PW 2 : Python - Conditions, functions

1- Conditions

1.1 - Booleans

Boolean variables were introduced in PW1. As a reminder, these are variables which can only take 2 values, `True` or `False` .

Exercise: In a new cell, assign the value `False` to a variable `test` and display its content.

Exercise :

In [1]:
```
test=False
print(test)
```
```
False
```

1.2 - Comparison operators

The python language includes several operators (`==` , `<=` , `>=` ...) to perform tests whose result is a boolean which is 'True' if the condition is true, `False` if it is not. For example, the `==` operator is used to test if two objects are equal (as in `75 == 24`) or identical (as in `"Jacques" == "Joe"`). Similarly, the `<=` operator tests whether one number is less than or equal to another.

Exercise: Display the result of the test `"James" == "Joe"` and `"James" == "James"` . Define a `test2` variable containing the value 25. Test if this variable is equal to 26. Test if this variable is less than 50. Test if this variable is greater than 50. Display the test results.

Exercise :

In [2]:
```
"James" == "Joe"
```
Out[2]:
```
False
```

In [3]:
```
"James" == "James"
```
Out[3]:
```
True
```

In [4]:
```
test2 = 25
test2 == 26
```

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

In [5]:
```
test2 < 50
```

Out[5]:
```
True
```

In [6]:
```
test2 > 50
```

Out[6]:
```
False
```

1.3 - Tests Combination

Tests can be combined using the pseudo-operators `and` and `or` . For `and` , the result of the combination is `True` if **both** tests are `True` , otherwise it is `False` . For the second, the result of the combination is `True` if **at least one** of the 2 tests is `True` , otherwise it is `False` .

Exercise: The tables giving the different combinations are called truth tables. Run the cell below and fill in the tables in the corresponding Markdown cell.

Exercise :

In [7]:
```
print(False and False)
print(False and True)
print(True and False)
print(True and True)
```

```
False
False
False
True
```

In [8]:
```
print(False or False)
print(False or True)
print(True or False)
print(True or True)
```

```
False
True
True
True
```

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

Exercise : It is possible to add the keyword `not` before a boolean variable or a test. Run the following cell. What do you conclude?

```
In [9]: print(not True)
        print(not False)
```

```
False
True
```

answer

The opposite of True is False and the opposite of False is True.

1.4 - Branching

In a program, it is very common to want to execute different instructions according to the different cases encountered. The `if` command allows to select the instructions to be performed according to the result of a test.

```
if test:
    instruction_if_test_is_True
else:
    instruction_if_test_is_False
```

Exercise: Write two tests reusing the boolean variables and tests examples seen previously.

Exercise :

In [10]:
```
if not test2 >= 50:
    print ("This value is less than 50 or this value equals 5
0.")
else:
    print ("This value is more than 50.")
```

```
This value is less than 50 or this value equals 50.
```

In [11]:
```
if "James" == "Joe" or "James" == "Jack":
    print ("James is an impostor.")
else:
    print("James is a crewmate.")
```

```
James is a crewmate.
```

Exercise: Run the following cell and check that the result is

In [12]:
```python
print("1st example")
if True:
    print("   This is True")
print()

print("2nd example")
if not True:
    print("   This is False")
print()

print("3rd example")
if False:
    print("   This is False")
print()

print("4th example")
if True and False:
    print("   This is False")
print()

print("5th example")
if False:
    print("   This is executed if test is True")
else:
    print("T   his is executed if test is False")
print()

print("6th example")
if "James" == "Joe" or test2<=50:
    print("   First test is False")
    print("   test2 =",test2,"which is smaller than or equal to 
50")
else:
    print("   First test is False")
    print("   test2 =",test2,"which is greater than 50")
print()
```

```
1st example
   This is True

2nd example

3rd example

4th example

5th example
T   his is executed if test is False

6th example
   First test is False
   test2 = 25 which is smaller than or equal to 50
```

Exercise :

The results were what I expected.

2 Loop with the command `for`

When you want to make the same block several times, it makes no sense to duplicate the instructions as many times as necessary. We use the `for` command, an example of which is given below.

In [13]:
```
for i in range(10):
    print(i)
```

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

In the above statement, the function "range(10)" creates a list of ten integers. Any list can be used

In [14]:
```
sum=0
for i in [1,2,3,5,7,11,13]:
    sum=sum+i
    print("i=",i," sum=",sum)
```

```
i= 1  sum= 1
i= 2  sum= 3
i= 3  sum= 6
i= 5  sum= 11
i= 7  sum= 18
i= 11  sum= 29
i= 13  sum= 42
```

exercice: Using the examples above, write a program that uses a loop and calculates the sum of the squares of the prime numbers between 1 and 13 included.

