Hello World!

In Python, it is possible to display a chain of character, called a string, in the console output.
This can be done by calling the function **'print()'**, and by putting the desired string between the parenthesis.
Here is an example with the string **'Hello, World!'**.

In [1]:
```
print('Hello, World!')
```

```
Hello, World!
```

While being in a Jupyter notebook environnement, it is possible to display an object in the console output without using the print function.
For this, you only need to mention it in the last line of your Jupyter cell.

In [2]:
```
'Hello, World!'
```

Out[2]:
```
'Hello, World!'
```

Variable

Declaration

A variable is an **object** designed to store a **value**. Once it is declared, it can be accessed and used later by mentionning its name.
The declaration of a variable is done as follows:

```
variable_name = value
```

Thus, the 'Hello, World!' string can be stored in a variable. In this case we will call it s.
The previously declared variable can be displayed in the console output using the function 'print()', similarily to a regular string.

In [3]:
```
s = 'Hello, World!'
```

In [4]:
```
print(s)
```

```
Hello, World!
```

In [5]:
```
s
```

Out[5]:
```
'Hello, World!'
```

NB: Notice the difference between the variable name and the content of the string.
The content in the **string** is written between **quotes**, while the variable name doesn't have them.
Indeed, to be interpreted as a string, a chain of characters needs to be placed between quotes (' or "), while any chain of characters not placed between quotes will be interpreted as a variable name.

In [6]:
```
print(s)
print("s")
```

```
Hello, World!
s
```

In [7]:
```
print(Hello)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-85bf5114fa6b> in <module>
----> 1 print(Hello)

NameError: name 'Hello' is not defined
```

In this case, the Hello is **not placed between quotes**, so it is **considered as a variable name**.
However, no variable name Hello was declared, the Python interpreter cannot find it and gives us a **NameError**.

There are few rules to name your variables:

1. Names can not start with a number.
2. There can be no spaces in the name, you can use _ instead.
3. Can't use any of these symbols :'",<>/?|()!@#$%^&*~-+
4. Avoid using words that have special meaning in Python like "list" and "str"

Updating

As suggested by the name "variable", the value of a variable can be changed after its declaration.
This is done in a same fashion as the declaration:

```
variable_name = new_value
```

By doing so, the previous **value of the variable will be overwritten**, and any later reference to the variable will use the new value as a result.

In [3]:
```
s2 = 'Hello, World!'
print(s2)
s2 = 'Goodbye, World!'
print(s2)
```

```
Hello, World!
Goodbye, World!
```

In [4]:
```
s2
```

Out[4]:
```
'Goodbye, World!'
```

Deleting

Variables can be declared and their value can be updated, but they can also be deleted. In Python, this is done using the operator **del**, as follows:

del variable_name

In [5]:
```
# Deleting s2
del s2
```

In [6]:
```
s2
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-630081a5992e> in <module>
----> 1 s2

NameError: name 's2' is not defined
```

Now that s2 was deleted, the Python interpreter cannot find it and gives us a **NameError**.
Deleting unused variables is particularily **useful while handling big datasets**, since every variable declared is stored in the RAM of the computer, which is limited.
If the Python interpreter tries to use more RAM than what the computer can offer, it will result in a **MemoryError** and the **interpreter will crash**.

Numbers in Python

Types of numbers

Python has various "types" of numbers. We will focus on three of those types: the **integers**, **floating point numbers** (also called float), and **complex numbers**.

Integers

An integer is a round number, with **no decimals**.
Integers can be postive as well as negative.

Examples of integers are: -1000, -10, -3, 2, 5, 235000

In Python, integers are referred to as **int**.

In [1]:
```
print(type(3))
```

```
<class 'int'>
```

Floating point numbers

Floating point numbers are numbers that include a **decimal part**.
Similarily to integers, they can be both positive or negative.

Examples of floating point numbers are: -1.92, -0.01, 1.2, 2.5, 5.9999

In Python, floating point numbers are referred to as **float**.

In [2]:
```
print(type(-2.3))
```

```
<class 'float'>
```

Complex numbers

Complex numbers are numbers that have an **imaginary part**.
In mathematics, this imaginary part is usually represented by an "i". However, in Python, the imaginary part is represented by a "**j**".

In [3]:
```
# Pure imaginary number
4j
```

Out[3]:
```
4j
```

In [4]:
```
# Imaginary number with a real part
2+3j
```

Out[4]:
```
(2+3j)
```

In Python, complex numbers are referred to as **complex**.

In [5]:
```
print(type(2+3j))
```

```
<class 'complex'>
```

Arithmetic operations in Python

In Python, most common arithmetic operations can be applied on numbers using **operators**, such as **+**, **-**, *, **/**,...

