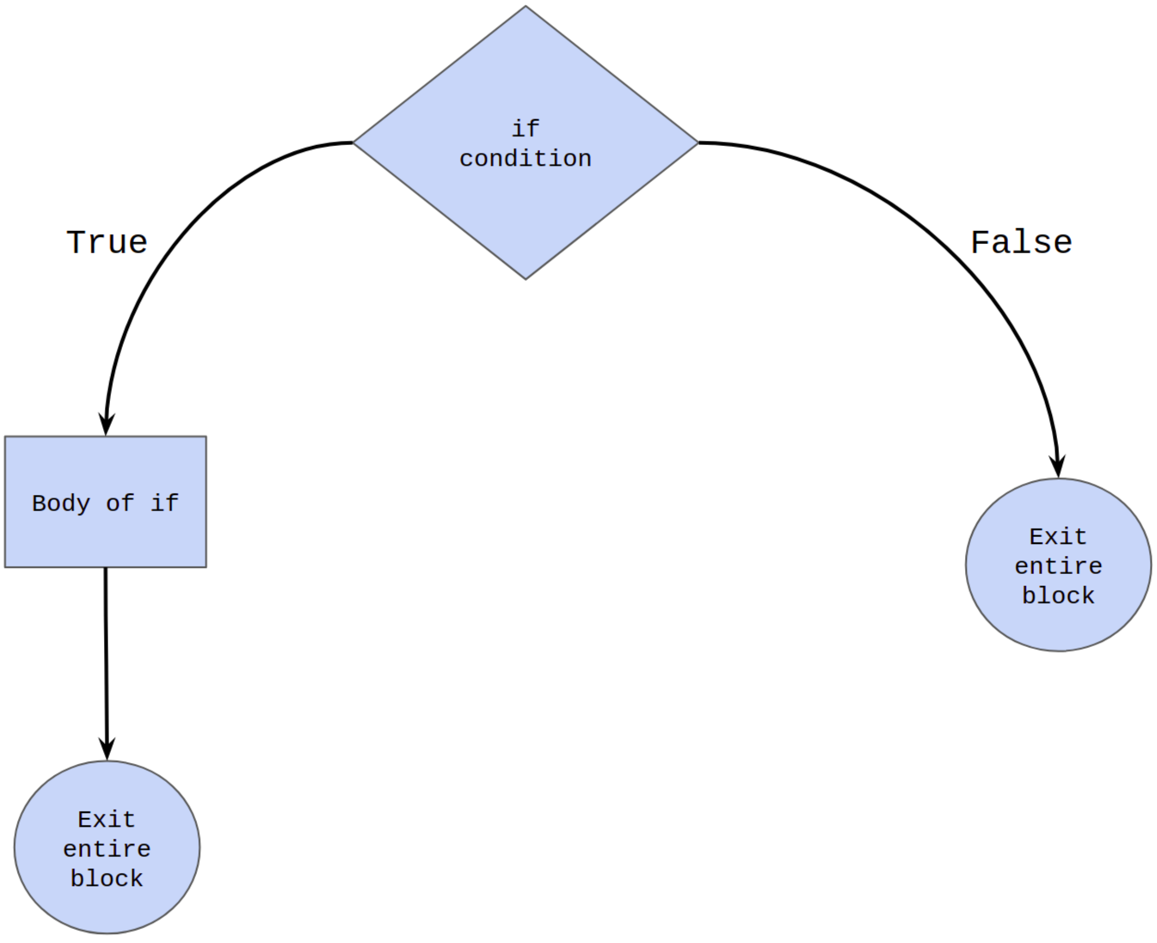
```
1  x = int(input())
2  if x >= 0:
3      print('non-negative')
```

`if` is a keyword in Python. The text adjacent to `if` is a boolean expression, usually called the **if-condition** or just the **condition**. Line-3 is the body of `if`. If the condition evaluates to `True`, then line-3 is executed. If it is `False`, then line-3 doesn't get executed. The following diagram captures the terms that have been introduced:



```
if x >= 0:
    print('non-negative')
    print('inside if')
```

The control flow of the if-statement as a flow chart is given below:



Coming back to the code snippet:

```
x = int(input())
if x >= 0:
    print('non-negative')
```

