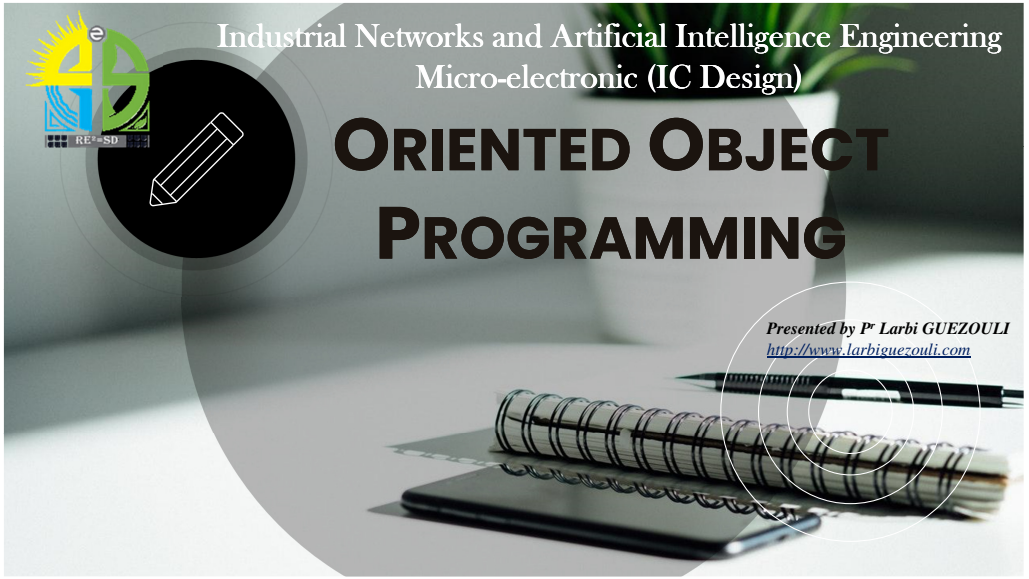


Objective of the OOP

The main purpose of OOP is to **link** data and the **functions** that use it.

The responsibility for **processing** these data remains with the functions that are linked to them, and the rest of the code is freed from this processing.

Syllabus

CHAPTER I. Notions of programming language paradigms (03h00)

1. Imperative programming
2. Functional programming
3. Introduction to object-oriented programming

CHAPTER II. Classes and objects (02h00)

1. Declaration of classes
2. Builders and destroyers
3. Access methods
4. Encapsulation

CHAPTER III. Basic types and strings in Python (02h00)

1. Primitive types
2. Strings

Syllabus(next)

CHAPTER IV. Inheritance and polymorphism (02h30)

1. Generalities
2. Overload and redefinition
3. Inheritance
4. Polymorphism
5. Abstract classes

CHAPTER V. Inner class (01h00)

1. Definition
2. Why do we need Inner classes?

CHAPTER VI. Collections (02h00)

1. Iterators
2. Lists & Arrays
3. Sets
4. Queues
5. Dictionaries

Syllabus(next)

CHAPTER VII. Graphics User Interface GUI (03h00)

1. The package 'Tkinter'
2. The window
3. Geometry of components
4. Using OOP in GUI programming
5. Some widgets
6. Event-based programming

CHAPTER VIII. Exception handling (02h00)

1. What is an exception?
2. How to handle an exception in a program
3. Interception of hierarchical exceptions

CHAPTER IX. Streams and files (02h00)

1. Text I/O
2. Binary I/O
3. RAW I/O

Syllabus(next)

Evaluation:

- Interrogations
- Practical work
- Homework
- Final exam

Bibliography

- Steven F. Lott, Dusty Phillips, "Python Object-Oriented Programming Fourth Edition", Packt Publishing Ltd., 4th ed., 2021, ISBN: 978-1-80107-726-2
- Vincent BOUCHENY, "Apprendre la Programmation Orientée Objet avec le langage Python", ENI, 2016, ISBN: 978-2409000997
- Gomez Richard, "Le petit Python orienté objet - Programmation orientée objet avec Python 3", Ellipses, 2022, ISBN: 9782340064065
- Official web site of Python: https://docs.python.org/3/tutorial

1 Notions of programming language paradigms

Basic concepts

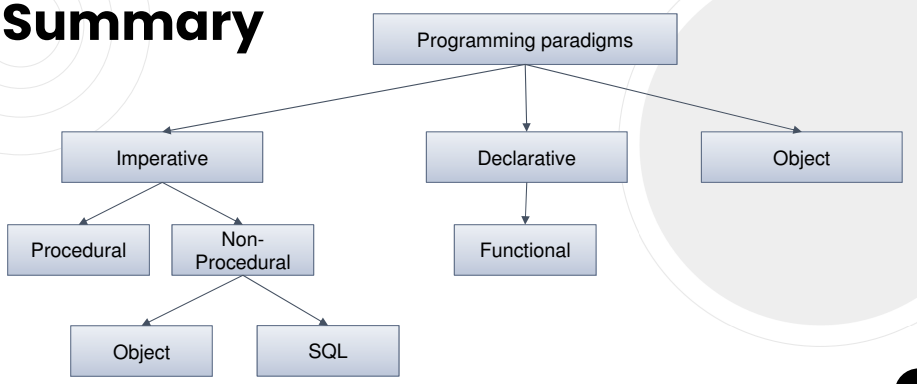
Introduction

A programming paradigm is a **style of computer programming** that focuses on how to formulate a solution to a problem.

We will see the three most common programming paradigms:

- The imperative paradigm;
- The functional paradigm;
- The object paradigm.

Summary



1. Imperative programming

This paradigm describes a program as a **sequence of instructions** to be executed by the computer, one after the other, producing changes of states.

Exp. 1

```
numbers = [3, 9, 6, 11]
sum = 0
sum += numbers[0]
print("sum = ", sum)
sum += numbers[1]
print("sum = ", sum)
sum += numbers[2]
print("sum = ", sum)
sum += numbers[3]
print("sum = ", sum)
```

Result:

sum = 3
sum = 12
sum = 18
sum = 29

Exp. 2

```
numbers = [3, 9, 6, 11]
sum = 0
for i in range(len(numbers)):
    sum += numbers[i]
    print("sum = ", sum)
```

Result:

sum = 3
sum = 12
sum = 18
sum = 29

2. Functional programming

This paradigm defines a program as a **mathematical function**. Inputs are processed to provide the same output result for the same input values.

Exp. 1

```
a = 50
def sum(b):
    global a
    return (a + b)

print(sum(10))
a = 70
print(sum(10))
```

Result:

60

80

This example does not respect the principle of the functional paradigm. There are several results for the same input value.

Exp. 2

```
def sum(a, b):
    return (a + b)

a = 13
print(sum(50, 10))
a = 45
print(sum(50, 10))
```

Result:

60

60

This example respects the principle of the functional paradigm. The same result for the same input values.

Exp. 3

```
def add(a, b):
    a.append(b)
    return a

numbers = [5, 6, 8]
print(numbers)
print(add(numbers, 7))
print(numbers)
```

The 'numbers' variable, external to the add() method, is modified at each call to this method, this is the **side effect**.

Result:
[5, 6, 8]
[5, 6, 8, 7]
[5, 6, 8, 7]

Exp. 4

To avoid the problem of the **side effect**, we make a copy of the variable 'numbers' inside the function.

```
def add(a, b):
    a1 = a.copy()
    a1.append(b)
    return a1

numbers = [5, 6, 8]
print(numbers)
print(add(numbers, 7))
print(numbers)
```

Result:
[5, 6, 8]
[5, 6, 8, 7]
[5, 6, 8]

3. Introduction to object-oriented programming

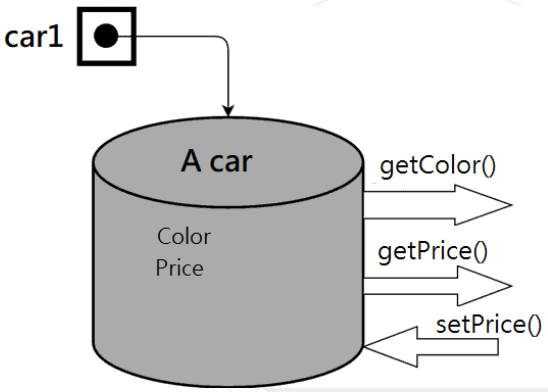
This paradigm allows to model a code in the form of objects having data called properties and functions called methods and which interact between them.

Definition

- **Object-oriented programming** is an approach in which programs are organized as **sets of cooperating objects**.
- Each object represents an **instance** of a certain **class**.
- All classes being members of a **class hierarchy** unified by inheritance relations.

The class "A car" contains two data "Color" and "Price" and three methods (getColor, getPrice, setPrice)

The object "car1" is an instantiation of the class "A car".



2

Classes and objects

From definition to use

1. Class

A **class** encapsulates a set of data (**attributes**) and a set of functions for processing these data (**methods**).

A particular method called **constructor** is called when an object of this class is created.

