Introduction to Python and Programming Concepts

Objectives

- Structure programmes according to an algorithm
- Master the elements of vocabulary and syntax of a language to write a programme
- Run a programme
- Debug a programme

Programme

01 **Introduction - programmes and programming rules**

02 **Variables**

03 **Operators and expressions**

04 **Control structures**

05 **Procedures and functions**

06 **Object-Oriented Programming (OOP)**

Foreword - Main source

Part of this educational material comes from the excellent course Introduction to Python programming for biology by Patrick Fuchs and Pierre Poulain, under a free licence (CC BY-SA 3.0 FR).
The adaptations made in this material relate to the visual identity of the slides and the development of particularly relevant points for this training.
Therefore, the information contained in this material is licensed under CC BY-SA 4.0.

01

Introduction - programmes and programming rules

Brief History of Python

- The Python programming language was created in 1989 by Guido van Rossum in the Netherlands.
- The name Python is a tribute to the television series Monty Python's Flying Circus, which G. van Rossum is a fan of.
- The first public version of this language was released in 1991.

Versions

- At the time of our training, the latest (sub-)version of Python is version 3.14, released in October 2025. The latest patch is 3.14.2. The (major) version 2 of Python is obsolete and is no longer maintained… do not use it (really!).

Python Documentation by Version

Python Documentation by Version

Some previous versions of the documentation remain available online. Use the list below to select a version to view.

For unreleased (in development) documentation, see In Development Versions.

- Python 3.12.2, documentation released on 6 February 2024.
- Python 3.12.1, documentation released on 8 December 2023.
- Python 3.12.0, documentation released on 2 October 2023.
- Python 3.11.8, documentation released on 6 February 2024.
- Python 3.11.7, documentation released on 4 December 2023.
- Python 3.11.6, documentation released on 2 October 2023.
- Python 3.11.5, documentation released on 24 August 2023.
- Python 3.11.4, documentation released on 6 June 2023.
- Python 3.11.3, documentation released on 5 April 2023.
- Python 3.11.2, documentation released on 8 February 2023.
- Python 3.11.1, documentation released on 6 December 2022.
- Python 3.11.0, documentation released on 24 October 2022.

- The Python Software Foundation is the organisation that manages the development of Python and leads the community of developers and users.

Features

Python has many interesting features:

- It is cross-platform. In other words, it runs on a wide range of operating systems: Windows, Mac OS X, Linux, Android, iOS, from Raspberry Pi microcomputers to supercomputers.
- It is free. You can install it on as many devices as you wish.
- It is a high-level language. To use it, relatively little knowledge of how a computer works is required. It is an interpreted language. A Python script does not need to be compiled to be executed, unlike languages like C or C++.
- It is object-oriented (but not only). In other words, it is possible to design in Python entities that imitate those of the real world (a car, a cell, etc.) with a certain number of rules of operation and interactions.
- It is relatively easy to learn...

➡ Due to these features, Python is now taught in many schools and universities, and it is also widely used in businesses. Companies use Python for various purposes such as web development, data analysis, machine learning, automation...

Download and install Python from the official site

- You can download and install Python directly from the official Python website. For example, you can follow the general steps of this tutorial, which is quite comprehensive on the different installation methods on a 'personal' machine.
- **However, it is essential to adhere to the guidelines and policies established by your company regarding the installation and use of software, in order to ensure compliance with security protocols and compatibility with existing systems.**
- There are also other options for working with Python, depending on your needs and preferences.

Download and install Python - Anaconda

- Anaconda is a popular distribution of the Python and R programming languages for scientific computing, data science, and machine learning.
- It includes a wide range of pre-installed packages and commonly used tools in data analysis and scientific research.
- Anaconda provides a user-friendly package manager called conda to manage environments and install additional packages.

Download and install Python - Docker

- Docker is a containerisation platform that allows you to package applications and their dependencies into containers.
- With Docker, you can create portable and reproducible environments for running Python applications, ensuring consistency across different systems.
- Docker containers offer isolation and flexibility, making them suitable for development, testing, and deployment workflows.
- You can find the official Python Docker images on the Docker Hub website.
- To use the official Python Docker images, you can pull the desired version of Python using Docker's command line interface (CLI). For example, to get the latest Python 3 image, you can use the following command:

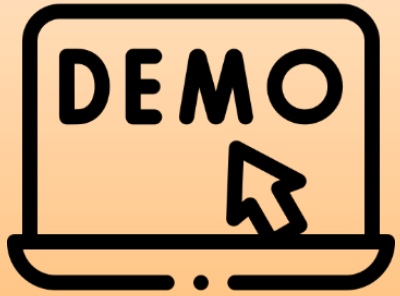
```
docker pull python:3
```

  Once the image is uploaded, you can create and run a container based on the Python image using the docker run command, specifying the additional necessary parameters.
- Docker provides comprehensive documentation and resources for working with containers and Docker images, making it easier to integrate Docker into your Python development workflow.

Manipulate Python without having to install it

- To manipulate Python without having to install it, you can use online Python interpreters or cloud-based platforms that offer Python environments.
- Online Python interpreters such as Repl.it, PythonAnywhere, IDEOne... provide online Python interpreters where you can write and execute Python code in your browser. These platforms offer basic Python environments and support for the most common Python libraries.
- Cloud-based Jupyter notebooks like Google Colab, Microsoft Azure Notebooks, or Databricks (which we will use today for those who are willing to create an account or use an existing account) provide environments where you can create and run Python notebooks without installing Python locally. These platforms offer additional features such as collaborative editing, cloud storage, and integration with other cloud services.

Download and install Python - Setup



Preparing our work environment

The Python interpreter 1/2

- Python is an interpreted language, which means that each line of code is read and interpreted to be executed by the computer. To see this in action, open a shell and enter the following command:

```
>>> python
```

- The command above will launch the Python interpreter. You should see something like this:

```
C:\Users\bspeziale>python
Python 3.14.2 (tags/v3.14.0:0fb18b0, 24 Jan 2026, 13:03:39) [MSC v.1935 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Screenshot on Windows. It would be similar on Mac and Linux.

The Python interpreter 2/2

- The Python interpreter is an interactive system in which you can enter commands that Python will execute before your eyes (when you confirm the command by pressing the Enter key). At this point, you can enter another command or exit the Python interpreter. To exit, you can type the command exit() and press Enter.
- There are many other interpreted languages like Perl or R. The great advantage of this type of language is that you can immediately test a command using the interpreter, which is very useful for debugging (finding and correcting potential errors in a program).

Editors and IDEs - Main IDEs

Major integrated development environments (IDEs) for Python:

- **Visual Studio Code (VSCode) Developed by Microsoft.**
  Lightweight yet feature-rich code editor, with a vast ecosystem of extensions.
  Supports Python through these extensions.
- **PyCharm Developed by JetBrains.**
  Powerful IDE with features such as code completion, syntax highlighting, debugging tools, integrated version control...
- **Open-source Spyder IDE.**
  Specifically designed for scientific computing and data analysis with Python.
  Includes features such as variable explorer, debugging, integration with scientific libraries...

Editors and other IDEs

- It is also possible to use web-based interactive development environments, such as JupyterLab for Jupyter notebooks, which are widely used in data science and research, and which support Python as well as other languages.
- Finally, you can use the text editor of your choice (Sublime Text, Atom, Vim, Emacs, Notepad++, Geany, Komodo Edit, Brackets, TextMate, GNU nano, Kate, gedit, Bluefish...).

Algorithm, programme - Definitions

Algorithm

- An algorithm is a finite and ordered sequence of steps or instructions that solves a problem or accomplishes a task.
- **Clarity: The steps must be clearly defined and understandable.**
- **Finiteness: The algorithm must terminate after a finite number of steps.**
- **Precision: Each step must be unambiguous.**
- **Efficiency: It must produce a correct result while being optimal in time and space.**
- **Example: Algorithm to calculate the sum of the first n integers.**

1. Initialise a variable sum to 0.
2. For each integer i from 1 to n, add i to sum.
3. Display the result sum.

Algorithm, programme - Definitions

Programme

- A programme is a sequence of instructions written in a programming language (such as Python) that are executed by a computer to accomplish a specific task.
- It contains:

  - instructions, which are the actions that the computer must perform, such as calculations, loops, or function calls.

  - structures, such as conditions (if/else), loops (for, while), and functions that organise the code.

  - data, information manipulated by the programme, stored in the form of variables, lists, dictionaries, etc.
- A Python programme can be used to automate tasks, perform calculations, manipulate files, interact with users…

A programme implements one or more algorithms using a programming language. The algorithm is therefore a kind of plan or recipe, while the programme is the concrete computer translation of that plan.

The Python .py file extension

A .py file is a Python source file containing Python code.

- Can include functions, classes, variables, and executable statements.
- Executed using the Python interpreter:

```
python filename.py
```

- Organizes and stores Python programs.
- Enables code reuse and modular programming.
- Standard file type for Python projects.
- Can be imported as a module in other Python scripts.

First Hello, World programme with Python installed

Here’s how to create your first 'Hello, World!' program in Python, depending on whether you've installed it locally or are working in an online notebook.

If you have Python installed locally:

- Open the text editor of your choice (such as Notepad on Windows, TextEdit on Mac, or a code editor like Visual Studio Code, PyCharm, etc.)
- Type the following code: print("Hello, World!")
- Save the file with the extension '.py' (for example, hello_world.py).
- Open a command prompt (terminal on Mac/Linux, Command Prompt on Windows).
- Navigate to the directory where you saved your file hello_world.py. Type the command python hello_world.py and press Enter.

First 'Hello, World' program with an online notebook

If you are working with an online notebook

- Open your online notebook platform.
- Create a new notebook or open an existing notebook.
- In a code cell, type the following code: print("Hello, World!")
- Run the code cell. In most notebooks, you can do this by pressing Shift + Enter. You should see the message "Hello, World!" printed beneath the code cell.

Programming rules: The PEPs (Python Enhancement Proposals)

- Official documents used in the Python community to propose, discuss, and document changes or developments to the Python language, libraries, or the processes overseeing its development. They help structure and standardise how ideas are presented and debated, while serving as a reference for future decisions.
- **Formal documentation:** The PEPs provide a clear and detailed description of new features or proposed changes.
- **Transparency:** They make the decision-making process participatory and open to the Python community.
- **Standardisation:** They establish standards (like coding style or the use of specific conventions) to enhance the consistency and readability of the code.
- **Historical references:** They keep track of past decisions, allowing an understanding of why certain features were adopted, modified, or rejected.

https://peps.python.org/

Programming rules: PEP 8

- "Style Guide for Python Code": one of the most iconic documents of the Python community. It defines the style conventions that Python developers must follow to write clean, readable, and consistent code.
- **Code readability:** The main objective is to improve the readability of Python code, thereby making it easier for other developers to understand.
- **Uniformity:** To promote common conventions so that all Python developers can work on projects with a uniform style.
- **Facilitating collaboration:** By following the same standards, teams can collaborate more effectively without wasting time reformatting or interpreting varied code styles.
- **Professionalism:** Adopting a consistent and professional style enhances the overall quality of Python projects.

https://peps.python.org/pep-0008/

Environments

In Python, environments are isolated spaces that allow for the management of dependencies and libraries for specific projects. They play a crucial role in version management and in avoiding dependency conflicts between different projects.

Why use environments?

- **Project isolation**

  Each project has its own libraries and versions without interfering with other projects. For example, one project uses Django 3.2 while another uses Django 4.0.

- **Avoiding dependency conflicts**

  Isolated environments ensure that project-specific dependencies do not interfere with other projects.

- **Ease of sharing and reproducibility**

  It is possible to share the exact environment of a project via a file like requirements.txt or environment.yml.

Environments

Virtual environments with venv or virtualenv

- These are local environments created for a specific project.
- Create an environment:

```
python -m venv myenv
```

```
python3.12 -m venv myenv
```

- Activate the environment:

Windows:

```
myenv\Scripts\activate
```

macOS/Linux:

```
source myenv/bin/activate
```

- Install libraries in the environment:

```
pip install <package>
```

- Deactivate the environment:

```
deactivate
```

Environments

Conda environments

- Conda is a broader tool used for managing environments and dependencies, often used with scientific projects
- Create an environment:

```
conda create -n myenv
```

```
conda create -n myenv python=3.12
```

- Activate the environment:

```
conda activate myenv
```

- Install libraries in the environment:

```
conda install <package>
```

- Deactivate the environment:

```
conda deactivate
```

Environments

Version and environment managers like pyenv

- Allow management of different versions of Python on a machine.
- Can be combined with tools like pyenv-virtualenv to create environments specific to Python versions.
- Install a version of Python:

```
pyenv install 3.14.0
```

- Create a virtual environment based on this specific version:

```
pyenv virtualenv 3.14.0 myenv
```

Environments

Files associated with environments

➡ requirements.txt (for pip):

```
pip freeze > requirements.txt
```

```
pip install –r requirements
```

```
numpy==1.21.2
pandas>=1.3.0
```

➡ environment.yml (for Conda)

```
conda env export > environment.yml
```

```
conda env create -f environment.yml
```

```
name: myenv
dependencies:
- python=3.14
- numpy
- pandas
```

Environments

- **conda in venv: This is not the ideal method, although technically possible, and it may lead to dependency management issues.**
- **pip in conda: Quite possible, and often used to add packages not present in the Conda channels. However, it is preferable to manage the main dependencies with Conda and use pip for additional packages.**

Environments

Best practices

- Create an environment for each project (this ensures isolation and avoids conflicts).
- Version control dependency files (i.e. include requirements.txt or environment.yml in version control).
- Deactivate an environment when it is no longer needed (avoids unexpected behaviour related to dependencies).
- Use Python version managers like pyenv (to easily test code on different versions of Python).

Environments

Typical workflow

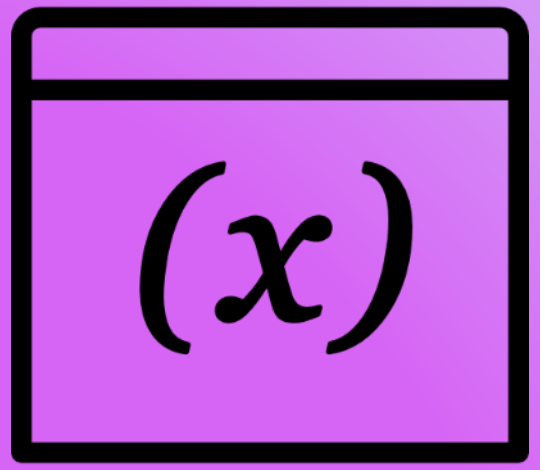
```
# Create an environment
python -m venv myenv

# Activate the environment
source myenv/bin/activate

# Install dependencies
pip install flask

# Generate a requirements file
pip freeze > requirements.txt

# Deactivate the environment
deactivate
```



02

Variables

Variables, names, and references

- In Python, a variable is a location in the computer's memory where a value is stored. For the programmer, this variable is defined by a name, whereas for the computer, it is actually an address that refers to a specific area of memory.
- In Python, declaring a variable and its initialisation (that is, assigning the first value) happens simultaneously.

```
>>> x = 2
>>> x
2
```

- On line 1 of this example, we declared and initialised the variable x with the value 2. Note that several things happened 'behind the scenes':

  1. Python 'guessed' that the variable was an integer. Python is known as a dynamically typed language.
  2. Python allocated (reserved) memory space to hold an integer. Each type of variable takes up more or less space in memory. Python also ensured that the variable could be found under the name x.
  3. Finally, Python assigned (attributed) the value 2 to the variable x.

- In other languages (such as C, for example), these different steps must be coded one by one. Python, being a high-level language, requires only the instruction x = 2 to accomplish all 3 steps at once!

Types of variables

- The type of a variable corresponds to its nature.
- We mainly use the following types to represent basic values:

int integer (integer)
float floating-point number (floating-point)
str string (string)
bool boolean (boolean)

- We will cover more advanced types later. For a comprehensive overview of existing types, you can refer to this link.
- The built-in function type() allows you to determine the type of a variable.

