Programming paradigms

A ***programming paradigm*** (a ***programming style***) is a method to conceptualize the way:

- of execution the calculations within a computer,
- of structuring and organizing the tasks responsible with these calculations

A programming language:

- ***offers support*** for a programming style if the programming language allows enough facilities that make it useful in this style
- ***allows*** only to use a programming style if the needed effort to write a program in this style is greater, the programming language does not offer enough facilities

A. Procedural programming

- It is ***one of the oldest and most used*** paradigms
- This paradigm implies the following steps:
  - a) the ***decomposition*** of the problem to be solved in smaller problems
  - b) ***finding*** for each small problem ***an optimal algorithm***
  - c) ***implementing of each algorithm*** by using functions and procedures of an appropriate programming language

Example. Determining if an integer is a prime number:

A) In C:

```
int Prime(int n) {
   int i;
    for (i=2; i<n; ++i)
       if (n%i == 0)
          return 0;
    return 1;
}
```

```
void PrimeFactors(int n) {
   int i;
   for (i=2 ; i<n/2 ; i++) {
   if (n%i == 0 && Prime(i)) {
      printf("%d\n", i);
     }
   }
}
```

B) In Python:

```
def prime(n):
    for i in range(2, n - 1):
        if n % i == 0:
            return False
    return True

def prime_factors(n):
    for i in range(2, n // 2):
        if n % i == 0 and prime(i):
            print(i)
```

- In Python functions are a powerful mechanism
- ***Nested functions***: functions defined in the scope of another functions

```
def outer(num1):
    def inner_increment(num1):
        return num1 + 1
    num2 = inner_increment(num1)
    print(num1, num2)

outer(10)
```

- An outer function can return an inner function

```
def fib(n):
    def f_rec():
        return fib(n-1) + fib(n-2)
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return f_rec()
for k in range(10):
    fib(k)
```

- A ***factory function***: can create several functions (a ***design pattern***)

```
def create_adder(x):
   def _adder(y):
       return x + y
   return _adder
add2 = create_adder(2)
add100 = create_adder(100)
```

```
print(add2(50))
print(add100(50))

>>> 52
>>> 150
```

- ***Function decorators***: wrappers to existing functions (a ***design pattern***)

```
def make_bold(fn):
    def wrapper():
        return "<b>" + fn() + "</b>"
    return wrapper

def get_text():
    return "hello"

bold_text = make_bold(get_text)
```

- Python's Decorator Syntax:

```
def make_bold(fn):
    def wrapper():
        return "<b>" + fn() + "</b>"
    return wrapper

@make_bold
def get_text():
    return "hello"

get_text()
```

B. Data encapsulation (modularization)

- The accent in procedural programming has moved from the ***function design*** to the ***data organization***.
  - ***Data*** are not regarded in isolation; they are regarded together with the functions that they process.
- In this paradigm the notion of ***module*** was defined as representing a *set of related functions*, together with *data processed* by these functions.
- A ***module*** contains:
  - An ***interface***, where data and functions accessible outside of the module are declared;
  - An ***implementation***, which is inaccessible outside to the module, where functions manipulating data from the module are defined.

- The C language ***allows*** only data encapsulation:
  - The *interface* part is usually specified in a header file that must be included in all the others files of a program that use the module functions;
  - The *implementation* part of the module is realized in a distinct file which must be included in the program project.

Example. Defining and using a module that allows the operations with integers:

```
//file ‘sequence.h’: the interface
#define max_dim 100
void Init() ;
int Summ() ;
void Sort() ;
void AddElement(int) ;
void Print() ;

 //file ‘sequence.c’: the implementation
#include “sequence.h“
static int dim ;
static int v[max_dim] ;
void Init() { dim = 0 ; }
void AddElement(int k) {
  v[dim++] = k ;
}
```

```
int Summ() {
  /* the code for the sum determination */
}
void Sort() {
  /* the code for sorting */
}
int Print() {
  /* the code for printing */
}

//file 'pr.c': using the module 'sequence'
#include "sequence.h"
void Processing() {
  int i, s, k, n = 0 ;
  Init() ;
  for(i=0 ; i<n ; i++) {
    scanf("%d", &k) ;
    AddElement(k) ;
  }
  s = Sum() ;
  printf("\nSum=%d", s) ;
  Sort() ;
  Print() ;
}
```

The Python language uses ***modularization*** in a different way:

- A ***module*** represents package of ***variable names*** and ***objects***, known as a ***namespace***
- A module is usually a Python file, the highest-level ***program organization unit***
- The names within a module are called ***attributes***
- An ***attribute*** is a variable name that is attached to a specific object

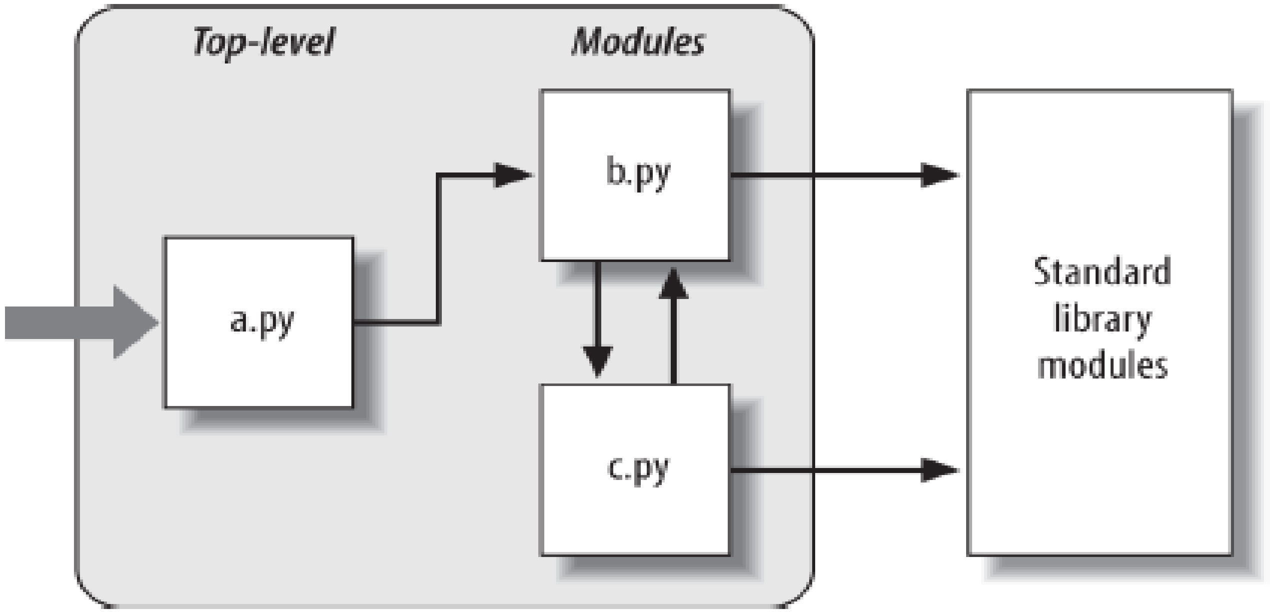
❑ Modules are processed with two statements:

- ***import***: allows a module client (importer) to fetch another module ***as a whole***
- ***from***: allows modules to fetch ***particular names*** from another module

❑ Import operations ***load*** a Python file and grant access to its contents

- The contents of a module are made available to the outside world through its attributes

- This module-based model represents the core idea behind ***program architecture***
  - Larger programs usually take the form of multiple module files, which import tools from other module files.
  - One of the modules is designated as the ***main*** or ***top-level*** file: the file launched to start the entire program



- There is a difference between a C **`#include`** macro and a Python **`import`** statement:
  - ***`import`*** is a runtime operation that performs three distinct steps:
    1. ***Find*** the module's file
    2. ***Compile*** it to byte code
    3. ***Run*** the module's code to build the objects it defines
  - All the above steps are carried out only the ***first time*** a module is imported during a program's execution

Example. Defining and importing a module:

```
# file m1.py
def my_print (x): # my_print is a module attribute
   print(x)

# file m2.py
import m1
m1.my_print('Hello world!')
```

When the file *m2.py* is loaded, it prints the message:

```
Hello world!
```

Remark. *m1* is an object, and *my_print* is an attribute (qualification is required)

- The ***from*** statement copies specific names from one module into another scope
  - It allows us to use the copied names directly, without the name of the imported module

Example. Using the from statement:

```
# file m3.py
from module m1 import my_print
my_print('Hello world!')
```

When the file *m2.py* is loaded, it prints the same message

- A special form of ***from***:
  - `from <module> import *`
  - it imports of all names assigned at the top level of the referenced module

- A Python module exports ***all*** the names assigned at the ***top level*** of its file
  - There is no way to prevent a client from changing names inside an imported module

- In Python, ***data hiding*** in modules is only a ***convention***, not a syntactical ***constraint***
  - ***Encapsulation*** in Python is more about ***packaging*** than about ***restricting***

- As a ***special case***, the names can be ***prefixed*** with a single underscore to ***prevent*** them from being ***copied out***, when a client imports a module's names with a ***`from *`*** statement

- Unfortunately, underscores are not "***private***" declarations:
  - an importer module can still ***see*** and ***change*** such names with other import forms, such as the ***`import`*** statement:

```
# md.py
a, _b, c, _d = 1, 2, 3, 4

# md1.py
from md import *
print(_b)
```

```
print(a, c)
```

```
(1, 3)
NameError: name '_b' is not defined
```

```
# md2.py
import md
print(md._b)
```

```
2
```

Example. The previous example using Python modules:

```
# module seq.py
import sys

L = []

def init_seq():
    L.clear()

def add_elem(a):
    L.append(a)

def sum():
    s = 0
    for a in L:
        s = s + a
    return s

def sort_seq():
    L.sort()

def print_seq():
    print(L)
```

```
# module proc.py
from seq import *

def seq_proc():
    init_seq()
    text_len = input('seq_len=')
    n = int(text_len)
    for i in range(n):
        text_elem = input('elem=')
        a = int(text_elem)
        add_elem(a)
    s = sum()
    print('sum = ', s)
    print_seq()
    sort_seq()
    print_seq()

seq_proc()
```

C. Object-Based Programming

- ***Object-based programming*** is a programming paradigm that use the notions of ***encapsulation*** and ***objects*** with ***operations***.
- ***Encapsulation*** is related to the notion of ***abstract data types***
  - ***Abstract data types*** are the basis of object-based programming
- An ***abstract data type*** (ADT) represents:
  - A set of operations that can be performed on the set of its elements (the ***interface***, which is accessible from outside);
  - A set of ***axioms***, which represents the way to describe the properties of the elements and of their operations;
  - A set of ***preconditions*** and ***postconditions*** that specify conditions in which each operation can be called, and the state of the system after the calling of each operation respectively
- The implementation of an ADT into a programming language represents a ***data type***.

- The implementation of a user data type is realized through the notion of ***class***.
- A ***class*** describe the common structure of a set of ***objects***
- All objects described by a class A are called ***instances*** of this class.
- Each ***object*** has its own ***state***: e.g. values for each ***component*** of the object
- Examples of ***object-based languages*** (that are not ***object-oriented***):
  - Early versions of ***Ada***
  - ***Visual Basic*** (before .NET)
  - ***Fortran 90***

- Sometimes, the term ***object-based*** is applied to ***prototype-based*** languages:
  - are partially ***object-oriented*** languages that ***do not have classes***,
  - in which ***objects inherit*** their code and data directly from other ***template objects***
- An example of a commonly used ***prototype-based*** language is ***JavaScript***

- In ***JavaScript***, any object has a ***prototype***, including functions
- The ***prototype*** is a simple way of ***adding object members*** to any newly created instance of the whole object

Example:

```
var constructor = function() { };
constructor.prototype.text = "hello world";
alert(new constructor().text); // This alerts hello world
```

- The C++ language allows the programmers to define user data type by using ***classes*** and ***operator overloading***.
- In C++, a ***class*** can be regarded as an extension of the ***structure*** of the C language, which allows to define inside of the class both data and functions using data.