In [6]:
```
# Additions
2 + 3
```

Out[6]:
```
5
```

In [7]:
```
#Substractions
3.1 - 0.5
```

Out[7]:
```
2.6
```

In [8]:
```
#Multiplications
11 * 53
```

Out[8]:
```
583
```

In [9]:
```
#Divisions
15 / 5
```

Out[9]:
```
3.0
```

NB: Notice that the result of the division comes out as a floating point number, even if the result doesn't have a decimal part.

In [10]:
```
print(type(15 / 5))
```

```
<class 'float'>
```

In Python, the classical operation priorities are conserved.

In [11]:
```
6 + 20 * 5 + 3
```

Out[11]:
```
109
```

Similarily, parenthesis can also be used to change the order of the operations

In [12]:
```
(6+20) * (5+3)
```

Out[12]:
```
208
```

Euclidian divisions

In Python, using the operator **/** while give us the result of the division of two numbers. Thus the result of the division of 15 by 7 is:

In [13]:
```
15 / 7
```

Out[13]:
```
2.142857142857143
```

However, if we want the result of the **Euclidian division** of 15 by 7, we need to use different operators.
The Euclidian division of x by y is such as **x = k*y + c**, where k is the floor division result, and c is the modulo.

In [14]:
```
#Floor division
k = 15 // 7
k
```

Out[14]: 2

In [15]:
```
#Modulo
c = 15 % 7
c
```

Out[15]: 1

In [16]:
```
7 * k + c
```

Out[16]: 15

Power

In Python, power is achieved using the operator **.

In [17]:
```
#Square
5 ** 2
```

Out[17]:
```
25
```

Knowing that square root is equivalent to power 0.5, square root can be done as follows:

In [20]:
```
#Square root
4 ** 0.5
```

Out[20]:
```
2.0
```

In [21]:
```
#Square root (alternative using division operator)
4 ** (1 / 2)
```

Out[21]:
```
2.0
```

Numbers and variables

Numbers from any type can be **stored in variables**, and as a result will be able to be accessed later.
This allows the **buffering of numerical values**, which will be particularily useful during calculation.

In [22]:
```
a = 10
b = 5.8
```

In [23]:
```
a
```

Out[23]:
```
10
```

In [24]:
```
b
```

Out[24]:
```
5.8
```

In [25]:
```
# Declare a complex number as a variable
z = 2 + 3j
```

In [26]:
```
# Accessing the real part of a complex number
z.real
```

Out[26]:
```
2.0
```

In [27]:
```
# Accessing the imaginary part of a complex number
z.imag
```

Out[27]:
```
3.0
```

Variables which store numerical values can be **used in mathematical operations**, by using the operators mentionned above.

In [28]:
```
# Example with an addition
a + b
```

Out[28]:
```
15.8
```

Being variables, their values **can be updated**. This characteristic can have a critical role in complex operations.
One of its most common use is to **increment** or **decrement** a variable.

Incrementation

In computer science, the term incrementation refers to an **increase of a variable** with the addition of a **constant value**.
For example, repetitvely adding 1 to a variable.

In [29]:
```
# Incrementing a by 1
a = a + 1
```

In [30]:
```
a
```

Out[30]:
```
11
```

This syntax is the most straightforward we could imagine, explicitely updating a variable by adding 1 to its previous value.
However, as most programing languages, Python offers a more efficient way to do it.

In [31]:
```
# Incrementing a by 1, but in Python style
a += 1
```

In [32]:
```
a
```

Out[32]: 12

Decrementation

In opposition to the incrementation, the decrementation refers to a **decrease of a variable** with the substraction of a **constant value**.
For example, repetitvely substracting 1 to a variable.

In [33]:
```
# Decrementing a by 1
a = a - 1
```

In [34]:
```
a
```

Out[34]:
```
11
```

Similarily to the incrementation, the decrementation has also an alternative syntax in Python

In [35]:
```
# Decrementing a by 1, but in Python style
a -= 1
```

In [36]:
```
a
```

Out[36]:
```
10
```

Generalization

This alternative syntax is not only specific to incrementation and decrementation, but can also be applied to **any mathematical operator**.

variable [operator]= numerical_value

In [37]:
```
# Dividing a by 5
a /= 5
```

In [38]:
```
a
```

Out[38]:
```
2.0
```

Exercise

Calculate the area of a circle that has a radius of r.

Booleans

Booleans are **logical units** that will reflect a state. In Python, they will be written **True** or **False**.
As any other type, their value can be assigned to a variable.

In [1]:
```
# Making a True
a = True
```

In [2]:
```
a
```

Out[2]:
```
True
```

In [3]:
```
# Making a False
a = False
```

In [4]:
```
a
```

Out[4]:
```
False
```

True and **False** have opposite values, and they can be inverted using the operator **not**.

