AGENDA

Part II – Functions
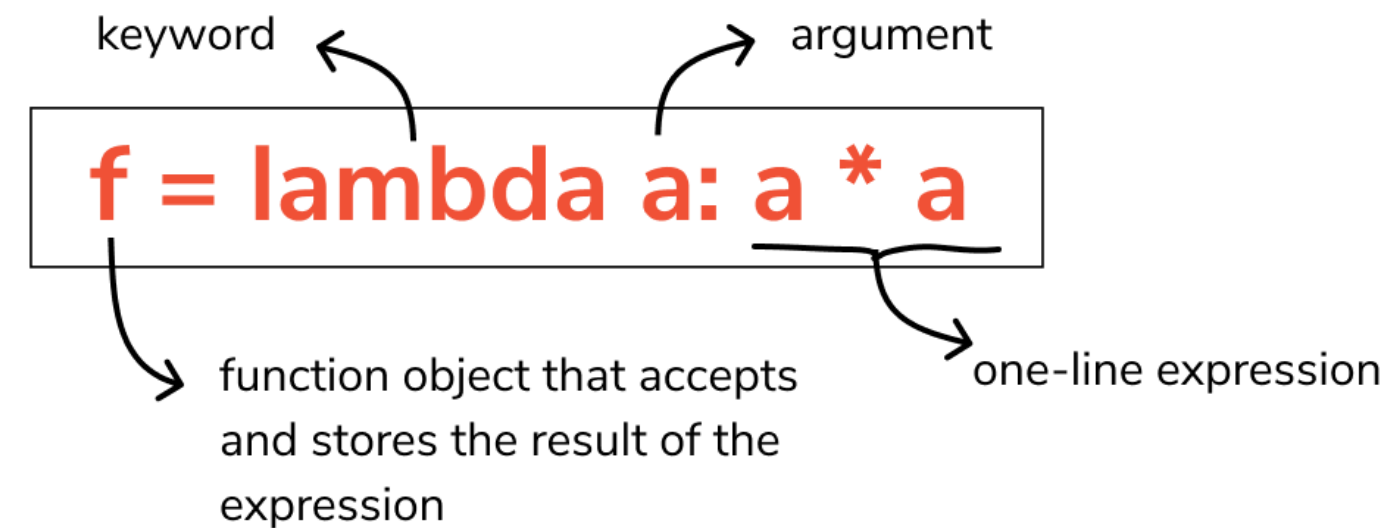
1. **Building lambda functions**
2. **Python integrated functions**
   a. Map
   b. Filter
   c. Any and all
   d. Generators
   e. Sorted
   f. Reversed
   g. Len, Abs, Sum and Round
   h. Zip

Lambda Functions

What is a lambda function?

Python Lambda Functions are **anonymous functions**; this means that the function has no name. As we already know, the **def** keyword is used to define a normal function in Python. Similarly, the **lambda** keyword is used to define an anonymous function.

keyword → ← argument

f = lambda a: a * a

function object that accepts and stores the result of the expression

one-line expression

Lambda Functions properties

Structure

This function can have **any number of arguments** but only **one expression**, which is evaluated and returned. They are syntactically restricted to a single expression.

Use case

One is free to use lambda functions wherever function objects are required.

How to build Lambda functions in Python?

As we already know, the **def** keyword is used to define the normal functions, while the **lambda** keyword is used to create anonymous functions.

Motivation.txt

```python
# Python code to illustrate the cube of a number
using lambda function
def cube(x):
    return x*x*x

cube_v2 = lambda x : x*x*x

print(cube(7))
>> 343
print(cube_v2(7))
>> 343
```

How to build Lambda functions in Python?

Suppose we want to create an output which capitalizes a given string and reverts the characters order.

Motivation.txt

```python
# Example: Lambda function to capitalize and
reverse a string
str1 = 'natixis'

rev_upper = lambda string:string.upper()[::-1]

print(rev_upper(str1))
>> SIXITAN
```

How to build Lambda functions in Python?