```
>>> x = 5
>>> print(type(x))
<class 'int'>

>>> y = 3.14
>>> print(type(y))
<class 'float'>

>>> z = "Hello, World!"
>>> print(type(z))
<class 'str'>
```

To display a result on the screen in Python, you can use the print() function.

The print() function takes one or more expressions separated by commas, converts them to strings if necessary, and then displays them on the screen.

Types of variables

To convert a Python variable to another type, you can use the built-in conversion functions to the target type. The name of these functions corresponds to the name of the associated type, as in this example:

```
>>> x = 10
>>> y = float(x)
>>> print(y)
10.0
```

Variables: operations - Operations with numerical types

- The four basic arithmetic operations are performed straightforwardly on numerical types (integers and floats):

```
>>> x = 45
>>> x + 2
47
>>> x - 2
43
>>> x * 3
135
>>> y = 2.5
>>> x - y
42.5
>>> (x * 10) + y
452.5
```

- Note, however, that if you mix integer and float types, the result is returned as a float (as this type is more general). Parentheses can also be used to manage priorities.

Variables: operations - Operations with numeric types

- The operator / performs division. Unlike the operators +, - and *, it always returns a float:

```
>>> 3 / 4
0.75
>>> 4 / 2
2.0
```

- The power operator is written as **:

```
>>> 2 ** 3
8
>>> 2 ** 4
16
```

- To obtain the quotient and the remainder of an integer division, the symbols // and % (modulo) are used respectively:

```
>>> 9 // 4
2
>>> 9 % 4
1
```

Variables: operations - Operations with numeric types

- The symbols +, -, *, /, **, //, and % are called operators, simply because they perform operations on variables.
- There are combined operators that perform an operation and an assignment in a single step. For example, the operator += adds and then assigns the result to the same variable. This operation is called increment:

```
>>> i = 0
>>> i = i + 1
>>> i
1
>>> i += 1
>>> i
2
>>> i += 2
>>> i
4
```

- The operators -=, *=, and /= behave similarly for subtraction, multiplication, and division.

Variables: operations - Operations on strings

For strings, two operations are possible: addition and multiplication.

```
>>> word = "Hi"
>>> word
'Hi'
>>> word + " Python"
'Hi Python'
>>> word * 3
'HiHiHi'
```

- The addition operator + concatenates (joins) two strings.
- The multiplication operator * between an integer and a string duplicates (repeats) a string multiple times.

Variables: operations - Operations on strings: behind the scenes

In Python, the __add__ method is a special method that allows objects to define how they behave when the + operator is used with them. This method is referred to as the 'addition' or 'concatenation' method, depending on the context in which it is used.

When you use the + operator between two objects, Python internally calls the add method of the left operand (the object to the left of the + operator) and passes the right operand as an argument to this method. The add method then performs the necessary operation and returns the result.

For example, if you have two objects a and b and you write a + b, Python translates this internally to a.__add__(b).

Variables: operations - Operations on strings

If you perform an illegal operation, you will receive an error message:

```
>>> "toto" * 1.3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError : can't multiply sequence by non-int of type 'float'
>>> "toto" + 2
Traceback (most recent call last) :
File "<stdin>", line 1, in <module>
TypeError : can only concatenate str (not "int") to str
```

Note that Python provides information in its error message. In the second example, it indicates that you must use a variable of type str, which means a string, and not an int, which means an integer.

Variables - Pratical work

Pratical work 1 ex. 1 to 3
Manipulating variables

Lists - Definition

- A list is a data structure containing a series of values.
- Python allows the construction of lists containing values of different types (for example, integer and string), which gives them great flexibility.
- A list is declared by a series of values (remember to use single or double quotes if strings are involved) separated by commas, and the whole is enclosed in brackets.
- Here are a few examples:

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> dimensions = [5, 2.5, 1.75, 0.15]
>>> mix = ['giraffe', 5, 'mouse ', 0.15]
>>> animals
['giraffe', 'tiger', 'monkey', 'mouse']
>>> dimensions
[5, 2.5, 1.75, 0.15]
mix
['giraffe', 5, 'mouse', 0.15]
```

When a list is displayed, Python presents it exactly as it was entered.

Lists - Usage

- One of the great advantages of a list is that you can access its elements by their position. This number is called the index of the list.

```
list: ['giraffe', 'tiger', 'monkey', 'mouse']
index: 0 1 2 3
```

- Please note that the indices of a list of n elements start at 0 and end at n - 1.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[0]
'giraffe'
>>> animals[1]
'tiger'
>>> animals[3]
'mouse'
```

- Consequently, if we call the element at index 4 of our list, Python will return an error message:

```
>>> animals[4]
Traceback (innermost last):
File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Lists - Operations on Lists

Like strings, lists support the + operator for concatenation, as well as the * operator for duplication:

```
>>> animals_1 = ['giraffe', 'tiger']
>>> animals_2 = ['monkey', 'mouse']
>>> animals_1 + animals_2
['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals_1 * 3
['giraffe', 'tiger', 'giraffe', 'tiger', 'giraffe', 'tiger']
```

Lists - Operations on Lists

You can also use the .append() method to add a unique element to the end of a list.

```
>>> a = []
>>> a
[]
>>> a = a + [15]
>>> a
[15]
>>> a = a + [-5]
>>> a
[15, -5]
>>> a.append(13)
>>> a
[15, -5, 13]
>>> a.append(-3)
>>> a
[15, -5, 13, -3]
```

Lists - Negative Index

```
list: ['giraffe', 'tiger', 'monkey', 'mouse']
positive index: 0 1 2 3
negative index: -4 -3 -2 -1
```

➡ It is therefore possible to access the last element of a list without using its length.

```
>>> animals = ['giraffe','tiger','monkey','mouse']
>>> animals[-1]
'mouse'
>>> animals[-2]
'monkey'
```

Lists - Slicing

It is possible to select a portion of a list using an index constructed according to the pattern [m:n] to retrieve all elements from element m inclusive to element n exclusive. In this case, we say that a slice of the list is retrieved.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[0:2]
['giraffe', 'tiger']
>>> animals[:3]
['giraffe', 'tiger', 'monkey']
>>> animals[:]
['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[1:]
['tiger', 'monkey', 'mouse']
>>> animals[1 :-1]
['tiger', 'monkey']
```

Lists - Slicing

You can also specify the step size by adding an additional ':' symbol and indicating the step size with an integer.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[:3:2]
['giraffe', 'monkey']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
>>> x[6:2:-3]
[6, 3]
```

Lists - Slicing, in summary

Accessing the content of a list is based on the pattern

list[start:end:step]

where start is included and end is excluded.

Lists - The len() function

The len() function is used to find out the length of a list, that is, the number of elements that the list contains.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> len(animals)
4
>>> len([1, 2, 3, 4, 5, 6, 7, 8])
8
```

Lists - The list() and range() functions

```
>>> iterable = (1, 2, 3, 4, 5)
>>> new_list = list(iterable)
>>> print(new_list)
[1, 2, 3, 4, 5] Output: [1, 2,
```

In Python, the list() function is used to create a new list object (from an Iterable, which we will see later).

The range() function is a special function in Python that generates integers within a given range. When used in combination with the list() function, it produces a list of integers.

The range() function works according to the model range([start,] end[, step]). The arguments in square brackets are optional. If a single argument is provided, it corresponds to the end, in which case the start is 0 and the step is 1. If two arguments are provided, they correspond to the start and the end, and the step is 1.

```
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
>>> list(range(15, 20))
[15, 16, 17, 18, 19]
>>> list(range(0, 1000, 200))
[0, 200, 400, 600, 800]
>>> list(range(2, -2, -1))
[2, 1, 0, -1]
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Lists - An example of a list of lists

```
>>> exhibit_1 = ['giraffe', 4]
>>> exhibit_2 = ['tiger', 2]
>>> exhibit_3 = ['monkey', 5]
>>> zoo = [exhibit_1, exhibit_2, exhibit_3]
>>> zoo
[['giraffe', 4], ['tiger', 2], ['monkey', 5]]
```

```
>>> zoo[1]
['tiger', 2]
>>> zoo[1][0]
'tiger'
>>> zoo[1][1]
2
```

Collections - Pratical work

Pratical work 2
Manipulating lists

Tuples - Comparison with lists

Tuples are similar to lists, with a few fundamental differences. They are defined with parentheses instead of brackets, and are immutable.