In [5]:
```
# Opposite of True
not True
```

Out[5]:
```
False
```

In [6]:
```
#Opposite of a
not a
```

Out[6]:
```
True
```

In [7]:
```
# Define b as the opposite of a
b = not a
```

In [8]:
```
b
```

Out[8]:
```
True
```

Booleans and comparison

Booleans being logical units, they can be used to **compare values or variables**.
In Python, we can check if two values are equal using the **is** operator.

In [9]:
```
# Checking equality of two strings
"Hello" is "Goodbye"
```

Out[9]:
```
False
```

Alternatively, the inequality can be tested using the **is not** operator.

In [10]:
```
# Checking inequality of two strings
"Hello" is not "Goodbye"
```

Out[10]:
```
True
```

Numerical comparisons

While handling numerical values and variables, comparisons can be done using the following **operators**.

Comparison	Operator
Equality	==
Inequality	!=
Superiority	>
Superiority or equality	>=
Inferiority	<
Inferiority or equality	<=

Lists

A list is a **data structure** and more particularilly a **sequence**, meaning that it is an ensemble of elements brought together in a single object.
Lists are declared between "**[ ]**", with every element being separated by "**,**".

In [ ]:
```
# Assigning a list to a variable
my_list = [1, 2, 3]
```

In [ ]:
```
my_list
```

Out[ ]:
```
[1, 2, 3]
```

Every element in a list is **indexed**, meaning that they are associated to a number, referring to their order.
In Python, as in most programming languages, indexing **starts at 0**.
Elements can be accessed as following:

In [ ]:
```
# Displaying the first, second and third element of a list
print(my_list[0])
print(my_list[1])
print(my_list[2])
```

```
1
2
3
```

Lists are **mutable**, which means that all of its elements **can be modified**.

In [ ]:
```
# Updating the  third element of a list
my_list[2] = 5
my_list
```

Out[ ]:
```
[1, 2, 5]
```

Indexing a list using a positive index will count the index from the start, while giving a **negative index** will count them **from the end of the list**.
As an example, the last element of a list can be accessed with the index -1.

In [ ]:
```
# Displaying the third element of a list from the end
my_list[-3]
```

Out[ ]:
```
1
```

If you try to access an index that is **not in a list**, the Python interpreter will not find it and give an **IndexError**.

In [ ]:
```
my_list[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-452c52ef7c16> in <module>
----> 1 my_list[100]

IndexError: list index out of range
```

Concatenation

You can **concatenate** multiple lists using the operator **+**.

In [ ]:
```
my_list = [1, 2, 3] + [4, 5]
my_list
```

Out[ ]:
```
[1, 2, 3, 4, 5]
```

List length and list slicing

The number of elements in a list can be determined using the function **"len()"**.

In [ ]:
```
len(my_list)
```

Out[ ]:
```
5
```

Slicing means creating a **subset based on indexing**. It is possible to slice a list by specifiying the **first element** of the list that has to be kept followed by the **first one that should be rejected** (last index to be kept +1) using the following syntax:

list[start:end]

In [ ]:
```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [ ]:
```
# Slicing a list, keeping from the 3rd element to the 6th
my_list[2:6]
```

Out[ ]:
```
[3, 4, 5, 6]
```

WARNING: keep in mind that **indexing starts at 0** (first element is considered index 0) and that while slicing, the end index is the **first element to be rejected**.

If the **first index is not provided** during slicing, it will consider that it is 0 and **slice the list from the start**.
If the **end index is not provided**, it will **slice the list to the end**.

In [ ]:
```
# Slicing a list, keeping from the 1st element to the 5th
my_list[:5]
```

Out[ ]:
```
[1, 2, 3, 4, 5]
```

In [ ]:
```
# Slicing a list, keeping from the 5th element to the end
my_list[5:]
```

Out[ ]:
```
[6, 7, 8, 9, 10]
```

In [ ]:
```
# Slicing a list, keeping from the 2nd element to the 3rd element from the end
my_list[2:-2]
```

Out[ ]:
```
[3, 4, 5, 6, 7, 8]
```

Element checking

It is possible to know if an element is in a list using the operator **in**

In [ ]:
```
#Checking if a number is in a list
100 in my_list
```

Out[ ]:
```
False
```

In [ ]:
```
#Checking if a number is not in a list
100 not in my_list
```

Out[ ]:
```
True
```

List nesting

In Python, data structures such as lists support **nesting**, which means that they can be host other data structures as elements.
For example, **a list can contain other lists**.

In [ ]:
```
# Creation of a list composed of three other lists
list_1=[1, 2, 3]
list_2=[4, 5, 6]
list_3=[7, 8, 9]
new_list = [list_1, list_2, list_3]
new_list
```

Out[ ]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Lists built-in functions

Append

The **"append()"** function allows to add a **new element** at the **end of a list**.

In [1]:
```
# Adding 5 at the end of a list
my_list = [6, 1, 3, 5, 7, 2, 4, 5]
my_list.append(5)
print(my_list)
```

