Course: Python language

Ali ZAINOUL

for Keyce Academy
May 10, 2023

Python ?!

- ■ **Python in brief**:
  - Powerful programming language;
  - Easy to use and learn;
  - Open source;
  - Dynamic typing language

- ■ **Advantages of Python**:
  - Adapted to newbies;
  - Used in a large scale and variety projects in industry;

Figure: Python logo

Necessary tools

- Integrated Development Environment (IDE)
  Choice between: PyDev, Microsoft Visual Studio, Sublimetext …

Figure: Best IDE

- Text Editor & compilation *via* terminal

Installing Python on Windows

- Download the latest Python installer from the official website.
- Open Command Prompt and navigate to the folder where the installer was downloaded.
- Run the installer by typing the following command: `python installer-file-name.exe`
- Follow the instructions on the screen to complete the installation.
- To verify that Python was installed correctly, open Command Prompt and type `python` to launch the Python interpreter.

Installing Python on Linux

- Open the terminal and type the following command: `sudo apt-get install python3`
- Enter your password when prompted.
- Wait for the installation to complete.
- To verify that Python was installed correctly, type `python3` in the terminal to launch the Python interpreter.

Installing Python on MacOS

- Open the terminal and type the following command: `xcode-select --install`
- Follow the instructions on the screen to install Xcode Command Line Tools.
- Once Xcode Command Line Tools is installed, type the following command in the terminal: `brew install python3`
- Wait for the installation to complete.
- To verify that Python was installed correctly, type `python3` in the terminal to launch the Python interpreter.

Ecosystem

49%	**Analyse de données**
46%	Développement Web
42%	Apprentissage machine
37%	Administration système / Création de scripts d'automatisation / Configuration d'infrastructure (DevOps)
37%	Programmation d'analyseurs / scrapers / crawlers web
25%	Tests de logiciels / Création de tests automatisés
22%	Prototypage logiciel
20%	À des fins éducatives
19%	Programmation réseau
12%	Développement desktop
6%	Graphismes informatiques
6%	Développement de technologies embarquées
5%	Développement de jeux
3%	Développement mobile
4%	Autre

Figure: Ecosystem Python

Most used libraries in Python

- Pandas: much used in data science. (CSV, TXT, SQL)
- Numpy: numerical analysis and calculus, mathematical functions, objects and operations, linear algebra.
- Matpotlib: library used in order to trace and visualize data with graphs.
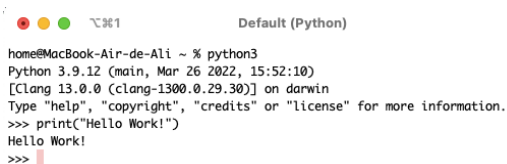
Python, quésaco ?!

Python is a programming language that is:

- Interpreted;
- Multiparadigm et multiplatforms;
- which promotes the:
  - *structured imperative programming*
  - *functionnal programming*
  - *Object Oriented Programming*

First program in Python

print("Hello Work!")

Figure: First program in Python

Plan of the course

- Introduction to Python
- Basic Syntax, Variables and Data Types
- Functions
- Modules
- Files I/O
- Exception Handling
- OOP

Basic Syntax, Variables, and Data Types

Indentation in Python

- Python uses indentation to define blocks of code. This is different from many other programming languages, which use curly braces to define blocks such as C, C++ or Java. Here's an example:

```python
if x > 0:
        print("x is positive")
else:
        print("x is not positive")
```

- The indentation tells Python which lines of code belong to each block.
- Use the **tab** key in order to indent your code.

Line Endings in Python

- In Python, a line of code ends with a newline character. You can also use a backslash (\) to indicate that a statement continues on the next line. Here's an example:

```python
x = 5 + \
10
print(x) # Output: 15
```

- In this example, the backslash tells Python that the + operator is part of the previous line, and the value of x is 15.

