DataScientest • com

Python for data-science

Loops

When building an algorithm, it often happens that we need to repeat the same lines of code several times. To this end, it is necessary to use **loops**, which will perform a series of operations as many times as necessary.

There are two types of loops in Python: The `for` and `while` loops.

1. The `while` loop

The **`while`** loop allows you to repeat a block of instructions **as long as** the starting condition is true.

For example, to determine the index of the word `"found"` in a list of words, you can iterate over all the indices in the list until you find the character string `"found"` :

```
# The wordlist in which we want to find the word "found".
sentence = ['The', 'while', 'loop', 'browses', 'all', 'the',
            'elements', 'from', 'the', 'list', "until", 'it',
            'has', 'found', 'what', 'it', 'seeks', '.']

# The variable i will store the starting index
i = 0

# As long as the word at the index where we are is different from "found"
while sentence[i] != "found":
    # We increase the value of i by 1 to go to the next index
    i + = 1

# The loop stops when we find the right word
print ("The word 'found' is at the index", i)
>>> The word 'found' is at the index 13
```

The general structure of a **`while`** loop is as follows:

```
while condition == True:
    instruction1
    ...
    instructionN

an_other_instruction
```

At each **iteration** of the **`while`** loop, the condition is evaluated. If the condition is true, the instruction block is executed, otherwise the loop ends

In [4]:

```
# Insert your code here
i = 1

while i < 11:
    print(i - 1)
    i += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

Hide solution

In [3]:

```
i = 1

# While i is less than 10
while i <= 10:
    # We print i
    print(i)

    # We increase i by 1
    i += 1
```

```
1
2
3
4
5
6
7
8
9
9
10
```

> We have a list of times performed by athletes in a 100m race. The results are **sorted in ascending order**.

- **(c)** Using a `while` loop, determine how many athletes achieved a time **lesser than 10s**.

In [7]:

```
results = [9.81, 9.89, 9.91, 9.93, 9.94, 9.95, 9.96, 9.97, 9.98, 10.03, 10.04, 10.05, 10.06, 10.08, 10.11, 10.23]

# Insert your code here
i = 0
n = 0
while i<len(results):
    if results[i]<10:
        n += 1
    else :
        pass
    i += 1
print(f'Number of Athletes less than 10s : {n}')
```

```
Number of Athletes less than 10s : 9
```

Hide solution

In [8]:

```
results = [9.81, 9.89, 9.91, 9.93, 9.94, 9.95, 9.96, 9.97, 9.98, 10.03, 10.04, 10.05, 10.06, 10.08, 10.11, 10.23]

# The variable n will count the number of athletes who have
# ran 100m in less than 10 seconds
n = 0

# The variable i will iterate through the indexes of the results list
i = 0

# While i is less than the length of the list
while i < len(results):
    # if the result of the athlete at index i is less than 10
    if results[i] < 10:
        # We increment n by 1
        n += 1

    # We increment i by 1 to go to the next athlete
    i += 1

print("The number of athletes with a time less than 10s is", n)
```

```
The number of athletes with a time less than 10s is 9
```

2. The `for` loop

The `for` loop allows to repeat an instruction block in a more controlled way. Indeed, it is not clear with a `while` loop how many times the loop will be executed.

The `for` loop is very **explicit** with respect to the variable that will be modified at each iteration. Moreover, the number of iterations the loop will perform will **always** be finite.

For example, we can use a `for` loop to display one by one the letters in the word "loop":

```
for letter in "loop":
   print(letter)
>>> l
>>> o
>>> o
>>> p
```

The general structure of a `for` loop is as follows:

```
for element in sequence:
    instruction1
    ...
    instructionN

other_instruction
```

The `for` loop executes the **instruction block** for each element in the **sequence**.

As with the `while` loop, lines outside the instruction block are not part of the loop, so they are only executed once when the loop is complete.

The actions take place in the following order:

- The `element` variable takes the value of the **first** element of `sequence` .
- The instruction block is executed.
- The `element` variable takes the value of the **second** element of the `sequence` .
- The instruction block is executed.
- ...
- ...
- The variable `element` takes the value of the **last** element of the `sequence` .
- The instruction block is executed and **the loop ends**.
- The `other_instruction` is executed.

The sequence object can be any type of **indexable** object such as a `list` , a `tuple` , a character string, etc.

In the **`for`** loop there is no need to change the `element` variable, Python takes care of it automatically. Be careful however not to forget in the syntax the **`in`** and **`:`** bits which are indispensable.

A teacher has undergraded his students, and wants to raise their grades so that the class average is above 10/20.

The students' grades have been included in the following list:

```
bad_marks = [0, 2, 3, 3, 3, 3, 4, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 8, 8, 8, 8, 8, 8, 9, 10, 10, 10, 11, 12, 14]
```

With the help of **`for`** loops :

- **(a)** Compute and display the class average. There are **30 students** in the class.
- **(b)** Create a `good marks` list where you will store the marks **increased by 4 points**. To do this you can create an empty list and then add the marks one by one using

In [14]:

```
bad_marks = [0, 2, 3, 3, 3, 3, 4, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 8, 8, 8, 8, 8, 8, 9, 10, 10, 10, 11, 12, 14]

# Insert your code here
# a)
sm = 0
for i in bad_marks:
    sm += i
print("Bad marks average :",sm/len(bad_marks))

# b)
good_marks = []
for i in bad_marks:
    i += 4
    good_marks.append(i)
#print(len(good_marks), good_marks)

# c)
sm = 0
for i in good_marks:
    sm += i
print("Good marks average :", sm/len(good_marks))
```

```
Bad marks average : 6.7
Good marks average : 10.7
```

Hide solution

In [12]:

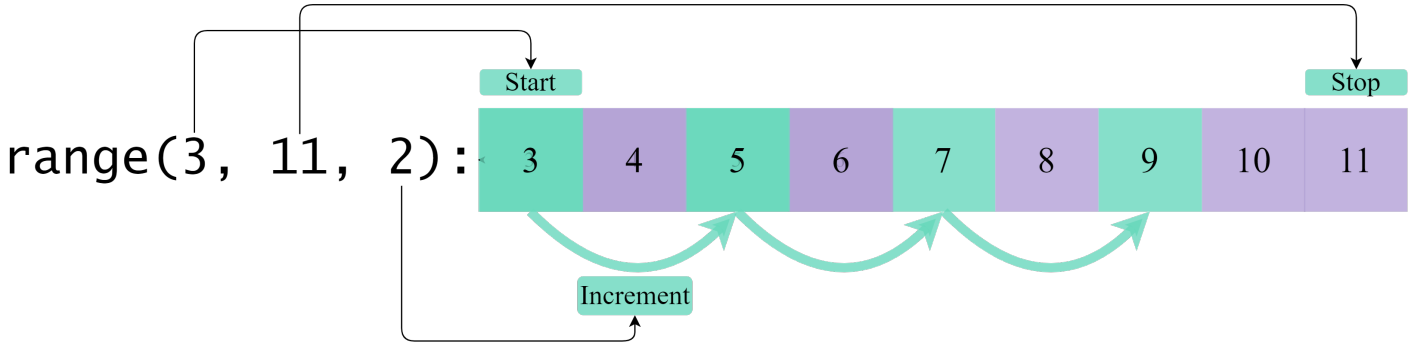
```
bad_marks = [0, 2, 3, 3, 3, 3, 4, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 8, 8, 8, 8, 8, 8, 9, 10, 10, 10, 11, 12, 14]

# Computation of the class' average mark

total = 0

# We sum all the students' marks
for mark in bad_marks :
    total += mark
# The sum of the grades is divided by the number of students to obtain the average
average = total / 30

print("Initial average:", average)

# The average is 6.7, so we add 4 points to each student:
good_marks = []

# For each mark in bad_marks
for mark in bad_marks :
    # We add to good_marks the mark increased by 4 points
    good_marks.append(mark + 4)

# Compute the new class average
total = 0
for mark in good_marks :
    total += mark
average = total / 30

print("Final average:", average)
```

```
Initial average: 6.7
Final average: 10.7
```

3. The **range** function

The **`range`** function is often used with `for` loops:

- It takes as argument a **start**, an **end** and a **step** value.
- It returns a sequence of numbers from the beginning value to the end value (the beginning number is included, but **the end number is excluded**) with the step defining the increment between two consecutive numbers.



By default the start is 0 and the step is 1.

Thus:

In [17]:

```
# The first two terms of the Fibonnaci Sequence
u = [0, 1]