```
[6, 1, 3, 5, 7, 2, 4, 5, 5]
```

Index

The **"index()"** function will return the first index of an element in a list. However, if the element is not in the list, it will give an **ValueError**.

In [2]:
```
# Find the index of 5
my_list = [6, 1, 3, 5, 7, 2, 4, 5]
my_list.index(5)
```

Out[2]:
```
3
```

Tuples

Tuples are also **data structures**, and as a matter of fact they are very **similar to lists**. But, in opposition to lists, they are **immutable**, which means that once the are declared, they **cannot be modified**.
As a result, they will share most of lists features and built-in functions, except the ones that are imply modification (i.e. "append()", "sort()", remove()",...).
Tuples are declared between **"( )"**.

In [ ]:
```
# Assigning a tuple to a variable
my_tuple = (1, 2, 3)
my_tuple
```

Out[ ]:
```
(1, 2, 3)
```

In [ ]:
```
# Displaying the second element of a tuple
print(my_tuple[1])
```

```
2
```

Sets

Sets are also **data structures**, but they are only composed of **unique elements**. This means that each element can only occur once inside the set.
NB: The elements are not ordered inside a set.

Other **data structures can be converted to sets** using the **"set()"** function.

In [ ]:
```
my_list = [6, 1, 3, 5, 7, 2, 4, 6, 6, 6, 7]
my_set = set(my_list)
my_set
```

Out[ ]:
```
{1, 2, 3, 4, 5, 6, 7}
```

And conversively, a set can be converted to a list.

In [ ]:
```
new_list = list(my_set)
new_list
```

Out[ ]:
```
[1, 2, 3, 4, 5, 6, 7]
```

Range

Range are a particular kind **data structures**, they are used to create an ordered list of number, in an ascending order. As a result, they are composed of **ordered unique elements**. They are also **immutable**.
They are declared using the function **"range()"**. This function has three arguments, the **first number to start** the range, the **first number to be rejected** (will not be included in the range), and the **increment**.

In [ ]:
```
# Creates a range that starts at 4, will stop at 12 and is incrementing by 2.
x = range(4, 12, 2)
x
```

Out[ ]:
```
range(4, 12, 2)
```

Displaying a range will not show the numbers that are contained inside. For this, it needs to be converted to a list using the **"list()"** function.

In [ ]:
```
list(x)
```

Out[ ]:
```
[4, 6, 8, 10]
```

It is **not necessary to provide the three arguments** to the range. For example, if the **incrementing argument** is not provided (only the start and the end are), the function will consider it **1 by default**.

In [ ]:
```
# Creates a range that starts at 4, will stop at 12 and is incrementing by 1.
x = range(4, 12)
list(x)
```

Out[ ]:
```
[4, 5, 6, 7, 8, 9, 10, 11]
```

Similarily, the **start argument is optional**. If it is not provided, it will be considered as **0 by default**.

In [ ]:
```
# Creates a range that starts at 0, will stop at 12 and is incrementing by 1.
x = range(12)
list(x)
```

Out[ ]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

As a matter of fact, only the **end argument is necessary**.

Strings and Characters in Python

In computer sciences, a **character** will refer to letters, numbers or symboles. A chain of characters will be called a **string**.
Strings are declared between " or between ', and refered to as **str**.

In [46]:
```
# Declaration of a string using ""
type("Hello, World!")
```

Out[46]:
```
str
```

In [47]:
```
# Declaration of a string using ''
type('Hello, World!')
```

Out[47]:
```
str
```

There is **no practical difference** between strings declared using " and ', as long as they have the same value.

In [48]:
```
# Comparison between strings declared with "" and ''
"Hello, World!" is 'Hello, World!'
```

Out[48]:
```
True
```

In opposition to other programming languages, **Python does not have a character type**.
So any isolated character will be considered a **string**.
Thus characters are of a lesser importance.

In [49]:
```
# Initiation of a single character using ''
type('c')
```

Out[49]:
```
str
```

However, characters can still be useful.
It is possible to know the **Unicode value** of a character using the function **'ord()'**.
And conversively, you can determine the character corresponding to a Unicode value using the function **'chr()'**

In [50]:
```
# Unicode value of character c
ord('c')
```

Out[50]:
```
99
```

In [51]:
```
# Character corresponding to Unicode value 99
chr(99)
```

Out[51]:
```
'c'
```

In [52]:
```
# But the function chr still returns a string
type(chr(99))
```

Out[52]:
```
str
```

The importance of the backslash

Escaping

One of the most critical aspect of declaring a string is to pay attention to the content of the string.
For example, if a string is declared with ', what will happen if the string itself contains a '?

In [53]:
```
string = 'I'm very happy to learn Python'
```

