Object-Oriented Programming and Python Application

Julia Clemente

Máster en Tecnologías de la Información Geográfica
Universidad de Alcalá

Objectives

1. Introduce basic programming concepts.
2. Understand the main characteristics of Object-Oriented Programming (OOP).
3. Use Python to implement hierarchies of basic classes.

Table of contents

Understanding concepts

Differentiate between . . .

Programming

Set of techniques that allow the development of programs using a programming language.

Programming language

Set of rules and instructions based on a familiar syntax and later translated into machine language which allow the elaboration of a program to solve a problem.

Paradigm

Set of rules, patterns and styles of programming that are used by programming languages [3].

Programming paradigms types (I)

Declarative programming

Describes **what** is used to calculate through conditions, propositions, statements, etc., but does not specify **how**.

- **Logic**: follows the first order predicate logic in order to formalize facts of the real world (e.g., *Prolog*).
  - Example: *Anne's father is Raul, Raul's mother is Agnes. Who is Ana's grandmother*
- **Functional**: is based on the evaluation of functions (like maths) recursively (e.g., *Lisp y Haskell*).
  - Example: *the factorial of 0 and 1 is 1 and the factorial of n is n * factorial (n-1). What is the factorial of 3?*

Programming paradigms types (II)

Imperative programming

Describes by a set of instructions that change the **program state**, **how** the task should be implemented.

- **Procedural**: organizes the program using collections of subroutines related by means of invocations (e.g., *C, Python*).
  - Example: *The cooking process consists of 20 lines of code. When it is used, it only calls the function (1 line).*
- **Structural**: is based on conditional statements, nesting, loops and subroutines. Sentences of `GOTO` type is forbidden (e.g., *C, Pascal*).
  - Example: *reviewing products of a shopping list and add the item X to the shopping if it is available.*

Programming paradigms types (III)

Object-Oriented Programming

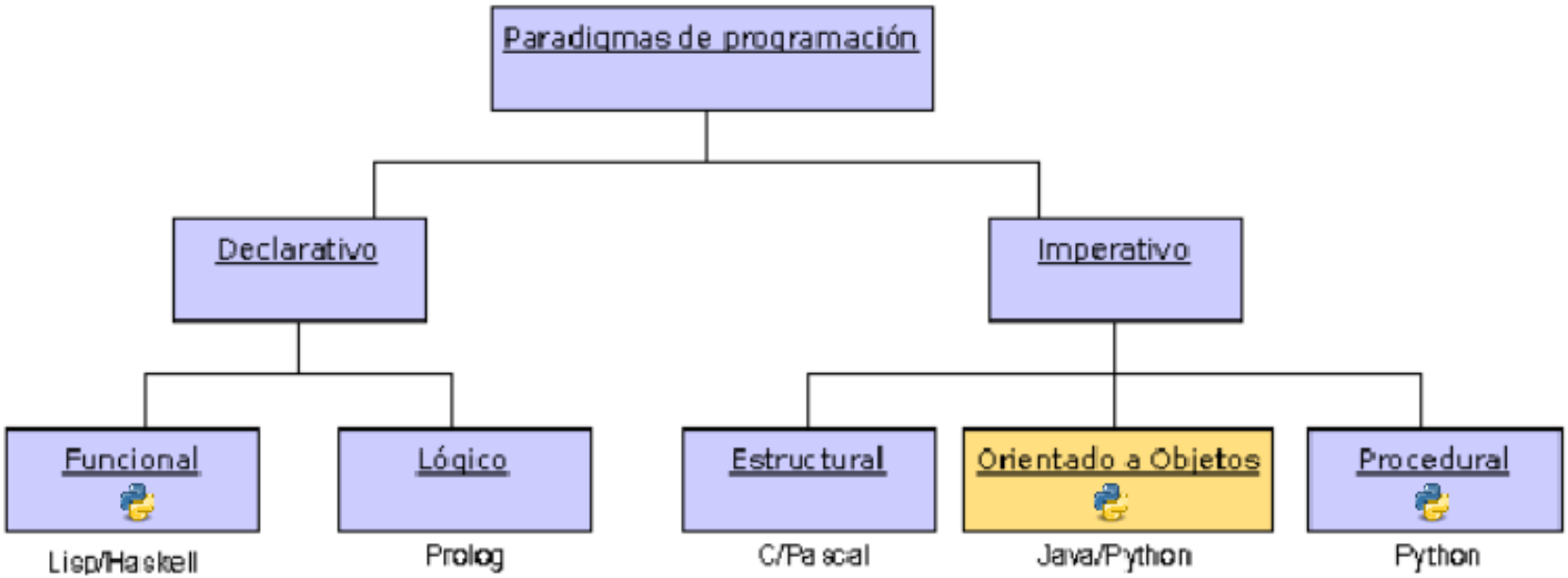
Evolves from imperative programming. It is based on **objects** that allow express the **characteristics** and **behavior** in a closer way to real life (Java, Python, C++).