	Tuple	List
Syntax	Parentheses : (1, 2, 3)	Square brackets : [1, 2, 3]
Mutability	**Immutable** (cannot be modified after creation)	**Mutable** (can be modified after creation)
Performance	Faster in terms of access and processing	Slightly slower due to mutability
Size	Fixed size once created	Variable size (elements can be added or removed)
Common use	When data integrity is important (e.g. GPS coordinates of a reference point)	When data manipulation is needed (e.g. a shopping list)
Available methods	Fewer methods available (e.g. no append(), remove())	More methods available (append(), remove(), etc.)

Tuples - Comparison with lists

In particular, the following syntax will raise an error:

```
my_tuple = (1, 2, 3)
my_tuple[0] = 4 # will raise an error
>>> TypeError: 'tuple' object does not support item assignment
```

Additional information on strings - Strings as sequences of characters

Character strings can be considered as 'lists' (of characters) of a particular kind:

```
>>> animal = 'big tiger'
>>> animal[:5]
'big t'
>>> animal[6:]
'ger'
>>> animal[:-2]
'big tig'
>>> animal[1:-2:2]
'i i'
```

Additional information on strings - Strings as sequences of characters

However, unlike lists, strings have a notable difference: they cannot be modified. Once a string has been defined, you cannot change any of its elements.

```
>>> animal = "big tiger"
>>> animal[5]
't'
>>> animal[5] = "T"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- For this reason, we might refer to them as "sequences" of characters, a broader term than "lists" that does not imply certain properties, particularly mutability.

- If you wish to modify a string, you will need to create a new one. To do this, remember that concatenation (+) and duplication (*) operators can be useful. You can also generate a list, which can be modified, and then revert to a string.

Additional information on lists - Other list methods: the .insert() method

The .insert() method inserts an object into a list at a given position (index):

```
>>> a = [1, 2, 3]
>>> a.insert(2, -15)
>>> a
[1, 2, -15, 3]
```

Additional information on lists - Other list methods: del

The del statement removes an element from a list at a given index. Unlike other list-related methods, del is a general Python statement that can be used for objects other than lists. It does not require parentheses.

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> a
[1, 3]
```

Additional information on lists - Other methods of lists: the .remove() method

The .remove() method removes an element from a list based on its value. More specifically, it removes the first occurrence of the specified element.

```
>>> a = [1, 2, 3]
>>> a.remove(3)
>>> a
[1, 2]
```

```
>>> b = [1, 2, 3, 2]
>>> b.remove(2)
>>> b
[1, 3, 2]
>>> b.remove(2)
>>> b
[1, 3]
```

Additional information on lists - Other list methods: the .sort() and reverse() methods

The .sort() method sorts a list, by default, from smallest to largest.

```
>>> a = [4, 1, 2]
>>> a.sort()
>>> a
[1, 2, 4]
```

The .reverse() method reverses a list.

```
>>> a = [4, 1, 2]
>>> a.reverse()
>>> a
[2, 1, 4]
```

Note that the .sort() followed by .reverse() sorts from largest to smallest. For the sake of simplicity, reverse can also be passed as an argument to .sort() to sort from largest to smallest.

```
>>> a = [4, 1, 2]
>>> a.sort(reverse=True)
>>> a
[4, 2, 1]
```

Additional information on Lists - Membership

The 'in' operator checks if an element is part of a list.

```
my_list = [1, 3, 5, 7, 9]
>>> 3 in my_list
True
>>> 4 in my_list
False
```

The variation with not allows, on the contrary, to check that an element is not in a list.

```
>>> 3 not in my_list
False
>>> 4 not in my_list
True
```

If you remember the slides on booleans, it would also be possible to reverse the resulting boolean obtained.

```
>>> not (3 in my_list)
False
>>> not 4 not in my_list
True
```

Additional information on Lists - Mutability of Lists and Copies

Mutable Objects

When you have a mutable object, such as a list, a dictionary, or an instance of a custom class, you can modify its content directly. This means that if you have two variables referencing the same mutable object, changes made via one variable will be reflected in the other variable, as both point to the same object in memory.

Copy by Reference

When you create a copy by reference, any subsequent changes made to the original object will be visible through the copied reference, and vice versa. The opposite would be a copy by value.

When you perform a copy operation on a mutable object in Python (like lists) using the method =, you create a new reference to the same object.

```
>>> x = [1, 2, 3]
>>> y = x
>>> y
[1, 2, 3]
>>> x[1] = -15
>>> x
[1, -15, 3]
>>> y
[1, -15, 3]
```

Python Tutor

Additional information on lists - Mutability of lists and copies

- Using the .copy() method on a list in Python creates a shallow copy of that list, meaning it copies the elements of the list into a new list object. Consequently, changes made to the original list will not affect the copied list, and vice versa.
- However, a shallow copy means that the top-level structure of the list is duplicated, but the elements themselves are not copied recursively. In other words, if the list contains references to other mutable objects (such as other lists or dictionaries), these references are copied, but the objects themselves are not duplicated.

```
>>> x = [1, 2, 3]
>>> y = x.copy()
>>> x[1] = -15
>>> x
[1, -15, 3]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, [7, 5]]
>>> y = x.copy()
>>> x[1].append(3)
>>> x
[1, 2, [7, 5, 3]]
>>> y
[1, 2, [7, 5, 3]]
```

Additional information on Lists - Mutability of Lists and Copies

To make a deep copy of a list in Python, you can use the copy.deepcopy() function from the copy module. This function creates a new object and recursively copies the original object as well as all its nested objects. This ensures that modifications made to the original object do not affect the copied object, even if the original object contains nested mutable objects.

```
>>> import copy
>>> x = [[1, 2], [3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> y = copy.deepcopy(x)
>>> x[1][1] = 99
>>> x
[[1, 2], [3, 99]]
>>> y
[[1, 2], [3, 4]]
```

Collections - Dictionaries

- Used to store collections of data in the form of key-value pairs.
- Each element in a dictionary is accessible by its key, rather than by its index as in lists or tuples.
- Dictionaries are mutable, meaning they can be modified after their creation.

```
>>> animal_1 = {}
>>> animal_1["name"] = "giraffe"
>>> animal_1["height"] = 5.0
>>> animal_1["weight"] = 1100
>>> animal_1
{'name' : 'giraffe', 'height' : 5.0, 'weight' : 1100}
```

```
>>> animal_1["height"]
5.0
>>> animal_1["sex"]
KeyError: 'sex'
>>> animal_1.get("height")
5.0
>>> animal_1.get("sex")
None
```

```
>>> animal_2 = {"name" : "monkey", "weight" : 70, "height" : 1.75}
>>> animal_2["age"] = 15
```

Collections - Dictionaries: .keys(), .values()

```
>>> animal_2.keys()
dict_keys(['weight', 'name', 'height'])
>>> animal_2.values()
dict_values([70, 'monkey', 1.75])
```

The mentions dict_keys and dict_values indicate that we are dealing with somewhat special objects.

They are not indexable (we cannot retrieve an element by index, for example some_dict.keys()[0] will raise an error).

```
>>> animal_2.keys()[0]
... TypeError: 'dict_keys' object is not subscriptable
```

If necessary, we can convert them to a list using the list() function. However, they are 'iterable' objects, so they can be used directly in a loop (we will revisit this in the next section dedicated to loops).

```
>>> for val in animal_2.values():
... print(val)
...
70
'monkey'
1.75
```

Collections - Dictionaries: .items()

The .items() method returns a view object that displays a list of a dictionary's key-value pairs. These pairs are presented as tuples, with each tuple containing a key and its corresponding value. This method allows you to access and iterate over both the keys and values of the dictionary simultaneously.

```
>>> a = {0 : 't', 1 : 'o', 2 : 't', 3 : 'o'}
>>> a.items()
dict_items([(0, 't'), (1, 'o'), (2, 't'), (3, 'o')])
```

Collections - Lists of dictionnaries

- When we create a list of dictionaries with shared keys, we are effectively building a structure similar to a database. Unlike lists, dictionaries provide a more explicit way to manage complex structures. Each dictionary entry represents a record, with keys serving as field names and the corresponding values holding the data associated with each field.
- This approach provides clarity and organization, making it easier to access and manipulate data within the structure. Dictionaries offer flexibility in representing real-world entities, allowing for more intuitive data management in Python programs.