Example. The definition of a data type representing the *rational numbers* (*fractions*):

```
//the file 'fraction.h': the interface part of the class
struct fraction {
  /* the numerator and the denominator */
  int p, q;
  /* constructor */
  fraction (int _p = 0, int _q = 1);
  /* operations */
  fraction Sum(fraction);
  fraction Mult(fraction);
  fraction Div(fraction);
};

//the file 'fraction.cpp': the implementation part of the class
#include "fraction.h"
fraction::fraction (int _p = 0, int _q = 1) {
  p = _p;
  q = _q;
}
fraction fraction::Sum(fraction f) {
  p = p * f.q + f.p * q;
```

```
  q = q * f.q;
  return *this;
}
fraction fraction::Mult(fraction f) {
  p = p * f.p;
  q = q * f.q;
  return *this;
}
fraction fraction::Div (fraction f) {
  p = p / f.q;
  q = q / f.p;
  return f;
}

//the file 'pr.cpp': using the class fraction
#include "fraction.h"
void Processing() {
   fraction f1(1, 2), f2(7, 4), f3;
   f3 = f1.Mult(f2).Sum(f1.Div(f2));
   printf("\nf3 = %d/%d", f3.p, f3.q);
}
```

Example. The above example of the rational numbers, where functions are defined as *overloaded operators*:

```
//the file ‘frac.h’ - definition of the fraction class
class fraction {
   int p, q;
public :
   //the constructor
   fraction(int a = 0, int b = 1) {
      p = a;
      q = b;
   }
   //declaration of the overloaded operators
   friend operator + (fraction, fraction);
   friend operator * (fraction, fraction);
   friend operator / (fraction, fraction);
};
```

```
//the file 'frac.cpp'- overloaded operators are implemented
#include "frac.h"
fraction operator+(fraction f1, fraction f2) {
  fraction f;
  f.p = f1.p * f2.q + f2.p * f1.q;
  f.q = f1.q * f2.q;
  return f;
}
fraction operator*(fraction f1, fraction f2) {
  fraction f;
  f.p = f1.p * f2.p;
  f.q = f1.q * f2.q;
  return f;
}

fraction operator/(fraction f1, fraction f2) {
  fraction f;
  f.p = f1.p / f2.q;
  f.q = f1.q / f2.p;
  return f;
}
```

```
//the file ‘pr.cpp’ - using the class fraction
#include “frac.h”
void Processing() {
  fraction f1(1, 2), f2(7, 4), f3;
  f3 = f1*f2 + f1/f2;
  // ...
}
```

Example. The previous example using Python classes:

```
# module frac.py
class Fraction:
    def __init__(self, a=0, b=1):
        self.p = a
        self.q = b
    def __add__(self, f):
        a = self.p * f.q + f.p * self.q
        b = self.q * f.q
        return Fraction(a, b)
    def __mul__(self, f):
        a = self.p * f.p
        b = self.q * f.q
        return Fraction(a, b)
    def __truediv__(self, f):
        a = self.p / f.p
        b = self.q / f.q
        return Fraction(a, b)
    def __str__(self):
        return "({0}/{1})".format(self.p, self.q)
```

```python
# module usingfrac.py
from frac import *

def processing():
    f1 = Fraction(1, 2)
    f2 = Fraction(7, 4)
    f3 = f1*f2 + f1/f2
    print(f3)

processing()
```

- Functions whose name contains double leading and trailing underscore (**__**) are called ***special functions*** in Python
- The ***`__init__()`*** function from a class represent its ***constructor*** that initializes data members (attributes) of a ***class instance***
  - Because all methods of a class must have ***self*** (representing the current instance of the class) as the first parameter,
    - All ***local*** attributes must be initialized in ***`__init__()`***:

```
def __init__(self, a=0, b=1):
    self.p = a
    self.q = b
```

  - There is no special section in a class for defining its attributes
    - All ***local*** attributes of class instances must be defined in the ***`__init__()`*** function
    - Attributes defined outside of class methods are ***global*** attributes of that class
- The special ***`__str__()`*** method specifies how values of the class' attributes will be printed

- It uses the ***format()*** method of the ***string*** built-in type in order to describe data conversion

```
def __str__(self):
    return "({0},{1})".format(self.p, self.q)
```

- Some special functions representing ***operator overloading***:

Operator	***Expression***	***Meaning***
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`

Bitwise XOR	**`p1 ^ p2`**	**`p1.__xor__(p2)`**
Bitwise NOT	**`~p1`**	**`p1.__invert__()`**
Indexing	**`p1[p2]`**	**`P1.__getitem__(p2)`**

- One important feature of data abstraction represented by ***data parametrization***. The C++ language *provide support* for parametrization by using the **`template`** mechanism.

- The problem of parametrization appears when the programmer wants to define some ***generic data types***, where the data type of the components is unspecified.

Example. The definition in the C++ language of a *vector* class, where the data type of the components is generic.

```
template<class T>
class vector {
  T *v;       // the array whose components
              // have the generic data type T
  int dim;    // dimension of the array
public:
  // the constructor
  vector(int n) {
    if (n > 0)
      v = new T[dim = n];
  }
```

```
    // the indexing operator
    T& operator[](int k) { return v[k]; }
    int Dimension() { return dim; }
  };
```

- An array object can be created by instantiating the generic class:

```
// an array with 20 integer components
vector<int> v1(20);
// an array with 10 real components
vector<double> v2(10);
v1[7] = 5;
v2[7] = 2.3;
```

- Python generally do not use ***generic data types***
  - Python is not a ***statically typed language***
  - It is instead a ***dynamically non-typed language***
- ***Generic data types*** can be viewed by using a ***convention***:
  - that define a ***contract*** for a class
  - and ***use*** this contract for creating class instances

Example:

```
# 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('aaa')          #add only strings
stack2.push('bbb')          #add only strings
y = stack2.pop()            #a homogeneous container
print(x)
stack3 = MyStack()          #contract is not respected
stack3.push('aaa')          #a heterogeneous container
stack3.push(27)
z = stack3.pop()
print(x)
```