Comments in Python

- You can add comments to your Python code to explain what the code does. Comments start with a hash (#) character and continue until the end of the line. Here's an example:

```
# This is a comment
print("Hello, world!")
```

- In this example, the first line is a comment, and Python ignores it when running the code. The second line prints the message "Hello, world!" to the console. Comments can be used to make your code more readable and easier to understand.

Variables in Python

- In Python, a variable is a name that refers to a value. You can assign a value to a variable using the assignment operator (=). Here's an example:

```python
x = 10
print(x)
```

- This code assigns the value 10 to the variable x and then prints the value of x to the console.
- Variables can hold different types of values, such as integers, floats, strings, and booleans. You can even assign a variable to another variable.

Variable Names in Python

- In Python, variable names can contain letters, numbers, and underscores. They must start with a letter or underscore, and they are case-sensitive. Here are some examples of valid variable names:

```
age = 30
name = "John"
my_score = 9.5
is_student = True
```

- It's important to choose meaningful and descriptive names for your variables, so that your code is easier to read and understand.

Variable Assignment in Python

- In Python, you can assign a value to a variable using the assignment operator (=). You can also assign the same value to multiple variables at once. Here's an example:

```
x = y = z = 0
print(x, y, z)
```

- This code assigns the value 0 to the variables x, y, and z all at once, and then prints their values to the console.
- You can also assign different values to multiple variables in a single line of code, separated by commas.

Data Types in Python: Integers

- Python supports several data types, including integers, floats, strings, booleans, lists, tuples, sets, and dictionaries. Let's take a closer look at each one.
- **Integers** are whole numbers, such as -3, -2, - 1, 0, 1, 2, 3 etc. You can perform basic arithmetic operations on integers, such as addition, subtraction, multiplication, and division.

```
x = 3
y = 2
print(x + y) # Output: 5
```

Data Types in Python: Floats and Strings

- **Floats** are decimal numbers, such as 3.14, 2.0, and -1.5. You can also perform basic arithmetic operations on floats.

```
x = 3.14
y = 2.0
print(x * y) # Output: 6.28
```

- **Strings** are sequences of characters, such as "hello", "world", and "Python". You can concatenate strings using the + operator.

```
x = "hello"
y = "world"
print(x + " " + y) # Output: "hello world"
```

Data Types in Python: Booleans

■ **Booleans** are either True or False. They are often used in conditional statements and loops.

```
x = 10
y = 5
print(x > y) # Output: True
```

■ **Lists, tuples, sets, and dictionaries** are collections of values. They have different properties and methods for manipulating and accessing their elements.

Lists in Python

■ A list is a collection of values that are ordered and mutable.
Lists are created using square brackets [] and each value is separated by a comma.

```
fruits = ["apple", "banana", "orange"]
print(fruits[0]) # Output: "apple"
fruits.append("grape")
print(fruits) # Output: ["apple", "banana", "orange", "grape"]
```

■ You can access and modify individual elements of a list using their index. You can also add or remove elements using built-in methods like append(), insert(), remove(), and pop().

Tuples in Python

■ A tuple is a collection of values that are ordered and immutable. Tuples are created using parentheses () and each value is separated by a comma.

```
colors = ("red", "green", "blue")
print(colors[1]) # Output: "green"
```

■ You can access individual elements of a tuple using their index, but you cannot modify the elements themselves. Tuples are useful for representing fixed collections of values.

Sets in Python

■ A set is a collection of values that are unordered and unique.
Sets are created using curly braces or the set() function and each value is separated by a comma.

```
fruits = { "apple", "banana", "orange" }
print("apple" in fruits) # Output: True
fruits.add("grape")
print(fruits) # Output: "apple", "banana", "orange", "grape"
```

■ You can check if a value is in a set using the in operator. You can also add or remove elements using built-in methods like add(), remove(), and discard().

Dictionaries in Python

■ A dictionary is a collection of key-value pairs that are unordered and mutable.

