# Lecture4

September 6, 2024

# 1 Lecture 4: Flow control

### 1.0.1 Objective:

This lecture aims to introduce `flow control` mechanisms in Python, focusing on `conditional statements`, `loops`, and `exception handling`. We will review essential constructs like `if`, `else`, and `elif` for decision-making, as well as `for` and `while` loops for iteration. ****

**Flow control** in Python is essential for directing the flow of a program's execution, enabling dynamic and adaptable behavior. `Conditional statements`, such as `if`, `elif`, and `else`, allow you to execute different blocks of code based on specific conditions, facilitating complex decision-making processes. `Loops`, including `for` and `while`, provide mechanisms for repeating code execution, which is invaluable for tasks that involve processing collections of data or performing actions multiple times. Mastering these flow control constructs is key to writing efficient, readable, and flexible code, as they allow you to tailor your program's behavior to meet varying conditions and requirements.

## 1.1 0. Conditional Structure

### 1.1.1 0.0. Instructions with the if…else clause

Conditional statements are operations that the program executes when a logical expression evaluates to `True` or `False` (depending on the case). Conditional statements are defined within an `if...else` clause.

The general structure of an `if...else` block is as follows:

```
if logical_expression:
    statement1
    statement2
    ...
    statementn
else:
    other_block_of_statements
```

Note the : symbol following the `if` or `else` keyword, which indicates the end of the `if` or `else` declaration and the beginning of the statement definitions. Also, note that the `else` clause is optional, meaning there can be an `if` clause without an `else` clause.

It is crucial to observe the essential role of *indentation*, which delineates each block of statements, and the presence of colons after the condition and after the `else` keyword.

[33]:

```
# Instructions with a simple "if" clause
a = 150
if (a > 100):
    print("a exceeds 100")

print("================================================")
print('\t')


# Instructions with the if...else clause
x = 65
if (x > 70):
    print("x exceeds 70")
else:
    print("x does not exceed 100")
```

```
a exceeds 100
================================================
	
x does not exceed 100
```

### 1.1.2 0.1. Instructions with the if...elif...else clause

The `if...elif...else` clause is used when there are multiple logical conditions, each associated with instructions to execute when they are met.

```
x = 0
if x > 0 :
    print("x is positive")
elif x < 0 :
    print("x is negative")
else:
    print("x is zero")
```

[34]:

```
x = 0
if x > 0 :
    print("x is positive")
elif x < 0 :
    print("x is negative")
else:
        print("x is zero")
```

```
x is zero
```

**Note:** As demonstrated by the example above, the `elif` clause can be used as many times as there are alternative conditions between the initial `if` clause and the final `else` clause. However, using `elif` is not mandatory for handling alternative conditions. You can simply use additional `if` clauses instead.

### 1.1.3 0.2. Defining nested `if` clauses**

The previous examples illustrate first-level `if` clauses. In complex programs, it is very common to have nested `if` clauses, meaning that `if` clauses are defined inside other `if` clauses, sometimes at multiple levels. See the example below:

```
a = 15
b = 10

if a > b:
    print("a is greater than b")
    if a - b > 5:
        print("The difference between a and b is greater than 5")
    else:
        print("The difference between a and b is 5 or less")
else:
    print("a is not greater than b")
```

In this example, there is an `if` statement nested within another `if` statement. This structure allows for more granular control over the conditions and the corresponding actions in your program.

[35]:

```
a = 15
b = 10

if a > b:
    print("a is greater than b")
    if a - b > 5:
        print("The difference between a and b is greater than 5")
    else:
        print("The difference between a and b is 5 or less")
else:
    print("a is not greater than b")
```

```
a is greater than b
The difference between a and b is 5 or less
```

[36]:

```
embranchement = "rien"
if embranchement == "vertébrés":
    if classe == "mammifères":
        if ordre == "carnivores":
            if famille == "félins":
                print("Il s’agit peut-être d’un chat")
        print("C’est en tous cas un mammifère")
    elif classe == "oiseaux":
        print("c’est peut-être un canari")
print("La classification des animaux est complexe")
```

```
La classification des animaux est complexe
```

## 1.2 1. Loop Instructions

In Python, there are two main types of loop instructions:

1. `while...` **loops**: These loops execute a block of code repeatedly as long as a specified condition remains `True`. They are useful when the number of iterations is not known in advance and depends on dynamic conditions evaluated during execution.
2. `for... in...` **loops**: These loops iterate over a sequence (such as a list, tuple, or string) or other iterable objects. They are ideal for iterating through known sequences or ranges of values, allowing for clear and concise iteration over collections or ranges.

Both types of loops are fundamental for controlling the flow of execution and repeating tasks efficiently in Python programs.