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

    def work(self):
        print(self.name, 'is working on', self.project)
```

Data Members

Method

1. Class (next)

<u>The constructor method</u>

Each class must define a particular method called **constructor**.

A constructor is a method <u>invoked when an object is instantiated</u>.

This method, which can be empty, performs the operations necessary for the initialization of an object.

In Python the `__init__()` method is called the constructor and is always called when an object is created.

Syntax of constructor declaration :

```
def __init__(self):
    # body of the constructor
```

1. Class (next)

Types of constructors :

Default constructor: The default constructor is a simple constructor which doesn't accept any arguments.

Parameterized constructor: It takes its first argument as a reference to the instance being constructed known as `self` and the rest of the arguments are provided by the programmer.

Exp.

```
class Rectangle:
    def __init__(self):
        self.longueur = 0
        self.largeur = 0
        self.origine_x = 0
        self.origine_y = 0
    def __init__(self, long, lar, ox, oy):
        self.longueur = long
        self.largeur = lar
        self.origine_x = ox
        self.origine_y = oy
    def move(self, x:float, y:float): #We can specify the variable type
        origine_x = self.origine_x + x
        origine_y = self.origine_y + y
    def surface(self):
        return self.longueur * self.largeur
```


2. Object

An object is an **instance** (a case) of a class and is represented by a variable with a state (or value).

To create an object of a class we have to declare a **variable** whose **type is the class** to be instantiated, and then to call a **constructor** of this class.

Exp.

```
class Rectangle:
    def __init__(self):
        self.longueur = 0
        self.largeur = 0
        self.origine_x = 0
        self.origine_y = 0
    def __init__(self, long, lar, ox, oy):
        self.longueur = long
        self.largeur = lar
        self.origine_x = ox
        self.origine_y = oy
    def move(self, x, y):
        origine_x = self.origine_x + x
        origine_y = self.origine_y + y
    def surface(self):
        return self.longueur * self.largeur

r = Rectangle(5, 5, 0, 0)
r.move(10, 10)
print("surface : ", r.surface())
```

3. Package

A package (library) is a container including several classes dedicated to a specific theme.

There are predefined packages, as well as the possibility to create your own package.

Popular predefined packages

Package	Description
NumPy	Tools to help build multi-dimensional arrays, solve algebraic formulas, perform common statistical operations, and much more.
Tkinter	GUI programming
Matplotlib	Data exploration and visualization library.

Exp.

```
import numpy as np
a = np.arange(15).reshape(3, 5)
print("a : ", a)
print("shape : ", a.shape)
print("ndim : ", a.ndim)
print("dtype.name : ", a.dtype.name)
print("itemsize : ", a.itemsize)
print("size : ", a.size)
print("type(a) : ", type(a))
b = np.array([6, 7, 8])
print("b : ", b)
```

Result:

```
a :  [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
shape :  (3, 5)
ndim :  2
dtype.name :  int32
itemsize :  4
size :  15
type(a) :  <class 'numpy.ndarray'>
b :  [6 7 8]
```


4. Access authorizations (or access modifiers)

In Python, there are several types of access authorization:

- public
- protected
- private

4.1. Public authorization

Members, methods and classes declared public are accessible from **anywhere**. This type of access does not put any restrictions on access.

All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

Exp.

```
class Addition:
    def add(self, a, b):
        return a + b
obj = Addition()
print(obj.add(10, 21))
```

We can call the public method `add()` of the `'Addition'` class from the outside of the class.

Result:

31

4.2. Protected authorization

Protected data members and methods are only accessible by **subclasses**.

Python's convention to make an instance variable **protected** is to add a prefix _ (single underscore) to it. This only allows access from a subclass.

Exp.

```
class Addition:
    def __init__(self):
        self._a = 0
    def _add(self, b):
        return self._a + b
obj = Addition()
obj._a = 10
print(obj._add(21))
```

`_a` and `_add()` are protected members.

Result:

31

However, it is still accessible in Python. Hence, the responsible **programmer** would **refrain** from accessing and modifying instance variables prefixed with '_' from outside its class and subclasses.

4.2. Protected authorization (next)

Best practices: How should we access protected properties?

We can use **getters** and **setters** to access protected properties from the outside of the class et subclasses.

Exp.

```
class Addition:
    def __init__(self):
        self._a = 0
    @property
    def a(self):
        return self._a
    @a.setter
    def a(self, value):
        self._a = value

    def add(self, b):
        return self._a + b

obj = Addition()

obj.a = 10
print("a : ", obj.a)
print(obj.add(21))
```

The property _a will be accessed through the getter and the setter.

Result:

a : 10

31

4.3. Private authorization

- The scope of the "**private**" authorization type is limited to the class only.
- **Private** data members and methods are only accessible **within the class**.
- Classes and interfaces **cannot** be declared as **private**.
- The double underscore '__' prefixed to a variable makes it private. It is strongly advised not to access it from outside the class.

Exp.

```
class Addition:
    def __init__(self, value):
        self.__a = value  # private instance attribute
    def __add(self, b):
        return self.__a + b
```

```
obj = Addition(10)
obj.__a = 5
print(obj.__a)
print(obj.__add(21))
```

New class attribute

Errors

Python changes the names of private variables to `_object._class__variable`.

It is possible to access them from outside the class with this new name, but you should refrain from this practice.

Exp.

```
class Addition:
    def __init__(self, value):
        self.__a = value  # private instance attribute
    def __add(self, b):
        return self.__a + b
```

≠

=

```
obj = Addition(10)
print(obj._Addition__a)
print(obj._Addition__add(21))
obj.__a = 5
print(obj.__a)
print(obj._Addition__a)
print(obj._Addition__add(21))
```

Not recommended

New object attribute

Result:

10

31

5

10

31

This practice is **not recommended**.

What is the recommended solution???

4.3. Private authorization (next)

Best practices: How should we access private properties?

We can use **getters** and **setters** to access private properties from the outside of the class.

Exp.

```
class Addition:

    def __init__(self, value):
        self.__a = value
    @property
    def a(self):
        return self.__a
    @a.setter
    def a(self, value):
        self.__a = value

    def __add(self, b):
        return self.__a + b
    def callAdd(self, b):
        return self.__add(b)
obj = Addition(10)
obj.a = 5
print("a : ", obj.a)
print(obj._Addition__add(21))
```

The property **__a** will be accessed through the getter and the setter.

Result:

a : 5

26

4.4. Summary

Access from:	Same class	Subclass	Another class
public	X	X	X
protected	X	X	
private	X		

5. Static Variable of Class variable

It is a class property declared inside the class but outside any method.

It can be referred through a class but not directly through an instance.

Exp.

```
class A:
    x = 9 # Static variable or class variable
print (A.x)
# Access through an instance
obj = A()
print(obj.x)
# Change within an instance
obj.x = 12
print(obj.x)
print(A.x)
# Change within the class
A.x = 10
print(A.x)
print(obj.x)
```

Result:

9
9
12
9
10
12

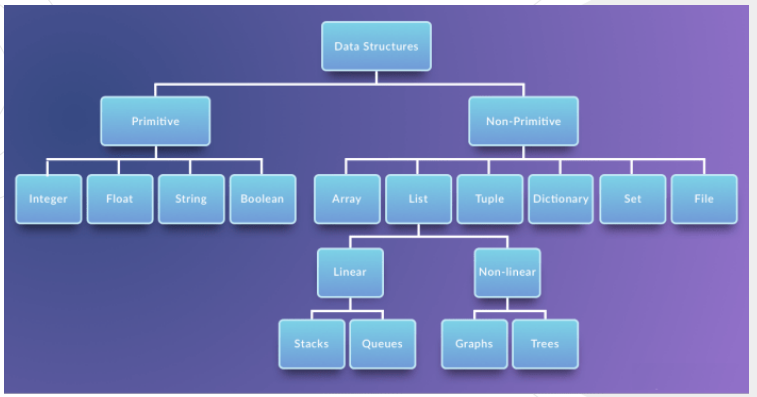
3 Basic types and strings in Python

Learn by example

1. Some basic types

The basic types known in the different languages also exist in the Python language.

Basic types in Python are divided into two categories: primitive/non-primitive.



Exp.