Note that line-3 in the solution code is indented. In this case, the indentation corresponds to four spaces. It is very important to keep this consistent throughout the program. In all lessons, the first level of indentation will have four space . To understand how indentation works and why it is necessary, consider the following code blocks:

```
1  # Left                                       | # Right
2  x = 1                                        | x = -1
3  if x >= 0:                                   | if x >= 0:
4      print('non-negative')                    |     print('non-negative')
5      print('inside if')                       |     print('inside if')
6  print('outside if')                          | print('outside if')
```

The output is:

```
non-negative                                 | outside if
inside if                                    |
outside if                                   |
```

Lines 3-5 in the code make up the **if-block**. Lines 4 and 5 which are indented make up the body of `if`. Whenever the if-condition evaluates to `True`, the interpreter enters the body of `if` and executes the lines sequentially. The indentation helps in separating the body of the if-block from the rest of the code.

Left: For the code on the left, the condition is `True`. So lines 4 and 5 are going to be executed. Once we exit the if-block, the interpreter will resume execution from line-6.

Right: For the code on the right, the condition is `False`. So, lines 4 and 5 are *not* going to be executed. The interpreter will skip the body of `if` and directly move to line-6.

if-else

Let us add one more level of complexity to the problem.

Problem

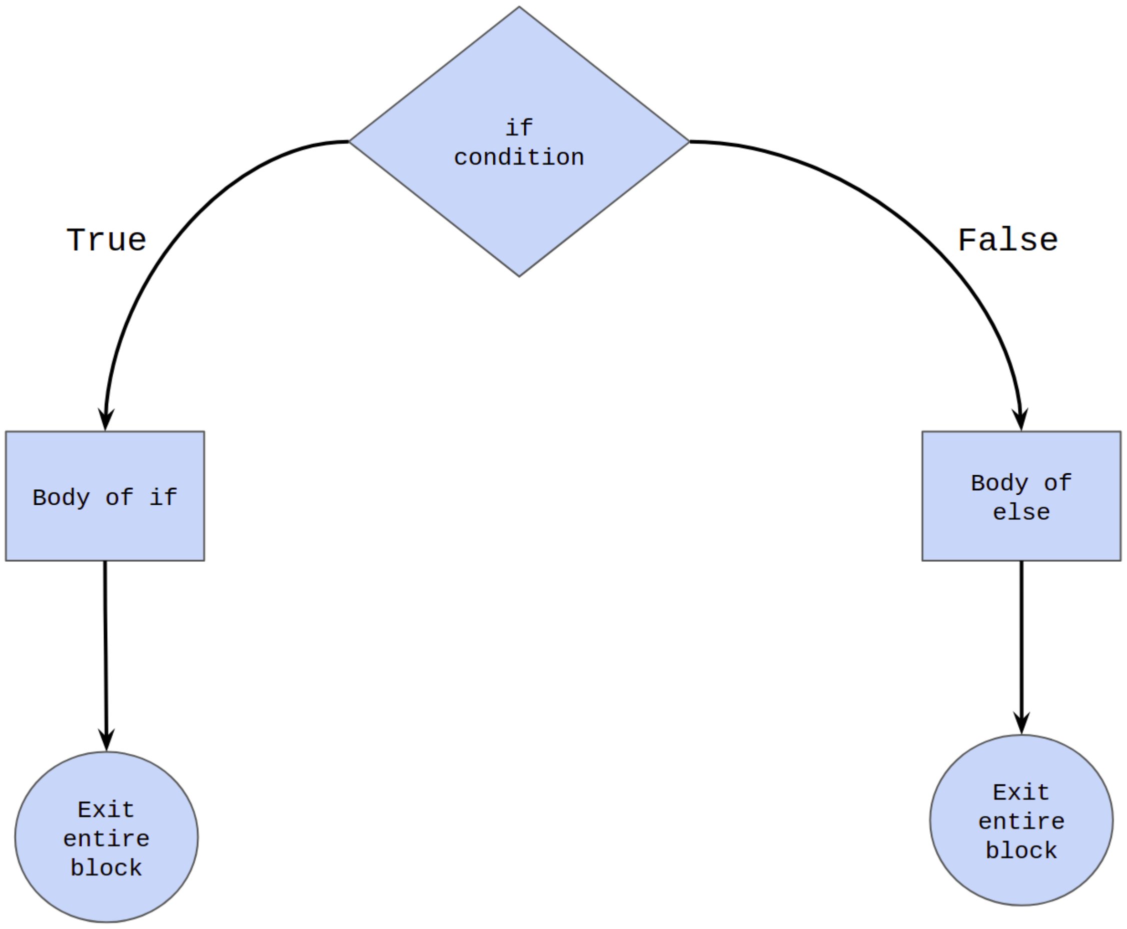
> Accept an integer as input from the user. If the number is greater than or equal to zero, print: `non-negative`. If the number is less than zero, print `negative`.