# Insert your code here
for i in range(98):
    x = u[-1] + u[-2]
    u.append(x)
print(u)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025,
121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 1
02334155, 165580141, 267914296, 433494437, 701408733, 1134903170, 1836311903, 2971215073, 4807526976, 7778742049, 12586269025,
20365011074, 32951280099, 53316291173, 86267571272, 139583862445, 225851433717, 365435296162, 591286729879, 956722026041, 1548
008755920, 2504730781961, 4052739537881, 6557470319842, 10610209857723, 17167680177565, 27777890035288, 44945570212853, 727234
60248141, 117669030460994, 190392490709135, 308061521170129, 498454011879264, 806515533049393, 1304969544928657, 2111485077978
050, 3416454622906707, 5527939700884757, 8944394323791464, 14472334024676221, 23416728348467685, 37889062373143906, 6130579072
1611591, 99194853094755497, 160500643816367088, 259695496911122585, 420196140727489673, 679891637638612258, 110008777836610193
1, 1779979416004714189, 2880067194370816120, 4660046610375530309, 7540113804746346429, 12200160415121876738, 19740274219868223
167, 31940434634990099905, 51680708854858323072, 83621143489848422977, 135301852344706746049, 218922995834555169026]
```

Hide solution

In [16]:

```
# The first two terms of the Fibonnaci Sequence
u = [0, 1]

# For i going from 2 to 100
for i in range(2, 100):
    # We compute u_i with u[i-1] and u[i-2]
    u_i = u[i-1] + u[i-2]

    # We add u_i at the end of the list u
    u.append(u_i)
```

4. Nested loops

Loops can be nested inside one another. For example, when you have a list of lists, it is possible to browse all its elements with two nested loops.

The syntax is as follows:

```
# For each list in the list of lists
for list in list_of_lists:
```

In [1]:

```
text = ['The', '21', 'World', 'Cup', 'tournaments', 'have', 'been', 'won', 'by', 'eight',
        'national', 'teams.', 'Brazil', 'have', 'won', 'five', 'times', ',', 'and',
        'they', 'are', 'the', 'only', 'team', 'to', 'have', 'played', 'in', 'every',
        'tournament', '.', 'The', 'other', 'World', 'Cup', 'winners', 'are', 'Germany',
        'and', 'Italy', ',', 'with', 'four', 'titles', 'each', ';', 'Argentina', ',',
        'France', ',', 'and', 'inaugural', 'winner', 'Uruguay,', 'with', 'two', 'titles',
        'each', ';and', 'England', 'and', 'Spain', ',', 'with', 'one', 'title', 'each', '.']

# Insert your code here
sum_e = 0
for word in text:
    for char in word:
        if "e" in char:
            sum_e += 1
print(sum_e)
```

```
34
```

Hide solution

In [2]:

```
text = ['The', '21', 'World', 'Cup', 'tournaments', 'have', 'been', 'won', 'by', 'eight',
        'national', 'teams.', 'Brazil', 'have', 'won', 'five', 'times', ',', 'and',
        'they', 'are', 'the', 'only', 'team', 'to', 'have', 'played', 'in', 'every',
        'tournament', '.', 'The', 'other', 'World', 'Cup', 'winners', 'are', 'Germany',
        'and', 'Italy', ',', 'with', 'four', 'titles', 'each', ';', 'Argentina', ',',
        'France', ',', 'and', 'inaugural', 'winner', 'Uruguay,', 'with', 'two', 'titles',
        'each', ';and', 'England', 'and', 'Spain', ',', 'with', 'one', 'title', 'each', '.']

# The number of times the character 'e' appears
# will be stored in the variable n
n = 0

# For each word in the text
for word in text :
    # For each letter in word
    for letter in word :
        # If the letter is 'e'
        if letter == 'e':
            # We increase n by 1
            n += 1

print("The character 'e' appears", n, "times in the text.")
```

```
The character 'e' appears 34 times in the text.
```

5. List comprehension

List comprehension is an extremely interesting concept in Python which fits in with the central objective of simplifying code and increasing productivity.

Using the `for` loop syntax, it allows a very **compact and elegant definition of a list** of values.

We want to store the first 10 integers squared in a list. To do this, we could create an empty list and use a `for` loop as before:

```
my_list = []

# For i going from 0 to 9
for i in range(10):
    my_list.append(i**2)
```

Python allows us to reduce this syntax, thanks to list comprehension:

```
my_list = [i**2 for i in range(10)]
```

These two syntaxes are strictly **equivalent**.

In [5]:
```
# The list of numbers for the last two questions
numbers_list = [10, 12, 7, 3, 26, 2, 19]

