Python for Data Analytics: Functions

⬅️ Comprehensions - Error catching ➡️

We finished the warm-up with comprehensions. We know to store data, to operate on it, and to alter the flow of code with conditionals and loops.

Now, we need to learn how to repeat blocks of code without having to write them over and over again. This is where functions come in.

Functions

Functions are blocks of code that are executed when called. They are used to group code that is saved in memory and we can call whenever we need it.

Functions are like machines that can take some input, process it, and return some output.

```
graph LR
    I[Input] --> A(Function)
    A --> O[Output]
```

An example could be a function that receives two numbers and returns their sum:

```
graph LR
    I(2, 7) --> A(Sum 2 and 7)
    A --> O[9]
```

We can define a function with the `def` keyword, followed by the name of the function, parentheses with the arguments, and a colon. The code block that defines the function is indented.

```python
def sum_numbers(a, b):
    final_sum = a + b
    return final_sum
```

Where:

- `def` is the keyword that tells Python we are defining a function
- `sum_numbers` is the name of the function that we can use to call it
- `a` and `b` are the arguments that the function receives, in this case, two numbers
- `final_sum` is an intermediate variable that stores the result of the sum
- `return` is the keyword that tells Python to return the value of `final_sum` when the function is called

We can call the function by using its name and passing the arguments:

```
result = sum_numbers(2, 7)

print(result)
```

```
9
```

The only mandatory elements in a function definition are the `def` keyword and the name of the function. The arguments and the return statement are optional.

- If the function doesn't receive arguments, we can define it with empty parentheses: `def my_function():`
- If the function doesn't return anything, we can omit the `return` statement.
    - In this case, the function will return `None` by default.

Positional arguments

Functions can receive arguments, which are the values that we pass to the function when we call it.

These arguements will be used in the code block of the function to perform some operation. If the parameter is not used, it will be ignored.

We can create functions with any number of arguments, including none.

```
def concatenate_strings(string_1, string_2, separator):
    return a + separator + b
```

After defining the function, we can call it with the arguments:

```
result = concatenate_strings('Oh', 'yeah', ' ')

print(result)
```

```
Oh yeah
```

If we don't pass the correct number of arguments, we will get an error:

```
result = concatenate_strings('Oh', 'yeah')

print(result)
```

```
TypeError: concatenate_strings() missing 1 required positional argument:
'separator'
```

Python checks for the number of arguments, but if we call our function with the arguments in a different order, Python will assign the values to the arguments by position as they were defined in the function

definition.

This sometimes will work, but often it will not give the expected result.

```
def slice_string(string, start, end):
    return string[start:end]

# correct order of arguments
result = slice_string('Python is awesome', 0, 6)

print(result)
```

```
Python
```

What is happening here is:

- Python is using the first argument as the string, the second as the start index, and the third as the end index.
- The arguments we have passed are `'Python is awesome'`, `0`, and `6`.
- The operation here is `'Python is awesome'[0:6]`, which returns `'Python'`.

```
# incorrect order of arguments
result = slice_string(0, 6, 'Python is awesome')

print(result)
```

```
TypeError: 'int' object is not subscriptable
```

What is happening here is:

- Python is using the first argument as the string, the second as the start index, and the third as the end index.
- The arguments we have passed are `0`, `6`, and `'Python is awesome'`.
- The operation here is `0['Python is awesome':6]`, which doesn't make sense.

So, the order of the arguments is crucial.

Keyword arguments

To avoid the confusion of the order of the arguments, we can use keyword arguments.

When we call a function with keyword arguments, we can specify the name of the argument and the value we want to pass. That way, we can pass the arguments in any order as long as we specify the name of the argument.

```
def slice_string(string, start, end):
    return string[start:end]
```

```
result = slice_string(string='Python is awesome', start=0, end=6)

print(result)
```

```
Python
```

We can call the function with the arguments in any order by specifying the name of the arguments:

```
result = slice_string(end=6, string='Python is awesome', start=0)

print(result)
```

```
Python
```

Perfect! Now we know how to define functions, how to pass arguments, and how to use keyword arguments.

Next, we will learn about default arguments.

Mixing positional and keyword arguments

We can mix positional and keyword arguments when calling a function.

When we mix them, we need to pass the positional arguments first and then the keyword arguments.

```
def power(base, exponent):
    return base ** exponent

result = power(2, exponent=3)

print(result)
```

```
8
```

This is correct, because we are passing the positional argument `2` and the keyword argument `exponent=3` after it.

But if we try to pass the keyword argument first, even if the order is the original one in the definition of the function, we will get an error:

```
result = power(base=3, 2)

print(result)
```

```
SyntaxError: positional argument follows keyword argument
```

Default arguments

Sometimes, we face situations in which the usual value of an argument will be the same in most cases. In these cases, we can define a default value for the argument.

If we call the function without passing the argument, the default value will be used.