```
a = 'c'
print("Type of a: ", type(a))
b = "Hello"
print("Type of b: ", type(b))
c = 5
print("Type of c: ", type(c))
d = 5.1
print("Type of d: ", type(d))
e = (5 > 1)
print("Type of e: ", type(e))
```

```
import array as arr
f = arr.array('i', [1,2,3]) #available types i, f,...
print("Type of f: ", type(f))
import numpy as np
g = np.array([1,"hello",3])
print("Type of g: ", type(g))
h = [1,"hello",3]
print("Type of h: ", type(h))
```

Result:

```
Type of a:  <class 'str'>
Type of b:  <class 'str'>
Type of c:  <class 'int'>
Type of d:  <class 'float'>
Type of e:  <class 'bool'>
Type of f:  <class 'array.array'>
Type of g:  <class 'numpy.ndarray'>
Type of h:  <class 'list'>
```


Array type codes: `arr.array('<type_code>', [1,2,3])`

Type code	C Type	Python Type	Minimum size in bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	wchar_t	Unicode character	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'q'`	signed long long	int	8
`'Q'`	unsigned long long	int	8
`'f'`	float	float	4
`'d'`	double	float	8

1.1. Operators

The operators available in Python are presented in the following table:

Operator	Name	Example
==	Equals	x == y
!=	Not equals	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y
and	Logical AND	x < 5 and x < 10
or	Logical OR	x < 5 or x < 4
not	Logical NOT	not(x < 5 and x < 10)

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%= (modulo)	x %= 3	x = x % 3
//= (division quotient)	x //= 3	x = x // 3
**= (power)	x **= 3	x = x ** 3
&= (Bitwise AND)	x &= 3	x = x & 3
|= (Bitwise OR)	x |= 3	x = x | 3
^= (Bitwise XOR)	x ^= 3	x = x ^ 3
~ (Bitwise NOT)	~x	Inverts all the bits
>>= (Signed right shift)	x >>= 3	x = x >> 3
<<= (Zero fill left shift)	x <<= 3	x = x << 3

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

2. Strings

A string is a chain of characters:

```
s1 = 'Hello'
s2 = "Hello"
```

Multiline String: (three quotes)

```
s1 = """Hello,
Good morning,
See you."""
s1 = '''Hello,
Good morning,
See you.'''
```

Result:

```
Hello,
Good morning,
See you.
```


2. Strings (next)

A string is an array of characters:

```
s1 = 'Hello'
print(s1[1])
```

Looping Through a String

```
for x in "hello":
  print(x)
```

Result:

```
h
e
l
l
o
```

2. Strings (next)

Check String

```
s1 = 'Hello'
if ("lo" in s1):
   print('OK')
if ("lo" not in s1):
   print('not OK')
```


2.1. Operations on Strings

String Length

```
print(len(a))
```

String Concatenation:

```
s = s1 + s2
```

String Comparison:

```
if ('fruit' == 'hello'):
if ('fruit' != 'hello'):
if ('fruit' < 'hello'):
if ('fruit' > 'hello'):
if ('fruit' <= 'hello'):
if ('fruit' >= 'hello'):
```


4

Inheritance and polymorphism

Principle and utility

1. Generalities

Object Oriented Programming is based on three fundamental concepts:

- Encapsulation
- Inheritance
- Polymorphism.

2. Encapsulation

Encapsulation in Python describes the concept of **bundling (grouping) data and methods within a single unit**.

A class is an example of encapsulation as it binds (groups) all properties and methods into a single unit.

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

    def work(self):
        print(self.name, 'is working on', self.project)
```

Data Members

Method

3. Principle of inheritance

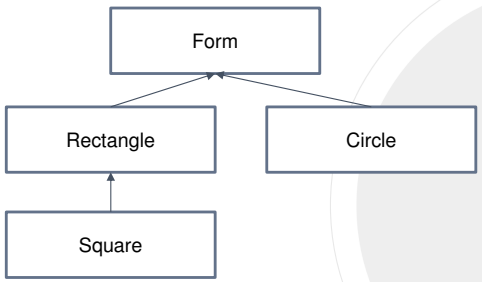
The main idea of inheritance is to organize classes in a hierarchical way.

The inheritance relationship is unidirectional.

It is used to create new types based on old ones.

A child class inherits the attributes and methods of the parent class.

Exp.



Exp.

```
Class Form:
    def __init__(self, n):
        self.name = n
    def setName(self, n):
        self.name = n
class Rectangle(Form):
    def __init__(self, n, w, l):
        Form.__init__(self, n)
        self.__width = w
        self.__length = l
    def getLength(self):
        return self.__length
    def getWitdh(self):
        return self.__width
    def surface(self):
        return self.__width * self.__length
```

```
    def display(self):
        print("Rectangle: ", self.name, " ",
        self.__length, "x", self.__width)
        print("Surface: ", self.surface())
r = Rectangle("my rectangle", 10, 20)
r.display()
```

Result:

```
Rectangle:  my rectangle   20 x 10
Surface:  200
```

Exp.

```
class Square(Rectangle):
    def __init__(self, n, l):
        Rectangle.__init__(self, n, l, l)
def display(self):
        print("Square: ", self.name, " ",
        self.getLength(), "x", self.getWitdh())
        print("Surface: ", self.surface())
s = Square("my square", 10)
s.display()
```

Result:

```
Square:  my square   10 x 10
Surface:  100
```


Exp.

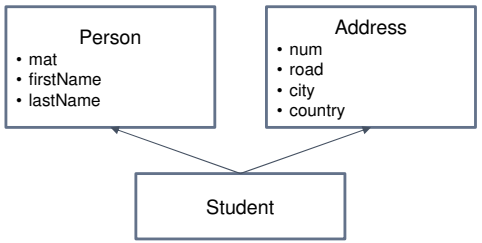
```
class Circle(Form):
    def __init__(self, n, r):
        Form.__init__(self, n)
        self.__radius = r
    @property
    def radius(self):
        return self.__radius
    @radius.setter
    def radius(self, r):
        self.__radius = r
    def surface(self):
        return 3.14 * self.__radius ** 2
    def display(self):
        print("Circle: ", self.name, " of radius: ", self.__radius)
        print("Surface: ", self.surface())
c = Circle("my circle", 20)
c.display()
```

Result:

```
Circle:  my circle  of radius:  20
Surface:  1256.0
```


3.1. Multiple inheritance

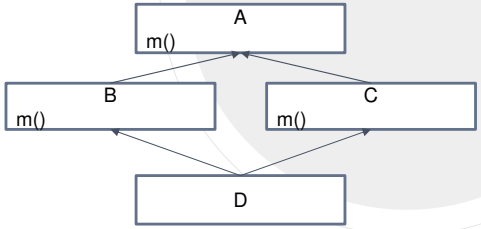
A child class inherits the attributes and methods of multiple parent classes.




```
class Person:
  pass
class Address:
  pass
class Student(Person, Address):
  pass
```


3.2. The Diamond Problem

Represents the ambiguity that arises when two classes `B` and `C` inherit from a superclass `A`, and another class `D` inherits from both `B` and `C`.



3.2. The Diamond Problem (next)

If there is a method `m()` in `A` that `B` or `C` (or even both of them) has **overridden**, then the question is which version of the method `m()` does `D` inherit? It could be the one from `A`, `B` or `C`.

Solution:

Python uses **MRO (Method Resolution Order)** to resolve this kind of problems.

3.3. Method Resolution Order (MRO)

It is the **order** in which a method is searched for in a classes hierarchy. It is especially useful in Python because it supports multiple inheritance.

In Python, the MRO is from bottom to top and left to right.

Rule:

"depth-first left-to-right traversal" + "removing duplicates expect for the last".

Exp. 1.

```
class A:
    def m(self):
        print("Diamond problem is bad")
class B(A):
    def __init__(self, x):
        self.x = x
    def m(self):
        print(self.x)
class C(A):
    def __init__(self, y):
        self.y = y
    def m(self):
        print(self.y)
```

```
class D(B, C):
    def __init__(self, x, y):
        B.__init__(self, x)
        C.__init__(self, y)
print(D.mro())
d = D(1, 2)
d.m()
```

Returns the mro of class D

The MRO of class D:

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>,
<class 'object'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>]
```

Result:

```
[<class '__main__.D'>, <class
'__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>]
1
```


Exp. 2.

In the previous example:

```
class Person:
  pass
class Address:
  pass
class Student(Person, Address):
  pass

print(Student.mro())
```

The MRO of class Student:

```
[<class '__main__.Student'>, <class '__main__.Person'>,
<class 'object'>, <class '__main__.Address'>, <class 'object'>]
```

Result:

```
[<class '__main__.Student'>, <class '__main__.Person'>,
<class '__main__.Address'>, <class 'object'>]
```

Difference between super() and calling superclass directly

The method `super()` uses the MRO of the class to choose which superclass to use.

So, in the previous example:

```
class D (B, C):
    def __init__ (self):
        super(D, self).m()
        super(B, self).m()
        C.m()

print(D.mro())
```

m() of class B

m() of class C

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```


4. The polymorphism

Allow to perform the same action in different ways. It can be found in:

- The Overloading
- The Overriding

4. The polymorphism(next)

4.1. The Overloading:

Function overloading is used in a single class where we have a method with default values for its arguments. Wo, we can call this method with different arguments.

4.2. The Overriding:

Child classes in Python can (**redefine**) provide a **specific implementation** of a **method** that is already provided by one of its **parent classes**.

Exp. (Method Overloading)

```
class Shape:
    def surface(self, a:float, b:float=0):     # Overloading
        if (b==0):
            print("Circle = ", 3.14 * a * a)
        else:
            print("Rectangle =", a * b)

myShape = Shape()
myShape.surface(5)              # Polymorphism
myShape.surface(6.0, 1.2)       # Polymorphism
```

Result:

```
Circle =  78.5
Rectangle = 7.199999999999999
```

Exp. (Method Overriding)

```
class Animal:
    def eat(self):
        print("Animals eat: ")
class herbivores(Animal):
    def eat(self):      # Method Overriding
        print("Herbivores eat plants.")
class omnivores(Animal):
    def eat(self):      # Method Overriding
        print("Omnivores eat plants and meat")
class carnivores(Animal):
    def eat(self):          # Method Overriding
        print("Carnivores eat meat")
```

```
a = Animal()
h = herbivores()        # Polymorphism
o = omnivores()         # Polymorphism
c = carnivores()        # Polymorphism
a.eat()
h.eat()
o.eat()
c.eat()
```

Result:

```
Animals eat:
Herbivores eat plants.
Omnivores eat plants and meat
Carnivores eat meat
```


5. Abstract class

An abstract class is a template class that can contains attributes, implemented methods and **non-implemented methods** (without body).

We cannot instantiate an abstract class because some of its methods are not implemented.

An abstract class can inherits from a class or from another abstract class.

5. Abstract class (next)

To define an abstract class, we use the '**abc**' (abstract base class) module.