# Insert your code here
# a)
powers_three = [3**x for x in range(10)]
#b)
double_list = [i*2 for i in numbers_list]
#c)
even_list = ["even" if i%2 == 0 else "odd" for i in numbers_list]
powers_three, double_list, even_list
```

Out[5]:
```
([1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683],
 [20, 24, 14, 6, 52, 4, 38],
 ['even', 'even', 'odd', 'odd', 'even', 'even', 'odd'])
```

Hide solution

In [ ]:
```
numbers_list = [10, 12, 7, 3, 26, 2, 19]

powers_three = [3**k for k in range(10)]

double_list = [number*2 for number in numbers_list]

even_list = ["even" if number%2 == 0 else "odd" for number in numbers_list]
```

6. The `enumerate` function

It is sometimes useful to have access to the **index** of an element in a sequence. To do this, it is possible to use the `enumerate` function in the statement of the `for` loop:

```
for index, element in enumerate(sequence):
    ...
    ...
```

For example, if we want to display the different positions of the word `"the"` in a text:

```
text = ["the", "word", "the", "is", "the", "word", "of", "which", "we", "search", "the", "position"]

# For each word in the text
for position, word in enumerate(text):
    # If the word is "the"
    if word == "the":
       # We display its position
        print(position)
>>> 0
>>> 2
>>> 4
>>> 10
```

- **(a)** Determine the index of the **maximum** of the list `L` using the `enumerate` function. To find this maximum, just store **the largest element seen** while iterating through the list.
- **(b)** Display the index and the value of the maximum of the list.

In [ ]:

```
L = [22, 65, 75, 93, 64, 47, 91, 53, 86, 53, 88, 17, 94, 39]

# Insert your code here
lst =  [0]
for index,value in enumerate(L):
    if value > lst[-1]:
        lst.append(value)
    else:
        lst.insert(0,value)

for i,v in enumerate(L):
    if lst[-1] == v:
        print(f"The maximum of the list is at the index {i} and value is {v}")
```

Hide solution

In [6]:

```
L = [22, 65, 75, 93, 64, 47, 91, 53, 86, 53, 88, 17, 94, 39]

maximum=0
max_index=0

# For each element in the list L
for index, element in enumerate(L):
    # If the element is greater than those we have seen before
    if element > maximum:
        # We overwrite the maximum with its value
        maximum = element
        max_index = index

print("The maximum of the list is at the index", max_index)
print("Its value is", maximum)
```

```
The maximum of the list is at the index 12
Its value is 94
```

7. The `zip` function

> The **`zip`** function allows you to iterate through **several sequences of the same length simultaneously** in a single **`for`** loop.
>
> The syntax is as follows:
>
> ```
> # At each iteration, we take an element of the first sequence and an element of the second
> for element1, element2 in zip(sequence1, sequence2):
>     ...
>     ...
> ```

In [7]:
```
incomes = [1200, 2000, 1500, 0, 1000, 4500, 1200, 500, 1350, 2200, 1650, 1300, 2300]
expenses = [1000, 1700, 2000, 700, 1200, 3500, 200, 500, 1000, 3500, 1350, 1050, 1850]

savings = []

# Insert your code here
for income, expense in zip(incomes,expenses):
    savings.append(income - expense)
savings
```

Out[7]: [200, 300, -500, -700, -200, 1000, 1000, 0, 350, -1300, 300, 250, 450]

Hide solution

In [8]:
```
incomes = [1200, 2000, 1500, 0, 1000, 4500, 1200, 500, 1350, 2200, 1650, 1300, 2300]
expenses = [1000, 1700, 2000, 700, 1200, 3500, 200, 500, 1000, 3500, 1350, 1050, 1850]

savings = []

for income, expense in zip(incomes, expenses):
    saving = income - expense
    savings.append(saving)

print(savings)
```

```
[200, 300, -500, -700, -200, 1000, 1000, 0, 350, -1300, 300, 250, 450]
```

Conclusion and recap

Loops are essential programming tools. They allow you to repeat instructions in a controlled manner.

In this notebook you have learned how to:

- Define a `while` loop which executes as long as the condition defining it is verified.
- Define a `for` loop that allows you to browse sequences.
- Define lists by **comprehension**, which is one of the most useful features tools in Python.
- Use the **`range`** function to iterate through lists of integers.