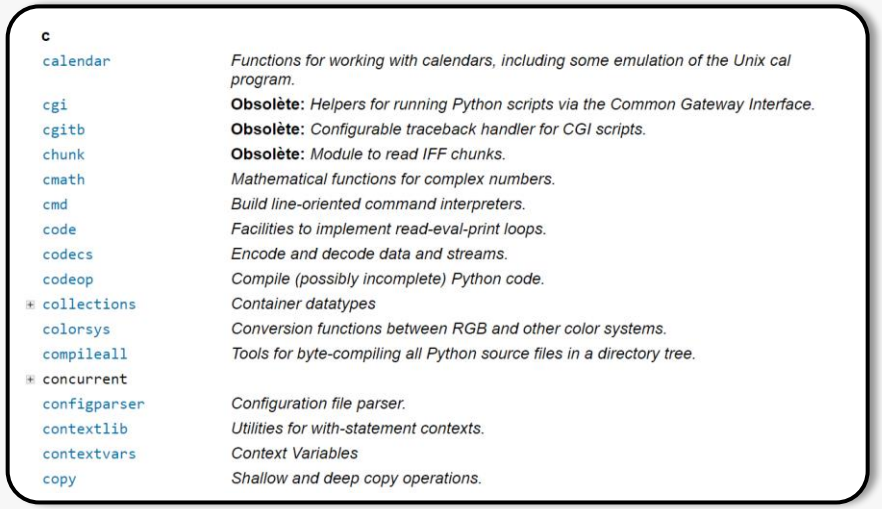
```
>>> animals = [animal_1, animal_2]
>>> animals
[{'name': 'giraffe', 'weight': 1100, 'height': 5.0}, {'name': 'monkey',
'weight': 70, 'height': 1.75}]
>>>
>>> for animal in animals :
...   print(animal['name'])
...
giraffe
monkey
```

Interlude

Some additional information on modules

Modules - What are they?

- Modules in Python are collections of reusable functions and code that developers can use to perform various tasks. In reality, they are simply Python files containing Python code.
- Although Python's standard library includes many modules covering a wide range of functionality, developers have also created countless third-party modules available for use.
- It is strongly recommended to check if the functionality you need already exists in the form of a module before writing your own code. The official Python documentation provides detailed information on standard modules, and additional documentation for third-party modules can often be found online.



c

calendar	*Functions for working with calendars, including some emulation of the Unix cal program.*
cgi	**Obsolète:** *Helpers for running Python scripts via the Common Gateway Interface.*
cgitb	**Obsolète:** *Configurable traceback handler for CGI scripts.*
chunk	**Obsolète:** *Module to read IFF chunks.*
cmath	*Mathematical functions for complex numbers.*
cmd	*Build line-oriented command interpreters.*
code	*Facilities to implement read-eval-print loops.*
codecs	*Encode and decode data and streams.*
codeop	*Compile (possibly incomplete) Python code.*
collections	*Container datatypes*
colorsys	*Conversion functions between RGB and other color systems.*
compileall	*Tools for byte-compiling all Python source files in a directory tree.*
concurrent	
configparser	*Configuration file parser.*
contextlib	*Utilities for with-statement contexts.*
contextvars	*Context Variables*
copy	*Shallow and deep copy operations.*

Although the standard library of Python alone contains over 300 modules, the Python ecosystem includes thousands of additional modules available on platforms like the Python Package Index (PyPI) and other repositories.

Modules - Importing modules

```
>>> import random
>>> random.randint(0, 10)
4
```

- In line 1, the import statement provides access to all the functions from the random module.
- In line 2, we use the randint() function from the random module. This function returns a randomly generated integer between 0 and 10.

```
>>> import math
>>> math.cos(math.pi / 2)
6.123233995736766e-17
>>> math.sin(math.pi / 2)
1.0
```

Modules - Importing modules

There is another way to import one or more functions from a module. By using the keyword from, you can import a specific function from a given module. In this case, you do not need to repeat the name of the module when using it, just the name of the function concerned.

```
>>> from random import randint
>>> randint(0,10)
6
```

Modules - Importing modules

Finally, you can also import all the functions of a module using import *.

```
>>> from random import *
>>> randint(0,10)
3
>>> uniform(0,2.5)
0.7494317476072795 1
```

This is a syntax you may encounter, but it is generally discouraged in production code as it can lead to namespace pollution and make the code harder to read and maintain. It imports all names defined in the module into the current namespace, which can cause conflicts and make it difficult to trace the origin of functions and variables.

Use the other two syntaxes.

Modules - Get help on imported modules

The help() command is actually a more general command that provides assistance on any object loaded in memory.

You can use the built-in help() function.

```
>>> import random
>>> help(random)
```

```
Help on module random :

NAME
random - Random variable generators.

MODULE REFERENCE
https ://docs.python.org/3.7/library/random
The following documentation is automatically generated from the Python
source files. It may be incomplete, incorrect or include features that
are considered implementation detail and may vary between Python
implementations. When in doubt, consult the module reference at the
location listed above.

DESCRIPTION
integers
    --------
uniform within range

sequences
    ---------
pick random element
pick random sample
```

Common modules - Some common modules

- There are a number of modules that you are likely to use if you programme in Python.
  - `math` provides basic mathematical functions and constants (sin, cos, exp, pi...),
  - `sys` interacts with the Python interpreter, notably for passing command-line arguments and accessing information about the execution environment,
  - `os` facilitates interaction with the operating system, allowing for operations such as file and directory manipulation, as well as accessing environment variables,
  - `random` generates random numbers,
  - `time` provides access to the computer's time and time management functions,
  - `urllib` retrieves data from the Internet directly within Python,
  - `re` handles regular expressions.
- Feel free to check the module page on the official Python website.
- There are many other external modules that are not installed by default with Python but are widely used in practice: Flask, Requests, TensorFlow, scikit-learn, Matplotlib, seaborn, SQLAlchemy…

Modules, packages, libraries, frameworks

In the world of Python development, there are a few distinct concepts that form the foundations of code structure and influence how developers organise and reuse code to build robust and modular applications: modules, packages, libraries (sometimes referred to as 'librairies' in a borrowing from English) and frameworks (which are rarely translated into French).

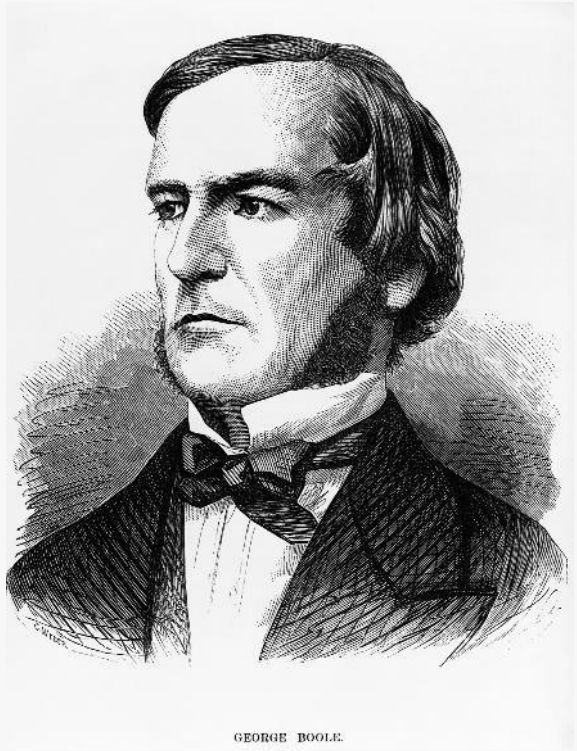
I recommend reading this excellent article from LearnPython that explains the utility of and the differences between these four concepts: Python Modules, Packages, Libraries, and Frameworks.