Dictionaries are created using curly braces and each key-value pair is separated by a colon :.

```
person = { "name": "John", "age": 30, "city": "New York" }
print(person["name"]) # Output: "John" person["age"] = 31
print(person)
# Output: { "name": "John", "age": 31, "city": "New York" }
```

■ You can access and modify values in a dictionary using their keys. You can also add or remove key-value pairs using built-in methods like update(), pop(), and copy().

Alternate and Repeating Conditional Structures and Loops

- In algorithms and programming, we call a **conditional structure** the set of instructions that test whether a condition is true or not. There are three of them:
  - The conditional structure **if**:

    ```
    if condition:
          my_instructions
    ```

Alternate and Repeating Conditional Structures and Loops

■ **Continuation**:
- The Conditional Structure **if ...else** :

```
if condition:
        my_instructions
else:
        my_other_instructions
```

Alternate and Repeating Conditional Structures and Loops

■ **Continuation:**

- The conditional structure **if …elif …else** :

```
if condition_1:
        my_instructions_1
elif condition_2:
        my_instructions_2
else:
        my_other_instructions
```

Alternate and Repeating Conditional Structures and Loops

- Furthermore, an **iterative conditional structure** allows the same series of instructions to be executed several times (iterations). The **while** instruction executes the instruction blocks as long as the while condition is true. The same principle for the **for** loop (for), the two structures are detailed below:
    - The iterative Conditional Structure **while**:

    ```
    while condition:
         my_instructions
    ```

Alternate and Repeating Conditional Structures and Loops

- **Continuation:**
  - The iterative conditional structure **for:**

    ```
    for i in range(start, stop, step) :
            my_instructions
    ```

- **Remark:** The major difference between for loop and the while loop is that for loop is used when the number of iterations is known, whereas execution is done in the while loop until the statement in the program is proved wrong.

Alternate and Repeating Conditional Structures and Loops

■ The iterative conditional structure **for**:

```
for variable in iterable:
    my_instructions
```

Functions in Python

- One may want to define a function in order to use it in a program, the general syntax of declaring and defining a function is as follow:

```
def theFunctionName(args if any):
    # function implementation
    # function implementation
    # function implementation
```

- A function will always be defined with the keyword **def**.
- Please note the importance of indentation in the function implementation.

Functions in Python: example 1

Defining a function without parameters and without any returns:

```python
# Defining the utility function printLine
def printLine():
    print("————")

# Testing our function
printLine()
```

Functions in Python: example 2

Defining a function that takes a number as a parameter and returns its square

```
# Defining our square function
def Square(n):
    return n**2

# Testing our function
Square(4)
```

Note the usage of the ****** operator in order to calculate the power of a number.

Notion of recursivity: recursive functions

Definitions:
- An algorithm is called **recursive** if it is defined by it-self.
- In the same manner, a function is called **recursive** if it references itself.

Functions in Python: example 3

Defining a recursive function with a parameter and no returns (the return is returning void):

```python
# Defining our recursive function Count
def Count(n):
    print(n)
    if n == 0 :
        print("Done.")
        return
    Count(n-1)

# Testing our function
Count(10)
```

Lambda Functions

- Python Lambda functions are:
    - Small
    - Anonymous
    - Subject to a more restrictive syntax
    - Concise
    - Taking any number of arguments, but can only have one expression.
- We define a Lambda function with the keyword **lambda**.
- The general syntax of a lambda function is the following:

```
lambda args : implementation
```

Lambda Functions: examples

```python
# Example 1: lambda function with 1 parameter
f = lambda x : 2 * x
print( f(50) )
#Output: 100.
```

```python
# Example 2: lambda function with 2 parameters
fun = lambda a, b : a ** b
print(fun(7, 2))
#Output: 49.
```

Usage of Lambda Functions

- The utility of a Lambda function is when you use it as an anonymous function inside another function.
- A complete example is shown below:

```python
def multiplier(n):
    return lambda x : x * n
Doubler = multiplier(2)
Tripler = multiplier(3)
print(Doubler(33)) # Outputs: 66
print(Tripler(33)) # Outputs: 99
```