```
  File "<ipython-input-53-efc1f3453059>", line 1
    string = 'I'm very happy to learn Python'
                ^
SyntaxError: invalid syntax
```

The simplest answer would be to use the other option to declare the string.
In this case ".

In [54]:
```
string = "I'm very happy to learn Python"
string
```

Out[54]:
```
"I'm very happy to learn Python"
```

Works like a charm! However, there is a more universal solution to this problem.
Indeed, we can use **escaping**. For that, we just need to use the character \ before the ' or the ". It will then be interpreted as the corresponding character.

In [55]:
```
string = 'I\'m very happy to learn Python'
string
```

Out[55]:
```
"I'm very happy to learn Python"
```

Formatting

The backslash is also used to perform **formatting** within a string.

In [56]:
```
# Using \t for tab
"Hello! \tGoodbye!"
```

Out[56]:
```
'Hello! \tGoodbye!'
```

Interestingly, it does nothing here. That is because in **interactive mode** (Jupyter Notebook or Python console), displaying a string only by calling it will only display its value.
For the formatting to be considered, the string needs to be displayed with the function **"print()"**.

In [57]:
```
# Using \t for tab, and seeing the result
print("Hello! \tGoodbye!")
```

```
Hello!  Goodbye!
```

In [58]:
```
# Using \b for backspace
print("Hello! \bGoodbye!")
```

```
Hello! Goodbye!
```

In [59]:
```
# Using \n for new line
print("Hello! \nGoodbye!")
```

```
Hello!
Goodbye!
```

In [60]:
```
# Using \r for carriage return
print("Hello!\rGoodbye!")
```

```
Goodbye!
```

If \ as a particular meaning within a string, it means that having a backlash in your string can cause problems.

In [61]:
```
print("The date is 30\11\2020 ")
```

```
The date is 30  0
```

The solution is to **escape the backslash** as well, writing two consecutive backslashes.

In [62]:
```
print("The date is 30\\11\\2020 ")
```

```
The date is 30\11\2020
```

String comparison

It is possible to know if two strings are equivalent using the **is** operator.

In [63]:
```
#Checking equality
"Hello" is "Goodbye"
```

Out[63]:
```
False
```

In [64]:
```
#Checking inequality
"Hello" is not "Goodbye"
```

Out[64]:
```
True
```

It is possible to know if a substring is inside a string using the **in** operator.

In [65]:
```
#Checking if a substring is in a string
"friend" in "Hello, my friend!"
```

Out[65]: True

In [66]:
```
#Checking if a substring is not in a string
"friend" not in "Hello, my friend!"
```

Out[66]: False

String manipulation

String length and string slicing

A string can practically be considered as a **list of characters**.
As such, the function **"len()"** can be used to determine the number of characters included in a string.

In [67]:
```
# Displaying number of characters in string
len("foo")
```

Out[67]:
```
3
```

Similarilly as a list, a string can be **sliced** using indices corresponding to characters.

In [68]:
```
# Keeping letters 3 to 6 of a string
string = "foobar"
string[2:6]
```

Out[68]:
```
'obar'
```

It is also possible to slice a string and keep only one letter in a given step.

In [69]:
```
# Keeping letters 2 to 8 of a string, keeping only 1 letter out of 3
string = "foobarfoo"
string[1:8:3]
```

Out[69]:
```
'oao'
```

Concatenation

String concatenation is the action of **putting strings back to back**, literally adding one to the other.
In Python, this is done using the operator **+**.

In [70]:
```
# Concatenating two strings
'foo' + 'bar'
```

Out[70]:
```
'foobar'
```

Duplication

If a string needs to be duplicated a certain amount of time, the operator * can be used.

Interesingly, Python accepts a negative number of the duplication. However, it will return an empty string

In [72]:
```
'foo' * -2
```

Out[72]:
```
''
```

Replacing

Part of a string can be replaced using the function **"replace()"** using the following syntax:

In [1]:
```
string = "Yesterday is a good day"
value_to_replace = "is"
new_value = "was"
string.replace(value_to_replace, new_value)
```

Out[1]:
```
'Yesterday was a good day'
```

String built-in functions

Format

The **"format()"** allows to easily **insert values into a string**. For this, their desired position in the string needs to be placed within **{}**.

In [4]:
```
"The year is {}".format(2020)
```

Out[4]:
```
'The year is 2020'
```

Also, this function allows to **nominatively place** those values. For this, you just have to put a name inside the {}.

In [5]:
```
"The date is {day}/{month}/{year}".format(day=30, month=12, year=2020)
```

Out[5]:
```
'The date is 30/12/2020'
```

Lower case

The **"lower()"** converts all characters in the string to lower case.

In [2]:
```
# Puts the string in lower case
string = "HELP, MY CAPS LOCK KEY IS BROKEN"
string.lower()
```

Out[2]:
```
'help, my caps lock key is broken'
```

Split

The **"split()"** function breaks down a string every given character into a list of substrings.

In [4]:
```
# Splits the string every '/'
string = '30/11/2020'
string.split('/')
```

Out[4]:
```
['30', '11', '2020']
```

Join

Alternatively, it is possible to concatenate every string in a list, separated by a given substring by using the **"join()"** function. However, having a non string element in the list will lead to an error.

In [89]:
```
list_to_join = ['April', 'May', 'June']
'/'.join(list_to_join)
```

Out[89]:
```
'April/May/June'
```

Dictionaries

Dictionaries are an other forme of **data structure**. But in opposition to lists or tuples, they store information in pairs, with a **key** and a **value**.
They can be declared using **{ }**.

In [3]:
```
# Declaring an empty dictionary
my_dict = {}
print(my_dict)
```

