Defining and using classes

A program which uses object-oriented programming paradigm involves:

- ***to define*** the classes that it uses,
- ***to use*** these classes.

In C++, ***to define*** a class consists of:

- ***declaring*** the class,
- ***implementing*** this class.

Using classes in an object-oriented application involves:

- ***to create*** a set of objects that are instances of these classes,
- ***to communicate*** with the created objects by using *messages*.

The mechanism for ***sending*** and ***receiving*** the messages to/from objects is represented by ***calling member functions***.

A) Example in C++

```
//file "stack.h"
class stack {
  int dim;              //data member
  char *buff;           //data member
public:
  stack(int);           //constructor
  ~stack();             //destructor
  void push(char);      //function member
  char& pop();          //function member
  bool isEmpty();       //function member
} ;

//file "stack.cpp"
#include "stack.h"
stack::stack(int n) {
  //code for the constructor
}
stack::~stack(int n) {
  //code for the destructor
}
```

```
void stack::push(char c) {
  //code for the function push
}
char& stack::pop() {
  //code for function pop
}
bool stack::isEmpty() {
  //code for function isEmpty
}

//file “main.cpp”
#include “stack.h”
int main()
{
  //declaring stack objects
  stack st1(100), st2(50);
  //using stack objects
  st1.push(‘a’);
  // ...
}
```

B) Example in Python

```
# module mystack.py
class MyStack:
    def __init__(self):
        self.items = []
   # the top of the stack is the last element
    def push(self, elem):
        self.items.append(elem)
    def pop(self):
        return self.items.pop()
    def empty(self) -> bool:
        return not self.items

# module usemystack.py
from mystack import *
stack1 = MyStack()
stack1.push(2)          #add only integers
stack1.push(3)          #add only integers
x = stack1.pop()        #a homogeneous container
print(x)
```

```
stack2 = MyStack()
stack2.push('a')      #add only strings
stack2.push('b')      #add only strings
y = stack2.pop()      #a homogeneous container
print(x)
```

Remark.

1. In ***Python*** there is no difference between *declaring* and *implementing* a class.
2. The ***class*** *statement* is an *executable statement* that ***creates*** a class object and ***assigns*** it a name.
3. ***Methods*** are *functions* that are defined ***inside classes***, being attached to classes and used to ***process instances*** of those classes.

A. Declaring classes

- **In C++,** the syntax for ***declaring a class*** is the following:

```
<class declaration> ::= <class header> [<member declaration>] ;
<class header>      ::= <class specifier>
       <class name>
       [: <base class> {“,” < base class>}]
<member declaration>   ::= “{“ {<specific member>} “}”
<class specifier>   ::= struct | class
<base class>  ::= [<class modifier access> :] <class name>
<specific member >  ::= [<member modifier access> :]
       < member declaration >
```

- The above definition is specified in a ***metalanguage*** called ***Backus-Nour form notation*** (***BNF***), used to describe ***programming languages***

- A definition in ***BNF*** consists in one ore more ***rules***, each rule defining a ***syntactic category***, called ***non-terminal variable***:

```
    <non-terminal> ::= <expression>
```

- In the expression describing a non-terminal can be used the following elements:
  - ***Terminals*** (string of characters)
  - ***Meta-characters***:
    - `[]` – optional element
    - `()` – parenthesis for grouping other elements
    - `{}` – braces, for representing the repetition of zero or many times of other elements

- The header of a class contains mandatory:
  - the ***name*** of the class
  - its ***specifier***
    - usually it is **`struct`** or **`class`**
    - it ***may*** be **`union`** (in this case the class cannot be a ***base*** class, nor a ***derived*** class).

- The declaration of a class can be ***incomplete***, if its members are not declared.

- For example:

```
struct ClA;

struct ClB {
  // ...
  ClA *a;
  // ...
} ;

struct ClA {
  // ...
  ClB *b;
  // ...
} ;
```

- The ***syntax*** for defining classes in Python:

```
class ClassName:
    <statement-1>
    . . .
    <statement-n>
```

- The ***statements*** inside a class definition will usually be *function definitions*, but other statements are allowed.
  - For example, an *assignment statement* can be used to define ***global*** or ***shared*** (*static* in C++) ***immutable variable*** members:

```
class Experiment:
   n = 0          # shared variable n
   s = 0.0        # shared variable s
   list = []      #  error, list is a mutable object
   def __init__(self, x):
      self.x = x       # x is an instance variable
      Experiment.n = Experiment.n + 1
      Experiment.s = Experiment.s + x
   def med():
      return Experiment.s/Experiment.n
   def getn():
      return Experiment.n
```