```
from abc import ABC
class AbstractClassName(ABC):
    pass
```


5. Abstract class (next)

To define an abstract method, you use the **@abstractmethod** decorator.

```
from abc import ABC, abstractmethod
class AbstractClassName(ABC):
    @abstractmethod
    def abstract_method_name(self):
        pass
```

Exp.

```
from abc import ABC, abstractmethod
class Shape(ABC):
    def __init__(self, x=0, y=0):
        self.__origin_x = x
        self.__origin_y = y
    @property
    def origin_x(self):
        return self.__origin_x
    @property
    def origin_y(self):
        return self.__origin_y
    @abstractmethod
    def display(self):
        pass
class Rectangle(Shape):
    def __init__(self, x=0, y=0):
        super().__init__(x, y)
    def display(self):
        print("Rectangle, origin: ", self.origin_x, " , ",  self.origin_y)
class Square(Rectangle):
    def __init__(self, x=0, y=0):
        super().__init__(x, y)
    def display(self):
        print("Square, origin: ", self.origin_x, " , ", self.origin_y)
class Circle(Shape):
    def __init__(self, x=0, y=0):
        super().__init__(x, y)
    def display(self):
        print("Circle, origin: ", self.origin_x, " , ", self.origin_y)
F1 = Rectangle(10, 10)
F1.display()
F2 = Square(20, 20)
F2.display()
F3 = Circle(30, 30)
F3.display()
```

Result:

```
Rectangle, origin:  10  ,  10
Square , origin:  20  ,  20
Circle , origin:  30  ,  30
```


5

Inner classes

Nested classes

1. Inner class (or Nested class)

Inner or **Nested** Class is a class defined inside another class.

An inner class can have access to the methods and attributes of the enclosing class.

```
# Outer class
class Outer:
    # Inner class
    class Inner1:
        pass
        # Multilevel inner class
        class InnerInner:
            pass
    # Another inner class
    class Inner2:
        pass
```


2. Why do we need Inner classes?

For grouping two or more classes.

For example, the classes **Car** and **Engine** can be grouped together.

A car needs an engine but the engine will not be used without a car.

So logically the Engine class can be an inner class of the Car class.

Exp. 1. (Inner class)

```
class A:
    def __init__(self):
        self.__x = 0
    @property
    def x(self):
        return self.__x
    @x.setter
    def x(self, value):
        self.__x = value
    class B:
        def __init__(self):
            A.x = 12
        def display(self):
            print("B.x= ", A.x)
```

Access the attribute 'x' of the enclosing class

```
obj1 = A()
print("obj1.__x= ", obj1.x)
obj2 = obj1.B()
obj2.display()
print("obj1.__x= ", obj1.x)
```

Create an object of the enclosed class

Result:

```
obj1.__x=  0
B.x=  12
obj1.__x=  12
```


Exp. 2

```
class Car:
    def __init__(self, m, y, p):
        self.model = m
        self.year = y
        self.eng = self.Engine(p)
    class Engine:
        def __init__(self, p):
            self.power = p
        def display(self):
            print("Power: ", self.power)
    def display(self):
        print("*** Car ***")
        print("Model: ", self.model)
        print("Year: ", self.year)
        self.eng.display()
obj1 = Car("Ford", 2023, 90)
obj1.display()
```

Result:

```
*** Car ***
Model:  Ford
Year:  2023
Power:  90
```


6

Collections

Iterator, List, Set & Queue

1. Iterators

'**Iterator**' is an object that is used to iterate over **iterable** objects like lists, tuples, dictionaries, and sets.

In Python, the iterator object is initialized using the `iter()` method. It uses the `next()` method for iteration.

- To create an iterator object:

```
L = [1,2,3]
L_iter = iter(L)
```

1. Iterators (next)

- To move to the next element:

```
print(next(L_iter))
```

- Looping through an iterator:

```
my_list = [1, 2, 3, 4]
L_iter = iter(my_list)
while True:
    try:
        element = next(L_iter)
        print(element)
    except StopIteration:
        break
```

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

The `for` loop creates an iterator object and executes the `next()` method for each loop.

Exp.

```
items = ["aaa", "bbb", "TWO", "ccc"]
L_iter = iter(items)
while True:
    try:
        elem = next(L_iter)
        if( elem == "TWO" ):
            items.remove(elem)
    except StopIteration:
        break
print(items)
```

```
items = ["aaa", "bbb", "TWO", "ccc"]
for elem in items:
    if (elem == "TWO"):
        items.remove(elem)
    print(elem)
print(items)
```

Result:

```
aaa
bbb
TWO
['aaa', 'bbb', 'ccc']
```


2. Lists & Arrays

A **List** is a data structure that holds a collection of items.

Lists have a number of important characteristics:

- List items are enclosed in square brackets, ex. [1, 2, 3].
- Lists are **ordered** – i.e. the items appear in a specific order.
- **Duplication** of items is possible, as each item has its own distinct location.
- The items can be of **different data types**.
- Lists are **modifiable**, which means that you can add or remove items after a list has been created.
- Preferred for **shorter sequence** of data items
- Consume **larger memory** for easy addition of elements

2. Lists & Arrays

Exp.

```
mylist = ['Batna', 5, 'Algiers', 16, 'Oran', 31, 'Adrar', 1, 'Batna', True]
print(mylist)
print(len(mylist))
mylist = list((False, 5, ['happy', 5.2]))
print(mylist)
```

Result:

```
['Batna', 5, 'Algiers', 16, 'Oran', 31, 'Adrar', 1, 'Batna', True]
10
[False, 5, ['happy', 5.2]]
```

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

2. Lists & Arrays

An **Array** is a data structure that holds a collection of items.

Arrays have a number of important characteristics:

- Array items are enclosed in square brackets, ex. [1, 2, 3].
- Arrays are **ordered** – i.e. the items appear in a specific order.
- **Duplication** of items is possible, as each item has its own distinct location.
- According to the **kind of array** used, the items **can or cannot** be of different data types.
- Arrays are **modifiable**, which means that you can add or remove items after an array has been created.
- Preferred for **longer sequence** of data items
- Comparatively more **compact in memory** size

2. Lists & Arrays

To use arrays in Python, you can import either an `array` module or a `NumPy` module.

2. Lists & Arrays

Exp. 1

```
import array as arr
myArray = arr.array('i', [1, 2, 3]) # same type for all items ‘i’ means Integer
print(myArray)

print(len(myArray))
```

Result:

```
array('i', [1, 2, 3])
3
```

2. Lists & Arrays

Exp. 1

```
import numpy as np
myArray = np.array([1, 2, 3, 'happy']) # different type for items accepted
print(myArray)

print(len(myArray))
```

Result:

```
['1' '2' '3' 'happy']
4
```


3. The sets

A **set** is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed). Set elements may be of different types.

Mathematical operations on sets are possible, such as union, intersection, symmetric difference, etc.

3. The sets (next)

In python, to create a set we can use **accolades** or the function `set()`.

Exp.

```
S1 = {1, 2, 3, 4, 3, 2}
S2 = set([1, 2, 3, 2])
S3 = {1.0, "Hello", False}
print(S1)
print(S2)
print(S3)
```

Result:

```
{1, 2, 3, 4}
{1, 2, 3}
{'Hello', 1.0, False}
```


3. The sets (next)

To modify a set we can use:

- `add()`
- `update()`
- `discard()`
- `remove()`
- `clear()`

Exp.