- However, starting with the version 3.5, the Python distribution contains a module called ***typing***, which defines the fundamental building blocks for the usage of ***static type checking***

- Among others, this module contains two important elements: ***TypeVar*** and ***Generic***

- In ***typing***, a theory of types is developed, which support statically type checking:
  - ***Type variables*** can be defined by using a factory function, ***TypeVar()***:

```
T = TypeVar('T')  # T is a type variable
```

  - A type variable can be instantiated with an existing type:

```
def do_sum(a: T, b: T):
   return a + b
x = do_sum(2, 5)              # T is an int
print(x)                      # 7
y = do_sum('abc', 'xyz')      # T is a string
print(y)                      # 'abcxyz'
```

- ***Generic classes*** can be constructed by inheriting from a generic base class, defined in ***`Generic`***:

```
class Stack(Generic[T]):
    pass
```

Example. Defining and using a generic stack class.

```
# module generic.py
from typing import TypeVar, Generic

T = TypeVar('T')     # Declare a type variable

class Stack(Generic[T]):
    # An empty list with items of type T
    def __init__(self) -> None:
        self.items = []
    def push(self, item: T) -> None:
        self.items.append(item)
    def pop(self) -> T:
        return self.items.pop()
    def empty(self) -> bool:
        return not self.items
```

```python
# module usegen.py
from generic import *

stack1 = Stack[int]()      # instantiation: T->int
stack1.push(2)             # OK
stack1.push(3)             # OK
x = stack1.pop()
print(x)                   # 3
stack1.push(‘abc’)         # Type error

stack2 = Stack[str]()      # instantiation: T->str
stack2.push('aa')          # OK
stack2.push('bb')          # OK
y = stack2.pop()
print(y)                   # ‘bb’
stack1.push(5)             # Type error
```

D. Object-oriented programming (OOP)

- The notion of ***class***:
  - is specific to the object-based programming paradigm (not to OOP);
  - is a basic element in the OOP paradigm.

- ***Properties*** (***components***) of a class can be described by:
  - data (***attributs***),
  - functions (***methods***).

- The instances of a class are called ***objects***. An object is uniquely identified by its name.

- For example, the definition of three instances of the class *fraction*:

```
fraction f1(1, 2), f2(7, 4), f3;
```

- An instance of a class defines the ***state*** of its corresponding object, which it is represented by the current values of the attributes of the object at a certain moment.

- A ***method*** describes the way an object reacts when it receives a certain message from another object. A ***message*** is a request of a certain object to the current object to invoke a specific method of the current object.

- In the C++ language ***sending a message*** to an object means the calling of a specific method of that object.

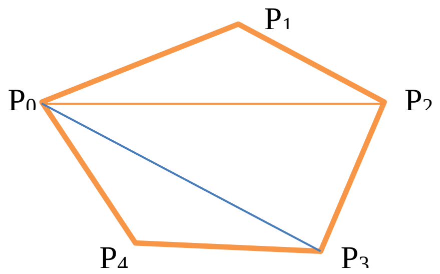
  - **<u>A first essential element</u>** of **OOP** is *to allow the difference between the general and the particular properties of objects*.

  - It follows an important property of object-oriented languages: they allow the ***partition*** of objects into classes, and also a mechanism for ***inheriting*** the properties of a class into another class.

  - In this way classes can form ***hierarchies of classes*** based on the inheritance mechanism

Example. Let us consider the following *polygonal figures* in a plane: triangles, quadrilaterals, pentagons, etc., each polygon being described by the coordinates of its vertexes in trigonometric order.

- One can define a class hierarchy, which has the *polygon* class as the root, the others classes inheriting the class *polygon*.
- Let us suppose that for the *polygon* class there are specified the coordinates for the first two vertices of the polygon, $P_0(x_0,y_0)$ and $P_1(x_1,y_1)$, the other classes having to store one after another the coordinates of the next vertex.




```
class Polygon {
  // ...
  public:
    Polygon(_x0, _y0, _x1, _y1);
    virtual ~Polygon();
    virtual double Perimeter();
  protected:
    double x0, y0, x1, y1;
    virtual double TwoEdges() = 0;
    virtual double OneEdge() = 0;
    // ...
} ;
```

```
class Triangle : public Polygon {
  // ...
  public:
    Triangle(_x0, _y0, _x1, _y1, _x2, _y2);
    double Perimeter();
  protected:
    double x2, y2;
    double TwoEdges ();
    double OneEdge();
    // ...
} ;

class Qadrilateral : public Triangle {
  // ...
  public:
    Qadrilateral(_x0, _y0, _x1, _y1, _x2, _y2, _x3, _y3);
    double Perimeter();
  protected:
    double x3, y3;
    double TwoEdges ();
    double OneEdge ();
    // ...
 };
```

- Some function implementations:

```
Polygon::Polygon(_x0, _y0, _x1, _y1):
  : x0(_x0), x1(_x1), y0(_y0), y1(_y1) {
}
double Polygon::Perimeter() {
  double l ;
  l = sqrt((x0-x1)*(x0-x1) + (y0-y1)*(y0-y1));
  return l ;
}
Triangle:: Triangle (_x0, _y0, _x1, _y1, _x2, _y2):
  : Polygon(_x0, _y0, _x1, _y1), x2(_x2), y2(_y2) {
}
double Triangle::TwoEdges() {
  double a, b ;
  a = sqrt((x0-x2)*(x0-x2) + (y0-y2)*(y0-y2));
  b = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
  return a + b;
}
```

```
double Triangle:: OneEdge () {
    double l;
    l= sqrt((x0-x1*(x0-x1) + (y0-y1)*(y0-y1));
    return l;
}
double Triangle::Perimeter() {
  double p ;
  p = Polygon::Perimeter() + TwoEdges();
  return p ;
}
```

- Because the functions *Perimeter*, *OneEdge* and *TwoEdges* are common to all classes of the hierarchy, but their implementations are specific to each class, these functions represents ***virtual functions***. Moreover, the functions *TwoEdges* and *OneEdge* cannot be implemented in the class *Polygon*, these functions representing ***pure virtual functions*** in this class.

- In the case of virtual functions, the selection of the effective function that will be called at a certain moment is automatically realized by the compiler.