- **Main characteristics**: abstraction, encapsulation, polymorphism, inheritance, modularity, etc.
- Example: *a car has a set of properties (color, fuel type, model) and a functionality (speed up, shift gears, braking).*

There are many other paradigms such as Event-Driven programming, Concurrent, Reactive, Generic, etc.

Programming paradigms types (IV)

Classification



Python supports the three major paradigms, although it stands out for the OOP and Imperative paradigms.

Object-Oriented Programming

Objectives

- **Reusability**: Ability of software elements to serve for the construction of many different applications.
- **Extensibility**: Ease of adapting software products to specification changes.
- **Maintainability**: Amount of effort necessary for a product to maintain its normal functionality.
- **Usability**: Ability of a software to be understood, learned, used and attractive to the user, under specific conditions of use.
- **Robustness**: Ability of software systems to react appropriately to exceptional conditions.
- **Correction**: Ability of software products to perform their tasks accurately, as defined in their specifications.

Object-Oriented Programming

Concepts (I)

Class

Generic entity that groups the properties and functions of the objects described by it [4], [1].

Object-Oriented Programming

Concepts (II)

Atribute

Individual **characteristics** that determine the qualities of an object.

Object-Oriented Programming

Concepts (III)

Method

Function responsible for performing operations according to input parameters.

Object-Oriented Programming

Concepts (IV)

Object or instance

Specific representation of a class with its corresponding attributes (data) and behaviour (methods).

Object-Oriented Programming

Concepts(V)

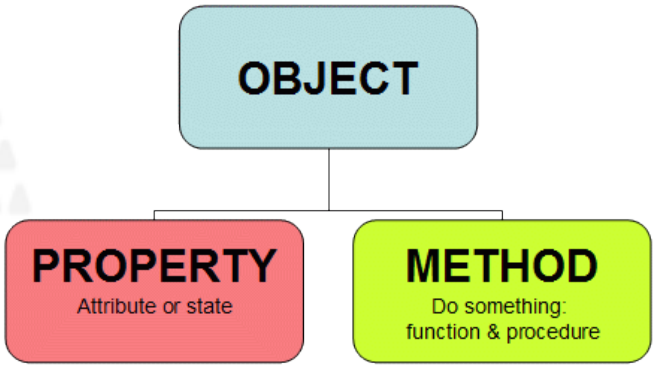
Constructor

Method called when an object is created. It allows the initialization of attributes.

Object-Oriented Programming

Synthesizing OOP terminology

- Software objects mimics physical objects.
  - An object contains *attributes* (state) and a *behaviour*.
  - Example: A dog has a name (state) and may be a little (behaviour).
- A **class** is a set of objects with common characteristics and behaviour.
- An **object** is called an **Instance** of a class.
- **Members** of a class:
  - **Properties**: Data describing an object.
  - **Methods**: What an object can do.



Source:
http://www.teachitza.com/delphi/oop.htm

Characteristics

Inheritance

Concept

Mechanism of **reusing** code in OOP. Consists of generating child classes from other existing (**super-class**) allowing the use and adaptation of the attributes and methods of the parent class to the child class.