```
{}
```

They are **mutable** and **unordered**.
As a result, we can now insert data in the declared dictionary. This is done by specifiying the **key** and the **value** we want to insert in the dictionary, as follows:

dictionary[key] = value

In [4]:
```
# Adding pairs of keys and values to the dictionary
my_dict['Age'] = 25
my_dict['Job'] = 'PhD Student'
print(my_dict)
```

```
{'Age': 25, 'Job': 'PhD Student'}
```

Accessing a value in a key-pair value can be done by calling the key within the dictionary.

In [5]:
```
# Accessing the value associated to the key 'Age'
my_dict['Age']
```

Out[5]:
```
25
```

In [6]:
```
# Declaring a dictionary with key-value pairs
my_new_dict = {'Age': 25, 'Job': 'PhD Student'}
print(my_new_dict)
```

```
{'Age': 25, 'Job': 'PhD Student'}
```

Dictionaries **do not tolerate duplicates of keys**. As such, if you enter a new value with a key that was already in the dictionary, it will only update the value associated to the key in the dictionary.

In [7]:
```
# Trying to add a new 'Age' key-value pair
my_dict['Age'] = 26
print(my_dict)
```

```
{'Age': 26, 'Job': 'PhD Student'}
```

Dictionary nesting and subkeys

As other **data structures**, dictionaries allow **nesting**, can have other data structures as elements, including other dictionaries.
To access a value in a dictionary within a dictionary, the **keys need to be called consecutively**.

In [10]:
```
# Accessing the value associated to the key 'Programming' within the key 'Skill
s'
my_new_dict = {'Age': 25, 'Skills': {'Programing':'Python', 'Software':'Excel
'}}
my_new_dict['Skills']['Programing']
```

Out[10]:
```
'Python'
```

Dictionary built-in functions

Keys

To retrieve the keys in a dictionary, use the **"keys()"** function.

In [1]:
```
# Retrieve the keys of a dictionary
my_new_dict = {'Age': 25, 'Job': 'PhD Student'}
my_new_dict.keys()
```

Out[1]:
```
dict_keys(['Age', 'Job'])
```

Values

To retrieve the values in a dictionary, use the **"values()"** function.

In [2]:
```
# Retrieve the values of a dictionary
my_new_dict = {'Age': 25, 'Job': 'PhD Student'}
my_new_dict.values()
```

Out[2]:
```
dict_values([25, 'PhD Student'])
```

Items

To retrieve the key-value pairs in a dictionary, use the **"items()"** function.

In [3]:
```
# Retrieve the values of a dictionary
my_new_dict = {'Age': 25, 'Job': 'PhD Student'}
my_new_dict.items()
```

Out[3]:
```
dict_items([('Age', 25), ('Job', 'PhD Student')])
```

If-else statements

If

The operator **if** allows the execution of a **block of code**, only if it is followed by a boolean having the value **True**.

In [1]:
```
# Example of a block of code executed in an if statement
if True:
    print('The statement was True!')
```

```
The statement was True!
```

A condition can follow the if operator, and the associated block of code will be executed if this condition is true.

In [2]:
```
# Example of an if statement
if 2 < 10:
    print("Wow, that is incredible!")
```

```
Wow, that is incredible!
```

If the condition needs to be false for the code to be executed, the operator **"not"** can be added before the condition.

In [3]:
```
if not 'H' in 'Goodbye':
    print('Goodbye')
```

```
Goodbye
```

Condition chaining

Multiple conditions can be added after an if.
In the case where you want them to be true at the same time to execute a piece of code, you can use the operator **"and"**.
Also, if you only need one or more of them to be true, you can use the operator **"or"**.

In [4]:
```
# Example of an if with two conditions and an "and"
if 2 < 10 and 'H' in 'Hello':
    print("Python is amazing!")
```