Condition Checking Using Python lambda function

```
                                    Motivation.txt

format_numeric = lambda num: f"{num:e}" if
isinstance(num, int) else f"{num:,.2f}"


print("Int formatting:", format_numeric(1000000))
>> Int formatting: 1.000000e+06

print("float formatting:",
format_numeric(999999.789541235))
>> float formatting: 999,999.79
```

Lambda function vs def defined function

A Lambda function can be written as a def function; the contrary is not always possible.

```
                                    Motivation.txt

def multiply(y, x):
    return y * x

lambda_multiply = lambda y, x: y * x

# Using function defined with def keyword
print("Using function defined with `def` keyword,
multiply:", multiply(5, 3))
>> Using function defined with `def` keyword,
multiply: 15

# Using the lambda function
print("Using lambda function, multiply:",
lambda_multiply(5, 3))
>> Using lambda function, multiply: 15
```

Lambda function vs def defined function

With lambda function	Without lambda function
Supports **single line statements** that returns some value.	Supports **any number** of lines inside a function block
Good for performing **short operations**/data manipulations.	Good for **any cases** that require **multiple** lines of code.
Using lambda function **can** sometime **reduce** the **readability** of code.	We can use comments and function descriptions for **easy readability**.

Python integrated functions

What is a Python integrated function?

Python has a wide range of **built-in functions** that are available without the need for any additional modules or packages.

Some examples →

print(): Used to print output to the console.
type(): Returns the type of an object.
len(): Returns the length of a sequence.
range(): Returns a sequence of numbers.
input(): Reads input from the user.
str(): Converts an object to a string.
int(): Converts a string or float to an integer.
float(): Converts a string or integer to a float.
abs(): Returns the absolute value of a number.
max(): Returns the maximum value in a sequence.
min(): Returns the minimum value in a sequence.
sum(): Returns the sum of all the values in a sequence.
round(): Rounds a number to a specified number of decimal places.
sorted(): Returns a sorted list.

Iterable vs Iterator

Before jumping into the integrated functions, let's **clarify two concepts**:

- **Iterable:** An object capable of returning its members one at a time (e.g. lists, strings, tuples). It implements the __iter__() method (e.g. list, tuples, dictionaries...)

- **Iterator:** An object that represents a stream of data; it returns the next item with __next__(). It is produced by calling iter() on an iterable.

```
                              Motivation.txt

# An iterable
numbers = [1, 2, 3]
# Getting an iterator from the iterable
iterator = iter(numbers)


# Using the iterator
print(next(iterator))
>> 1
print(next(iterator))
>> 2
print(next(iterator))
>> 3
```

The map() function

map() function returns a map object (which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.) In this example, it returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)

```
                                        Motivation.txt

# Return double of n
def addition(n):
    return n + n
# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
>> [2, 4, 6, 8]

# Double all numbers using map and lambda
numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))
>> [2, 4, 6, 8]
```

The filter() function

filter() function filters the given sequence with the help of a function that tests each element in the sequence to be true or not

Example using a function defined with def keyword

```
                                    Motivation.txt

# Function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if variable in letters:
        return True
    else:
        return False

sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
# using filter function
filtered = filter(fun, sequence)
print('The filtered letters are:')
for s in filtered:
    print(s)
>> The filtered letters are:
>>    e
>>    e
```

The filter() function

filter() function filters the given sequence with the help of a function that tests each element in the sequence to be true or not

Example using a lambda function

Motivation.txt

```python
# A list contains both even and odd numbers
seq = [0, 1, 2, 3, 5, 8, 13]
# Result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))
>> [1, 3, 5, 13]

# result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))
>> [0, 2, 8]
```

any() vs all() functions

	Any	All
All truthy values	True	True
All falsy values	False	False
One truthy value (all others are falsy)	True	False
One falsy value (all others are truthy)	True	False
Empty iterable	False	True

The any() function

any() returns **True** if any of the items is True. It returns False if empty or all are false. It can be thought of as a sequence of OR operations on the provided iterables. It shorts circuit the execution, i.e. stops the execution as soon as the result is known.

```
# Since all are false, false is returned
print(any([False, False, False, False]))
>> False

# The method will short-circuit at the second
item (True) and will return True
print(any([False, True, False, False]))
>> True

# The method will short-circuit at the first
(True) and will return True
print(any([True, False, False, False]))
>> True
```