03

Operators and expressions

Booleans - The 'boolean' data type

- Booleans are a type of data that represents one of two possible values: true or false.
- Booleans are named after the mathematician George Boole, who first formulated Boolean algebra, a branch of algebra where the variables are either true or false.
- In Python, the boolean data type is represented by the keywords True and False. These keywords are case-sensitive.



Booleans - Usage

- Booleans can be combined using logical operators such as and, or, and not to perform logical operations.
- Booleans are often used in expressions involving comparison operators such as == (equal to), != (not equal to), < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to).
- Booleans are commonly used in conditional statements such as if, elif, and else to control the flow of execution of the programme based on certain conditions.

Booleans - Comparisons

Python Syntax Meaning

== is equal to
!= is not equal to
> is strictly greater than
>= is greater than or equal to
< is strictly less than
<= is less than or equal to

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
>>> x < 10
True
```

```
>>> animal = "tiger"
>>> animal == "tig"
False
>>> animal != "tig"
True
>>> animal == "tiger"
True
```

```
>>> "a" < "b"
True
>>> "ali" < "alo"
True
>>> "abb" < "ada"
True
```

Booleans - Conditional Expressions

Conditional tests are an essential part of any programming language if one wishes to give it a certain complexity, as they allow the computer to make decisions. For this, Python uses the if statement along with a comparison.

```
>>> x = 2
>>> if x == 2:
...   print("The test is true!")
...
The test is true!
```

```
>>> x = "mouse"
>>> if x == "tiger":
...   print("The test is true!")
...
```

- In the first example, since the test is true, the instruction print("The test is true!") is executed.
- In the second example, the test is false and nothing is displayed.
- The instruction blocks in tests must be indented. The indentation indicates the scope of the instructions to execute if the test is true.
- The line containing the if statement ends with the colon character ":".

Booleans - Conditional expressions: else

Sometimes, it is useful to test whether a condition is true or false in the same if statement:

```
>>> if x == 2:
...   print("The test is true!")
... else:
...   print("The test is false!")
...
The test is true!
```

```
>>> import random
>>> base = random.choice(["a", "t", "c", "g"])
>>> if base == "a:",
...   print("Choice: adenine")
... elif base == "t":
...   print("Choice: thymine")
... elif base == "c":
...   print("Choice: cytosine")
... elif base == "g":
...   print("Choice: guanine")
...
Choice: cytosine
```

Booleans - Multiple Tests

```
True OR True = True
True OR False = True
False OR True = True
False OR False = False
```

```
True AND True = True
True AND False = False
False AND True = False
False AND False = False
```

Booleans - Multiple Tests

In Python, we use the reserved word and for the AND operator and the reserved word or for the OR operator. Pay attention to the case of the operators and and or, which are written in lowercase in Python.

```
>>> x = 2
>>> y = 2
>>> if x == 2 and y == 2:
... print("x and y are both 2")
...
x and y are both 2
```

The same result would be obtained by using two nested if statements:

```
>>> x = 2
>>> y = 2
>>> if x == 2:
...   if y == 2:
...     print("x and y are both 2")
...
x and y are both 2
```

Booleans - Multiple Tests

You can also directly test the effect of these operators using True and False (case-sensitive).

```
>>> True or False
True
```

Finally, you can use the logical negation operator not, which reverses the result of a condition:

```
>>> not True
False
>>> not False
True
>>> not (True and True)
False
```

Booleans - Test for the value of floats in data analysis (1/3)

When you want to test the value of a floating variable, your initial instinct would be to use the equality operator, like:

```
>>> 1/10 == 0.1
True
```

However, it is not really advisable to do so. Python stores the numeric values of floats as floating-point numbers (hence the name!), which leads to certain limitations.

```
>>> (3 - 2.7) == 0.3
False
>>> 3 - 2.7
0.2999999999999998
```

Booleans - Test for the value of floats in data analysis (2/3)

- In fact, this issue does not arise from Python, but rather from the way a computer handles floating-point numbers (like a ratio of binary numbers).
- This means that some floating values can only be approximations.
- One way to realise this is to use formatted writing by requesting the display of a large number of decimals:

```
>>> 0.3
0.3
>>> "{:.5f}".format(0.3)
'0.30000'
>>> "{:.60f}".format(0.3)
'0.299999999999999988897769753748434595763683319091796875000000'
>>> "{:.60f}".format(3.0 - 2.7)
'0.299999999999999822364316059974953532218933105468750000000000'
```

Booleans - Test for the value of floats in data analysis (3/3)

For these reasons, you should not test if a float is equal to a certain value. The best practice is to check if a float lies within a range of a certain precision.

```
>>> delta = 0.0001
>>> var = 3.0 - 2.7
>>> 0.3 - delta < var < 0.3 + delta
True
>>> abs(var - 0.3) < delta
True
```

Boolean variables practical work

Practical work 1 ex. 4

Manipulating variables

04

Control structures

FOR loops - Principle

- In programming, you often need to repeat an instruction multiple times.
- Loops are an essential part of any programming language and help us do this in a compact and efficient manner.
- Imagine, for example, that you want to display the elements of a list one after the other. With what we have covered so far in this training, you would have to type something like :

```
animals = ['giraffe', 'tiger', 'monkey', 'mouse']
print(animals[0])
print(animals[1])
print(animals[2])
print(animals[3])
```

- If your list only contains 4 items, it's still manageable, but imagine if it contained 1000...
- To remedy this, we use loops.

FOR loops - Principle

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...     print(animal)
...
giraffe
tiger
monkey
mouse
```

- The variable animal is called the iteration variable and takes on the different values from the animal list at each iteration of the loop.
- You can choose whatever name you like for this variable. It is created by Python the first time the line containing it is executed (if it already existed, its content would be overwritten). Once the loop is finished, this iteration variable animal will not be destroyed and will therefore hold the last value from the animal list (in this case, the string mouse).

FOR loops - Principle

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...   print(animal)
...
giraffe
tiger
monkey
mouse
```

- Note the types of variables used here: animals is a list that we iterate over, and animal is a string, as each element of the list is a string.
- The iteration variable can be of any type, depending on the list being iterated over. In Python, a loop always iterates over a so-called sequential object (that is, an object made up of other objects) such as a list.
- It is possible to iterate over other sequential objects with a loop.

FOR loops - Principle

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...   print(animal)
...
giraffe
tiger
monkey
mouse
```

- Note also the ':' at the end of the line starting with for. This means that the for loop expects a block of instructions, in this case, all the instructions that Python will repeat on each iteration of the loop. This block of instructions is called the body of the loop.
- How do we indicate to Python where this block starts and ends? This is indicated solely by indentation, that is, by shifting the line(s) of the block of instructions to the right.

Iterables - Some definitions

List A list is a built-in data structure in Python that contains an ordered collection of items.

You can iterate over a list using a for loop to access each element sequentially.

```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
  print(item)
```

Sequence In Python, a sequence is a generic term that refers to any iterable object that supports indexed access, such as strings, lists, tuples, and ranges. As a result, it has a known length. Some properties, such as mutability, are not fixed.

You can iterate over sequences using a for loop to access each element.

```
my_string = "Hello, world!"
for char in my_string:
  print(char)
```

Iterable An iterable is any object in Python that can be iterated over, meaning it can produce one element at a time sequentially. In addition to sequences, other examples of iterables include dictionaries, sets, file objects, and generator objects.

```
my_dict = {"a": 1, "b": 2,
"c": 3}
for key in my_dict:
  print(key, my_dict[key])
```

Iterables - Overview

A loop in Python can iterate over various data structures, all of which are very useful in data analysis:

Lists A loop can iterate over the elements of a list.
Tuples Similarly, a loop can iterate over the elements of a tuple.
Strings A loop can iterate over the characters of a string.
Ranges A loop can iterate over the elements of a range object.
Dictionaries A loop can iterate over the keys, values, or key-value pairs of a dictionary.
Sets A loop can iterate over the elements of a set.
File objects A loop can iterate over the lines of a file object when reading a file.