- The declaration of a Python class may be ***incomplete*** by using the keyword ***pass***:

```
  class Experiment:
      pass
```

- The *reason* for defining incomplete classes is totally different from C++: the programmer is not interested in defining a certain class at a certain moment

- Recursive Python classes can be defined without using an incomplete definition:

```
class A:
   def increment(self, n):
      if n > 10:
         return B().decrement(n)
      else:
         return n + 1
class B:
   def decrement(self, n):
      if n < 0:
         return A().increment(n)
      else:
         return n - 1

n = A().increment(7)
print(n)
```

- However, using recursive object members in classes is not possible. The following definition represents an infinite recursion:

```python
class A:
   def __init__(self, a):
      self.a = a
      self.b = B(a)
   def increment(self):
      return self.a + 1
   def decrement_b(self):
      return self.b.decrement()

class B:
   def __init__(self, a):
      self.a = A(a)
      self.b = a
   def decrement(self):
      return self.b - 1
   def increment_a(self):
      return self.a.increment()
```

- A class can be ***derived*** from one or more classes, which are called ***base classes*** for the derived class.

- ***Base classes*** for a derived class must be specified by specifying their *names* and their ***access types***. The ***default*** access type for **`class`** is ***private***, and for **`struct`** the access type is ***public***.

Example:

```
class Triunghi : public Poligon {
  // ...
  double x2, y2;
  double Perimeter();
  double TwoEdges();
  // ...
} ;
```

- The declaration of a class member can be preceded, optionally, by an ***access modifier*** of the respective member (that is different from the access modifier of a base class).

- ***The access modifier*** for a class member specifies the way in which the respective member can be seen outside the class:
  - a ***public*** member is ***visible*** outside and it can be accessed,
  - a ***private*** member is ***inaccessible***,
  - a ***protected*** member ***can be accessed only in a class derived*** from the respective class with the ***public*** access modifier.

- An access modifier affects the accessibility of all declared member after this in the current class, until another access modifier is encountered.
  - If the ***first declared*** member of class does not have specified an access modifier, then, by ***default*** this is **`private`** for ***class*** and **`public`** for ***struct***.

Example. The *polygon* class stores pointers to the polygon vertices (not the vertex coordinates).

```
struct point {
  double x, y;
  point double x0=0, double y0=0)
   { x = x0; y = y0; }
} ;
```

```
class polygon {
  //private members
  int nr_vertices;
  point **vertices;
  double area, perimeter;
  void ComputePerimeter();
  void AdjustArea();
public :
  //public members
  polygon();
  ~polygon();
  int NrVertices() const { return nr_vertices; }
  void AddVertex (point*);
  point* operator[](int);
  double Area() const { return area; }
  double Perimeter() const { return perimeter; }
} ;
```

- ***Public members*** of a class can be accessed by ***outside*** of the class and they represent the ***interface*** of that class (the way the class communicates with the exterior). The ***private members*** are ***local*** to the respective class.

- The ***scope*** of the members of a class is represented by the ***class definition***.
    - This allows defining members in different classes with the same name, which represent different members.

- In ***Python*** (***absolutely***) *private members* of a class ***do not exists***
- However, there are some *conventions* related to the *access mode* to class members:

1. *Using single leading underscore*:
   - represents a weak "internal use" indicator
   - ***`from <module> import *`*** does not import objects whose name starts with an underscore
   - One leading underscore can be used (by convention) for *non-public* methods and instance variables
2. *Using double leading underscore*:
   - for a class attribute, it invokes a mechanism called ***name mangling***:
     - inside a class C: an attribute **`__x`** becomes **`_C__x`**
     - such an attribute cannot be accessed by **`C.__x`**
     - however, it can be accessed by calling **`C._C__x`**
   - Double leading underscores should be used only to *avoid name conflicts* with attributes in classes designed to be ***subclassed***

- **Remark**: Python does ***not support strong encapsulation*** (it does *not enforce encapsulation*, as C++ or Java), but it ***allows encapsulation*** by ***convention***
  - **Question**: is Python ***less*** object-oriented than C++ or Java?
  - **Hint**: *Smalltalk* (considered as a pure O-O language) uses also a *convention* for encapsulation (as Python, Smalltalk does not enforce encapsulation)
  - **Hint**: ***Data encapsulation*** is about using ***interfaces***
  - *Martin Fowler* says:
    - There is really room for another access type: *PublishedInterface*
    - There is a fundamental *difference* between features *exposed to other classes* within a project team and those *exposed to other teams*
    - The ***distinction*** between *published* and *public* is more important than that between *public* and *private*

