EM Lyon - Python coding bootcamp - Session 1

Quick tour of Python

1. Key Concepts
2. Built-in Types
3. Control Flow Statements
4. Functions
5. Classes
6. Input and Output
7. Modules
8. Glimpse of the standard library

1. Key Concepts

1.1 Interpreter

`Python` is an interpreted language. Statements are processed by the interpreter line by line and top down.

When an expression is valid, it is evaluated and a result is printed.

In the code, comments are materialized by a hash `#`. All characters placed on the right of a hash are not interpreted, unless the hash is within a string literal (see below). The language does not support multi-lines comments.

1.2 Built-in Types

The `Python` language defines built-in elementary types, such as booleans, integers, floats, and also built-in data structure objects, such as lists, tuples, strings, dictionaries, which intend to represent collections of objects.

A built-in object might be represented with a literal, i.e. as a sequence of characters which is directly recognized by the interpreter.

Objects can also be represented by an identifier, i.e. as a sequence of characters which refers to an object after being assigned to it and considered as a variable. Assignment of a variable is performed by using the equal character `=`. Multiple assignments are also supported.

`Python` is a dynamically typed language: a variable can be successively assigned to objects of different types. Moreover, data structure objects can contain objects of different types.

Circa 30 identifiers are reserved keywords in `Python`. They cannot be used for naming variables, functions or classes (see below).

1.3 Control Flow Statements

The `Python` language defines some control flow statements, such as conditional or loop, which enable to manage precisely the execution of code.

1.4 Functions

The `Python` language also defines built-in functions. A function is a block of code which only runs when it is called.

A function might define some arguments which have to be passed to it when the call is performed. The language offers a variety of kinds of argument (positionnal, keyword, default value, variable-length) which empower the definition of functions.

The language enables to implement user-defined functions: either named functions or lambda.

1.5 Classes and Instances

In `Python`, any object is an instance of some class (or type).

The class defines the structure of its instances (i.e. their attributes or instance variables, which can refer to any object) and their behavior (i.e. some methods i.e. kind of functions). Access to the instance variables or execution of methods is performed by using the attribute notation materialized by a dot `.`.

Built-in types are implemented as classes. The language enables to implement user-defined classes.

`Python` manages class inheritance where a class may benefit from the structure and behavior defined by its superclass and to specialize some behavior.

1.6 Modules

In `Python`, a module is a file, or a set of files, which contains statements that are executed when the module is imported. A module can contain executable statements as well as function or class definitions, which will be available whenever the module has been imported.

The `Python` language implements a comprehensive collection of modules. They offer a wide range of facilities. This collection is completed by a huge library of open source modules available on `PyPI`. It can also be extended by user-defined modules.

A modules defines implicitely a namespace. It enables to access to the objects, functions and classes that the module defines. One can access to them by using the module name and the attribute notation.

1.7 Software Engineering

Functions, classes and modules are useful software engineering concepts which improve:

- *reusability* of code: write one run everywhere
- *readibility* of code: do not enter into details of implementation when it is not necessary
- *maintainability* of code: debugging, improving efficiency or extending functionalities are localized

2. Built-in Types

`Python` provides a wide range of built-in types. We only present a subset of this very rich collection and their main features.

Type	Description
NoneType	absence of value
bool	booleans
int	integers
float	floating point numbers
list	mutable sequences
tuple	immutable sequences
range	immutable sequences of int
str	immutable sequences of Unicode characters
dict	mutable mapping structures
function	function
module	module

2.1 None type

`None` is the sole instance of the `NoneType` class. This object enables to handle the absence of value in `Python`.

This object is used when there is a need to use a variable without assigning an initial value.