### 1.2.1 1.0. while... loops

After introducing conditional structures, we turn our attention to loop structures: these structures allow a block of instructions to be executed multiple times. These repetitions fall into two categories:

- **Conditional repetitions**: The block of instructions is repeated as long as a condition is true.
- **Unconditional repetitions**: The block of instructions is repeated a specified number of times.

The general syntax for defining `while` loop instructions is as follows:

```
Initialize increment variable
while condition:
     block_of_instructions
     increment
```

As with the `if` structure, the condition is first evaluated, and if it is true, the block of instructions is executed. However, with the `while` loop, after executing the block of instructions, the condition is evaluated again. This process repeats until the condition becomes false. Therefore, it is essential to define an increment variable whose value changes after each execution so that the condition can eventually become false. This increment is necessary to prevent the instructions from being evaluated indefinitely, creating an infinite loop (or endless loop). Such a situation requires forcibly stopping the program's execution.

[37]:

```
x = 1 # Initialization of the variable x
while (x < 10):
    print('The value of x is ', x)
    x = x + 1 # Increment of the variable x
```

```
The value of x is  1
The value of x is  2
The value of x is  3
The value of x is  4
The value of x is  5
The value of x is  6
The value of x is  7
The value of x is  8
The value of x is  9
```

[38]:

```
fruits = ['pommes', 'oranges', 'fraises', 'bananes']
i = 0
while i < len(fruits):
    print (fruits[i])
    i = i + 1
```

```
pommes
oranges
fraises
bananes
```

### 1.2.2 1.1. Loop Instructions: `for... in...`

When we want to repeat a block of instructions a specified number of times, we can use a *counter*, which is a variable that counts the number of repetitions and controls the exit of the `while` loop. This is illustrated in the following example, where a function takes an integer `n` as an argument and prints the same message `n` times.

To perform such a repetition, we have a loop structure that saves us from initializing the counter (`i = 0`) and incrementing it (`i += 1`): this is the structure introduced by the `for` keyword.

The general syntax for defining `for... in...` loop instructions is as follows:

```
for element in sequence_of_values :
     block of instructions
```

[39]:

```
n = 5
for i in range(n):
    print('I repeat myself {} times.' ' (i={})'.format(n, i))
```

```
I repeat myself 5 times. (i=0)
I repeat myself 5 times. (i=1)
I repeat myself 5 times. (i=2)
I repeat myself 5 times. (i=3)
I repeat myself 5 times. (i=4)
```

[40]:

```
# Instruction dans une boucle « for.. in ... » simple
for i in range(1,11) :
    print(i)
```

```
1
2
3
4
5
6
7
8
```

```
9
10
```

[41]:

```
# Boucle « for.. in... » sur une chaine de caractères
listch = "Hello world"
for i in listch :
    print ( i )
```

```
H
e
l
l
o
 
w
o
r
l
d
```

[42]:

```
# Combining a `for...in...` Loop with an `if` Clause
listnb = [4, 5, 6]
for i in listnb:
    if i == 5:
        print("The condition is verified for the element", i)
    else :
        print("The condition is not verified for the element", i)
```

```
The condition is not verified for the element 4
The condition is verified for the element 5
The condition is not verified for the element 6
```

[43]:

```
# Combining a `for...in...` Loop with an `if` Clause
mych = "Hello World"
for lettre in mych :
    if lettre in "AEIOUYaeiouy":
        print ('La lettre', lettre, 'est une voyelle')
    else :
        print ('La lettre', lettre, 'est une consonne')
```

```
La lettre H est une consonne
La lettre e est une voyelle
La lettre l est une consonne
La lettre l est une consonne
La lettre o est une voyelle
La lettre   est une consonne
La lettre W est une consonne
La lettre o est une voyelle
La lettre r est une consonne
La lettre l est une consonne
La lettre d est une consonne
```

[44]:

```python
# Combining a `for...in...` Loop with an `if` Clause
# Considering the `space` character
mych = "Hello World"
for letter in mych:
    if letter in "AEIOUYaeiouy":
        print('The letter', letter, 'is a vowel')
    elif letter == " ":
        print("This is likely a space")
    else:
        print('The letter', letter, 'is a consonant')
```

```
The letter H is a consonant
The letter e is a vowel
The letter l is a consonant
The letter l is a consonant
The letter o is a vowel
This is likely a space
The letter W is a consonant
The letter o is a vowel
The letter r is a consonant
The letter l is a consonant
The letter d is a consonant
```