- In conclusion, **the second essential element** of the OOP programming consists of the ***mechanism of polymorphism*** (in C++ are used ***virtual functions***), where the calling of a member function of an object depends of the type of that object.

- For example, in the case of the polygonal figures, the following declarations and statements:

```
Polygon* f1 = new Triangle(0,0,0,1,1,0);
Polygon* f2 = new Quadrilater(0,0,0,1,1,0,1,1);
double p1 = f1->Perimeter() ;
double p2 = f2->Perimeter() ;
```

  allow the correct selection of the *Perimeter* function by the compiler, for each object.

- ***A similar example in Python***

```
# module 'polygons.py'
from math import sqrt, inf, fabs, pi
# class Polygon is abstract because the method area_calc
# is not defined
class Polygon:
   def area(self):
      return self.area_calc()
class Point(Polygon):
   def __init__(self, x, y):
      self.x = x
      self.y = y
   # length of the segment [self, p]
   def segment(self, p):
      return sqrt((self.x - p.x) * (self.x - p.x) +
         (self.y - p.y) * (self.y - p.y))
   # the slope of the segment [self, p] with the Ox axis
   def slope(self, p):
      if p.x - self.x == 0:
         return None
      else:
         return (p.y - self.y) / (p.x - self.x)
```

```python
    def area_calc(self):
        return 0

# a segment has two points: Point(x0, y0), and p1(x1, y1)
class Segment(Point):
    def __init__(self, x0, y0, x1, y1):
        Point.__init__(self, x0, y0)  # the first point
        self.p1 = Point(x1, y1)       # the second point
        # the slope of the segment [Point, p1] with the Ox axis
        self.slope = Point(x0, y0).slope(self.p1)
        # the length of the segment [Point, p1]
        self.length = Point(x0, y0).segment(self.p1)
    # the segment [Point, p1] is perpendicular on Ox axix
    def is_perpend(self, d):
        if (self.slope != None) and (d.slope != None)
              and (self.slope * d.slope == -1):
            return True
        elif (self.slope == None) and (d.slope == 0.0):
            return True
        elif (self.slope == 0.0) and (d.slope == None):
            return True
        else:
            return False
```

```
    def area_calc(self):
        return 0
# a triangle has 3 segments having the length l1, l2, l3
class Triangle(Polygon):
    def __init__(self, x0, y0, x1, y1, x2, y2):
        self.l1 = Point(x0, y0).segment(Point(x1, y1))
        self.l2 = Point(x1, y1).segment(Point(x2, y2))
        self.l3 = Point(x2, y2).segment(Point(x0, y0))
    # using the Heron's formula
    def area_calc(self):
        p = (self.l1 + self.l2 + self.l3) / 2
        return sqrt(p * (p-self.l1) * (p-self.l2) *
            (p-self.l3))
# a rectangle has 4 points and length
# of 2 distinct perpendicular segments
class Rectangle(Polygon):
    def __init__(self, x0, y0, x1, y1, x2, y2, x3, y3):
        # the list of points
        self.rp = [Point(x0, y0), Point(x1, y1),
              Point(x2, y2), Point(x3, y3)]
        # the length of the first segment
        self.l1 = self.rp[0].segment(self.rp[1])
```

```
        # the length of the second segment
        self.l2 = self.rp[1].segment(self.rp[2])
        # all 4 segments
        s1 = Segment(x0, y0, x1, y1)
        s2 = Segment(x1, y1, x2, y2)
        s3 = Segment(x2, y2, x3, y3)
        s4 = Segment(x3, y3, x0, y0)
        # all 4 segments must to be perpendicular
        assert (s1.is_perpend(s2) and s2.is_perpend(s3)
           and s3.is_perpend(s4) and s4.is_perpend(s1)),
           'Quadrilateral must be a rectangle'
    # area of a rectangle: a = l1 * l2
    def area_calc(self):
        return self.l1 * self.l2
# a square is a particular rectangle, when l1 == l2
class Square(Rectangle):
    def __init__(self, x0, y0, x1, y1, x2, y2, x3, y3):
        Rectangle.__init__(self, x0, y0, x1, y1, x2, y2,
              x3, y3)
        # condition to be a square (l1 == l2)
        assert (self.l1 == self.l2),
           'Quadrilateral must be a Square'
```

```
    def area_calc(self):
        return super().area_calc()

# a circle is represented by the center of the circle
# and its radius
class Circle(Point):
    def __init__(self, x, y, r):
        # center of the circle (a point)
        Point.__init__(self, x, y)
        # the radius
        self.r = r
    def area_calc(self):
        return pi * self.r * self.r

# a triangular right prism is defined by its base
# (a triangle) and the length of a perpendicular edge
class Prism(Triangle):
    def __init__(self, x0, y0, x1, y1, x2, y2, h):
        # the base of the prism (a triangle)
        Triangle.__init__(self, x0, y0, x1, y1, x2, y2)
        # the height of the prism
        self.h = h
```