If we call the function with the argument, the value we pass will be used.

```
def root(base, exponent=2):
    return base ** (1/exponent)

result = root(3)

print(result)
```

```
1.7320508075688772
```

In this case, we are defining a function that calculates the root of a number. If we don't pass the exponent, the function will calculate the square root.

If we pass the exponent, the function will calculate the root of the number with the exponent we pass.

```
result = root(8, 3)

print(result)
```

```
2.0
```

We can define as many default arguments as we want, but they must be defined after the non-default arguments, if any.

```
def volume_of_parallelipiped(length, width, height=1):
    return length * width * height

result = volume_of_parallelipiped(2, 3)

print(result)
```

```
6
```

```
result = volume_of_parallelipiped(2, 3, 4)

print(result)
```

```
24
```

Using functions as arguments

There are cases in which we need to pass a function as an argument to another function.

We will see 3 examples of this:

- `map`
- `filter`
- `reduce`

`map`

`map` is a function that applies a function to each element of an iterable object.

The syntax is:

```
map(function, iterable)
```

Where:

- `function` is the function we want to apply to each element
- `iterable` is the object we want to iterate over

Keep in mind that the function we pass to `map` must receive only one argument, and it must be already defined.

Let's see it in an example: we want to calculate the square of each number in a list.

```
def square(number):
    return number ** 2

numbers = [1, 2, 3, 4, 5]

squared_numbers = map(square, numbers)

print(list(squared_numbers))
```

```
[1, 4, 9, 16, 25]
```

This is not an operation that we couldn't do with a list comprehension, but it is a good example of how to use a function as an argument. Also, map is usually faster than a list comprehension.

`filter`

`filter` is a function that applies a function to each element of an iterable object and returns only the elements that satisfy a condition.

The syntax is:

```
filter(function, iterable)
```

Where:

- `function` is the function we want to apply to each element, it should return a boolean, and only the elements that return `True` will be returned
- `iterable` is the object we want to iterate over

Let's see it in an example: we want to filter the even numbers from a list.

```
def is_even(number):
    return number % 2 == 0

numbers = [1, 2, 3, 4, 5]

even_numbers = filter(is_even, numbers)

print(list(even_numbers))
```

```
[2, 4]
```

reduce

`reduce` is a function that applies a function pairwise to the elements of an iterable object, and returns a single value.

The syntax is:

```
from functools import reduce

reduce(function, iterable)
```

Where:

- `function` is the function we want to apply to each pair of elements, it should receive two arguments and return a single value
- `iterable` is the object we want to iterate over

A classic example is the sum of all the elements in a list using `reduce`:

```
flowchart LR
    O["[1, 2, 3, 4, 5]"] --> O1["1"]
    O --> O2["2"]
    O --> O3["3"]
```

```
    O --> O4["4"]
    O --> O5["5"]

    O5 --> S4["sum"]
    R3 --> S4
    S4 --> R4["15"]

    O4 --> S3["sum"]
    R2 --> S3
    S3 --> R3["10"]

    O3 --> S2["sum"]
    R1 --> S2
    S2 --> R2[6]

    O2 --> S1["sum"]
    O1 --> S1
    S1 --> R1["3"]
```

Basically, `reduce` works in a similar way to us humans when we sum a list of numbers: we sum the first two numbers, then we sum the result with the next number, and so on.

Lambda functions

We have learned how to save operation blocks in memory with functions.

Also, how to use functions as arguments in other functions, like `map`, `filter`, and `reduce`. But sometimes we will not have a function defined to pass as an argument, so we will need to define it on the fly.

For these cases, we can use lambda functions.

Lambda functions are anonymous functions that we can define in a single line of code.

```
lambda x: x ** 2
```

This lambda function receives one argument `x` and returns the square of `x`.

We can use this lambda function as a regular function, assigning it to a variable:

```
square = lambda x: x ** 2

result = square(3)

print(result)
```

```
9
```

But if we are going to use it beyond its definition, it is better to use a regular function.

So, lambda functions are useful when we need to define a function on the fly, like when we are going to pass it as an argument to another function.

Let's see an example with `map`:

```
numbers = [1, 2, 3, 4, 5]

squared_numbers = map(lambda x: x ** 2, numbers)

print(list(squared_numbers))
```

```
[1, 4, 9, 16, 25]
```

And with `filter`:

```
numbers = [1, 2, 3, 4, 5]

even_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even_numbers))
```

```
[2, 4]
```

And with `reduce`:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

sum_of_numbers = reduce(lambda x, y: x + y, numbers)

print(sum_of_numbers)
```

```
15
```

Function	When to use	Example
User-defined function	When we are going to use the function more than once	`def square(x): return x ** 2`
Lambda function	When we are going to use the function only once	`lambda x: x ** 2`

⬅️ Comprehensions - Error catching ➡️