```
S1 = {1, 4}
print(S1)
S1.add(2)
print(S1)
S1.update([2, 3, 4])
print(S1)
S1.discard(4)
print(S1)
S1.remove(2)
print(S1)
S1.clear()
print(S1)
S1.discard(2)
print(S1)
S1.remove(2)
print(S1)
```

Result:

```
{1, 4}
{1, 2, 4}
{1, 2, 3, 4}
{1, 2, 3}
{1, 3}
set()
set()
Traceback (most recent call last):
  File ".\Exp_6_3_2.py", line 22, in <module>
    S1.remove(2)
KeyError: 2
```


Mathematical set operations

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print('A union B: ', A | B)
print('A union B: ', A.union(B))
print('A intersection B: ', A & B)
print('A intersection B: ', A.intersection(B))
print('A difference B: ', A - B)
print('A difference B: ', A.difference(B))
print('A symmetric difference B: ', A ^ B)
print('A symmetric difference B: ', A.symmetric_difference(B))
```

Result:

```
A union B:  {1, 2, 3, 4, 5, 6, 7, 8}
A union B:  {1, 2, 3, 4, 5, 6, 7, 8}
A intersection B:  {4, 5}
A intersection B:  {4, 5}
A difference B:  {1, 2, 3}
A difference B:  {1, 2, 3}
A symmetric difference B:  {1, 2, 3, 6, 7, 8}
A symmetric difference B:  {1, 2, 3, 6, 7, 8}
```


4. The Queues

A queue is a linear data structure that stores items in **First In First Out** (FIFO) manner.

Operations associated with queue are:

- **Enqueue**: Adds an item to the queue.
- **Dequeue**: Removes an item from the queue.
- **Front**: Get the front item from the queue
- **Rear**: Get the last item from queue.

4. The Queues (next)

To create a queue we use 'queue.Queue' module of Python.

There are various functions available in this module:

- **maxsize** – Number of items allowed in the queue. (maxsize=0 means infinite)
- **empty()** – Returns True if the queue is empty, False otherwise.
- **full()** – Returns True if there are maxsize items in the queue.

- **get()** – Removes and returns an item from the queue. If queue is empty, **wait** until an item is available.
- **get_nowait()** – Returns an item if one is immediately available, else raise exception QueueEmpty.
- **put(item)** – Puts an item into the queue. If the queue is full, **wait** until a free slot is available.
- **put_nowait(item)** – Puts an item into the queue. If no free slot is immediately available, raise exception QueueFull.
- **qsize()** – Returns the number of items in the queue.

Exp.

```
from queue import Queue
q = Queue(maxsize = 3)
print('Queue size: ', q.qsize())
q.put('a')
q.put('b')
q.put('c')
print("Full: ", q.full())
while (not(q.empty())):
    print('dequeue: ', q.get())
print('Queue size: ', q.qsize())
```

Result:

```
Queue size:  0
Full:  True
dequeue:  a
dequeue:  b
dequeue:  c
Queue size:  0
```


5. The Dictionary

Dictionaries are used to store data values in the form of pairs <key:value>.

A dictionary is an ordered collection (Python ≥ 3.7), changeable and does not allow duplicates.

Items in a dictionary can be referred to by using the key value.

5. The Dictionary (next)

Exp.

```
myDict =       {
  "Trademark": "Audi",
  "Model": "A1",
  "Year": 2022
}
print(myDict)
print(myDict["Model"])
print(len(myDict))
```

Result:

```
{'Trademark': 'Audi',
'Model': 'A1', 'Year':
2022}
A1
3
```

Looping Techniques

Exp.

```
myDict = {
  "Trademark": "Audi",
  "Model": "A1",
  "Year": 2022
}

for k, v in myDict.items():
  print(k, v)
for v in myDict.values():
  print(v)
for k in myDict.keys():
  print(k, myDict[k])
```

Result:

```
Trademark Audi
Model A1
Year 2022
Audi
A1
2022
Trademark Audi
Model A1
Year 2022
```


Add/update items

Exp.

```
myDict = {
  "Trademark": "Audi",
  "Model": "A1",
  "Year": 2022
}

myDict["color"] = "blue"
myDict.update({"Year": 2021})
myDict.update({"Power": 175})
print(myDict)
```

Result:

```
{'Trademark': 'Audi', 'Model': 'A1',
'Year': 2021, 'color': 'blue',
'Power': 175}
```


Remove items

Exp.

```
myDict = {
  "Trademark": "Audi",
  "Model": "A1",
  "Year": 2022
}

myDict.pop("Year")
print(myDict)
del myDict["Model"]
print(myDict)
myDict.clear()
print(myDict)
```

Result:

```
{'Trademark': 'Audi', 'Model': 'A1'}
{'Trademark': 'Audi'}
{}
```


7

Graphics User Interface GUI

A bit of front-end

1. The package Tkinter

Tkinter (**Tk inter**face) is an open source graphical user interface (GUI) library used in Python.

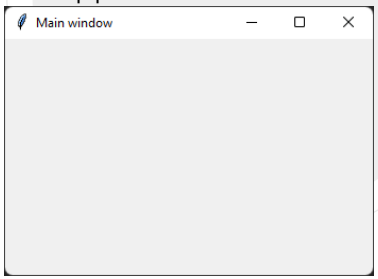
It comes from an adaptation of the **Tk** graphic library written for the Tool Command Language (Tcl) noted Tk/TCL.

2. The window

Tk is a class defined in the **`Tkinter`** module which allows to create the **master window** of the application.

`Exp.`

```
from tkinter import Tk
root=Tk()
root.mainloop()
```



2. The window (next)

The `mainloop()` of Tkinter is a gigantic infinite while loop.

It is useful to redraw the GUI and show the changes.

3. Geometry of components

The placement of components is done through a geometry manager.

Tkinter have three geometry managers: pack, grid et place.

3. Geometry of components (next)

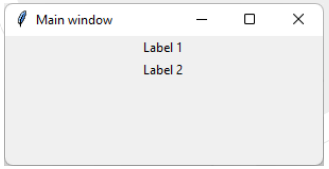
pack: Vertical or horizontal stacking.

grid: Placement according to a 2D grid.

place: Placement at a defined position in pixels.

Exp. 1 (pack):

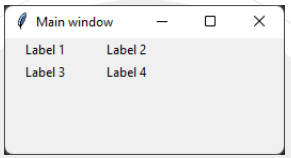
```
from tkinter import *
root = Tk()
root.title("Main window")
l1 = Label(root, width='10', text="Label 1")
l1.pack()
l2 = Label(root, width='10', text="Label 2")
l2.pack()
root.mainloop()
```



Exp. 2 (pack):

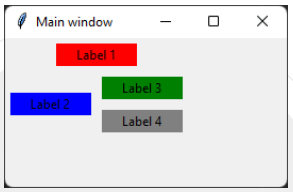

```
from tkinter import *
root = Tk()
root.title("Main window")
l1 = Label(root, width='10', text="Label 1", background = "blue")
l1.pack(side = LEFT, padx=5, pady=5)
l2 = Label(root, width='10', text="Label 2", background = "red")
l2.pack(side = LEFT, padx=5, pady=5)
root.mainloop()
```


Exp. 3 (grid):



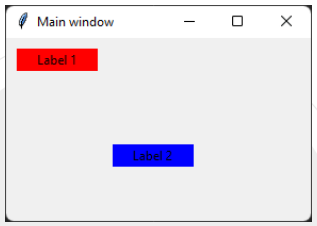

```
from tkinter import *
root = Tk()
root.title("Main window")
l1 = Label(root, width='10', text="Label 1")
l1.grid(row=2, column=0)
l2 = Label(root, width='10', text="Label 2")
l2.grid(row=2, column=1)
l3 = Label(root, width='10', text="Label 3")
l3.grid(row=3, column=0)
l4 = Label(root, width='10', text="Label 4")
l4.grid(row=3, column=1)
root.mainloop()
```

Exp. 4 (grid):




```
from tkinter import *
root = Tk()
root.title("Main window")
l1 = Label(root, width='10', text="Label 1", background="red")
l1.grid(row=0, column=0, columnspan=2, padx=5, pady=5)
l2 = Label(root, width='10', text="Label 2", background="blue")
l2.grid(row=1, column=0, rowspan=2, padx=5, pady=5)
l3 = Label(root, width='10', text="Label 3", background="green")
l3.grid(row=1, column=1, padx=5, pady=5)
l4 = Label(root, width='10', text="Label 4", background="gray")
l4.grid(row=2, column=1, padx=5, pady=5)
root.mainloop()
```


Exp. 5 (place):



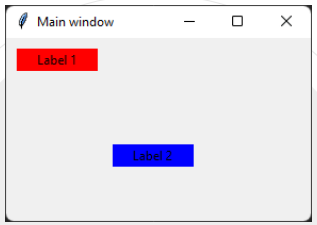

```
from tkinter import *
root = Tk()
root.title("Main window")
l1 = Label(root, width='10', text="Label 1", background="red")
l1.place(x=10, y=10)
l2 = Label(root, width='10', text="Label 2", background="blue")
l2.place(x=100, y=100)
root.mainloop()
```


4. Using OOP in GUI programming

We will see the advantage of using classes to manage a Tkinter GUI.

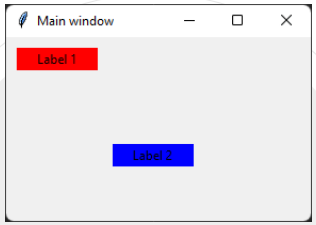
A graphical interface must use variables and functions. The latter are only used for this interface, so it is useful to **group them together** and **avoid** defining them as **global** variables and functions.