In [ ]:
```
# NoneType
a = None
print(a)
```

2.2 Booleans

The `bool` type has 2 instances:

- `True`: TRUE logical value
- `False`: FALSE logical value

Logical operations are:

- x `and` y: logical AND of x and y
- x `or` y: logical OR of x and y
- `not` x: logical NOT of x

In [ ]:
```
# logical operations with booleans
a = True
b = False
print(a and b, a or b, not a, not b)
```

Tips

- Few objects are considered as `False` such as zeros and empty collections, all other objects are considered as `True`.
- Booleans are implemented as integers: `True` stands for `1` and `False` for `0`.
- The `Python` interpreter performs a lazy evaluation of the second argument of the logical operations `and` and `or`:
  - x `and` y: if x is `False`, y is not evaluated
  - x `or` y: if x is `True`, y is not evaluated

Comparison

`Python` defines comparison operators between objects. They return booleans.

Operator	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

In [ ]:
```
# multiple assignment and comparison
a, b = 1, 2
print(a < b)
```

2.3 Integers and floating point numbers

The `int` type represents positive or negative integers.

The `float` type represents positive or negative floating point numbers.

Common arithmetic operations are available. Python manages standard priority between operations which can be tuned by using parenthesis `()`.

Operation	Result
x + y	sum of x and y
x - y	difference of x and y
x * y	product of x and y
x / y	quotient of x and y (Python 2: 3/2→1; Python 3: 3/2→1.5)
x // y	floored quotient of x and y
x % y	remainder of x / y
-x	x negated
abs(x)	absolute value of x
x ** y or pow(x, y)	x to the power y

In [ ]:
```
print(1 + 2 - 3 * 4 / 6)  # priority between operators
```

In [ ]:
```
print(1 + 2 - 3 * 4 // 6)  # integers only
```

In [ ]:
```
# floor division and modulus
a, b = 20, 7
print(a // b, a % b)
```

Exercise 1

- Compute how many seconds there are in a day and in a standard year.

In [ ]:
```
# %load session1/ex_01.py
```

Tips

- Zeros `0` and `0.0` are considered as `False`.
- Ingeters and floating point numbers support a notation where the underscore `_` may be used for grouping decimal numbers by thousands: e.g., `1_000`, `1_000_000.0`
- Floating point numbers support the standard floating point with the scientific notation using `e` followed by a power of `10`: e.g., `314e-2`
- When mixing `int` and `float` in an arithmetic operation, the result is cast to a `float`.
- Python deals with integers with arbitrary precision and floating point numbers with limited precison.
- In Python 3, `/` denotes the standard division and `//` denotes the integer division, whereas in Python 2, `/` denotes the integer division.
- Many operations and mathematical functions exist in `Python`, either in the language it self, or in the `math` module, or again in the `scipy` and `numpy` modules (see below).

2.4 Data structures

`Python` defines many built-in data structures. We present here the most used data structures: list, tuple, range, str and dict.

Some other data structures are also useful: e.g., set, frozenset, namedtuple.

2.4.1 List

An object of type `list` is an ordered mutable sequence. The empty list is `[]`.

In [ ]:
```
# creation of a list
a = [1, 2, 3]
print(a)
```

Extending lists

In [ ]:
```
# appending a single element
a.append(1)
a
```

In [ ]:
```
# appending several elements
a.extend([2, 3, 1])
a
```

Useful functions

Available for list, tuple, range and str (see below).

In [ ]:
```
# length
len(a)
```

In [ ]:
```
# min
min(a)
```

In [ ]:
```
# max
max(a)
```

In [ ]:
```
# sum
sum(a)
```

Testing elements

Available for list, tuple, range and str (see below).

In [ ]:
```
# testing whether 4 is in a
4 in a
```

In [ ]:
```
# testing whether 5 is not in a
5 not in a
```

Accessing elements and slices

The `[]` operator enables to access to a single element or to a slice of a sequence by position.

In `Python`, by convention the first element is starting at position `0` and the upper limit of the slice is not included in the selection.

This works for list, but also for tuple, range and str (see below).

Operator	Meaning
s[i]	access to the ith element of s
s[i:j]	slice of s from i to j-1
s[i:j:k]	slice of s from i to j-1 with step k
s[i:]	slice of s from i to len(s) - 1
s[:j]	slice of s from 0 to j-1
s[::-1]	reverse s

In [ ]:
```
# using the [] operator
print(a)
print(a[2], a[1:3], a[1:4:2])
print(a[1:], a[:3])
```

Concatenation

Available for list, tuple and str (see below).