18

The all() function

all() returns **True** if all items in an iterable object are true. If the iterable object is empty, the all() function also returns True.

```
# All the iterables are True so all will return
True and the same will be printed
print(all([True, True, True, True]))
>> True

# The method will short-circuit at the first item
(False) and will return False
print(all([False, True, True, False]))
>> False

# This statement will return False, as no True is
found in the iterables
print(all([False, False, False]))
>> False
```

Generators

A **generator-function** is defined like a normal function, but whenever it needs to generate a value, it does so with the **yield** keyword rather than return. If the body of a def function contains yield, the function automatically becomes a generator function. Values are yield one at a time.

Motivation.txt

```python
# A generator function that yields 1 first time, 2
second time, and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
>>    1
>>    2
>>    3
```

Generators

Generator functions return a **generator object**. Generator objects are used either by **calling the next method** on the generator object or using the generator object in a **for loop** (as in the example of the previous slide). **Generators are iterators.**

```
Motivation.txt

# A generator function that yields 1 first time, 2
second time, and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3


x = simpleGeneratorFun()
print(next(x))
>> 1
print(next(x))
>> 2
print(next(x))
>> 3
```

Generators

Why are they useful?

Generators are useful because they let you **work with sequences of data lazily**, meaning they **generate values on demand** instead of building the entire sequence in memory. **Generators save memory, can improve performance, and make your code more expressive for handling large or infinite sequences of data.**

Generators

Imagine you're an administrator and need to scan a 10 GB server log file for lines containing "ERROR". If you read it all into a list, this could crash your machine or be very slow, since it loads all 10 GB at once.

With generators, you have only 1 line in memory at the time

```
                                    Motivation.txt

#### Inefficient way #############
with open("server.log") as f:
    lines = f.readlines()  # ❌ loads
everything into memory
errors = [line for line in lines if "ERROR"
in line]
##########################

#### Efficient way #############
def error_lines(filepath):
    with open(filepath) as f:
        for line in f:
            if "ERROR" in line:
                yield line # generate only when needed

for err in error_lines("server.log"):
    print(err) # each error line is processed immediately
```

sorted() vs reversed() functions

Python **sorted()** function returns a sorted list from the iterable object.

Python **reversed()** method returns an iterator that accesses the given sequence in the reverse order.

```
                                          Motivation.txt

# Original list
x = [2, 8, 1, 4, 6, 3, 7]

# Sorted
print("Sorted List returned:", sorted(x))
>> Sorted List returned : [1, 2, 3, 4, 6, 7, 8]

print("Reverse sort:", sorted(x, reverse=True))
>> Reverse sort : [8, 7, 6, 4, 3, 2, 1]

# Reversed
print("Reversed:", list(reversed(x)))
>> Reversed : [7, 3, 6, 4, 1,  8, 2]
```

The len() function

The **len()** function is used to get the length of a string, list, tuple, or any other sequence in Python.

```python
# Example 1: Get the length of a string
string = "Hello, world!"
length = len(string)
print(length)
>> 13


# Example 2: Get the length of a list
my_list = [1, 2, 3, 4, 5]
length = len(my_list)
print(length)
>> 5
```

The abs() function

The **abs()** function returns the absolute value of a number.

Motivation.txt

```python
# Example 1: Get the absolute value of a positive number
num1 = 5
abs_value = abs(num1)
print(abs_value)
>> 5

# Example 2: Get the absolute value of a negative number
num2 = -7
abs_value = abs(num2)
print(abs_value)
>> 7
```

The sum() function

The **sum()** function is used to get the sum of all the elements in a list.

```python
# Example 1: Get the sum of all the elements in a
list
my_list = [1, 2, 3, 4, 5]
total = sum(my_list)
print(total)
>> 15

# Example 2: Get the sum of all the elements in a
list of floats
my_list = [1.5, 2.5, 3.5, 4.5]
total = sum(my_list)
print(total)
>> 12.0
```

27

The round() function

The **round()** function is used to round off a number to a specified number of digits.

```
                                    Motivation.txt

# Example 1: Round off a number to 2 decimal places
num1 = 3.14159
rounded_num = round(num1, 2)
print(rounded_num)
>> 3.14

# Example 2: Round off a number to the nearest
integer
num2 = 3.7
rounded_num = round(num2)
print(rounded_num)
>> 4
```

The zip( ) function

The **zip()** function takes iterables and returns a single **iterator of tuples**, where each tuple contains elements from the iterables at the same position (index). It is commonly used to combine multiple iterables so they can be iterated over in parallel.