Exp. 1 (class & GUI):




```
from tkinter import *
class Exp_7_4_1:
    def __init__(self):
        root = Tk()
        root.title("Main window")
        l1 = Label(root, width='10', text="Label 1", background="red")
        l1.place(x=10, y=10)
        l2 = Label(root, width='10', text="Label 2", background="blue")
        l2.place(x=100, y=100)
        root.mainloop()
Exp_7_4_1()
```

Exp. 2 (class & GUI):




```
import tkinter as tk
class Exp_7_4_2(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.l1 = tk.Label(self, width='10', text="Label 1", background="red")
        self.l1.place(x=10, y=10)
        self.l2 = tk.Label(self, width='10', text="Label 2", background="blue")
        self.l2.place(x=100, y=100)
if __name__ == "__main__":
    app = Exp_7_4_2()
    app.mainloop()
```

The process created by the command line is called "__main__".
For this, we test the name of the process before running the main program.

5. Some widgets

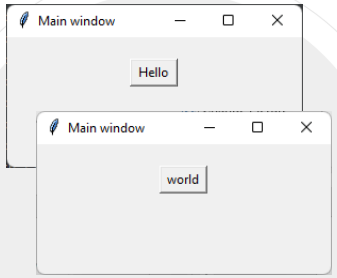
5.1. Label

Label is a class in the library tkinter. It allows to create a widget consisting of a short text message.

5.2. Button

The class Button of tkinter allows to create widget button.

Exp. (Button):




```
import tkinter as tk
class Exp_7_5_2(tk.Tk):
    def play(self):
        self.b['text'] = 'world'
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.b = tk.Button(self, text="Hello", command=self.play, width=5)
        self.b.pack(padx=50, pady=20)
if __name__ == "__main__":
    app = Exp_7_5_2()
    app.mainloop()
```


5. Some widgets (next)

5.3. Entry

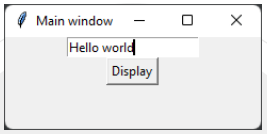
The Entry class of tkinter allows to create an editable text zone to enter a string.

5.4. Menu

The Menu class of tkinter allows to create drop-down menu.

Exp. (Entry):

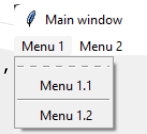
```
import tkinter as tk
class Exp_7_5_3(tk.Tk):
    def display(self):
        print(self.ent1.get())
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.ent1 = tk.Entry(self)
        self.ent1.pack()
        self.b1 = tk.Button(self, text="Display", command=self.display)
        self.b1.pack()
if __name__ == "__main__":
    app = Exp_7_5_3()
    app.mainloop()
```



Result:
Hello world

Exp. (Menu):

```
import tkinter as tk
class Exp_7_5_4(tk.Tk):
    def com1(self):
        print("Menu 1.1 cliqued")
    def com2(self):
        print("Menu 1.2 cliqued")
    def com3(self):
        print("Menu 2.1 cliqued")
    def com4(self):
        print("Menu 2.2 cliqued")
    def __init__(self):
        super().__init__()
        self.title("Main window")
        # Menu bar
        self.bm1 = tk.Menu(self)
        self.config(menu=self.bm1)
        # Menu 1
        self.m1 = tk.Menu(self.bm1)
        self.m1.add_command(label="Menu 1.1",
command=self.com1)
        self.m1.add_separator()
        self.m1.add_command(label="Menu 1.2",
command=self.com2)
        self.bm1.add_cascade(label="Menu 1",
menu=self.m1)
        # Menu 2
        self.m2 = tk.Menu(self.bm1)
        self.m2.add_command(label="Menu 2.1",
command=self.com3)
        self.m2.add_separator()
        self.m2.add_command(label="Menu 2.2",
command=self.com4)
        self.bm1.add_cascade(label="Menu 2",
menu=self.m2)
if __name__ == "__main__":
    app = Exp_7_5_4()
    app.mainloop()
```



5. Some widgets (next)

5.5. Frame

The Frame class of tkinter allows to create a widget used as a **container**. Itis very useful to **group** and **organize** interfaces containing many widgets.

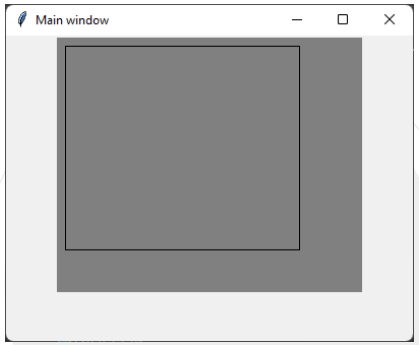
5.6. Canvas

The Canvas is a **rectangular** area intended for **drawing** pictures or other complex layouts. You can place graphics, text, widgets or frames on a Canvas.

Exp. (Frame):

```
import tkinter as tk
class Exp_7_5_5(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.frm = tk.Frame(self, background="gray")
        self.frm.pack()
        self.ent1 = tk.Entry(self.frm)
        self.ent1.pack(padx=5, pady=5)
        self.b1 = tk.Button(self.frm, text="Display")
        self.b1.pack(padx=5, pady=5)
if __name__ == "__main__":
    app = Exp_7_5_5()
    app.mainloop()
```

Exp. (Canvas):




```
import tkinter as tk
class Exp_7_5_6(tk.Tk):
    def play(self):
        self.b['text'] = 'world'
    def __init__(self):
        super().__init__()
        self.geometry("400x300")   # Set Window Size
        self.title("Main window")
        self.c = tk.Canvas(self, width=300, height=250, background='gray')
        self.rect = self.c.create_rectangle(10, 10, 240, 210)
        self.c.pack()
if __name__ == "__main__":
    app = Exp_7_5_6()
    app.mainloop()
```


5. Some widgets (next)

5.7. Radiobutton

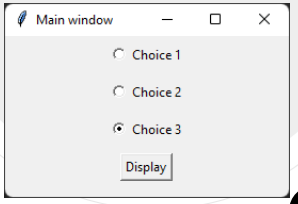
The **Radiobutton** class of tkinter allows to create a widget used in choices leading to a **single answer**.

5.8. Checkbutton

The **Checkbutton** class of tkinter allows to create a widget used in choices leading to **multiple answers**.

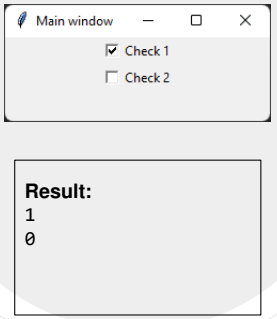
Exp. (**Radiobutton**):

```
import tkinter as tk
class Exp_7_5_7(tk.Tk):
    def com0(self):
        print(self.v.get())
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.v = tk.IntVar()
        self.r1     =     tk.Radiobutton(self,
variable=self.v, text="Choice 1", value=1)
        self.r1.pack(padx=5, pady=5)
        self.r2     =     tk.Radiobutton(self,
variable=self.v, text="Choice 2", value=2)
        self.r2.pack(padx=5, pady=5)
        self.r3         =         tk.Radiobutton(self,
variable=self.v, text="Choice 3", value=3)
        self.r3.pack(padx=5, pady=5)
        self.b1         =         tk.Button(self,
text="Display", command=self.com0)
        self.b1.pack(padx=5, pady=5)
if __name__ == "__main__":
    app = Exp_7_5_7()
    app.mainloop()
```



Exp. (**Checkbutton**):

```
import tkinter as tk
class Exp_7_5_8(tk.Tk):
    def com1(self):
        print(self.v1.get())
        print(self.v2.get())
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.v1 = tk.IntVar()
        self.ch1 = tk.Checkbutton(self,
text='Check 1', command=self.com1,
variable=self.v1)
        self.ch1.pack()
        self.v2 = tk.IntVar()
        self.ch2 = tk.Checkbutton(self,
text='Check 2', command=self.com1,
variable=self.v2)
        self.ch2.pack()
if __name__ == "__main__":
    app = Exp_7_5_8()
    app.mainloop()
```



5. Some widgets (next)

5.9. Listbox

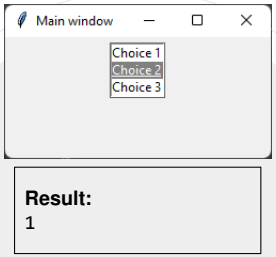
The **Listbox** class of tkinter allows to create a widget which proposes a list whose elements are **selectable**.

5.10. Combobox

The **Combobox** class of tkinter allows to create a widget that combines a text field with a pop-down list of values.

