DataScientest • com

Python for Data Science

Functions

1. Functions

A **function** in Python is a reusable block of code that is used to perform a specific operation.

We have already seen examples of built-in functions in Python such as:

- The `print` function: Used to display an object.
- The `range` function: Used to iterate through a series of integers.

We do not know the exact code inside these functions, yet we are able to predict their output. This is due to the fact that their result depends **only** on their **parameters** (or arguments) which are given to them as **input**.

The syntax to define a function is as follows:

```
def my_function(parameter):
    # Instruction block
    ...
    ...
    ...
    # The result of the function is given by the variable a_value
    return a_value
```

The keyword **`return`** defines the **result** of the function. This keyword also **ends the execution** of the function once the Python Interpreter encounters it. Everything that follows in the definition of the function will not be executed.

```
# We define a function that determines whether a number is even or odd
def is_even(number):
    # Is the number even?
    if number % 2 == 0:
```

In [5]:
```
# Insert your code here
def double(number):
    return 2*number

double(number=4), double(4)
```

Out[5]: (8, 8)

Hide solution

In [ ]:
```
def double(number) :
    return 2*number

double(number=4)
```

- **(c)** Write a function named **`list_product`** which takes as argument a **list** of numbers and computes using a loop the **product** of all the numbers in the list.
- **(d)** Evaluate this function on the list `[1, 0.12, -54, 12, 0.33, 12]`. Its result should be `-307.9296`.

In [23]:
```
test_list = [1, 0.12, -54, 12, 0.33, 12]

# Insert your code here

def list_product(list):
    product = 1
    for number in test_list:
        product *= number
    return product

print(list_product(test_list))
#print(product)
```

```
-307.9296
```

Hide solution

In [ ]:
```
def list_product(list_of_numbers):
    # We initialize the product to 1
    product = 1

    # For each number in the list
    for number in list_of_numbers:
        # We multiply the product with this number
        product *= number

    return product

test_list = [1, 0.12, -54, 12, 0.33, 12]

print(list_product(test_list))
```

- **(e)** Write a function named **`uniques`** which takes a list as argument and returns a new list containing the unique values of this list.

The term "**unique values**" does not mean values that appear only once in the list but rather the distinct **values** present.

Hence, `uniques([1, 1, 2, 2, 2, 3, 3, "Hello"])` should return **`[1, 2, 3, "Hello"]`**.

This terminology is very often used even though it does not have the same meaning as in everyday life.

You can check whether a value is part of a list using the membership operator **`in`** :

```
3 in [3, 1, 2]
>>> True

-1 in [3, 1, 2]
>>> False
```

In [26]:

```
# Insert your code here
test_list = [1, 1, 2, 2, 2, 3, 3, "Hello"]
def uniques(list_):
    list_uniques = []
    for i in list_:
        if i not in list_uniques:
            list_uniques.append(i)
        else:
            pass
    return list_uniques

uniques(test_list)
```

Out[26]: [1, 2, 3, 'Hello']

Hide solution

In [ ]:

```
def uniques(list_of_elements):
    # We initialize the list of unique values
    unique_values = []

    # For each item in the list
    for element in list_of_elements:
        # If the element is not in the list of unique values
        if element not in unique_values:
            # We add it
            unique_values.append(element)

    return unique_values


print(uniques([1, 1, 2, 2, 2, 3, 3, "Hello"]))
```

A function can have several parameters and several outputs. The general syntax of a function is therefore as follows:

```
def my_function (parameter1, parameter2, parameter3, ...):
# Instruction block
...
...
...
return output1, output2, output3 ...
```

When a function returns **multiple outputs**, the result of the function is actually a **tuple**. We can use **tuple assignment** to store the outputs in different variables:

```
# Definition of a function that returns the first and last
# elements of a list
def first_and_last(a_list):
    return a_list[0], a_list[-1]

# We use tuple assignement to assign each output
# of the funciton to a different variable
first, last = first_and_last([-2, 32, 31, 231, 4])

# Display of results
print(first)
>>> -2

print(last)
>>> 4
```

- **(f)** Create a function **`power4`** , which takes as argument a number `x` and returns the first 4 powers of this number (i.e. x^1, x^2, x^3, x^4).
- **(g)** Test this function on `x = 8` and store the results in 4 variables `x_1` , `x_2` , `x_3` and `x_4` .
- **(h)** Create a function `power_diff` taking as argument 4 numbers `x_1` , `x_2` , `x_3` and `x_4` which returns:
  - The difference between `x_2` and `x_1`
  - The difference between `x_3` and `x_2`

In [29]:
```
# Insert your code here

#f)
def power4(x):
    return x, x**2, x**3, x**4

x_1, x_2, x_3, x_4 = power4(8)

print(power4(8))
#H)
def power_diff(x1,x2,x3,x4):
    return x2-x1, x3-x2, x4-x3

power_diff(x_1, x_2, x_3, x_4)
```

```
(8, 64, 512, 4096)
```