In [ ]:
```
# concatenating 'a' with itself
a + a
```

In [ ]:
```
# concatenating 'a' with itself 3 times
a * 3  # or 3 * a
```

Sorting in place

Available for list only.

In [ ]:
```
# sorting 'a' in place
a.sort()
a
```

List comprehension

`Python` provides a very powerful syntactic way of building lists by comprehension.

In [ ]:
```
# list comprehension with the squares of elements of a
[i * i for i in a]
```

In [ ]:
```
# list comprehension with the squares of odd elements of a
[i * i for i in a if i % 2 == 1]
```

In [ ]:
```
# nested list comprehension
[i * j for i in [0, 1, 2] for j in [1, 2, 3]]
```

The `list()` function cast any sequence to a kind of a list.

In [ ]:
```
# list from a string (see below)
list('Python')
```

2.4.2 Tuple

An object of type `tuple` is an ordered immutable sequence. The empty tuple is `()`.

In [ ]:
```
# creation of a tuple
t = (1, 2, 3)
t
```

In some cases, parenthesis are not mandatory.

In [ ]:
```
# creation of a tuple
t = 1, 2, 3
t
```

The `tuple()` function cast any sequence to a kind of a tuple.

In [ ]:
```
# list from a string (see below)
tuple('Python')
```

2.4.3 Range

An object of type `range` is an ordered immutable sequence of integers. It is often used to build loops based on integers.

- `range(i)`: all integers starting at 0 until i - 1
- `range(i, j)`: all integers starting at i until j - 1
- `range(i, j, k)`: all integers starting at i until j - 1 with step k

Creating range uses the same notation than the access operator.

In [ ]:
```
r1 = range(10)  # integers from 0 to 9
r2 = range(1, 11)  # integers from 1 to 10
r3 = range(0, 100, 2)  # even integers from 0 to 100 excluded
print(list(r1))
print(list(r2))
print(list(r3))
```

Exercise 2

- Compute all cubes of odd integers from 1 to 19 included.

```
In [ ]: # %load session1/ex_02.py
```

2.4.4 String

An object of type `str` represents a string of characters (encoded in UTF-8 in Python 3).

A string might be delimited by 4 different means:

- 2 single quotes `'a string'`
- 2 quotes `"another string"`
- 2 triple single quotes `'''also a string'''`
- 2 triple quotes `"""yet another string"""`

When a string contains a special character (single quote, quote or backslash), the character should be escaped by being preceded by a backslash `\`.

Some special characters are also represented using a backslash:

- line feed: `\n`
- carriage return: `\r`
- tabulation: `\t`

```
In [ ]: s1 = 'this is a string'
        s2 = "it's a string"
        s3 = 'it\'s a string again'
        s4 = 'C:\\Users\\Guest\\'
        s5 = '''Monday,
        Tuesday,
        Wednesday,
        ...'''
        print(s1, s2, s3, s4, s5, sep='\n')
```

Formatting strings

Several techniques enable to format strings.

- i) new style: using literal string interpolation or `f-strings`
- ii) recent style: using the `format()` method
- iii) old style: using the `%` operator

All these techniques are available in Python 3.

i) Literal string interpolation or `f-strings`

With `f-strings`, identifiers put between curly brackets are replaced by their string representation.

```
In [ ]: # f-string
        name = 'Mary'
        age = 25
        f'{name} is {age} years old'
```

ii) `format()` method

With the `format()` method, curly brackets are replaced by the named or positionned arguments.

In [ ]:
```
# format with named arguments
'{name} is {age} years old'.format(name='Mary', age=25)
```

In [ ]:
```
# format with positionned arguments
'{1} is {0} years old'.format(25, 'Mary')
```

In [ ]:
```
# format with implicit positionned arguments
'{} is {} years old'.format('Mary', 25)
```

iii) `%` operator

With the `%` operator, specifiers marked with `%` are replaced by the positionned arguments.