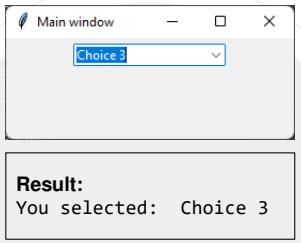
Exp. (**Listbox**):

```
import tkinter as tk
class Exp_7_5_9(tk.Tk):
    def com1(self, event):
        print(self.lbox1.curselection()[0])
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.L = ['Choice 1','Choice 2', 'Choice 3']
        z = tk.StringVar(value=self.L)
        self.lbox1 = tk.Listbox(self, width=8, height=len(self.L), selectbackground="gray",
listvar=z)
        self.lbox1.pack(padx=5, pady=5)
        self.lbox1.bind('<<ListboxSelect>>', self.com1)
if __name__ == "__main__":
    app = Exp_7_5_9()
    app.mainloop()
```



Exp. (**Combobox**):

```
import tkinter as tk
from tkinter import ttk
class Exp_7_5_10(tk.Tk):
    def com1(self, event):
        print('You selected: ', self.z.get())
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.z = tk.StringVar()
        self.cbox1 = ttk.Combobox(self, values=['Choice 1', 'Choice 2', 'Choice 3'],
                     textvariable=self.z)
        self.cbox1.pack(padx=5, pady=5)
        self.cbox1.bind('<<ComboboxSelected>>', self.com1)
if __name__ == "__main__":
    app = Exp_7_5_10()
    app.mainloop()
```



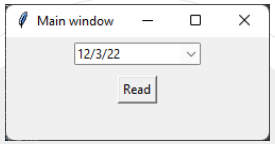
5. Some widgets (next)

5.11. DateEntry

The **DateEntry** class of tkinter allows to create a widget to choose a date from a calendar.

Exp. (DateEntry):

```
import tkinter as tk
from tkcalendar import DateEntry
class Exp_7_5_11(tk.Tk):
    def com1(self):
        print('You selected: ', self.de.get_date())
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.de = DateEntry(self, width=16, bd=2)
        self.de.pack(padx=5, pady=5)
        self.b1 = tk.Button(self, text='Read', command=self.com1)
        self.b1.pack(padx=5, pady=5)
if __name__ == "__main__":
    app = Exp_7_5_11()
    app.mainloop()
```



Result:
You selected: 2022-12-22

6. Event-based programming

When the user clicks on a component, moves the mouse over it or …, Tkinter creates an object of **type event** to represent this action.

An **event** is an object that represents a user's interaction with a GUI component and can be manipulated by our programs.

6. Event-based programming

`Tkinter` listens to some mouse or keyboard events through the infinite `mainloop()`.

A **keyboard event** is the pressing or releasing of a keyboard key.

A **mouse event** is the clicking or releasing of a button, moving the mouse on a certain area, turning the scroll wheel.

An event **happened** from a window or a widget can be **linked** to a function or method.

6.1. Keyboard event

To **link** a keyboard event to a function we use the method `bind()` of Tkinter as shown on this example:

```
from tkinter import *
def keyPressed(event):
    t=event.keysym
    print('Key %s pressed' %t)
root = Tk()
root.bind('<Key>', keyPressed)
root.mainloop()
```

Variable containing the key that caused the event

any key code

6.1.1. Codes of some keys

Names of events representing "<event>" and associated with the **pressing** or **releasing** of certain keys:

- **Pressing** a character key, for example '*a*', is called *<KeyPress-a>* or simply *<a>*.
- For the **release** of the key '*a*', we write *<KeyRelease-a>*.
- For a key combination like **ALT+CTRL-a** (simultaneous press), the event will be named <Control-Shift-KeyPress-a>.

- Arrows: <Left>, <Right>, <Up>, <Down>
- SPACE: <space>
- ENTER key: <Return> or, on the numeric keypad, <KP_Enter>
- ESCAPE key: <Escape>
- The keys of the numeric keypad: <KP_1>, <KP_2>, ...

A **complete list** of key event names is available on the following link:

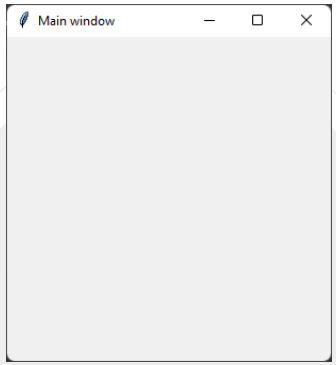
http://tkinter.fdex.eu/doc/event.html#noms-des-touches

6.2. Mouse event

A window or a widget can capture mouse actions.

To **link** a mouse event to a function we use the method bind() of Tkinter as shown on the following example:


```
from tkinter import *
import tkinter as tk
class Exp_7_6_2_1(tk.Tk):
    def clic(self, event):
        x, y = event.x, event.y
        print(x, y)
def __init__(self):
        super().__init__()
        self.title("Main window")
        self.cnv = Canvas(self, width=300, height=300)
        self.cnv.pack()
        self.cnv.bind("<Button-1>", self.clic)
if __name__ == "__main__":
    app = Exp_7_6_2_1()
    app.mainloop()
```



Result:

```
236 120
102 58
163 247
300 300
3 1
```

6.2.1. Codes of some mouse events

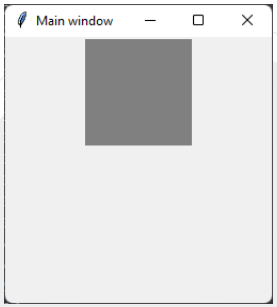
Mouse event	Action
<Button>	Button click
<Button-1>	Left click
<Button-2>	Center click
<Button-3>	Right click
<Button-4>	Wheel up
<Button-5>	Wheel down
<ButtonRelease>	Release a button
<Motion>	Moving the cursor of the mouse

Some combinations:

- <B1-Motion>: Button-1 pressed and drag the mouse.
- <Enter>: Mouse entered into the space of the widget.
- <Leave>: Mouse leaved the space of the widget.


```
from tkinter import *
import tkinter as tk
class Exp_7_6_2_2(tk.Tk):
    def mouvement(self, event):
        print("x = ", event.x, ' - y = ', event.y)
    def __init__(self):
        super().__init__()
        self.title("Main window")
        self.geometry("250x250")
        self.cnv = Canvas(self, width=100, height=100, background='gray')
        self.cnv.pack()
        self.cnv.bind("<B1-Motion>", self.mouvement)
if __name__ == "__main__":
    app = Exp_7_6_2_2()
    app.mainloop()
```

Click left button and move de mouse



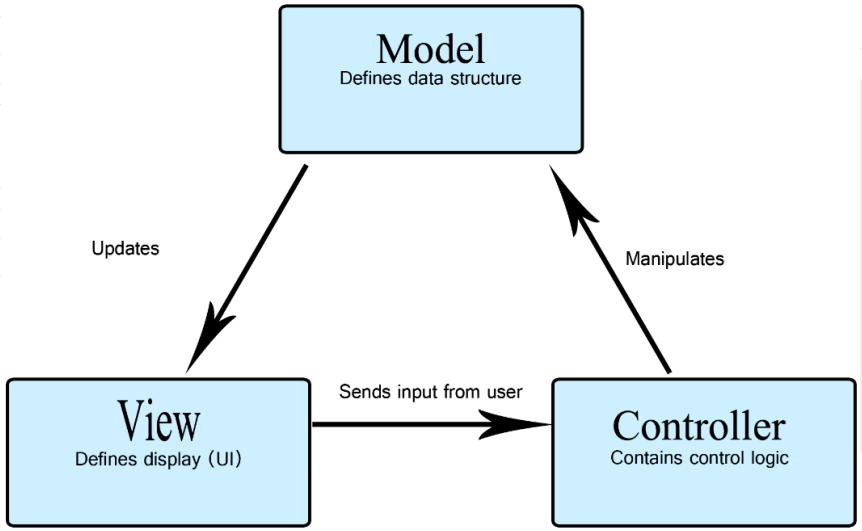
Result:

```
x =  96  - y =  61
x =  92  - y =  59
x =  86  - y =  59
x =  84  - y =  58
```


7. MVC Architecture (Model-View-Controller)

This architecture allows to organize an application in three layers in order to separate responsibilities:

- The model: is responsible for the logic and data part (business layer)
- The view: is responsible for the presentation of the model data to the user.
- The controller: it ensures the coordination between the model and the view to answer the user's request.



Exp.

A student management application for the school.

```
class StudentModel:
    def __init__(self, m, f, l):
        self.__mat = m
        self.__firstName = f
        self.__lastName = l
    @property
    def mat(self):
        return self.__mat
    @mat.setter
    def mat(self, value):
        self.__mat = value
    @property
    def firstName(self):
        return self.__firstName
    @firstName.setter
    def firstName(self, value):
        self.__firstName = value
    @property
    def lastName(self):
        return self.__lastName
    @lastName.setter
    def lastName(self, value):
        self.__lastName = value
class StudentView:
    def display(self, m, l, f):
        print("Student n°: ", m, " Last name: ",
l, " First name: ", f)
```


Exp. (suite)

```
class StudentController:
    def __init__(self, m:StudentModel,
v:StudentView):
        self.__model:StudentModel = m
        self.__view:StudentView = v
    @property
    def model(self):
        return self.__model
    @model.setter
    def model(self, value):
        self.__model = value
    @property
    def view(self):
        return self.__view
    @view.setter
    def view(self, value):
        self.__view = value
    def majView(self):
        self.__view.display(self.model.mat,
self.model.lastName, self.model.firstName)
def getStudentFromDatabase():
# We simulate the extraction of data from the database
    std = StudentModel(5, "TITI", "titi")
    return std
model = getStudentFromDatabase()
view = StudentView()
controller = StudentController(model, view)
controller.majView()
controller.model.lastName = "TOTO"
controller.model.firstName = "toto"
controller.majView()
```

Result:

```
Student n°:  5  Last name:
titi  First name:  TITI
Student n°:  5  Last name:
TOTO  First name:  toto
```


8

Exception handling

and how it works

1. What is an exception?

Errors (exceptions) that are generated during the execution of a program cause the program to stop if they are not handled.