```
Python is amazing!
```

Else

In the case you want to execute a different block of code if the condition is true or false, you can use **"else"** following an if statement.

In [6]:
```
# Example of an if statement followed by an else
if 'H' in 'Goodbye':
    print('Goodbye')
else:
    print("Do you even know how to read?")
```

```
Do you even know how to read?
```

Elif

In Python, there is a possibility of creating more complicated conditional structures. For example, if you want to execute a given block of code if a condition is true, but if it is false you want to **verify an other condition to execute an other block of code**, you can use **"elif"**.

In [7]:
```
# Example of an if block followed by an elif
# The first if checks if x is lesser than 10,
# Otherwise the elif checks if x can be divided by 2.
x = 1024
if x < 10:
    print("x is to low")
elif x % 2 == 0:
    print("x can be divided by 2")
```

```
x can be divided by 2
```

NB: As a sidenote, since if, elif and else use boolean values, you can **store your statements in variables** and use them in if blocks later.

In [13]:
```
x = 2
statement =  x < 10
print(statement)
```

```
True
```

In [14]:
```
if statement:
    print('Statement is True')
```

```
Statement is True
```

Exercise

Create an if-else block that, for a given input month will return the season of the year corresponding to it.
The input name can be either a string (name of the month), or an integer (number of the month).

For loops

We saw that some **data structures** were a collection of elements.
Those data structures are called **iterables**, meaning that you can separate all of its elements and **sequentially execute a block of code** on all of them using a **for loop**.

In [1]:
```
# Iterates over all elements of a list and print them
my_list = [5, 4, 3, 2, 1, "Happy New Year!"]

for element in my_list:
    print(element)
```

```
5
4
3
2
1
Happy New Year!
```

In this situation, a loop is created and will iterate through the list. At every step of that loop, the variable "element" takes the value of the next element of the list.
At then end of the loop, the execution of the rest of the code is resumed.

Exercise

Compute an approximation of pi using Wallis' product:

$\pi = 2\prod_{n=1}^{\infty} \frac{4n^2}{4n^2 - 1}$

NB: You can use the ranges, that are declared using the **range()** function.

While loops

While loops are an other type of loops that will iteratively run a block of code.
However, in opposition to for loops, they do not rely on an iterable, but rather on a **condition**.
They will execute a block of code **as long as the condition they rely on is true**.

In [3]:
```
# Print a message while the variable i is inferior to 3.
i = 0

while i < 3:
    print("I iterate")
    i += 1
print("I sleep")
```

```
I iterate
I iterate
I iterate
I sleep
```

Exercise

We are almost done with loops! Now to show that you understood while loops, try to create one that will duplicate the string 'Hello' until the total number of characters exceeds 50.
After this, display the result.

Break

Loops can take a while to execute. But more importantly, they are situations where you want to **exit the loop** at a certain moment (i.e. if a threshold is crossed).
For this, you can use **"break"** within a loop.

In [5]:
```
i = 0

# An infinite loop of incrementation
while True:
    i += 1

    # Breaks the loop if i has crossed a threshold
    if i >= 1000:
        print(i)
        break
```

```
1000
```

Functions

A function is a **block of code** stored inside an object that, once declared, can be called when wanted.
Functions are declared using **"def"** followed by the name of the function, and **()**, as follows:

In [9]:
```
def say_hello():
    print('Hello')
```

In [10]:
```
say_hello()
```

```
Hello
```

Parameters

The **brackets** used during the declaration and the calling of a function have a very specific purpose, they allow the use of **parameters**.
Parameters are variables that will be passed to a function, and that will be **useable within the code block** of the function.

In [3]:
```
def add(a, b):
    print(a)
    print(b)
    print(a + b)
```

In [4]:
```
add(2, 3)
```

```
2
3
5
```

NB: Notice that the parameters are **ordered**.

Parameters can be set to have a **default value**. If so, the function can be called without giving a value to these parameters, and thus will take their default values.
NB: Parameters that have default values should be **placed last** while **declaring the function**.

In [5]:
```
def multiply(a, b=2):
    print(a)
    print(b)
    print(a * b)
```

In [6]:
```
multiply(8)
```

```
8
2
16
```

The parameters are ordered by default. However, the parameters can be assigned **out of order** if they are **called with their name**.
By doing so, you directly create the variables that will be used within the block of code, thus their order do not matter anymore.
NB: Parameters that are assigned by name **should be assigned last** while **calling the function**.

In [7]:
```
multiply(b=3, a=1)
```