In [ ]:
```
# implicit positionned arguments
'%s is %d years old' % ('Mary', 25)
```

Further reading: Format Specification Mini-Language for `f-string` and `format()`

- Alignment
- Numbers
- ...
- Format Specification Mini-Language (https://docs.python.org/3/library/string.html#formatspec)

Accessing elements with negative positions

In `Python` the first position of a string of length `L` is `0` and the last position is `L - 1`.

`Python` enables to access by using negative positions from `-L` to `-1`.

P	y	t	h	o	n
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

This notation is very handful; for instance, the last position of `s` can be accessed with `s[-1]`.

This notation works also for other sequences: list, tuple, range.

In [ ]:
```
# slicing string
s = 'Python'
s[1:-1]
```

In [ ]:
```
# reversing string
s[::-1]
```

String methods

The `str` class provides many methods dedicated to string manipulation.

In [ ]:
```
# concatenation
'1' + '2'
```

In [ ]:
```
# lower case
s.lower()
```

In [ ]:
```
# testing start
s.startswith('P')
```

In [ ]:
```
# testing end
s.endswith('P')
```

In [ ]:
```
# testing substring
s.find('th')
```

In [ ]:
```
# spliting string
s = "123,456,789"
s.split(",")
```

In [ ]:
```
# joining strings
abc = ['A', 'B', 'C']
'-'.join(abc)
```

Two convenient functions return ASCII code from a character and vice-versa.

In [ ]:
```
# character to ascii
ord('A')
```

In [ ]:
```
# ascii to character
chr(64)
```

The `str()` function cast any object to a kind of a str. This is performed by using the `__str__()` method of its class (see below).

In [ ]:
```
# str of any object
str(1 * 2 / 3)
```

Further reading: String Methods

- String Methods (https://docs.python.org/3.7/library/stdtypes.html#string-methods)

2.4.5 Dict

An object of type `dict` represents a mapping mutable object which associates keys (generally strings but not mandatory) to objects of any kind. The empty dict is `{}`.

Accessing to value from a key is performed by using the `[]` operator.

Removing a key is performed by using the `del` operator.

In [ ]:
```
# example
d = {'a':1, 'b':2, 'c':3}
d['a']
```

In [ ]:
```
# get function
d.get('a', 0)
```

In [ ]:
```
# deleting
del d['a']
d
```

In [ ]:
```
# get function
d.get('a', 0)
```

The `dict()` function cast any mapping sequence to a kind of a dict.

In [ ]:
```
# dict from a mapping sequence
d = dict([['a', 1], ['b', 2], ['c', 3]])
print(d)
```

2.4.6 Set

An object of type `set` represents an unordered colection of distinct elements.

In [ ]:
```
# example
s = set()
s.add(1)
s.add(1)
s.add(2)
s.add(2)
s.add(3)
s
```

In [ ]:
```
# set form a list
print(a, set(a))
```

3. Control Flow Statements

3.1 The *if* statement

The `if` statement followed by any object (which will be interpreted as True or False) enables to execute conditionnaly a piece of code.

It might be followed by one or several `elif` for further conditions and an `else` by default. All these statements need a colon `:` character at the end of line.

The conditionnal code should be indented by using a number of spaces (generally 4) or a tabulation. It is possible to nest several `if` statements which need appropriate identation.

In [ ]:
```
age = 19
if age in range(10, 20):
    print("teenager")
```

In [ ]:
```
if age < 18:
    print("minor")
else:
    print("adult")
```

In [ ]:
```
if age < 10:
    print("child")
elif age < 20:
    print("teenager")
else:
    print("adult")
```

In [ ]:
```
# idem with nested statements
if age < 20:
    if age < 10:
        print("child")
    else:
    print("teenager")
else:
    print("adult")
```

A one line if statement also exists (ternary conditional operator).