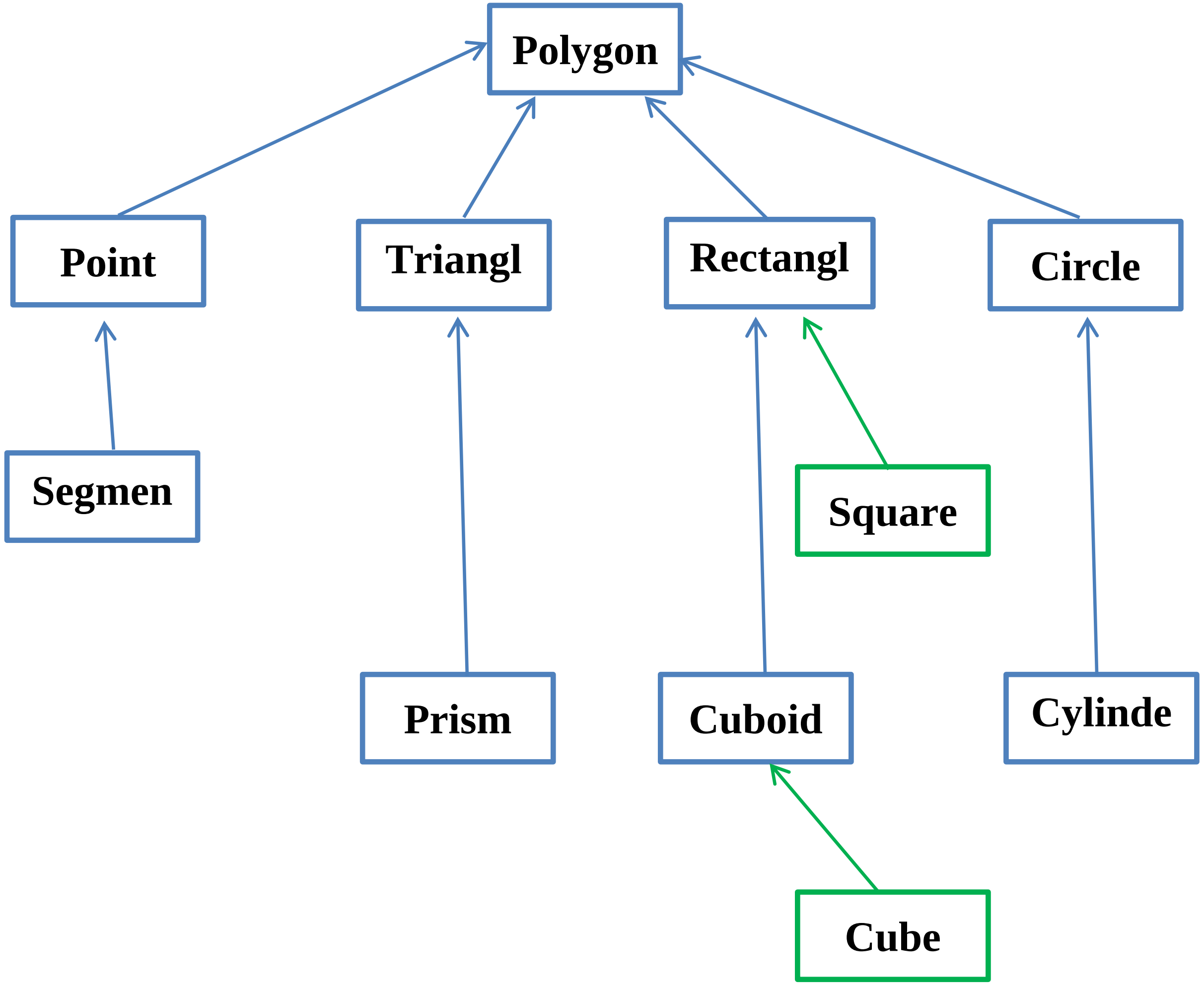
```
    # area of the prism
    def area_calc(self):
        return 2 * super().area_calc() + self.h *
            (self.l1 + self.l2 + self.l3)
# a cuboid is a rectangular parallelipiped
# is defined by its base (a rectangle) and its height
class Cuboid(Rectangle):
    def __init__(self, x0, y0, x1, y1, x2, y2, x3, y3, h):
        # the base of the cuboid (a rectangle)
        Rectangle.__init__(self, x0, y0, x1, y1, x2, y2,
              x3, y3)
        # the height of the cuboid
        self.h = h
    # total area of the cuboid
    def area_calc(self):
        return 2 * super().area_calc() + 2 * self.h *
             (self.l1 + self.l2)
# a cube is a special cuboid having all sides equals
class Cube(Cuboid):
    def __init__(self, x0, y0, x1, y1, x2, y2, x3, y3, h):
        Cuboid.__init__(self, x0, y0, x1, y1, x2, y2, x3,
              y3, h)
```

```
        # condition for the sides of the cube
        assert ((self.l1 == self.l2) and (self.l1 == self.h)),
           'Cuboid must be a cube'
    # total area of the cube
    def area_calc(self):
        return 6 * self.l1 * self.l2

# a right cylinder is defined by its base (a circle)
# and its height
class Cylinder(Circle):
    def __init__(self, x, y, r, h):
        # the base of the cilindre (a circle)
        Circle.__init__(self, x, y, r)
        # the height
        self.h = h
    # total area
    def area_calc(self):
        return 2*super().area_calc() + 2*pi*self.r*self.h
```

```
# module 'usepolygon.py'
from polygons import *
L = [Triangle(0, 0, 1, 0, 0, 1),
Rectangle(0, 0, 2, 0, 2, 1, 0, 1),
Square(0, 0, 1, 0, 1, 1, 0, 1),
Circle(0, 0, 1),
Prism(0, 0, 1, 0, 0, 1, 2),
Cuboid(0, 0, 2, 0, 2, 1, 0, 1, 2),
Cube(0, 0, 1, 0, 1, 1, 0, 1, 1),
Cylinder(0, 0, 1, 2)]
for obj in L:
    a = obj.area()
    print(a)
```

Remark. In ***Python*** the ***polymorphism*** mechanism is inherent to the language



Polygon
Point
Triangl
Rectangl
Circle
Segmen
Square
Prism
Cuboid
Cylinde
Cube

- There are two types of relations:
  - ***Inheritance*** (with blue), a static relation concerning code reuse
  - ***Subtyping*** (with green), a dynamic relation concerning using object at runtime
- Inheritance is similar to the same relation form the C++ language
- ***Polymorphism*** is inherent in Python due the ***dynamic typing*** and ***dynamic binding*** (there is no static binding in Python)
- ***Virtual functions*** from C++ are replaced by ***delegate functions*** (function ***`area()`*** in the example)
- ***Abstract classes*** are not defined by some syntactic constructions
  - An ***abstract class*** is a class that contains a method that is ***not implemented*** in the class (function ***`area_calc()`*** in the example)
  - However, there exists a ***decorator***, called ***`@abstractmethod`*** that can be used to define ***abstract methods***