- In Python, the syntax for the inheritance relation is slightly different:
  - A ***base class*** is called a *superclass* (while a ***derived class*** is called a *subclass*)
  - ***Superclasses*** are specified between parenthesis:

```
class C(A, B):
    pass
```

- Other difference: the ***inheritance*** is always *public* (in the sense of the C++ language)

Example. The same Polygon and Point classes:

```
# module 'polygon.py'
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    #overloaded __str__ function for printing
    def __str__(self):
        return "({0},{1})".format(self.x, self.y)
```

```
class Polygon:
   def __init__(self):
      self.dimension = 0
      self.area = 0.0
      self.perimeter = 0.0
      self.vertices = []
   def compute_area(self):
      pass
   def coppute_perimeter(self):
      pass
   def add_vertex(self, p):
      self.vertices.append(p)
      self.dimension = self.dimension + 1
      self.area = self.compute_area()
      self.perimeter = self.coppute_perimeter()
   # overloaded [] operator
   def __getitem__(self, item):
      return self.vertices[item]
```

```
# module `usepolygon.py'
from polygon import *
p1 = Point(1, 0)
p2 = Point(2, 0)
p3 = Point(2, 1)
print(p1)
print(p2)
print(p3)
plg = Polygon()
plg.add_vertex(p1)
plg.add_vertex(p2)
plg.add_vertex(p3)
ar = plg.area
pr = plg.perimeter
print(ar, pr)
p = plg[1]
print(p)
```

- According to the *new-style* (available in **3.x**), all classes are inherited (directly or indirectly) from the *root class* of the Python *class hierarchy*, called **`object`**

- For example:

```
    class C:
       pass
```

  is equivalent to:

```
    class C(object):
      pass
```

- The *reason* for using **`object`** as the *root* of the Python class hierarchy:
  - It contains important *special functions* (overloaded operators) that can be inherited by *user classes*:
    - **`__init__`**, **`__del__`**, **`__str__`**, etc.

Python metaclasses

- The term ***metaprogramming*** refers to the potential for a program to have knowledge of or manipulate itself.
  - Python supports a form of metaprogramming for classes called ***metaclasses***

- In Python, everything is an object. Classes are objects as well. As a result, a class must have a type. The ***type*** of a class is ***type***

- ***type*** is a ***metaclass***. Any class in Python **3.x**, is an instance of the ***type*** metaclass

Example:

```
class A:
   pass
for t in int, float, list, A:
   print(type(t))
# <class 'type'>
# <class 'type'>
# <class 'type'>
# <class 'type'>
```

- The type of the ***type*** metaclass is also ***type***

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

- Instances of Python classes are created by using the corresponding metaclass by using two methods of **`type`**:
  - **`__new__()`** for creating the class
  - **`__init__()`** for initializing the class

- Remark: **`__new__()`** and **`__init__()`** are also methods of the class **`object`**

- When a class is created, the interpreter:
  - Gets the name of the class
  - Gets the base classes of the class
  - Gets the metaclass of the class
  - Gets the variables/attributes in the class and stores them as a dictionary
  - Passes this information to metaclass as

```
metaclass(name_of_class, base_classes, attributes_dictionary)
```

    and it returns a class object

- Another method of **`type`** is **`__class__()`**, which is used for creation of object instances

- For the class

```
    class A:
        pass
```

the creation of an object

```
a = A()
```

imply the following actions:

- Calling the method **`__call__()`** of the metaclass of **`A`** (in this case the metaclass **`type`**)
- The **`__call__()`** method in turn invokes the following methods of the parent of the class **`A`** (in this case the class **`object`**):
  - **`__new__()`** for creating the object
  - **`__init__()`** for initializing the object state
- If the class **`A`** does not define **`__new__()`** and **`__init__()`**, default methods are inherited from **`A`**'s ancestry

❑ From the metaclass ***type*** can be derived custom metaclasses

❑ Th **`type`** metaclass is the root of all custom metaclasses, as the **`object`** class is the root of all classes from a Python application

Example:

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        x = super().__new__(cls, name, bases, dct)
        x.attr = 100
        return x
```

- In the above example, the metaclass **`Meta`**:
  - Delegates via **`super()`** to the **`__new__()`** method of the parent metaclass (**`type`**) to actually create a new class
  - Assigns the custom attribute **`attr`** to the class, with a value of 100
  - Returns the newly created class

- **`Meta`** can be used as a metaclass for a class:

```
class B(metaclass=Meta):
    pass

print(B.attr)    # 100
```

- The member functions are usually only ***declared*** in a class declaration. But some simple functions can be both ***declared*** and ***implemented*** inside the declaration part of a class. These functions implemented inside the declaration of a class represent ***inline*** functions.
- The objects of a class can be declared as ***constants***, as other variables in a C program. The C++ language allows for a *constant* object to call only ***constant member functions*** of the class.
- A ***constant member function*** is specified in the class declaration with the keyword **`const`** specified after the function header. Such a function must not modify the member data values of the class from where it belongs.

Example. The *circle* class definition:

```
class circle {
  double xc, yc;
  double r;
public :
  circle(double a, double b, double c)
    { xc = a; yc = b; r = c; }
  double GetXc () const { return xc; }
  double GetYc () const { return yc; }
  double GetR() const { return r; }
  void Translate(double dx, double dy)
    { xc += dx; yc += dy; }
};
```

Circle class utilization:

```
Circle c1(0, 0, 10);
const circle c2(8, 7, 5);
c1.Translate(2, 3);           // correct
c2.Translate(2, 3);           // incorrect
double x = c2.GetXc ();       // correct
```

- However, the **`mutable`** qualifier can be added to a member variable of a class. In this case, this member can be modified even though the member is part of an object declared as **`const`**.

```
class circle {
  double xc, yc;
  mutable double r;
public :
  circle(double a, double b, double c)
    { xc = a ; yc = b ; r = c ;}
  double GetXc () const { return xc; }
  double GetYc () const { return yc; }
  double GetR() const { return r; }
  void SetR(double a) { r = a; }
  void Translate(double dx, double dy)
    { xc += dx ; yc += dy; }

};

const circle c3(0, 0, 2);
c3.SetR(3); // correct
```

Remark. The above example is correct regarding the definition of ***`mutable`*** keyword, but it is wrong regarding to its way of usability.

- There is a difference between ***semantic immutability*** and ***syntactic immutability***.
- The ***semantic immutability*** does not affect the ***externally visible state*** of the object.
- The ***syntactic immutability*** does not affect the ***entire state of the object***, including non-visible values.
- The ***`mutable`*** keyword is used for *semantic immutability*, when some non-visible members can be modified.

- In the previous example, for a constant circle object it does not make sense to change its radius.

- However, there are cases in which some non-public member variables of a constant object should be modified. For example, calculating and caching values once, for a quick access.

Example.

```
class polygon {
  //private members
  int nr_vertices;
  point **vertices;
  mutable double cachedPerimeter{0};
  void ComputePerimeter();
public :
  //public members
  polygon();
  ~polygon();
  double Perimeter() const {
    auto perimeter = cachedPerimeter; // only once
    if (primeter == 0) {
      perimeter = ComputePerimeter();
      cachedPerimeter = perimeter;
    }
    return perimeter;
  }
} ;
```

- In a similar way a C++ program can use ***volatile*** objects and ***volatile*** member functions.
- For example, a class that controls a hardware device by placing appropriate ***values*** in ***hardware registers*** at known ***absolute addresses***.

```
// file devregs.h
struct devregs {
  // control-status-register
  unsigned short volatile csr;
  // data
  unsigned short const volatile data;
  // Busy-wait function to read a byte from device
  unsigned int read_dev() volatile;
  // constructor
  devregs() { csr = 0; data = 0; }
};

// file devregs.cpp
// bit patterns in the control-status-register
#define ERROR   0x1
```

```
#define READY   0x2
#define RESET   0x4

unsigned int devregs::read_dev() volatile {
  while ((csr & (READY | ERROR)) == 0)
    ;  // NULL - wait till done
  if (csr & ERROR){
    csr = RESET;
    return 0xffff;
  }
  return data & 0xff;
}

// file main.cpp
#include devregs.h
void process(void) {
  volatile devregs dvp;
  unsigned int ret;
  ret = dvp->read_dev();   // OK
}
```

- In **Python** there are not *constant* and *variable* objects
- There are instead:
  - ***Immutable objects*** (*numerical* objects, *strings*, and *tuples*)
  - ***Mutable objects*** (*lists*, *dictionaries*, *sets*, and *class instances*)
- All objects in Python have ***three*** different ***features***:
  - A ***type*** (data type)
    - The type can be determined by using the function **`type()`**
  - An ***identity*** (typically the ***memory address*** of the object)
    - The identity can be determined by using the function **`id()`**
  - A ***value***
- The value of an ***immutable*** object ***cannot be modified***
- The value of a ***mutable*** object ***can be modified***

- For **example**:

```
tp = (1, 2, 3)
print(tp)             # (1, 2, 3)
print(id(tp))         # 24735289
tp += (5, 6)
print(tp)             # (1, 2, 3, 5, 6)
print(id(tp))         # 38426517
x = 10
print(x)              # 10
print(id(x))          # 53764251
x = 11
print(x)              # 11
print(id(x))          # 34196577
l = [1, 2, 3]
print(l)              # [1, 2, 3]
print(id(l))          # 64155936
l += [5, 6]
print(l)              # [1, 2, 3, 5, 6]
print(id(l))          # 64155936
```

- A variant for creating a ***constant object*** (similar to C / C++) is to use an ***immutable object***:
  - Not a ***numerical object***
  - But a ***tuple*** with a single component
- For example, a ***mutable point*** class can be defined as follows:

```
class MutPoint:
   def __init__(self, x, y):
      self.x = x
      self.y = y
   def __str__(self):
      return "({0},{1})".format(self.x, self.y)
   def change_point(self, dx, dy):
      self.x = dx
      self.y = dy

p1 = MutPoint(3,3)
p1.change_point(4, 5)
print(p1)
>>> (4, 5)
```

- An ***immutable point*** class can be defined as follows:

```
class ImutPoint:
   def __init__(self, x, y):
      self.x = (x,)
      self.y = (y,)
   def __str__(self):
      return "({0},{1})".format(self.x[0], self.y[0])
   def change_point(self, dx, dy):
      self.x[0] = dx
      self.y[0] = dy
p2 = ImutPoint(1,2)
print(p2)   # >>> (1, 2)
p2.change_point(3, 4) # error
```

- A ***constant object*** of the class **`MutPoint`** can be created in a similar way:

```
p3 = (MutPoint(1,1),)
p3[0] = p1  # error
```

- However, the following sequence is correct:

```
p3[0].change_point(4)  # OK
print(p3[0])
```

- ***Access functions*** represent a group of member functions very used in C++ programs. These are functions, usually defined as inline, which allow to read or to modify the value of the private data members of the classes, where the user does not have direct access.
- The functions reading values are usually called ***accessors***, while the functions writing values are called ***modifiers***.
- There are not predefined rules for naming these functions, but usually the ***accessors*** are prefixed by ***Get***, while ***modifiers*** are prefixed by ***Set***.
- For example, the functions *GetXc*, *GetYc* and *GetR* from the *Circle* class are accessors. A modifier can be defines as::

```
void SetXc(double x) { xc = x; }
```

- ***Another way*** used is that of writing ***overloaded functions***, for ***accessors*** and for ***modifiers***. For example, for member data *xc* of *Circle* class, the following access functions can be defined:

```
void Xc(double x) { xc = x; }
double Xc() const { return xc; }
```

Remark. It is ***not recommended*** that accessors to ***return references***, nor ***not constant pointers at the private data*** of the classes (in this case they allow the direct access to the private data members).

- Because all members of a class are *visible* (public access), in Python there is not necessary to write *accessor* and *modifier* functions (***getters*** and ***setters***)
- However, it is possible to make attributes of a class to be *private* by convention using double leading underscore. In this case, one can write *getters* and *setters* (***not a Pythonic way!***).

```
class PrivAttr:
    def __init__(self,x):
        self.__x = x
    def get_x(self):
        return self.__x
    def set_x(self, x):
        self.__x = x

p1 = PrivAttr(43)
k = p1.get_x()
print(k)          # >>> 43
p1.set_x(44)
n = p1.get_x()
print(n)          # >>> 44
```

- The ***Pythonic way*** to deal with the above problem is to use ***property***
  - a mechanism that provides another way for ***new-style*** classes to define methods called automatically for access or assignment to instance attributes
- A ***property*** is a *type of object* assigned to a *class attribute name*
- A ***property*** can be generated by calling the ***property built-in function***, passing in up to three accessor methods: handlers for ***get***, ***set***, and ***delete*** operations
- Usually properties are related to ***one leading underscore attributes*** (that are by *convention* ***non-public***)

- Example of a class with *property* function:

```
class ProAttr:
    def __init__(self, x):
        self._x = x
    def get_x(self):
        return self._x
    def set_x(self, x):
        self._x = x
   x = property(get_x, set_x)
obj = ProAttr(7)
k = obj.x
print(k)     # >>> 7
obj.x = 9
print(obj.x)   # >>> 9
```

- A second example uses the ***property decorator***
  - A ***decorator*** is simply a function that *takes another function as an argument* and adding to its behavior by wrapping it
  - Syntactically it consists of the `@` symbol, followed by a ***metafunction***: a function that manages another function.