- ***Superclass***: "Father" of a class.
- ***Subclass***: "Child" of a class.
- A subclass inherits all the fields and methods from its superclass.
  - ***Fields***: Variable that is part of an object.
- A subclass has **one** or **more** superclass.
- A superclass has **at least one** subclass.
- ***Class hierarchy***: A set of classes related by inheritance.

Characteristics

Inheritance (II)

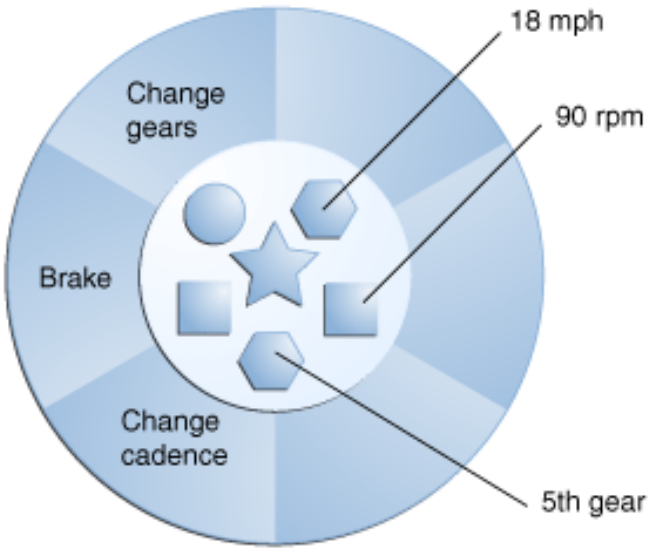
Types of inheritance

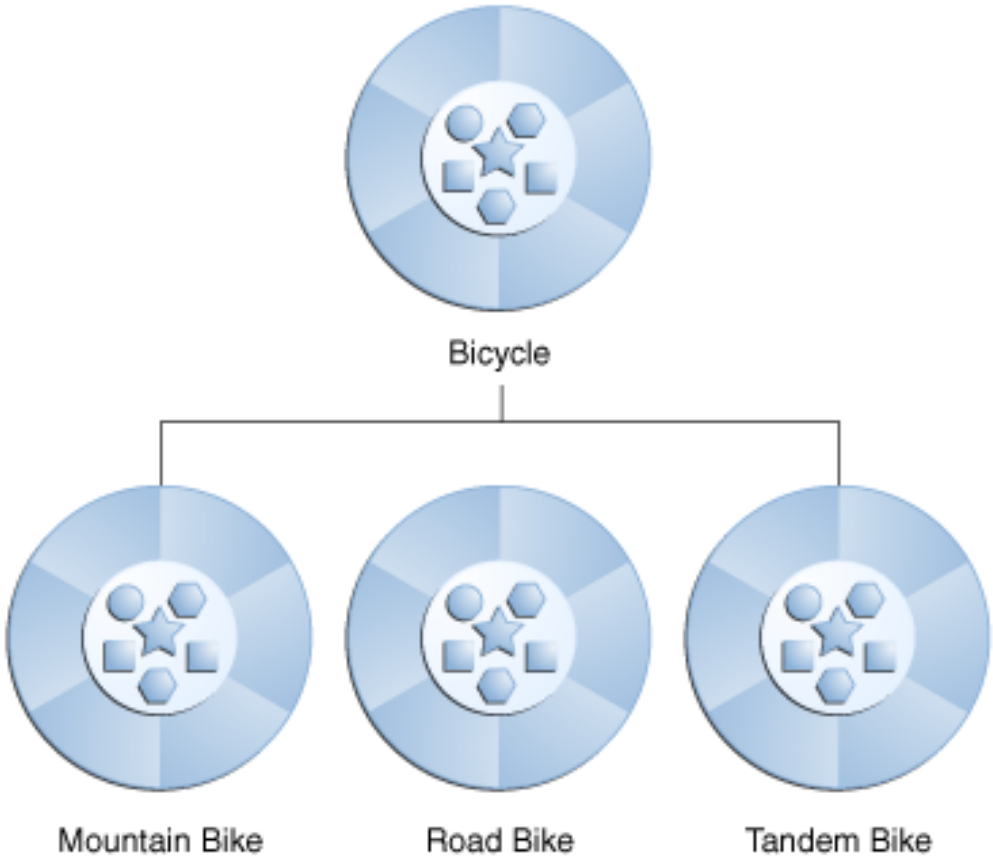
- If the child class inherits from a single class is called **single inheritance**.
- If it inherits from more classes is **multiple inheritance**.