E. A short history of object-oriented languages

- ***ALGOL*** (*Algorithmic Language*) is a programming language developed in the late of 1950s, which is the *most influential* language in all times
- Most languages in use today owe something to ALGOL
- ALGOL’s syntax and structure *directly* influenced a *large number* of other languages, known as “*Algol-like*” languages, such as:
  - Simula, C, Pascal, and Ada
- *Indirectly* he influenced the *most part* of imperative (and also a part of functional) programming languages
- Unfortunately, *ALGOL* has been little used in industry, because of large companies (IBM, for example), promoted the *Fortran* language
- It was, however, extensively used in *academic computer science*, and was the *standard language for algorithmic description* well into the 1980s and 90s

- **Example** of a procedure in *Algol 60* (from Wikipedia):

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
    value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a,
     of size n by m, s transferred to y, and the subscripts
     of this element to i and k;
begin
    integer p, q;
    y := 0; i := k := 1;
    for p := 1 step 1 until n do
        for q := 1 step 1 until m do
            if abs(a[p, q]) > y then
                begin y := abs(a[p, q]);
                    i := p; k := q
                end
end Absmax
```

- ***SIMULA*** (1967) was the ***first*** object-oriented language in history
- Originally designed for the purpose of ***simulation***
- SIMULA was designed as an extension and modification of ***Algol 60***
- Some SIMULA features:
  - ***Objects***: A SIMULA object is an ***activation record*** produced by call to a class
  - ***Classes***: A SIMULA class is a ***procedure that returns a pointer to its activation record***. The body of a class may initialize the objects it creates
  - ***Dynamic lookup*** (***polymorphism***): Operations on an ***object are selected from the activation record*** of that object
  - ***Abstraction***: ***Hiding data*** was not provided in SIMULA 67 but *was added later* and *used as the basis for C++*
  - ***Subtyping***: Objects are typed according to the classes that create them. Subtyping is determined by ***class hierarchy***
  - ***Inheritance***: A SIMULA class may be defined, by ***class prefixing***, as an extension of a class that has already been defined including the ability to redefine parts of a class in a subclass

- ***Smalltalk*** (1972) was the ***second*** object-oriented language, designed by ***Alan Kay*** (a visionary computer scientist)
- It was inspired by the simplicity of ***LISP*** and the classes and objects of ***Simula67***, but it was a completely ***new language***, with new terminology and an ***original syntax***
- It was written as an ***operating system*** for Dynabook, “A Personal Computer for Children of All Ages” – a concept of Alan Key for a very thin portable computer (similar, but more complex than a notebook or a tablet PC)
  - The ***Dynabook*** was never built, simply because it was too far ahead of technologies in the 1960s and 1970s
  - Instead of ***Dynabook***, the ***Alto*** computer was developed at Werox PARC (where Alan Key worked) in 1973, the ancestor of modern PC computers (having a ***mouse***, a **desktop**, and a ***graphical user interface***)
    - All ***Altos*** computers were connected to the *first local area network* (LAN)
    - ***Apple II*** was released in 1977, and ***Atari*** was released in 1981 (without mouse)

- ***Smalltalk*** encapsulated all of the pieces related to a modern personal computer, including many of the features that were desired in the Dynabook.
  - ***Smalltalk*** systems were the first to have ***bit-mapped displays***, ***overlapping windows***, ***menus***, ***icons***, and a ***mouse*** pointing device.
    - ***Microsoft Windows***, ***UNIX X-Windows***, and the ***Macintosh operating systems*** all have their roots in ***Smalltalk***
- ***Smalltalk*** was implemented as a ***bytecode compiler***. Smalltalk code was actually compiled into a virtual machine language
  - This technique was used later in the ***Java*** compilers and ***.NET*** languages
- ***Smalltalk environments*** were the first to develop what are now object-oriented software ***design patterns***. The ***model–view–controller*** (MVC) pattern was used in Smalltalk environments for ***user interface design***.
- In ***Smalltalk*** everything is an ***object***; even a ***class***. All ***operations*** are ***messages*** to objects

- Some ***Smalltalk*** features:
  - ***Objects***: A Smalltalk object is ***created*** by a ***class***. At run time, an object stores its instance variables and a pointer to the instantiating class
  - ***Classes***: A Smalltalk class defines ***variables***, ***class methods***, and the ***instance methods*** that are shared by all objects of the class
  - ***Abstraction***: Abstraction is provided through ***protected instance variables***. All methods are ***public*** but instance variables are ***protected***
  - The ***type*** of an object in Smalltalk is its ***interface***, i.e. the ***set of messages*** that can be sent to the object
  - ***Subtyping***: Subtyping arises implicitly through ***relations between the interfaces*** of objects. ***Subtyping*** depends on the set of messages that are understood by an object, not on the representation of objects
  - ***Inheritance***: Smalltalk subclasses ***inherit*** all ***instance variables*** and ***methods*** of their ***superclasses***. Methods defined in a superclass may be redefined in a subclass or deleted
  - The run-time structures used for ***Smalltalk*** classes and objects support ***dynamic lookup*** (***polymorphism***)

- ***C*** (developed in 1972 by Dennis Ritchie) seems to be the ***second influential language*** after ***ALGOL***
  - It has influenced almost every programming language that came after it
- The ***C*** programming language is still a ***very popular language*** (according to Tiobe index, he is ranked in the first or second place since 1989 until now)
- In the middle 1980s two different OOP languages have been developed based on C:
  - ***Objective C*** (1985) – by adding ***Smalltalk-style*** messaging to the ***C*** programming language
  - ***C++*** (1982-1985: ***Bjarne Stroustrup***) – by adding ***OOP features*** to ***C***, in a similar way that ***Simula*** added OOP features to ***ALGOL***
- ***Objective C***:
  - Was the main programming language supported by ***Apple*** for the ***macOS***, ***iOS*** operating systems
  - Use ***dynamic typing*** (***static typing*** is optional) and ***single inheritance***

- ***C++***:
  - Is used in different domains: ***embedded devices*** programming, ***game programming***, and also in most ***system programming*** where the ***large software systems*** can be developed