Variable number of parameters in a function

- Python allows you to create functions with a variable number of parameters (e.g. print() is a function that can receive an arbitrary number of parameters). The general syntax of such functions is:

```
def functionName(*args):
    # instructions
```

```
# Example:
def foo(*args):
    for arg in args:
        print(arg)
foo("Hello", 1, 2, 3, "World")
```

Functions with key/value parameters

- Parameters can also be managed as keys/values to populate a dictionary:

```
def functionName(**kwargs) :
    # instructions
```

```
# Example:
def printDict(**kwargs):
for (k, v) in kwargs.items() :
        print(k, v)

printDict(Day_1="Monday", Day_2="Tuesday"))
```

Generators

- A generator is implemented with the **yield** keyword, it permits to creates a flux of data. A kind of data reservoir; which can be called up piecewise by **next()**.

```python
def Generator() :
    for i in range(10, 0, -1):
        yield i
gen = Generator()
print(next(gen)) # Outputs : 10
print(next(gen)) # Outputs : 9
```

Modules and Importing

- **What are modules in Python?** In Python, a module is a file containing Python code that can be imported into other Python scripts. Modules are used to organize code into separate files and provide a way to reuse code across different programs. To create a module, you simply write Python code in a file with a .py extension. You can then import the module into another Python script using the **import** statement.
- **How do you import modules in Python?**
  To import a module in Python, you use the **import** statement followed by the name of the module. For example, if you have a module named "my_module.py", you can import it into another Python script with the following statement:

```
import my_module
```

Modules and Importing

- Once you have imported the module, you can use its functions and variables in your code. To call a function from the module, you use the dot notation, like this:

```
my_module.my_function()
```

- You can also use the **from** keyword to import specific functions or variables from a module, like this:

```
from my_module import my_function, my_variable
```

- This allows you to use the imported functions and variables without having to prefix them with the module name.

Best practices for modules in Python

- When importing modules in Python, there are some best practices you should follow to ensure that your code is clean, readable, and maintainable. Here are a few tips:
  - Use absolute imports to specify the full path to the module you want to import. This makes it clear which module you are importing and helps avoid naming conflicts.
  - Avoid using wildcard imports (e.g. from my_module import *) as they can make it difficult to understand where functions and variables are coming from.
  - Use aliases to shorten module names and make them easier to use. For example, you could import the numpy module as np like this: import numpy as np.

Best practices for modules in Python

- Continuation:
  - Group related imports together at the top of your script to make it clear which modules your code depends on.
  - Avoid circular imports, where two modules depend on each other. This can create confusing dependencies and make your code harder to understand.
- By following these best practices, you can ensure that your Python code is well-organized, easy to read, and maintainable over time.

Files I/O in Python

Files I/O in Python
- Input and output operations on files are an essential part of many programs.
- In Python, you can use the built-in functions open() and close() to work with files.
- To read data from a file, you can use the read() method.
- To write data to a file, you can use the write() method.

Files I/O in Python Reading Data Example

■ Here's an example of how to read data from a file:

```python
# Open the file in read mode
file = open("myfile.txt", "r")
# Read the contents of the file
data = file.read()
# Close the file
file.close()
# Print the data
print(data)
```

Files I/O in Python Writing Data Example

■ And here's an example of how to write data to a file:

```python
# Open the file in write mode
file = open("myfile.txt", "w")
# Write some data to the file
file.write("Hello, world!")
# Close the file
file.close()
```

Working with files in Python: specifications 1

By default, open() opens the file in text mode, which means that it reads and writes strings. If you want to work with binary data, you can use the open() function with the "b" mode modifier (wb or rb). Remember to always close your files when you're done working with them, as this releases any system resources associated with the file and helps prevent data corruption.

Working with files in Python: specifications 2

Working with File Paths in Python

- When working with files in Python, it's important to specify the correct file path.
- The file path is the location of the file on your computer.
- In Windows, file paths use the backslash (\) character to separate directories.
- In Linux and macOS, file paths use the forward slash (/) character.