[45]:

```python
fruits = ['mangue', 'orange', 'pomme', 'banane']
costs = [49, 99, 15, 32]
for fruit, price in zip(fruits, costs):
    print("A", fruit, "costs", price, "Rwandan francs.")
```

```
A mangue costs 49 Rwandan francs.
A orange costs 99 Rwandan francs.
A pomme costs 15 Rwandan francs.
A banane costs 32 Rwandan francs.
```

[46]:

```python
costs = {'mangue': 49, 'orange': 99, 'pomme': 15, 'banane': 32}
for fruit, price in costs.items():
    print("A", fruit, "costs", price, "Rwandan francs.")
```

```
A mangue costs 49 Rwandan francs.
A orange costs 99 Rwandan francs.
A pomme costs 15 Rwandan francs.
A banane costs 32 Rwandan francs.
```

### 1.3 2. List Comprehension

In Python, a convenient and expressive syntax for creating lists is provided by list comprehensions. This method allows you to generate lists in a very concise way, often eliminating the need for explicit loops when elements need to be tested or processed before being included in the list.

The syntax for defining a list comprehension is similar to the mathematical notation for defining a set comprehension:

```
[expression for item in iterable if condition]
```

Here's a breakdown of the components: - `expression`: The value or transformation to apply to each item. - `item`: The variable representing each element in the iterable. - `iterable`: The collection or sequence you are iterating over. - `condition` (optional): A filter that determines which items to include.

**Example:**

Suppose you want to create a list of squares for all even numbers between 1 and 10. Using a list comprehension, you can achieve this concisely:

```
squares = [x**2 for x in range(1, 11) if x % 2 == 0]
print(squares)
```

**Explanation:** - `x**2` is the expression that computes the square of each item. - `x` is the item variable iterating over the numbers in `range(1, 11)`. - `range(1, 11)` is the iterable providing numbers from 1 to 10. - `if x % 2 == 0` is the condition filtering only even numbers.

The output will be:

```
[4, 16, 36, 64, 100]
```

This approach is both readable and efficient for generating lists based on specific criteria or transformations.

[47]:
```
liste = [2, 4, 6, 8, 10]

print([3*x for x in liste])
print("================================================")
print('\t')

print([[x, x**3] for x in liste])
print("================================================")
print('\t')

print([3*x for x in liste if x > 5])  # filtering with a condition
print("================================================")
print('\t')

print([3*x for x in liste if x**2 < 50])  # same as above
print("================================================")
print('\t')

liste2 = list(range(3))
print([x*y for x in liste for y in liste2])
```

```
[6, 12, 18, 24, 30]
================================================
[[2, 8], [4, 64], [6, 216], [8, 512], [10, 1000]]
==================================================

[18, 24, 30]
==================================================

[6, 12, 18]
==================================================

[0, 2, 4, 0, 4, 8, 0, 6, 12, 0, 8, 16, 0, 10, 20]
```

[48]:

```
# Here is an example of an efficient way to get a list of leap years within a given range:
leap_years = [year for year in range(2000, 2100) if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)]

print(leap_years)
```

```
[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036, 2040, 2044, 2048,
2052, 2056, 2060, 2064, 2068, 2072, 2076, 2080, 2084, 2088, 2092, 2096]
```

## 1.4 3. Break and Continue statements in `while...` or `for... in...` loops

The reserved keywords `break` and `continue` are used to alter the behavior of `for...in` or `while...` loops. The `break` statement allows for the termination of the loop and the exit from it prematurely, even if the main condition defining the loop remains true. On the other hand, the `continue` statement is used to skip the execution of the remaining instructions in the current iteration of the loop and proceed to the next iteration when the main condition is satisfied. The following examples illustrate their use.

```
# Loop with the break statement
for i in range(5):
    if i > 2:
        break
print(i)
```

[49]:

```
for i in range(5):
    if i > 2:
        break
print(i)
```

```
3
```

**Explanation:**

- The loop iterates over a range of values from 0 to 4.
- When `i` becomes greater than 2, the `break` statement is executed, which exits the loop immediately.
- Therefore, the loop stops, and the value of `i` at the time the loop is exited is printed. In this case, the output will be `3`.

```
# Loop with the continue statement
for i in range(5):
    if i == 2:
        continue
    print(i)
```

[50]:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

```
0
1
3
4
```

**Explanation:**

- The loop iterates over a range of values from 0 to 4.
- When `i` equals 2, the `continue` statement is executed. This causes the loop to skip the rest of the code inside the loop for this iteration and proceed to the next iteration.
- As a result, `2` is not printed. The output will be `0`, `1`, `3`, and `4`.

## 1.5 Further reading

0. Programming with Python; Section 13
1. Programming with Python; Section 16
2. Programming with Python; Section 27
3. Programming with Python; Section 28