```
1
3
3
```

Return value

Ultimately, a function can return a value. This is done using **"return"** at the end of the block of code.

In [8]:
```
def divide(a, b=2):
    return a / b
```

In [9]:
```
divide(8, 2)
```

Out[9]:
```
4.0
```

NB: It is not necessary to put the **"return"** at the end of the block of code. For example, you can put it several times within a if-else block, as you want to return different values. However, remember that executing "return" will stop the execution of a function.

In [12]:
```
def divide(a, b=2):
    if b == 0:
        return "Wait... What?"
    else:
        return a / b
```

In [13]:
```
divide(8, 0)
```

Out[13]:
```
'Wait... What?'
```

Exceptions

In computer sciences, things can go sideways pretty fast. The good news is that you can prevent this using **exceptions**.
You can see the exceptions as a **shield** that will protect a part of your code and **prevent crashing**. If an error was supposed to happen during the execution of your code (i.e. IndexError), the execution of the code will be stopped, and an other block of code can be executed.

In [14]:
```
a = 8
b = 0

try:
    # Sensitive block of code
    c = a / b
except:
    # Code to be executed if the sensitive code fails
    print('Operation cannot be performed')

# The code continues
print("Hey, my code didn't break!")
```

```
Operation cannot be performed
Hey, my code didn't break!
```

Exercise 1: Capitalization

Remember the "capitalize()" built-in function of the strings? Try to recreate it by doing something similar, a function that will take the first letter of a string and make it upper case.

Hint 1 : You can use other strings built-in functions
Hint 2 : Google is your best friend
Bonus point if you manage to deal with the case where the input is not a string

In [3]:
```
def capitalize(string):
    if type(string) == str:
        first_char = string[0]
        return first_char.upper() + string[1:]
    else:
        return "This is not a string"

my_string = 'hello, how are you?'
capitalize(my_string)
```

Out[3]:
```
'Hello, how are you?'
```

Exercise 2: Fibonacci sequence

The Fibonacci sequence is a sequence of integers such as the nth element of the sequence is the sumn of the n-1th and n-2th element. Make a function that will return a list containing the Fibonacci sequence up to a certain number, that will be the input argument of the function.

In [7]:

```
def fibonacci(limit):
    n0 = 0
    n1 = 1
    list_fibonacci = [n0, n1]
    while n0 + n1 < limit:
        nth = n0 + n1
        n0 = n1
        n1 = nth
        list_fibonacci.append(nth)
    return list_fibonacci

fibonacci(10)
```

Out[7]:

```
[0, 1, 1, 2, 3, 5, 8]
```

Exercise 3: Decryption

You were contacted by the CIA, who needs help to decipher crypter transmissions to prevent a catastrophic event.
They have the decryption table, however none of their agents is skilled enough in programming to pull off an automatic way to decrypt those numbers.
They heard about your skills in Python and they are asking your help to make a function that will decrypt any input string that contains only letters into the corresponding number using the following table.
The survival of the world lies in your hands, and on your keyboard. **Good luck**!

Decrypted Character	Crypted Number
a	14
b	15
c	16
d	17
e	18
...	...

In [28]:

```
import string
alphabet = string.ascii_lowercase
print(alphabet)

coded_string = '21/18/25/25/28/36/28/31/25/17'

def decrypt(my_string):
    # Separates numbers
    list_numbers = my_string.split("/")

    # Decrypts
    code = []
    for number in list_numbers:
        code.append(alphabet[int(number)-14])
    code = ''.join(code)
    return code

decrypt(coded_string)
```

```
abcdefghijklmnopqrstuvwxyz
```

Out[28]:

```
'helloworld'
```

Exercise 4: Artificial Intelligence

Guessing a number between 0 and 100 is a simple game, we all know how to play it. But you feel kinda lazy, you would prefer your computer to play it for you!
Try to create an artificial intelligence (well... a function) that will automatically solve this game, and display the number of trials it took to solve it.

In [31]:

```
def check_guess(guess, target):
    "Check if we guessed the correct value"
    if guess < target:
        return "lower"
    elif guess > target:
        return "higher"
    else:
        return "correct !"


def guess(target, min_value=0, max_value=100):
    "Guess a target number between 0 and max_value"
    attempts = 0
    output = ""

    while output != "correct !":
        # TODO
        my_guess = min_value + int((max_value - min_value) / 2)
        output = check_guess(my_guess, target)
        if output is "lower":
            # If the guess is inferior to the number, updates inferior limit
            min_value = my_guess + 1
        elif output is "higher":
            # If the guess is superior to the number, updates superior limit
            max_value = my_guess - 1
        attempts += 1
    print("Attempt number : {}".format(attempts))

guess(50)
guess(25)
guess(0)
guess(100)
```

```
Attempt number : 1
Attempt number : 6
Attempt number : 6
Attempt number : 7
```