Working with files in Python: specifications 3

- You can use the os module in Python to work with file paths.
- The os.path.join() method is a platform-independent way of joining path components.
- Here's an example of how to use os.path.join() to construct a file path:

Working with files in Python: specifications 4

```python
import os
# Join path components to construct a file path
filepath = os.path.join("mydir", "myfile.txt")
# Open the file
file = open(filepath, "r")
# Read the contents of the file
data = file.read()
# Close the file
file.close()
# Print the data
print(data)
```

Working with files in Python: specifications 5

- In this example, os.path.join() joins the directory name "mydir" and the file name "myfile.txt" to construct the file path "mydir/myfile.txt".
- This is done in a platform-independent way, so the code will work on both Windows and Unix-based systems.
- By using the os module and the open() function, you can work with files and file paths in a way that is both easy and reliable.

Appending Data to a File in Python

- In addition to reading and writing data to a file, you can also append data to an existing file in Python. This allows you to add new data to the end of a file without overwriting its existing contents.

- To append data to a file, you can use the a mode modifier when opening the file. Here's an example:

```python
# Open the file in append mode
file = open("myfile.txt", "a")
# Write some data to the file
file.write("Hello, again!")
# Close the file
file.close()
```

Appending Data to a File in Python Continuity

- In this example, the file "myfile.txt" is opened in append mode using the a mode modifier.
- The write() method is used to add the string "Hello, again!" to the end of the file, without overwriting its existing contents.
- Remember to always close your files when you're done working with them,
  as this releases any system resources associated with the file and helps prevent data corruption.

Exception Handling

Exception Handling in Python

- In programming, errors and exceptions are common occurrences that can cause programs to crash or behave unexpectedly. In order to handle these errors, Python provides a mechanism called "**exception handling**."

- **Exception handling** allows you to catch and handle errors in your code so that your program can continue running gracefully even if errors occur. You can use the **try, except**, and **finally** keywords to handle exceptions in your code.

Exception Handling

Exception Handling in Python

- The **try block** contains the code that might raise an exception, and the **except block** contains the code that is executed if an exception is raised.
- You can catch specific types of exceptions by specifying the exception class after the **except** keyword.
- The **finally block** contains code that is always executed, regardless of whether an exception is raised or not. This block is typically used to clean up resources that were used in the try block.

Exception Handling

■ Here's the basic syntax for exception handling in Python:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Code to handle the actual exception
finally:
    # Code that is always executed
```

■ Exception handling make your programs more robust and less likely to crash or behave unpredictably when errors occur.

Exception Handling: Example 1

Example 1: Let's say you want to open a file in Python and perform some operations on it. However, there's a chance that the file might not exist or that you don't have permission to access it. In this case, you can use exception handling to catch the error and handle it gracefully.

Exception Handling: Example 1 Continuity

```python
try:
    f = open("myfile.txt", "r")
    # Performing some operations on our file myfile.txt
    f.close()
except FileNotFoundError:
    print("File not found.")
except PermissionError:
    print("You don't have permission to access this file.")
except Exception as e:
    print("An error occurred:", e)
```

Exception Handling: Example 2

Example 2: You might encounter an error if you try to divide a number by zero. To avoid this error, you can use exception handling.

```python
try:
        result = 10 / 0
except ZeroDivisionError:
        print("You cannot divide by zero.")
except Exception as e:
        print("An error occurred:", e)
```

Exception Handling: Example 3

Example 3: Let's say you have a list of numbers and you want to iterate over them and perform some calculations. However, some of the numbers might not be valid, so you need to use exception handling to catch the errors.

Exception Handling: Example 3 Continuity

```python
my_list = [ 1, 2, 3, "four", 5 ]
for num in my_list:
      try:
            result = 10 / num
            print(result)
      except TypeError:
            print("Invalid data type:", num)
      except ZeroDivisionError:
            print("You cannot divide by zero.")
      except Exception as e:
            print("An error occurred:", e)
```