In [ ]:
```
# one line if/else statement
status = 'minor' if age < 18 else 'adult'
print(status)
```

Exercise 3

- Take a number if it is even divide it by 2, if it is odd add 1.

In [ ]:
```
# %load session1/ex_03.py
```

3.2 The *for* statement

The `for` statement enables to loop the execution of a piece of code over a sequence object (or a generator).

It is used along with the `in` keyword and a colon at the end of the line.

Here again, the code in loop should be indented.

In [ ]:
```
# simple loop
for i in range(10):
    print(i * i)
```

`Python` provides the `enumerate()` function with enables to loop on a sequence along with an index.

In [ ]:
```
# enumerate function
a = list('abcdefghij')
print(a)
for i, e in enumerate(a):
    print(i, e)
```

Exercise 4

- Take a list of all alphabetical letters and print only those corresponding to positions that are multiple of 3.

In [ ]:
```
# % load session1/ex_04.py
```

In [ ]:
```
# looping on dict items
d = dict(zip(list('abcde'), [1, 2, 3, 4, 5]))
print(d)
for key, value in d.items():
    print(key, value)
```

3.3 The *while* statement

The `while` statement enable to loop the execution of a piece of code on a boolean condition. It is followed by a colon. The code in loop is executed while the condition is considered as True.

Here again, the code in loop should be indented.

In [ ]:
```
a = 0
while a < 5:
    print(a)
    a += 1
```

Exercise 5

- Take 10 and print the sentence: "The countdown is ..." with 10, then remove 1 until it reaches 0.

In [ ]:
```
# %load session1/ex_05.py
```

3.4 The *break*, *continue*, *pass* and *else* statements for loops

The `break` statement breaks out any *for* or *while* loop.

The `continue` statement interrupt the current loop and proceed to the next one.

The `pass` statement does nothing and is used when a statement is syntactically needed.

The `else` statement put after a loop is only executed when the loop has not been interrupted by a break.

In [ ]:
```
# example
for i in range(10):
    if i == 5:
        continue
    if i > 7:
        break
    print(i)
else:
    print("over")
```

Exercise 6

- Take a number, e.g. 100. Implement a loop where the number is divided by 2 if it is even and added 1 if it is odd. Stop when number is 1.

In [ ]:
```
# %load session1/ex_06.py
```

4. Functions

In `Python` functions are instances of the *function* class.

A function is defined by using the statement `def` followed by the name of the function, the arguments of the function and ending the line with a colon.

The code of the function itself should be indented. It is possible to return a value by using the `return` statement, if not `None` is returned by the function.

It is possible to define functions with an arbitrary number of arguments (e.g., the *print* function).

It is possible to define default values for some arguments which are used when the argument is not given.

Standard functions

In [ ]:
```
# function call
print('Hello World!')
```

In [ ]:
```
# user-defined function
def square(x):
    return x * x
```

In [ ]:
```
# function call
square(16)
```

In [ ]:
```
# function call in a list comprehension
[square(i) for i in range(10)]
```

In [ ]:
```
# function with default values
def area(length=20, width=10):
    return length * width
```

In [ ]:
```
# all arguments are given
area(30, 5)
```

In [ ]:
```
# only first argument is given
area(30)
```

In [ ]:
```
# only second argument is given
area(width=20)
```

In [ ]:
```
# no argument is given
area()
```

lambda

The `lambda` statement enables to create anonymous functions which can be used punctually in a piece of code.

In [ ]:
```
# creation of a list of tuples
a = list(zip(range(10), list("Python 3.6"), range(20, 10, -1)))
a
```

In [ ]:
```
# sorting a using the second element of tuples
a.sort(key=lambda x: x[1])
a
```

In [ ]:
```
# sorting a using the third element of tuples
a.sort(key=lambda x: x[2])
a
```

Functional programming

`Python` defines 2 functions which enable functionnal programming:

- `map`: map(function, sequence) provides an iterator with the application of the function on each element of the sequence
- `filter`: filter(function, sequence) provides an iterator with the sequence for which the function returns True

In [ ]:
```
# map example
for e in map(lambda x: x * x, range(10)):
    print(e)
```

In [ ]:
```
# filter example
for e in filter(lambda x: x % 2 == 1, range(10)):
    print(e)
```

Exercise 7

- Take the loop defined in Exercise 6 and implement it as a function which takes the number as argument.
- Run it with 1000.
- Run it with all multiples of 100 until 1000.