Exercise :

In [15]:
```
sum_squares=0
for i in [1,2,3,5,7,11,13]:
    sum_squares = sum_squares + i**2
    print("i=",i," sum_squares =",sum_squares)
```

```
i= 1  sum_squares = 1
i= 2  sum_squares = 5
i= 3  sum_squares = 14
i= 5  sum_squares = 39
i= 7  sum_squares = 88
i= 11  sum_squares = 209
i= 13  sum_squares = 378
```

3 - Functions

Loops are suitable when an instruction block is used at a single point in the program. Otherwise, it is better to use a function. In this course we will use many functions written by others. A library is a set of functions to answer a given problem.

In a program a function must be defined before it is called, that is, used as if it were a python command like the `print` function.

Your functions and libraries allow you to enrich your programs so that they meet your needs.

3.1 an example

In [16]:
```
def myfunction(i):
    j=i**2
    return j

sum=0
for i in [1,2,3,5,7,11,13]:
    sum=sum+myfunction(i)
print(" sum=",sum)
```

```
 sum= 378
```

The previous example just uses a function to do the equivalent of the previous program. The `return` instruction at the end allows to return a value in the main part of the program (here the square of the parameter `i`) The interest of a function is to be able to make several calculations and to return several results simultaneously. As in the example below.

In [17]:
```
def myfunction(i):
    j=i**2
    k=j*i # ie k=i**3
    return j,k

sum2=0
sum3=0
for i in [1,2,3,5,7,11,13]:
    i2,i3=myfunction(i)
    sum2=sum2+i2
    sum3=sum3+i3
print(" sum i^2 =",sum2)
print(" sum i^3=",sum3)
```

```
 sum i^2 = 378
 sum i^3= 4032
```

In [18]:
```
def myfunction(i,n):
    j=i**n
    k=j*i # ie k=i**3
    return j,k

sum2=0
sum3=0
for i in [1,2,3,5,7,11,13]:
    i2,i3=myfunction(i,2)
    sum2=sum2+i2
    sum3=sum3+i3
print(" sum i^2 =",sum2)
print(" sum i^3=",sum3)
```

```
 sum i^2 = 378
 sum i^3= 4032
```

exercice: In this exercise, you are asked to make a function that takes a list as of prime an argument, and returns a list containing all of its elements, plus the next prime numbers . We will proceed step by step. The objective being here to give you the basics allowing you to modify source code, the implementation is not optimal and will put the priority on the reuse of the notions seen previously.

step 1: Create a function `TestPrime` that takes a list `prime` as first argument, then an integer `n` as second argument, and returns the boolean variable True. Test this function with the list `[2]` and `n=4` .

step 2: Modify this function to insert a loop on the values included in the `primes` list. Display each value taken

step 3: For each value `i` taken from the list `prices` , check if the number `n` is a multiple. To do this use the result of the test `n%i==0` (the operator `%` gives the remainder of the integer division). In that case, return `False` .

step 4: Create a second function `AddOnePrime` that takes as a single argument a list of integers, finds the largest value in that list and stores it in `imax` variable, then returns the string `ERROR` .

step 5: In the function `AddOnePrime` loop over the integers `i` between `imax+1` and `2*imax+1` . For each value, test if it is a prime number thanks to the previous `TestPrime` function to which you will send the list `primes` and the integer `i` . You will display the result of this test. You will test this function with the list `primes=[2]`

test 6: When the test is `True` , you have found a new prime number. Add it to the list `primes` with the command `prices.append(i)` and return the new list `prices` as the result of the function.

step 7: Make a loop calling 20 times this function on the list `primes` which you have initialized with the value `[2]` . Check that you find the first 21 prime numbers `[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73]` .

Exercise :

In [41]:
```
def TestPrime (primes, n):
    for i in primes:
        #print(i)
        if n%i==0:
            return False
    return True

TestPrime([2], 4)
```

```
2
```

Out[41]:
```
False
```

In [42]:
```
def AddOnePrime(primes):
    imax = primes[0]
    for i in primes:
        if i > imax:
            imax = i
    for i in range(imax + 1, 2 * imax + 1):
        test = TestPrime(primes, i)
        #print(test)
        if test:
            primes.append(i)
            return primes
    return "ERROR"

AddOnePrime([2])
```

```
2
True
```

Out[42]:
```
[2, 3]
```

In [40]:
```
primes = [2]

for i in range (20):
    primes = AddOnePrime(primes)

print(primes)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
61, 67, 71, 73]
```