  - Use ***static typing*** and ***multiple inheritance***
- ***List*** (***List Processor***) is another very old (but different from others) programming language, developed in 1958 at MIT:
  - It is a ***functional*** programming language, based on ***lambda calculus***
- ***Functional programming*** and ***object-oriented programming*** are two different paradigms:
  - Between 1980s and 2000s there was a growth in the popularity of ***object-oriented programming*** languages
  - In the last decade there is an increasing effort to ***integrate these two programming*** paradigms:
    - Some languages, such as ***Scala***, ***Clojure*** and ***Swift*** explicitly implement ***features from both paradigms***
    - Some OO languages, such as ***C++***, ***Java***, ***Python*** and ***Ruby*** ***are converging*** to some point

- However, the first language that integrate the two programming paradigms is ***CLOS*** (Common Lisp Object System)
  - It was developed in the late 1980s
  - It is based on the ***Lisp*** syntax by adding OO concepts from ***Smalltalk***
- Also, in the late 1980s, ***Eiffel***, another OO programming language was developed:
  - It was developed by ***Eiffel Software*** (a company founded by ***Bertrand Meyer***), which contains a detailed treatment of the concepts and ***theory of the object technology*** that led to Eiffel's design
  - It is based on two programming languages: ***ADA*** and ***Smalltalk***
  - The most important contribution of ***Eiffel*** to ***software engineering*** is ***design by contract*** (DbC), in which ***assertions***, ***preconditions***, ***postconditions***, and ***class invariants*** are employed to help ensure ***program correctness***
- In 1990s several OO programming languages were developed, in response to the need for ***rapid application development*** (RAD) and the opportunity created by the ***Internet Age***

- ***Python*** was developed in 1991, as a successor of the ***ABC*** language
  - It is ***dynamically typed*** and ***garbage-collected*** and ***pure object-oriented***
  - It supports ***multiple programming paradigms***, including ***procedural***, ***object-oriented***, and ***functional*** programming
  - ***Python standard library*** provides tools for many tasks, such as: ***Internet applications***, creating ***graphical user interfaces***, using ***relational databases***, ***scientific computing***, ***regular expressions***, and ***unit testing***
  - The ***Python Package Index*** (PyPI), the official repository of Python contains many ***packages*** with a wide range of functionality, including:
    - **Graphical user interfaces**
    - **Machine learning**
    - **Web frameworks**
    - **Multimedia**
    - **Databases**
    - **Networking**
    - **Test frameworks**
    - **Documentation**
    - **System administration**
    - **Scientific computing**
    - **Text processing**

        - **Image processing**
    - Since 2003, Python has consistently ranked in the ***top ten most popular programming languages*** in the TIOBE Programming Community Index
    - Python is not named after the ***snake***. It's named after the British TV show ***Monty Python***

- ***Java*** vas released in 1995 at the ***Sun Microsystems***, as an ***object-oriented language*** (which is ***not pure OO***) by borrowing some ideas from other OO languages:
    - ***Syntax*** from ***C++***
    - ***Compiling to a virtual machine*** from ***Smalltalk***
    - ***Interfaces*** from ***Eiffel***

- Sun Microsystems released Java for providing ***no-cost run-times*** on ***popular platforms***

- Major ***web browsers*** incorporated the ability to run ***Java applets*** within web pages, and Java ***quickly became popular***

- The ***Java Class Library*** is the standard library, developed to support application development in Java, which is controlled by ***Sun Microsystems*** in cooperation with others through the ***Java Community Process*** program
- The ***core libraries*** include:
  - **IO/NIO**
  - **Networking**
  - **Reflection**
  - **Concurrency**
  - **Generics**
  - **Scripting/Compiler**
  - **Functional programming (Lambda, Streaming)**
  - **Collection libraries that implement data structures**
  - **XML Processing**
- Since 1999, Python has consistently ranked in the ***top three most popular programming languages*** in the TIOBE Programming Community Index
- ***Ruby*** was developed in 1995 in Japan, as an ***interpreted***, ***high-level***, ***general-purpose*** programming language

- It supports ***multiple programming paradigms***, including ***procedural***, ***object-oriented*** (*pure OO*), and ***functional*** programming (similar to Python)
- ***Ruby*** was influenced by ***Perl***, ***Smalltalk***, ***Eiffel***, ***Ada***, ***Basic***, and ***Lisp***
- A framework called ***Ruby on Rails*** has helped to increase its usage for ***web programming***
- Actually, ***Ruby*** is ranked as 13th in the TIOBE Index

❑ ***Current trends*** – OO languages developed ***after 2000***:

- ***C#*** - a ***general-purpose***, ***multi-paradigm programming*** language containing: ***strong typing***, ***imperative***, ***declarative***, ***functional***, ***object-oriented***, and ***component-oriented*** programming disciplines
    - Developed around 2000 by ***Microsoft*** as part of its ***.NET*** initiative (designed for the ***Common Language Infrastructure CLI***)
    - The language is intended for use in developing ***software components*** suitable for ***deployment in distributed environments***
    - ***C#*** was influenced by ***Java***, ***C++***, ***Eiffel*** and ***ADA***
    - Initially, James Gosling, who created Java in 1995, called ***C#*** an ***"imitation" of Java***

    - However, since 2005, the ***C#*** and ***Java*** languages have evolved on divergent trajectories, becoming two very different languages
    - Since 2004, ***C#*** has consistently ranked in the ***top seven most popular programming languages*** in the TIOBE Index
- ***Kotlin*** - ***statically-typed*** programming language that supports both ***object-oriented*** and ***functional*** programming
    - ***Kotlin*** provides similar syntax and concepts from other languages, including ***C#***, ***Java***, and ***Scala***, among many others
    - ***Kotlin*** was announced as an official ***Android*** development language at ***Google I/O*** 2017
    - ***Kotlin*** is ranked as 35^{th} in the TIOBE Index
- ***Swift*** is an alternative to the ***Objective-C*** language that employs a simpler syntax
    - During its introduction, it was described simply as ***Objective-C without the C***
    - It was developed by ***Apple*** for ***iOS*** and ***macOS*** operating systems
    - ***Swift*** took language ideas from ***Objective-C***, ***Rust***, ***Haskell***, ***Ruby***, ***Python***, ***C#***, and other many languages

- ***Swift*** is ranked as 12th in the TIOBE Index