In [ ]:
```
# %load session1/ex_07.py
```

5. Classes and instances

A class is a data model which defines attributes (instance variables) and some behavior (methods). A class enables instance creation which share the same structural model and also the behavior.

A class is defined by using the statement `class` followed by the name of the class and ending the line with a colon.

The code of the class itself should be indented.

Methods are define like functions of which the first argument represents the instance itself (generally named `self` but not mandatory). This argument is not used when the method is invoked.

The method `__init__()` enables to initialize the instance after being created.

The method `__str__()` is invoked when using the `str()` function and returns a string which represents the instance.

The method `__eq__()` is invoked when using the `==` operator and returns a boolean whether 2 instances are equal.

Attributes and methods are accessed by using the dot `.` operator.

Let us define the `Point` class which represents points in 2 dimensions.

An instance of `Point` has 2 variables `x` and `y` which represent the coordinates of the point.

The class `Point` defines also few methods.

In [ ]:
```
# small example : the POint class

class Point :
    def __init__(self, a=0.0, b=0.0):
        self.x = a
        self.y = b
    def __str__(self):
        return f"({self.x}, {self.y})"
    def is_diagonal(self):
        return self.x == self.y
    def transpose(self):
        return Point(self.y, self.x)
    def middle_with(self, autrePoint):
        return Point((self.x + autrePoint.x) / 2, (self.y + autrePoint.y) / 2)
    def __eq__(self, p):
        return (self.x == p.x) and (self.y == p.y)
```

An instance of a class is created by calling its class as a function with the appropriate arguments.

In [ ]:
```
# instance creation
p1 = Point()
print(p1)
```

In [ ]:
```
# accessing attribute
p1.x
```

In [ ]:
```
# diagonal?
p1.is_diagonal()
```

In [ ]:
```
# other instance
p2 = Point(10, 6)
p2.is_diagonal()
```

In [ ]:
```
# transposing
p3 = p2.transpose()
print(p3)
```

In [ ]:
```
# middle
p4 = p2.middle_with(p3)
print(p4)
```

In [ ]:
```
# diagonal?
p4.is_diagonal()
```

It is possible to create a new class `Polygon` which will reuse the `Point` class.

A `Polygon` instance is compound of a list of `Point` instances.

The `Polygon` class defines methods which enable to add some `Point` instances to the `Polygon` instance.

Look at the chain of methods: add_polygon() -> add_points() -> add_point()

In [ ]:
```
class Polygon:
    def __init__(self):
        self.points = []
    def __str__(self):
        return ("Poly{%s}" % ', '.join([str(p) for p in self.points]))
    def add_point(self, point):
        if point not in self.points: # ensure that all points are different
            self.points.append(point)
    def add_points(self, points):
        for point in points:
            self.add_point(point)
    def add_polygon(self, polygon):
        self.add_points(polygon.points)
    def name(self):
        names = {0:"empty", 1:"point", 2:"segment", 3:"triangle", 4:"quadrilater
al", 5:"pentagon"}
        nb = len(self.points)
        return names.get(nb, "polygon")
```

In [ ]:
```
# instance creation
p = Polygon()
print(p, p.name())
```

In [ ]:
```
# adding points
p.add_point(p1)
p.add_point(p2)
print(p, p.name())
```

In [ ]:
```
# other instance
q = Polygon()
q.add_points([p2, p3, p4])
print(q, q.name())
```

In [ ]:
```
# adding polygons
p.add_polygon(q)
print(p, p.name())
```

6. Input and output

6.1 User terminal

The `print()` function takes any number of arguments, plus 2 keywords arguments `sep` and `end`.