```
    class C:
        @my_decorator  # Function decoration syntax
        def meth():
            pass
```

- Internally, this syntax has the following effect:
  - passing the function through the decorator and assigning the result back to the original name

```
        meth = my_decorator(meth)
```

- The ***@property*** decorator (used as ***getter***):

```
@property
def x(self):
    return self._x
```

Is equivalent to the following code:

```
def x(self):
     return self._x
x = property(x)
```

- Then, on can use the decorator **`@x.setter`** as a setter, which is equivalent to:

```
def x_setter(self, value):
    self._x = value
x = x.setter(x_setter)
```

- In conclusion, using the ***`@property`*** decorator, the previous class can be written as follows

```
class ProDec:
    def __init__(self, x):
        self._x = x
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
obj1 = ProDec(5)
k = obj1.x
print(k)         # >>> 5
obj1.x = 6
print(obj1.x)    # >>> 6
```

B. Class implementation. The resolution operator

- For a complete class definition all member functions from the respective class declarations that are not ***inline*** must be implemented.
  - Usually the implementation of a ***non-inline*** function is made in a ***distinct*** source file.

- The C++ language has a new operator called ***resolution operator*** (denoted **::**), in order to be able to specify in the case of every member function the class where it belongs:

```
<class name> :: < function name>
```

Example. For example, the function *AddVertex* from the *polygon* class can be defined as follows:

```
void polygon::AddVertex(point* p) {
  point **v = new point *[nr_vertices+1];
  for(int i=0 ; i< nr_vertices ; i++)
    v[i] = vertices[i];
  v[nr_vertices++] = p;
  delete[] vertices;    //free memory for vertices
  vertices = v;
  Compute Perimeter();
  AdjustArea();
}
```

- The resolution operator can be used in this case as a ***unary operator*** which prefixes a name. In this case it refers the most outside appearance of the respective name, declared at the file level.

- A usual utilization is the reference of a name which is hidden in a block.

- There is no *resolution operator* in ***Python*** because all methods are written inside the class

Example: Using the unary form of the resolution operator:

```
int k;

f1() {
  // k is visible in this block
}

f2() {
  // k defined at the file level
  // it is not visible in f2
  int k = 0;
  k = k+2;          // using k at the block level
  ::k = ::k+2;      // using k at the file level
}
```

- The class ***constructors*** and ***destructors*** are member functions of the class.
  - They are special functions that ***do not have*** any ***returned value*** (not even **`void`**).
  - Moreover, the destructors ***cannot have arguments***.

Example:

```
polygon::polygon() {     // constructor
   vertices = 0;
   nr_vertices = 0;
   area = perimeter = 0;
}

polygon::~polygon() {    // destructor
   delete[] vertices;
}
```

- In ***Python***, the special function **`__init__`** is used as a *constructor* for a class
- ***Python*** uses *references* (as Java) and a *garbage-collector* mechanism
- However, the special function **`__del__`** (from the **`object`** root class in most cases) is called when an object is destroyed (the reference count become zero)
- This special function **`__del__`** in Python is equivalent to a *destructor* in C++:
  - This function can be overloaded for a user class, if necessary

- Another special category of member function are the ***operators***. The C++ language allows ***overloading*** the common ***operators*** of the language in order to be able to define other operations.

- The specification of an operator as a function is done with the help of the keyword **`operator`** used as prefix of the respective operator.

Example. In the class *polygon* it is overloaded the indexing operator:

```
point* polygon::operator[](int k)    {
  if (k < 0)  {
    cout << “\n Negativ index”;
    return 0;
  }
  return vertices[k];
}
```

- In ***Python*** only *special built-in functions* can be *overloaded*

C. Using classes

- ***Using of a class*** means the ***creation*** of some objects of these classes and the ***communication*** with the respective objects with the help of ***messages***, that is calling their member functions.

- The ***creation of an object*** involves two distinct operations, regardless of the allocation zone:
  - ***The allocation of a memory zone*** having an appropriate dimension;
  - ***The call of a constructor function*** of the class where the object belongs to, in order to initialize the member data with initial values.

- The ***dimension of the memory zone*** allocated for an instance object is, in general, given by the ***sum*** of the dimensions of its ***data members***, but this dimension ***depends on the implementation***.

- There are situations when the memory zone dimension of an object is greater than this sum. For example:
    - in the case of polymorphism,
    - in the case of some classes which do not contain only member functions.

Example. For a Visual C++ compiler, the following sequence displays the values 4 and 1:

```
#include <iostream>
using namespace std;

struct A {
  int n;
  A(int k) { n = k; }
  int N() { return n; }
};

struct B {
  void Print() { cout << "B"; }
};

int main() {
  A a;
  B b;
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
  retrun 0;
}
```

- Although the ***memory zone for the member functions*** is different from the ***memory zones of the object instances***, the calling of an object member function is strictly related with the memory zone of respective object, by passing of a ***hidden parameter***, which refers the memory address of its associated zone.

- **Example**. Defining a class *p_poligon*, which is be able to use more polygons stored into a doubled linked list. At the *polygon* class must be added 2 new members having the pointer type:

```
class p_polygon {
  int nr_vertices;
  point **vertices;
  double area, perimeter;
  p_polygon *succ, *pred;
  void ComputePerimeter();
  void AdjustArea();
public :
  p_polygon() {
    vertices = 0;
    nr_vertices = 0;
    area = perimeter = 0;
    succ = pred = nullptr;
  }
  ~p_polygon() ;
  int NrVertices() const { return nr_vertices; }
  void AddVertices (point*);
  point* operator[](int);
```

```
    double Area() const { return area; }
    double Perimeter() const { return perimeter; }
    p_polygon* Pred() const { return pred; }
    p_polygon* Succ() const { return succ; }
    void AddPolygon(p_polygon*);
  } ;
```

- The implementation of the functions of the class *p_polygon* is the same as in the case of the class *polygon*, with the exception of the new added one. The *AddPolygon* function adds a new polygon in the list, as successor of the current polygon:

```
void p_polygon::AddPolygon(p_polygon* p) {
  p->succ = succ;
  p->pred = this;
  succ->pred = p;
  succ = p;
}
```

- The keyword, **`this`** is called the ***self pointer*** and it points always on the current object.

- In the case of a class *X*, the parameter

  `X* this`

  is passed in all the non-static member functions of the *X* class.

- The using of the ***hidden parameter*** this is ***not absolutely necessary***, only in the case when an explicitly reference at the memory address of the current object is made. For example, the function *AddPolygon* can be also written:

```
void p_poligon::AddPoligon(p_poligon* p) {
  p->succ = this->succ;
  p->pred = this;
  this->succ->pred = p;
  this->succ = p;
}
```

- In Python the parameter ***self*** is no hidden in methods; it is mandatory

- The same example as in C++

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y
   def __str__(self):
      return "({0},{1})".format(self.x, self.y)

class LPolygon:
    def __init__(self):
        self.vertices = []
        self.area = 0.0
        self.perimeter = 0.0
        self.succ = None
        self.pred = None
    def compute_perimeter(self):
        pass
    def adjust_area(self):
        pass
```

```
def nr_vertices(self):
    return len(vertices)
def add_vertex(self, point):
    pass
def __getitem__(self, item):
    pass
def add_polygon(self, polygon):
    polygon.succ = self.succ
    polygon.pred = self
    self.succ.pred = polygon
    self.succ = polygon
```

D. Static members of a class

- ***Every object*** of a class has usually a ***copy of the member data of the class*** to which it belongs. For this reason, any modification of the value of an object member is local to the respective instance and it is not visible in other instances of the same class.

- The C++ language allows, in addition, the possibility of defining ***members*** having ***values that can be used in common by all other class instances***. These members are called ***static*** members and they are declared with by using the keyword **`static`**.

Example. Let us consider a class *Experiment*, which allows the description of the observations on a physic measure. Each class object stores a measured value of the physic measure. The *Experiment* class must determine the observations number at a certain time, and also the average of the observed values.

```
// experiment.h file
class Experiment {
  double x;
  static int n;
  static double s;
public:
  Experiment(double);
  double X() const { return x; }
  static double Med() { return s/n; }
  static int N() { return n; }
};
```

- Data members having ***static*** type are common for all class objects and they have allocated a memory zone that is different from the zone of non-static data. In this way it can be realized the simple and efficient communication between different objects belonging to the same class.

- The effective definition (***memory allocation*** and ***initialization*** with values) of the ***static*** member data must be realized ***outside*** of the class declaration and in a single place in the program.
    - Usually the definition of the static member date is realized in the file which contains the implementation of the class, avoiding in this way multiple definitions.

- For the previous example, in the implementation file of the experiment class must be added the following definitions:

```
// experiment.cpp file
int Experiment::n = 0;
double Experiment::s = 0;

Experiment::Experiment(double v) {
  x = v;
  n++;
  s += v;
}
```

- ***Static data members*** of a class can be **`const`** (**`static const`**)
  - In this case, **`static const`** variables can be ***initialized inside of the class*** declaration (they are still **`static`**):

```
class X {
    static const int m = 7;
    // …
}
```

- The using of static member data lead to a better structuring of information in a program, because these values are global only for class objects where they have been declared.