In reality, a loop in Python can iterate over any iterable, which means any object that supports the iteration protocol: if an object implements the __iter__() method to return an iterator, a loop can iterate over the elements produced by that iterator.

WHILE loops - Description

An alternative to the commonly used for statement in programming is the while loop. A series of statements is executed as long as a condition is true.

```
>>> i = 1
>>> while i <= 4:
...     print(i)
...     i += 1
...
1
2
3
4
```

A while loop generally requires three elements to function correctly:

- 1. Initialising the iteration variable before the loop (line 1).
- 2. Testing the iteration variable associated with the while statement (line 2).
- 3. Updating the iteration variable in the body of the loop (line 4).

WHILE loops - Example

A while loop combined with the input() function can be very useful when you want to ask the user for a numeric value.

```
>>> i = 0
>>> while i < 10:
...     response = input("Enter an integer greater than 10:")
...     i = int(response)
...
Enter an integer greater than 10: 4
Enter an integer greater than 10: -3
Enter an integer greater than 10: 15
>>> i
15
```

The input() function takes a prompt (as a string) as an argument, asks the user to enter a value, and returns it as a string. This value must then be converted to an integer (using the int() function).

Loops - The importance of indentation

Code 1:

```
numbers = [4, 5, 6]
for nb in numbers:
  if nb == 5:
    print("The test is true")
    print(f"because nb is {nb}")
```

Code 2:

```
numbers = [4, 5, 6]
for nb in numbers:
  if nb == 5:
    print("The test is true")
  print(f"because nb is {nb}")
```

Result 1:

```
The test is true
because nb is 5
```

Result 2:

```
because nb is 4
The test is true
because nb is 5
because nb is 6
```

Loops - The importance of indentation

An error often leads to "infinite loops" (which never stop).

You can always stop the execution of a Python script with the Ctrl-C key combination (that is, by pressing the Ctrl and C keys simultaneously).

```
i = 0
while i < 10:
  print("Python is cool!")
```

Here, we omitted updating the variable i in the body of the loop. As a result, the loop will never stop (except by pressing Ctrl-C), since the condition i < 10 will always be true.

Loops - BREAK and CONTINUE statements

These two statements can be used to modify the behaviour of a loop (for or while) with a test.

- The break statement 'exits' (stops) the loop.

```
>>> for i in range(5):
...   if (i > 2) and (i < 4):
...     break
...   print(i)
...
0
1
2
```

- The continue statement goes to the next iteration, without executing the rest of the block of instructions in the loop.

```
>>> for i in range(5):
...   if (i > 2) and (i < 4):
...     continue
...   print(i)
...
0
1
2
4
```

Loops: back to lists - Iteration over lists: iterating over indices

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for i in range(4):
...   print(animals[i])
...
giraffe
tiger
monkey
mouse
```

Loops: back to lists - Iteration over lists: direct iteration

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...   print(animal)
...
giraffe
tiger
monkey
mouse
```

Loops: back to lists - Iteration over lists: enumerate

- The most efficient method is the second, which iterates directly over the elements.
- However, there may be cases during a loop where we need the indices. In this case, we will need to iterate over the indices.
- Python provides the enumerate() function, which allows us to iterate over both the indices and the elements themselves. This offers a convenient way to access the elements and their corresponding indices in a loop.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for i, animal in enumerate(animals):
...   print("Animal {} is a(n) {}").format(i, animal)
...
Animal 0 is a(n) giraffe
Animal 1 is a(n) tiger
Animal 2 is a(n) monkey
Animal 3 is a mouse
```

Loops: back to lists - Iteration over lists: list comprehension

List comprehensions in Python offer several advantages, making the code more concise and readable (once you get used to it) on the one hand, and more efficient on the other.

```
>>> x = []
>>> for i in range(21):
...     if i % 2 == 0:
...         x.append(i)
...
>>> print(x)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
>>> x = [i for i in range(21) if i % 2 == 0]
>>> print(x)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Loops - practical work

Practical work 3

Manipulating loops (and lists)

Loops and condition tests - practical work

Practical work 4
Loops and condition tests

Returning to collections in light of lists - practical work

Practical work 5
Other collections

05

Procedures and functions

Principles and generalities of functions

- In programming, functions are very useful for performing the same operation multiple times within a programme. They also make the code clearer and more readable by dividing it into logical blocks.
- When used, a function can be seen as a sort of 'black box':

1. To which you pass no, one, or several variable(s) in brackets. These variables are called arguments. They can be of any type of Python object.
2. That executes an action.
3. And that returns a Python object or nothing at all.

Functions - Examples

```
>>> len([0, 1, 2])
3
```

1. We call the len() function, passing a list as an argument (in this case, the list [0, 1, 2]).
2. The function calculates the length of this list (in this case, 3).
3. It returns an integer equal to that length.

Functions - Examples

```
>>> some_list.append(5)
```

1. You pass the integer 5 as an argument.
2. The .append() method adds the integer 5 to the some_list object.
3. It returns nothing.

Functions - Modularity

- A function accomplishes a task. To do this, it can receive arguments and return a result. The algorithm used within the function has no direct interest for the user.

  ➡ For example, it is not necessary to know how the math.cos() function calculates a cosine. All you need to know is that it requires an angle in radians as an argument and it will return the cosine of that angle. What happens inside the function is solely the programmer's business.

- In general, each function performs a unique and specific task. If things become complex, it is recommended to write several functions (which can also call each other). This modularity improves the overall quality of the code as well as its readability.

Functions - Definition of a function

- To define a function, Python uses the keyword def.
- If you want the function to return something, use the keyword return. When a return statement is encountered, the execution of the function call stops immediately.

```
>>> def square(x):
...   return x ** 2
...
>>> print(square(3))
9
```

```
>>> res = square(3)
>>> print(res)
9
```

```
>>> def hello():
...   print("hi")
...
>>> hello()
hi
```

```
>>> var = hello()
hi
>>> print(var)
None
```

Functions - Return multiple results

- Functions can return multiple objects at once.

```
>>> def square_cube(x):
...   return x**2, x**3
...
>>> square_cube(3)
(9, 27)
```

- In reality, Python only returns a single object, but this object can be sequential, meaning it can contain other objects. In our example, Python returns an object of type tuple. We could just as easily have written our function to return a list.

```
>>> def square_cube(x):
...   return [x**2, x**3]
...
>>> square_cube(3)
[9, 27]
```

- Returning a tuple or a list containing two or more elements is compatible with multiple assignment, allowing for the retrieval of several values returned by a function and assigning them to different variables in one operation.

```
>>> z1, z2 = square_cube(3)
>>> z1
9
>>> z2
27
```

Functions - Definition of a function with default argument values

```
>>> def sum(a, b=1):
...   return a + b
...
>>> sum(3)
4
>>> sum(3, 3)
6
```

Functions - Positional and keyword arguments

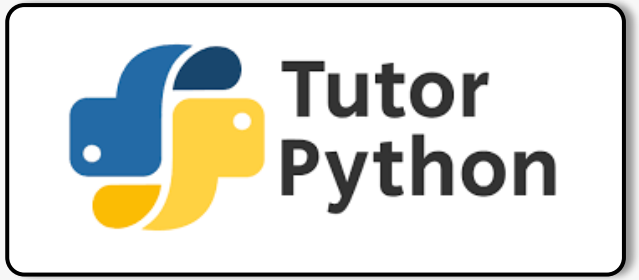
```
>>> def fct(a, b):
...   return a - b
...
>>> fct(3, 2)
1
>>> fct(a=7, b=2)
5
>>> fct(b=12, a=14)
2
```

Functions - Call a function within another function

- When working with functions, it is essential to understand the behaviour of variables.
- A variable is said to be local when it is created inside a function. It exists and is visible only during the execution of that function.
- A variable is said to be global when it is created in the main programme. It is visible everywhere in the programme.

```
def square(x):
  y = x ** 2
  return y

z = 5
result = square(z)
print(result)
```



Python Tutor

LGI Rule

When Python encounters a variable, it will resolve its name with particular priorities:

1. Firstly, it will check if the variable is local.
2. Next, if it does not find the variable locally, it will check if it is global.
3. Finally, if it is not global, it will check if it is internal (for example, the len() function is regarded as an internal function in Python, it exists whenever you start Python).