In [ ]:
```
# sep
print(1, 2, 3, sep=';')
```

In [ ]:
```
# end
for i in range(10):
    print(i, end=' ')
```

The `input()` function waits for the user to input some data. (In Python 2, the equivalent of `input()` is `raw_input()`, since the data of `input()` are interpreted by `Python`).

In [ ]:
```
# input number
age = int(input('What is your age? '))
age
```

Exercise 8

- Write a piece of code which asks for an integer and then prints the multiplication table of this integer.
- Example with table of 1:
  - 1 x 1 = 1
  - 1 x 2 = 2
  - 1 x 3 = 3
  - 1 x 4 = 4
  - 1 x 5 = 5
  - 1 x 6 = 6
  - 1 x 7 = 7
  - 1 x 8 = 8
  - 1 x 9 = 9
  - 1 x 10 = 10

In [ ]:
```
# %load session1/ex_08.py
```

6.2 Files

6.1 Standard files

The `open()` function enables to open textual and binary files in diverse modes (read, write, append). The function takes the name of the file, the open mode (`'rt'` by default), the encoding (UTF-8 by default). It is possible to provide another encoding, for instance `'latin-1'`.

Opening mode	Meaning
r	read (by default)
w	create and write
a	create and append
t	text (by default)
b	binary

The function returns an object which enables to access to the file. After used, a file need to be closed in order to free it. The context manager defined by the `with` statement enables to free automatically any resource.

In [ ]:
```
# create and write a file
with open('data.txt', 'w') as f:
    f.write('This is a simple file.\n')

# open and read a file
with open('data.txt') as f:
    content = f.read()

print(content)
```

In [ ]:
```
# writing a file
with open('data.txt', 'a') as f:
    for i in range(10):
        f.write(f'{i}\n')

# reading it line by line
with open('data.txt') as f:
    for line in f:
        print(line.rstrip('\n'))
```

6.2 JSON files

The JSON (JavaScript Object Notation) format is useful to store and share data accross computer languages.

The `json` module of the standard library manages such operations. (See modules in next paragraph).

In [ ]:
```
# creatin and object
data = {'language': 'Python', 'version': [3, 6]}
data
```

In [ ]:
```
# save it to json
import json
with open('example.json', 'w') as f:
    json.dump(data, f)

# load it from json
with open('example.json') as f:
    data2 = json.load(f)

print(data2 == data)
```

6.3 Databases

Python is able to access to most of standard databases (e.g., MySQL, PostgreSQL, SQLServer, Access, SQLite, Oracle).

There are several techniques to access to such databases:

- using a dedicated module
- using the ODBC (open data base connectivity) protocol and a driver
- using the SQLAlchemy module + a dedicated module or an ODBC connexion

In [ ]:
```
# example with SQLite and a dedicated module
import sqlite3
import os

# remove 'test.db' file if exists
if os.path.exists('test.db'):
    os.remove('test.db')

# create and fill a small database
with sqlite3.connect("test.db") as db:
    cursor = db.cursor()
    cursor.execute('''CREATE TABLE clients
                      (id integer, lastname text, firstname text)''')
    cursor.execute('''INSERT INTO clients VALUES (1, 'Smith', 'John')''')
    cursor.execute('''INSERT INTO clients VALUES (2, 'Doe', 'John')''')
    cursor.execute('''INSERT INTO clients VALUES (3, 'Smith', 'Ann')''')
    db.commit()
    print("database created")
```

In [ ]:
```
# open and request the database
with sqlite3.connect("test.db") as db:
    cursor = db.cursor()
    cursor.execute('''SELECT * FROM clients WHERE lastname=?''', ('Smith',))
    row = cursor.fetchone()
    while row:
        print(row)
        row = cursor.fetchone()
```

7. Modules

Modules are extensions of the `Python` language. A module is made of one or several `.py` files which define objects, functions and classes.

Different kinds of mudoles exist:

- Modules of the `Python` standard library, i.e. `sys`, `os`, `math`, `re`
- Open sources modules, i.e. `numpy`, `scipy`, `matplotlib`, `pandas`
- In-house modules implemented by anyone.