- ***Static functions*** cannot use the hidden parameter **`this`**. It follows that the ***static member functions*** can ***have access only at the static members*** of the respective class (data or functions). In the previous example, the *Med* function cannot modify the *x* member value and it cannot call other member functions of the class (*X* for example).

- The ***public static members*** can be accessed ***directly***, without the using an object instance.

- For example, a file using the experiment class could be the following (the following sequence of values is considered: 0.5, 1.5, … , 9.5):

```
//main.cpp file
int main() {
  for (int i=0; i<10; i++)
    Experiment e(i+0.5);
  int n = Experiment::N();
  double m = Experiment::Med();
  cout<<"n = "<<n<<endl<<"Med = "<<m<<endl;
  return 0;
}
```

- ***One way*** that a ***static member function*** can ***have access*** at the ***non-static member of the class*** to which it belongs, is by ***passing as parameter*** in the ***static function*** of an ***object*** of the class.

Example. The class *Folder* stores the path for a current folder and a predefined path, unique for all class objects. The static function *preset* allows setting the current path for a folder which is passed as parameter.

```
class Folder {
public:
  static void setpath(char const *newpath);
  static void preset(Folder &dir, char const *path);
private:
  string Currentpath;
  static char path[];
};

char Folder::path[200] = "C:\\";

void Folder::setpath(char const * newpath) {
  strcpy(path, newpath);
}

void Folder::preset(Folder &dir, char const * newpath) {
  dir.Currentpath = newpath;
}
```

```
int main() {
   Folder dir;
   Folder::setpath("D:\\");
   dir.setpath("D:\\");
   Folder::setpath (dir, "D:\\OOP");
   dir.setpath (dir, "D:\\OOP");
   return 0;
}
```

Remarks.

1. The ***non-static member functions*** can refer ***static members*** of the respective class (for example, the case of the constructor of *Experiment* class).
2. A *static member function* which is ***private***, ***cannot be called by means of a class object***.
3. If a member function is declared *static* in a class, but it not defined as *inline*, the effective *definition* of this function *does not contain* the word **`static`**.

For **example**:

```
class A {
 // ...
 static int x;
 static void SetX(int);
 // ...
} ;

void A::SetX(int k) {
  x = k;
}
```

- ***Static members*** in ***Python*** can be:
  - *Static data members* in C++ → *Static attributes* (*class variables*)
  - *Static function members* in C++ →
    - *Class methods*
    - *Static methods*

- ***Class variables*** are those attributes defined *outside* of the **`__init__`** function:
  - They are shared by all class instances

- For example:

```
class C:
    a = 1               # class variable
    def __init__(self, b_):
        self.b = b_     # instance variable
```

- In Python there are three types of *methods*:
  - ***Instance methods*** (they have **`self`** as first argument)
  - ***Class methods*** (they have **`cls`** as first argument)
  - ***Static methods*** (they do not have **`self`**, neither **`cls`**)

- Both *class methods* and *static methods* can use *class variables*
  - A class method by using the **`cls`** argument
  - A static method by using the name of the class
- Both types of methods can be defined by using Python ***decorators***
- Generally, ***static methods***:
  - are used to group functions which have some logical connection with a class
  - *cannot modify class variables* (neither instance variables)
  - *cannot be used when subclassing* (inheritance)
- Example of using instance, class, and static methods of a Python class:

```
class Experiment:
    n = 0
    s = 0.0

    def __init__(self, x_):
        self.x = x_
        Experiment.n = Experiment.n + 1
        Experiment.s = Experiment.s + self.x
    @classmethod
    def med(cls):
        return cls.s / cls.n
    @staticmethod
    def getn():
        return Experiment.n

def test()
    for i in range(10):
        e = Experiment(i+0.5)
    n = Experiment.getn()
    m = Experiment.med()
    print(n, m)
```

Remark. Both ***class methods*** and ***static methods*** can ***be called by using an instance*** of the corresponding class

- In C++ this is not possible

```
    e = Experiment(8)
    n = e.N()
    m = e.Med()
```

- ***Class methods***:
  - Can *modify class variables* (but not *instance variables*)
  - Can be used in *subclassing*
  - Usually, they can be used *to create factory methods* (because Python does not support method overloading)

Example:

```
from datetime import date

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

    # Create a Person object by birth year (like a constructor)
    @classmethod
    def createFromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # Static method to check if a Person is adult or not
    @staticmethod
    def isAdult(age):
        return age >= 18
```

```
person1 = Person('John', 22)
person2 = Person.createFromBirthYear('Mary', 1999)

print (person1.age)              # 22
print (person2.age)              # 20
print (Person.isAdult(22))       # True
```