Out[29]:
```
(56, 448, 3584)
```

Hide solution

In [28]:
```
def power4(x):
    return x**1, x**2, x**3, x**4

x_1, x_2, x_3, x_4 = power4(x = 8)

def power_diff(x_1, x_2, x_3, x_4):
    diff1 = x_2 - x_1
    diff2 = x_3 - x_2
    diff3 = x_4 - x_3

    return diff1, diff2, diff3

diff1, diff2, diff3 = power_diff(x_1, x_2, x_3, x_4)

print(diff1, diff2, diff3)
```

```
56 448 3584
```

When calling a function, it is possible to **not specify the value** of a parameter if it has a **default value**.

To give a parameter a default value, all you have to do is assign it with a value in the definition of the function:

```
# Definition of a function which calculates the product between two numbers
def product(a = 0, b = 1):
    return a*b

product(a = 4) # By default, b takes the value 1
>>> 4

product(b = 3) # By default, a takes the value 0
>>> 0

product(a = 4, b = 3)
>>> 12
```

ℹ It is not necessary to write the name of the parameters when calling a function (for example `product(3, 4)` returns `12`), but **the parameters should be given in the same order as in the definition of the function**. In this case, the first argument will be the value of `a` and the second argument the value of `b`.

2. Documentation of a function

In order to share a function with other users, it is common to write a small **description** that explains **how** the function must be used.

This description is called the **documentation** of a function, which is the equivalent of a user manual.

The documentation should be written at the start of the definition of a function:

```
def sort_list(a_list, order = "ascending"):
"""
This function sorts a list in the order specified by the 'order' argument.

Parameters:
-----------
list: the list to sort.

order: Must take the value "ascending" if we want to sort the list in ascending order.
       Must take the value "descending" otherwise.
```

In [4]:

```
# Insert your code here
def total_len(list_of_lists):
    """
    This function determines the total number of elements in the list of lists.

    Parameters:
    -----------
    list_of_lisst: the list of lists to find elements.


    Returns:
    --------
    number_of_elements: total number of elements in list_of_lists.
    """
    # instructions
    number_of_elements = 0
    for list_ in list_of_lists:
        number_of_elements += len(list_)
    return number_of_elements

test_list = [[1, 23, 1201, 21, 213, 2],
             [2311, 12, 3, 4],
             [11, 32, 1, 1, 2, 3, 3],
             [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

print(total_len(test_list))
```

Hide solution

In [ ]:

```
help(len)
# The len function returns the number of elements in a container

def total_len(list_of_lists):
    """
    This function counts the total number of items in a list of lists.

    Parameters:
    -----------
    list_of_lists: A list of lists.

    Returns:
    --------
    n_elements: the total number of elements in list_of_lists.
    """
    # We initialize the number of elements to 0
    n_elements = 0

    # For each list in the list of lists
    for a_list in list_of_lists:
        # We count the number of elements in the list
        n_elements += len(a_list)

    return n_elements

test_list = [[1, 23, 1201, 21, 213, 2],
             [2311, 12, 3, 4],
             [11, 32, 1, 1, 2, 3, 3],
             [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

print("The list test_list contains", total_len(test_list), "elements.")
```

3. Recursive functions

Recursion is the property of a function to calls itself during its execution.

This kind of syntax is able to solve some problems very simply, but it is not widely used in Python as it is harder to predict the final result of a recursive function.

The idea of recursive functions is to **simplify a problem until the solution is trivial**.

For example, if **N** people have to shake hands to greet each other, **how many handshakes are needed**?

Suppose that one person among the Ns shook hands with the other N-1 people. We can alread count **N-1** handshakes and it is now sufficient to count the handshakes for the remaining N-1 people, which is the same problem but with 1 less person to take into account.

We count this way until there are only 2 people left. In this case there is only one possible handshake left.

In Python, to count the number of handshakes between N people, we can define a recursive function:

```
def how_many_handshakes(N):
    """
    This function counts the number of handshakes
    needed for N persons to greet each other.
    """
    # If there are only 2 people
    if N == 2:
        # There can only be one handshake
        return 1
    # Otherwise
```

In [5]:

```
# Insert your code here

def factorial(n):
    if n < 0:
        return "Negative number."
    if n == 0 :
        return 1
    else:
        return n * factorial(n-1)
```

In [6]:

```
print(factorial(5))
```

```
120
```

Hide solution

In [2]:

```python
def factorial(n):
    if n < 0:
        return "Negative number." # Stops the function if the number is negative

    # The simple case where n == 0
    if n == 0:
        return 1
    else :
        # We use the recurrence n! = n * (n-1)!
        return n*factorial(n-1)

print(factorial(n=5))
```

```
120
```

In [ ]:

Unvalidate