Python allows both; simple and multiple inheritance.

Characteristics

Examples of simple inheritance (I)

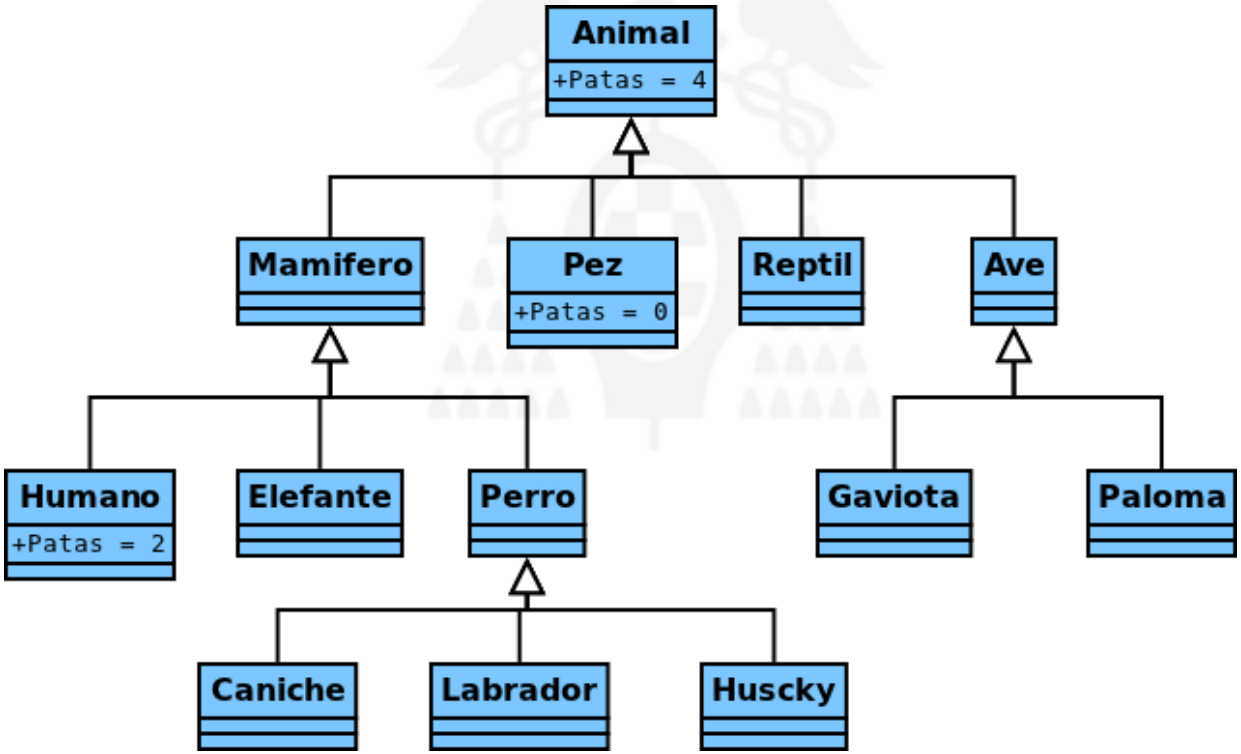




Source: http://docs.oracle.com/javase/tutorial/java/concepts/object.html
Source: http://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html