The `import` statement followed by the name of a module (or its path separated by `.`) enables to import a module.

By default, the namespace is the name of the module, i.e., any object, function or class defined by the module need to be prefixed by the name of the module. It is possible to change the name of a module when it is imported. It is possible to import directly some of all definitions of a module but warn collisions of names. It is also possible to import selectively few objects defined in a module.

When a module is first imported, it is precompiled in a `.pyc` file which will be directly used for further calls, unless the code of the module has been changed and need to be compiled again.

In [ ]:
```
# example: this module = Python philosophy
import this
```

In [ ]:
```
# example
import math
math.pi
```

In [ ]:
```
# example
import math as m
print(m.factorial(100))
```

In [ ]:
```
# example
from math import factorial as fact
print(fact(200))
```

Define your own module. Take this code and save it in a file named 'hyp.py'.

```
# coding: utf-8
from math import sqrt

def hypothenuse(x, y):
    return sqrt(x * x + y * y)
```

In [ ]:
```
# import module
from hyp import hypothenuse as h

h(3, 4) + h(5, 12)
```

Further reading: Python exercises with solutions

- Python Exercises, Practice, Solution (https://www.w3resource.com/python-exercises/)

8. Glimpse of the standard library

This paragraph gives a very small glimpse of the standard library, since it contains more than 200 modules.

Few modules and few features which are frequently encountered are listed here.

8.1 sys - System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

sys.argv

The list of command line arguments passed to a Python script. argv[0] is the script name. Other arguments are available in argv[1], argv[2], etc.

See https://docs.python.org/3/library/sys.html (https://docs.python.org/3/library/sys.html).

8.2 os - Miscellaneous operating system interfaces

This module provides a portable way of using operating system dependent functionality.

See https://docs.python.org/3/library/os.html (https://docs.python.org/3/library/os.html).

os.getcwd()

Return a string representing the current working directory. When used in a notebook, it is the directory where the notebook has been launched.

os.listdir(path)

Return a list containing the names of the entries in the directory given by path.

os.path - Common pathname manipulations

This sub-module implements some useful functions on pathnames.

See, https://docs.python.org/3/library/os.path.html#module-os.path (https://docs.python.org/3/library/os.path.html#module-os.path).

os.path.exists(path)

Return True if path refers to an existing path or an open file descriptor.

os.path.join(path, *paths)

Join one or more path components intelligently.

8.3 re - Regular expression operations

This module provides regular expression matching operations similar to those found in Perl.

See, https://docs.python.org/3/library/re.html (https://docs.python.org/3/library/re.html).

8.4 datetime - Basic date and time types

This module supplies classes for manipulating dates and times in both simple and complex ways.

See, https://docs.python.org/3/library/datetime.html (https://docs.python.org/3/library/datetime.html).

See also modules: time, dateutil, calendar, etc.

8.5 json - JSON encoder and decoder

This module provides reading and writing facilities for the JSON format.

See, https://docs.python.org/3/library/json.html (https://docs.python.org/3/library/json.html).

8.6 pickle - Python object serialization

This module implements binary protocols for serializing and de-serializing a Python object structure.

See, https://docs.python.org/3/library/pickle.html (https://docs.python.org/3/library/pickle.html).

8.7 collections - Container datatypes

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

See, https://docs.python.org/3/library/collections.html (https://docs.python.org/3/library/collections.html).

collections.defaultdict

dict subclass that calls a factory function to supply missing values.

8.8 urllib.request - Extensible library for opening URLs

This module module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

See, https://docs.python.org/3/library/urllib.request.html#module-urllib.request (https://docs.python.org/3/library/urllib.request.html#module-urllib.request).

8.9 IPython - IPython: Productive Interactive Computing

IPython provides a rich architecture for interactive computing (not in the standard library).

See, https://ipython.org/ (https://ipython.org/).

IPython.display

This sub-module provides a public API for display tools in IPython.

See, https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html (https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html).