Solution

```
x = int(input())
if x >= 0:
    print('non-negative')
else:
    print('negative')
```

`else` is a keyword in Python. When the if-condition evaluates to `True`, the statements inside the body of the if-block are evaluated. When the condition evaluates to `False`, the statements inside the body of the else-block are evaluated.

A visual representation of the control flow:



Points to remember:

- `if` and `else` are at the same level of indentation.
- `else` can never occur independent of an `if` in conditional statements.
- `else` cannot have any new condition associated with it.

The following code demonstrates the last two points:

```
##### Alarm! Wrong code snippet! #####
else:
    print(1)
##### Alarm! Wrong code snippet! #####

##### Alarm! Wrong code snippet! #####
x, y = 1, 2
if x >= y:
    print(1)
else x < y:
    print(1)
##### Alarm! Wrong code snippet! #####
```

if-elif-else

Time for another bump in the level of complexity:

> Accept an integer as input from the user. If the number is greater than zero, print: `positive`. If the number is less than zero, print `negative`. If the number is equal to zero, print `zero`.

```
1  x = int(input())
2  if x > 0:
3      print('positive')
4  elif x == 0:
5      print('zero')
6  else:
7      print('negative')
8  # End of code
```

`elif` is a keyword in Python. It is a shorthand for else-if.

To understand how this works, let us consider three different inputs and the corresponding outputs.

Input	Output
x = 1	positive
x = 0	zero
x = -1	negative

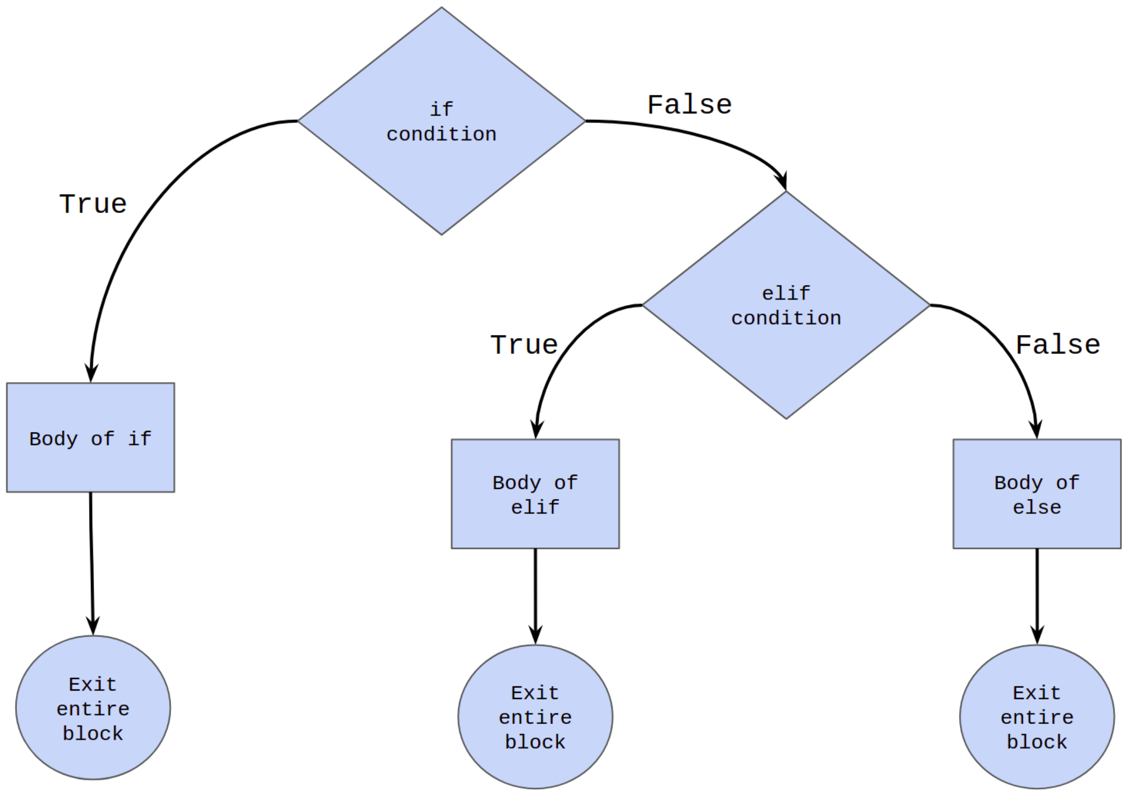
The entire `if-elif-else` block has three sub-blocks in it:

- if-block: lines 2-3
- elif-block: lines 4-5
- else-block: lines 6-7

This is the process followed by the interpreter in executing the `if-elif-else` block:

- If the if-condition evaluates to `True`, line-3 is executed and then the control transfers to line-8.
- If the if-condition evaluates to `False`, the control transfers to the elif-block. If the elif-condition evaluates to `True`, then line-5 is executed and then the control transfers to line-8.
- If the elif-condition is `False`, the control transfers to the else-block and line-7 is executed. As there are no more conditions to check, control naturally transfers to line-8.

A visual representation of the process is given below:



The general syntax:

```
if <condition-1>:
    <statement-1>
elif <condition-2>:
    <statement-2>
else:
    <statement-3>
```

Some features to note:

- Exactly one of the three statements gets executed.
- The moment either an `if` or an `elif` condition evaluates to `True`, the body of that block is executed and the flow exits out of the entire `if-elif-else` block.
- There could be multiple `elif` conditions after the `if`.
- An `else` condition cannot come before an `elif`. The final `else` block is not mandatory and can be removed. If the `else` is present, it can only come at the end.

Nested conditional statements

Consider the following problem:

> Accept three distinct integers as input from the user. If the numbers have been entered in ascending order, print `in ascending order`. If not, print `not in ascending order`.

An incomplete solution is given below:

```python
# Incomplete solution
x = int(input())
y = int(input())
z = int(input())

if x < y:
    print('in ascending order')
else:
    print('not in ascending order')
```

The problem with the above solution is that it doesn't check if `y < z`. So, for an input like `x, y, z = 1, 3, 2`, it will print `in ascending order`, which is incorrect. The complete solution is given below:

```python
 1  x = int(input())
 2  y = int(input())
 3  z = int(input())
 4
 5  if x < y:
 6      if y < z:
 7          print('in ascending order')
 8      else:
 9          print('not in ascending order')
10  else:
11      print('not in ascending order')
```

Whenever a new if-block is introduced, its body should have exactly one level of indentation with respect to its if-condition. Since line-7 makes up the body of the if-block starting at line-6, it has one level of indentation with respect to line-6. However, line-6 is already at the first level of indentation with respect to line-5, so line-7 has two levels of indentation with respect to line-5. According to the convention we have chosen, two levels of indentation will correspond to eight spaces.

Having a conditional statement inside another conditional statement is called nesting. The if-block from lines 5-9 forms the outer block. The if-else block from lines 6-9 forms the inner block. The `else` in line-8 is paired with the `if` in line-6 as they are at the same level of indentation. For similar reasons, the `else` in line-10 is paired with the `if` in line-5.

Defining variables inside `if`

Consider the following snippet of code:

```python
x = int(input())
if x % 5 == 0:
    output = 'the number is divisible by 5'
print(output)
```

Run the code multiple times, varying the input each time. What do you observe?

Whenever the input is a multiple of 5, the code runs without any error. When the input is not divisible by 5, the code throws a `NameError`. This is because, we are trying to reference a variable that has not been defined. The variable `output` is created only if line-3 is executed during run-time. Its mere presence in the code is not enough.