Motivation.txt

```
name = ["Manjeet", "Nikhil", "Shambhavi", "Astha"]
roll_no = [4, 1, 3, 2]


# using zip() to map values
mapped = zip(name, roll_no)
print(list(mapped))
>> [('Manjeet', 4), ('Nikhil', 1), ('Shambhavi',
3), ('Astha', 2)]
```

Why use Python integrated functions?

All python built-in functions:

https://docs.python.org/3/library/functions.html

Class wrap-up

What have we learned today?

- **Building Lambda Functions:**
  - Started with lambda functions, a way to create anonymous functions for short-term use.
  - Explored the syntax and use cases for lambda functions, understanding their simplicity and flexibility.
- **Python Integrated Functions:**
  - Delved into several built-in functions that Python provides for various tasks.
  - Discussed the map function, enabling efficient transformation of data by applying a given function to each item in an iterable.
  - Explored the filter function, allowing the selection of elements from an iterable based on a specified condition.
  - Introduced the any and all functions, facilitating Boolean evaluations over iterables.
  - Examined generators, a memory-efficient way to iterate over large datasets using the yield keyword.
  - Explored the sorted function, providing a sorted version of any iterable, with optional custom sorting criteria.
  - Discussed the reversed function, which reverses the order of elements in an iterable.
  - Covered the functions len, abs, sum, and round, essential for obtaining the length of iterables, absolute values, summing elements, and rounding numbers, respectively.
  - Concluded with the zip function, facilitating the combination of multiple iterables into tuples.

PRACTICE PRACTICE PRACTICE

You won't master a skill if you don't practice!

Exercises – Learn by doing!

In order to facilitate the learning process of Python **we have prepared for each session a python file** where you can find **exercises** that will help you to grasp the introduced Python concepts.

Visual Studio Code
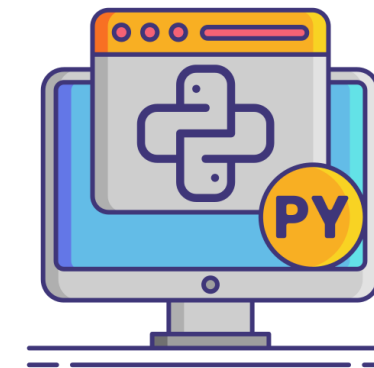
We will use **VS CODE** as our Python program IDE

Exercises for today

Link to exercises: https://github.com/MAlexandraOliveira/natixis-python-level-2/blob/main/exercises/Class2_exercises.py

Why should you deactivate Copilot? (for now)

As **beginners in Python programming**, it's crucial to focus on truly understanding how code works, rather than just seeing it appear. Tools like GitHub Copilot can be tempting, but they **often offer solutions without explanation**, making it easy to skip the learning process. While these tools are designed to assist, **not replace your thinking**, they can encourage you to rely on solutions you don't fully grasp—and they're not always correct. To truly learn, you need to write, debug, and explore code on your own. **By turning off Copilot** during the early stages of your learning, you give yourself the opportunity to develop real problem-solving skills, build confidence, and create a strong foundation. Later, when you have a solid grasp of the basics, Copilot can serve as a useful support tool, but always approach its suggestions with a critical mindset, not blind trust.

Steps to turn-off GitHub Copilot:

1. Go to Settings (File > Preferences > Settings or press Ctrl+,).
2. In the search bar, type: Copilot.
3. Find the setting GitHub Copilot: Enable.
4. Uncheck it to disable Copilot globally.

THANK YOU 😊

Questions?

Alexandra Oliveira 👤 ✉ m.alexandra.ro@gmail.com