LGI = Local > Global > Internal

```
>>> def my_function():
...     x = 4
...     print(f'Function: x = {x}')
...
>>> x = -15
>>> my_function()
Function: x = 4
>>> print('Module: x = {x}')
Module: x = -15
```

Functions - Practical work

Practical work 6
Manipulating functions

Lambda Expressions

- A lambda function is an anonymous function, that is to say, a function that does not need to be defined with the def keyword and has no specific name. It is often used for simple operations, particularly when it is unnecessary to define a complete function for a task that will only be used once.
- The syntax is as follows:

```
lambda arguments: expression
```

Example of use:

```
items = [(1, 'banana'), (2, 'apple'), (3, 'pear')]
sorted_items = sorted(items, key=lambda item: item[1])
print(sorted_items)
```

Lambda Expressions

Interest:

- **Conciseness:** They allow you to write short and simple functions in a compact way, without having to use multiple lines for a complete definition.
- **Temporary use:** They are more optimal in cases where the function will only be used once or in a local situation (for example, in functions like map(), filter(), or sorted()), as they do not involve storing in a variable.
- **Clarity (in certain situations):** In particular when using functions that accept other functions as arguments.

Limits:

- **Reduced readability:** Lambda functions can make the code less readable if they become too complex.
- **No documentation:** Unlike traditional functions, lambda functions cannot have docstrings (explanatory comments).

It should be noted that it is possible to store a function defined by a lambda expression in a variable, but the most common usage is to switch to the syntax with def.

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

Built-in functions

Built-in functions are functions that are available by default in a programming language, without needing to import any module. In Python, examples include:

- `print()`
- `len()`
- `type()`
- `sum()`
- `range()`

They are used for common, fundamental operations.

Lists, loops, a bit of everything in fact...

Pratical work 7
Warm-up...

FINAL BOSS

Object-oriented programming in Python

Defining a class

A class is a blueprint that defines the structure and behaviour of its objects.

Basic syntax:

```
class ClassName:
  # constructor (special method called at creation)
  def __init__(self, param1, param2):
    self.param1 = param1
    self.param2 = param2

  # a method (behaviour)
  def method_1(self):
    pass
```

```
class Car:
  def __init__(self, brand, speed):
    self.brand = brand
    self.speed = speed

  def drive(self):
    print(f"The {self.brand} is going at {self.speed} km/h.")
```

Object-oriented programming in Python

Instantiating (creating) objects

Instantiation consists of creating an object from the class, as if we were using a "mould".

```
v1 = Car("Peugeot", 120)
v2 = Car("Renault", 90)
```

v1 and v2 are two different instances of the Car class. Each has its own attributes (brand, speed).

Once an object is created, it is possible to access its attributes and methods.

```
print(v1.brand) # Peugeot
print(v2.speed) # 90

v1.drive() # The Peugeot is going at 120 km/h.
v2.drive() # The Renault is going at 90 km/h.
```

Object-oriented programming in Python - The role of the self parameter

In a class's methods, the first parameter is always self. It represents the current instance (the object you are working on). Python automatically passes it during the call.

```
class Car:
  def __init__(self, brand, speed):
    self.brand = brand
    self.speed = speed

  def drive(self):
    print(f"The {self.brand} is going at {self.speed} km/h.")
```

Object-oriented programming in Python - Modifying and adding attributes

It is possible to modify and/or add attributes on the fly.

```
v1.colour = "Red" # Adding a new attribute
print(v1.colour) # Red
```

Object-oriented programming in Python - Class attribute vs instance attribute

So far we have seen attributes referred to as 'instance' attributes. A class attribute is shared by all instances of a class.

```
class Car:
  wheels = 4 # class attribute, all cars have 4 wheels

  def __init__(self, brand, speed):
    self.brand = brand # instance attribute
    self.speed = speed # instance attribute

  def drive(self):
    print(f"The {self.brand} is driving at {self.speed} km/h.")
```

Object-oriented programming in Python – Practical work

Practical work - Part 1
OOP

Object-oriented programming in Python - Single (or simple) inheritance, multiple inheritance, polymorphism

Single inheritance

Involves creating a child class that inherits from a single parent class. This allows the child class to reuse the methods and attributes of the parent class.

```
class Animal:
  def __init__(self, name):
    self.name = name

  def speak(self):
    print(f"{self.name} makes a noise") # Displays a generic noise

# Child class inheriting from Animal
class Dog(Animal):
  def speak(self):
    print(f"{self.name} barks") # Redefinition of the speak method

# Test
my_dog = Dog("Rex")
my_dog.speak() # Displays: Rex barks
```

Object-oriented programming in Python - Single (or simple) inheritance, multiple inheritance, polymorphism

➡ **The inheritance system is dynamic!**

Python supports method resolution order (MRO).

```
class A:
  pass
class B(A):
  pass
print(B.__mro__)
# (<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Inheritance can even be dynamically modified at runtime.

```
class C:
  def hello(self):
  print("Hi from C")

B.__bases__ = (C,) # changing the parent class at runtime

b = B()
b.hello() # "Hi from C"
```

Object-oriented programming in Python - Single (or simple) inheritance, multiple inheritance, polymorphism

Multiple inheritance

Involves creating a class that inherits from multiple parent classes. Python handles the method resolution order through the MRO.

```
class Flyable:
  def describe(self):
    print("I can fly")

class Swimmable:
  def describe(self):
    print("I can swim")

# Class that inherits from two classes
class Duck(Flyable, Swimmable):
  pass

d = Duck()
d.describe() # Displays: I can fly (according to the MRO order: Flyable takes
priority)
```

Object-oriented programming in Python - Single (or simple) inheritance, multiple inheritance, polymorphism

Polymorphism

Allows the same interface to be used for objects of different classes. In other words, different classes can respond to the same method in a specific way.

```
class Cat(Animal):
  def speak(self):
    print(f"{self.name} meows") # Specific for the cat

# List of animals
animals = [Dog("Rex"), Cat("Mimi")]

# Same method speak(), different behaviours
for a in animals:
  a.speak()
# Displays:
# Rex barks
# Mimi meows
```

Object-oriented programming in Python - Single (or simple) inheritance, multiple inheritance, polymorphism

Polymorphism

Allows the same interface to be used for objects of different classes. In other words, different classes can respond to the same method in a specific way.

```
# Example with multiple inheritance and polymorphism
class Robot:
  def describe(self):
    print("I am a robot")

class RobotDuck(Duck, Robot):
  def describe(self):
    print("I am a robotic duck") # Redefinition (override)

rd = RobotDuck()
rd.describe() # Displays: I am a robotic duck
```

Object-oriented programming in Python – Practical work

Pratical work - Part 2
OOP

Object-oriented programming in Python - The specifics of Python's object model

➡ Everything is an object!

Numbers, functions, classes, modules, types… Yes, classes too! In Python, classes are instantiated by a metaclass, usually type, which means that classes are also instances of a special class.

```
class A:
  pass

print(type(A)) # <class 'type'>
```

```
A = type("A", (), {"x": 42})
print(A.x) # 42
```

Object-oriented programming in Python - The specifics of Python's object model

➡ Attributes are stored in a dictionary.

Python objects store their attributes in an internal dictionary called __dict__.

```
class Person:
def __init__(self, name):
  self.name = name

p = Person("Alice")
print(p.__dict__) # {'name': 'Alice'}
```

Object-oriented programming in Python - The specifics of Python's object model

Dunder methods

Dunder methods (short for "double underscore methods") are special methods in Python whose names begin and end with double underscores. They define how objects behave with built-in operations. They are usually called implicitly, not directly.

Common examples:

- __init__() → object initialization
- __str__() → string representation
- __len__() → length (len(obj))
- __eq__() → equality (==)

```
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return self.title
```

End of this part

Thank you!