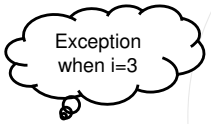
When writing a program it is necessary to **manage** the **errors** (**exceptions**) that can be generated during its execution.

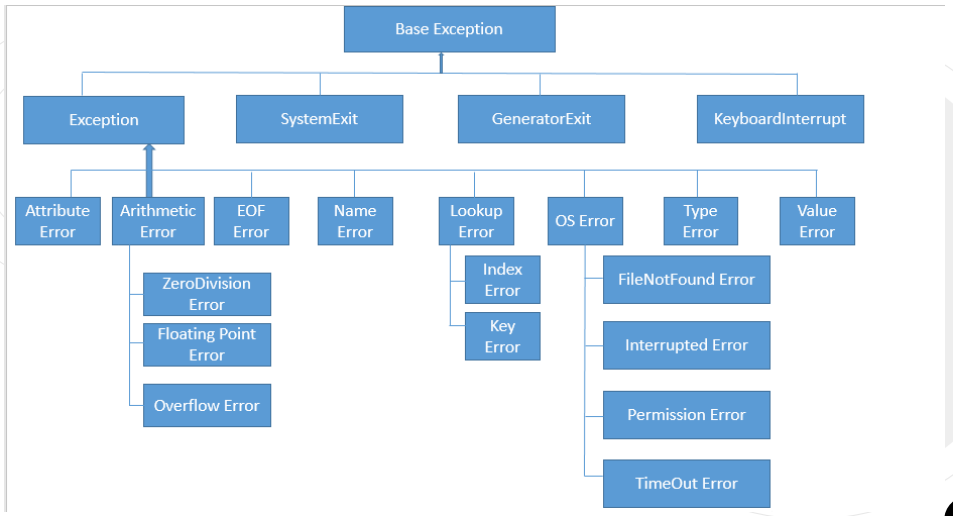
1. What is an exception? (next)

Exp.

```
class Exp_8_1:
    def f(self):
        T = [1, 2]
        for i in range(0, 3):
            print(T[i])

obj = Exp_8_1()
obj.f()
```





1. What is an exception? (next)

Frequent exception classes

Classe	Description
SystemExit	Raised by the sys.exit() function.
ArithmeticError	Base class for all errors that occur for numeric calculation.
EOFError	Raised when there is no input and the end of file is reached.
KeyboardInterrupt	Raised when pressing Ctrl+c
IndexError	Raised when an index is not found in a sequence.
IOError	Raised when an input/ output operation fails.

2. How to handle exceptions in a program

To handle the exception and avoid the sudden stop of a program, we must use the statement:

- try/except/else/finally;

2. How to handle exceptions in a program (next)

Syntax

```
try:
   Operations to do
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
else:
   If there is no exception then execute this block.
finally:
   This would always be executed.
```

2. How to handle exceptions in a program (next)

Exp.

```
def divide(x, y):
    try:
        result = x // y   # Gives only fractional part as answer
    except ZeroDivisionError:
        print("You are dividing by zero ")
    else:
        print("Your answer is :", result)
    finally:
        # This block is always executed regardless of exception generation.
        print('This is always executed')
divide(3, 2)
divide(3, 0)
```

Result:

```
Your answer is : 1
This is always executed
You are dividing by zero
This is always executed
```


3. Intercepting hierarchical exceptions

This is the case of a block that can generate several types of exceptions.

In this case we will use successive catches to catch several exceptions

3. Intercepting hierarchical exceptions (next)

Exp. (catch successifs)

```
try:
    L = [1, 2, 3]
    x = int(input())
    L[1] = 30 / x
    L[x] = 30
except ArithmeticError as e:
    print(f"{e}, {e.__class__}")
except IndexError as e:
    print(f"{e}, {e.__class__}")
except Exception as e:
    print(f"{e}, {e.__class__}")
print("Program continuation ...")
```

Result:

```
0
division by zero, <class
'ZeroDivisionError'>
Program continuation ...

5
list assignment index out of range,
<class 'IndexError'>
Program continuation ...

A
invalid literal for int() with base
10: 'A', <class 'ValueError'>
Program continuation ...
```


9

Streams and files

Data

The **input/output** module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O: **text I/O**, **binary I/O** and **raw I/O**.

1. Text I/O

The **text I/O** waits for and generates str objects.

To use a text stream from a file we use the methods `open()`, `read()`, `readLine()`, `write()`, `writeLines()`, `close()`, of module `io`.

r: read
w: write
a: append

Syntax:

```
f = open('fileName', 'mode')
s = f.read()      or   f.readLine()
f.write('a string')    or    f.writeLines('line1', …)
f.close()
```

Exp.

```
print("test 1")
f = open('test1.txt', 'w')
string = "Hello world\n"
f.write(string)
f.close()
```

```
print("test 2")
f = open('test1.txt', 'r')
string = f.read()
print(string)
f.close()
```

```
print("test 3")
f = open('test3.txt', 'a')
f.write('Hello world\n')
f.close()
```

```
print("test 4")
f = open('test4.txt', 'a')
lines = ['Hello world\n', 'Good
morning\n', 'Good afternoon\n']
f.writelines(lines)
f.close()
```

```
print("test 5")
f = open('test4.txt', 'r')
while True:
    line = f.readline()
    if line == '':
        break
    print(line)
f.close()
```

Result:

```
test 1
test 2
Hello world

test 3
test 4
test 5
Hello world

Good morning

Good afternoon
```


1. Text I/O (next)

In-memory text streams are also available as `StringIO` objects:

Syntax:

```
f = io.StringIO("text content of the in-memory stream")
s = f.read()      or   f.readLine()
f.write('a string')    or    f.writeLines('line1', …)
f.close()
```


Exp.

```
import io
myStream = io.StringIO()
myStream.write('Hello world.\n')
myStream.seek(0)  # Points to the beginning of the stream
print(myStream.readline())
myStream.close()  # Free memory buffer
```

Result:

```
Hello world.
```


2. Binary I/O

rb: read binary
wb: write binary
ab: append binary

Binary I/O (or buffered I/O) waits for bytes-like objects and generates bytes objects (non-text data).

To use a binary stream from a file we use the methods `open()`, `read()`, `write()`, `close()`, of module `io`.

Syntax:

```
f = open('fileName', 'mode')
s = f.read()
f.write(data)
f.close()
```

Exp.

```
f = open('test.dat', 'wb')
b = bytearray(b'Hello world')
f.write(b)
f.close()

f = open('test.dat', 'rb')
print(f.read())
f.close()
```

Result:
`b'Hello world'`

2. Binary I/O (next)

In-memory binary streams are also available as `BytesIO` objects:

Syntax:

```
f = io.BytesIO(b"initial binary data: \x00\x01")
s = f.read()
f.write('a string')
f.close()
```


Exp.

```
import io
stream_str = io.BytesIO(b"Hello world \x00\x01")
stream_str.write(b"Good bye")
print(stream_str.getvalue())
stream_str.close()
```

Result:
`b'Good byerld \x00\x01'`

3. RAW I/O

rb: read binary
wb: write binary
ab: append binary

Raw I/O (also called unbuffered I/O) is generally used as a low-level building-block for binary and text streams.

To use a raw stream from a file we use the methods `open(), read(), write(), close()`, of module `io`.

Syntax:

```
f = open('fileName', 'mode' , buffering=0)
s = f.read()
f.write(data)
f.close()
```

Exp.

```
import io

f = io.open("image.png", "rb", buffering=0)
print(f.read())
f.close()
```

Result:

```
O<S\xcd\x8bI\x96\xf2E\x03\xb3~
e\x1a>\x96\x89\xff\x00\xbdh\xf
7\xf9\xf5U\x1e\x84`X\xe9r\xf2t
\xb6\xab\xbc\xe8\x9b\xa3\xe9r\
xa9\xae\xc7\xf3\x0e
\xccg7c\x1b\'\x11\xaf\x0c\xd8\
x8e6\x0fB\x9a\x9d\xbf"\x16z\xd
euw\x00\xde\xb7.\xe0X6]\xc2\xe
3\xc2\xf0,.\x8f\x0b\xa1\x8e\xf
b\x10R@\xd8\xa3\x04\xef6\x1c{V
\xd5\x00DD\x01\x11\x10\x04D@\x
11\x11\x00DD\x01\x11\x10\x04D@
\x7f\xff\xd9'
```