Characteristics

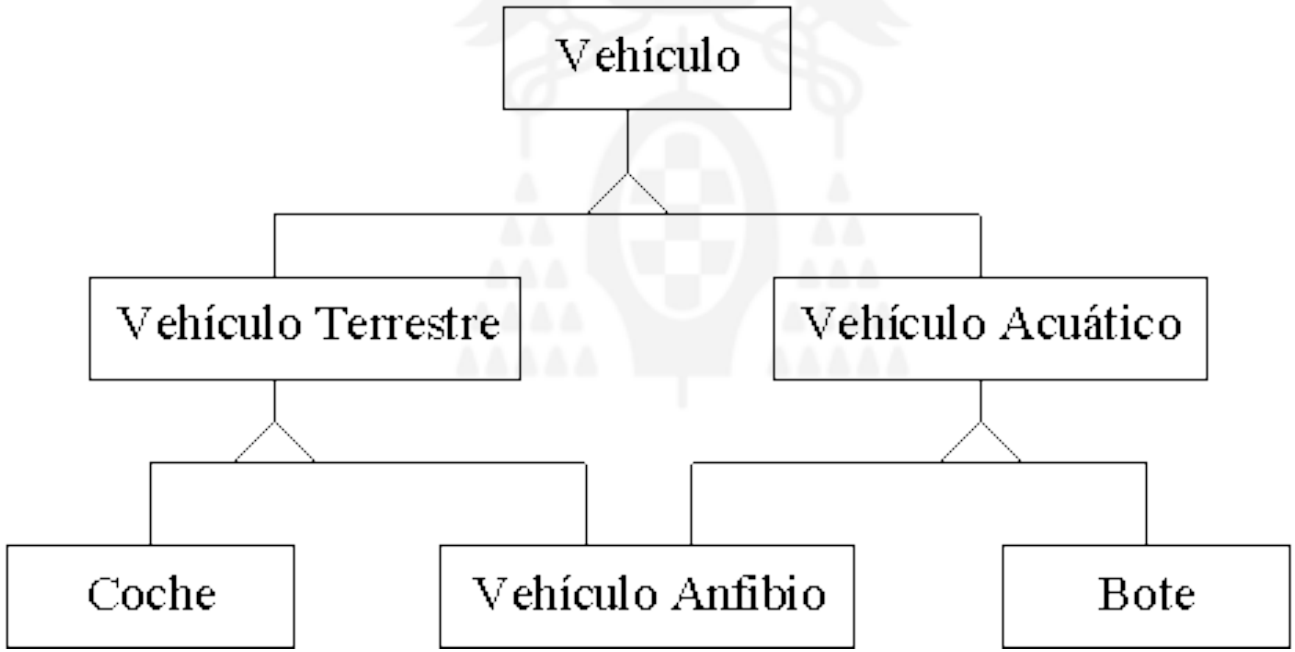
Examples of simple inheritance (II)



Source: https://android.scenebeta.com/tutorial/herencia-y-polimorfismo

Characteristics

Example of multiple inheritance



Source: http://www.chambers.com.au/Sample_p/c_inhert.htm

Characteristics

Polymorphism

Polymorphism

Mechanism of object-oriented programming that allows to invoke a method whose implementation will depend on the object that does it.

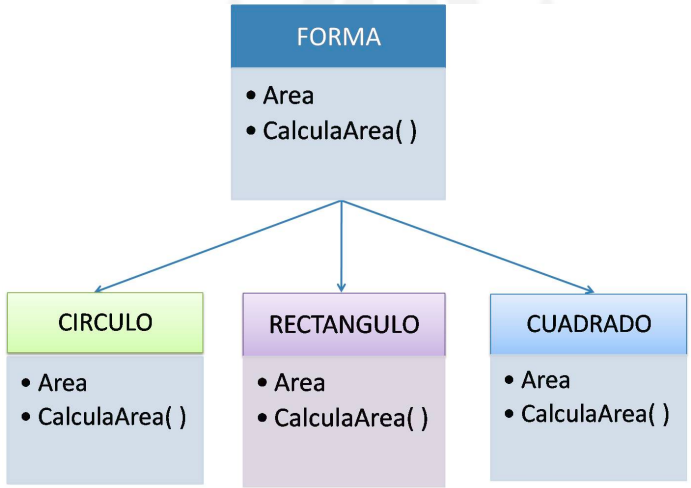
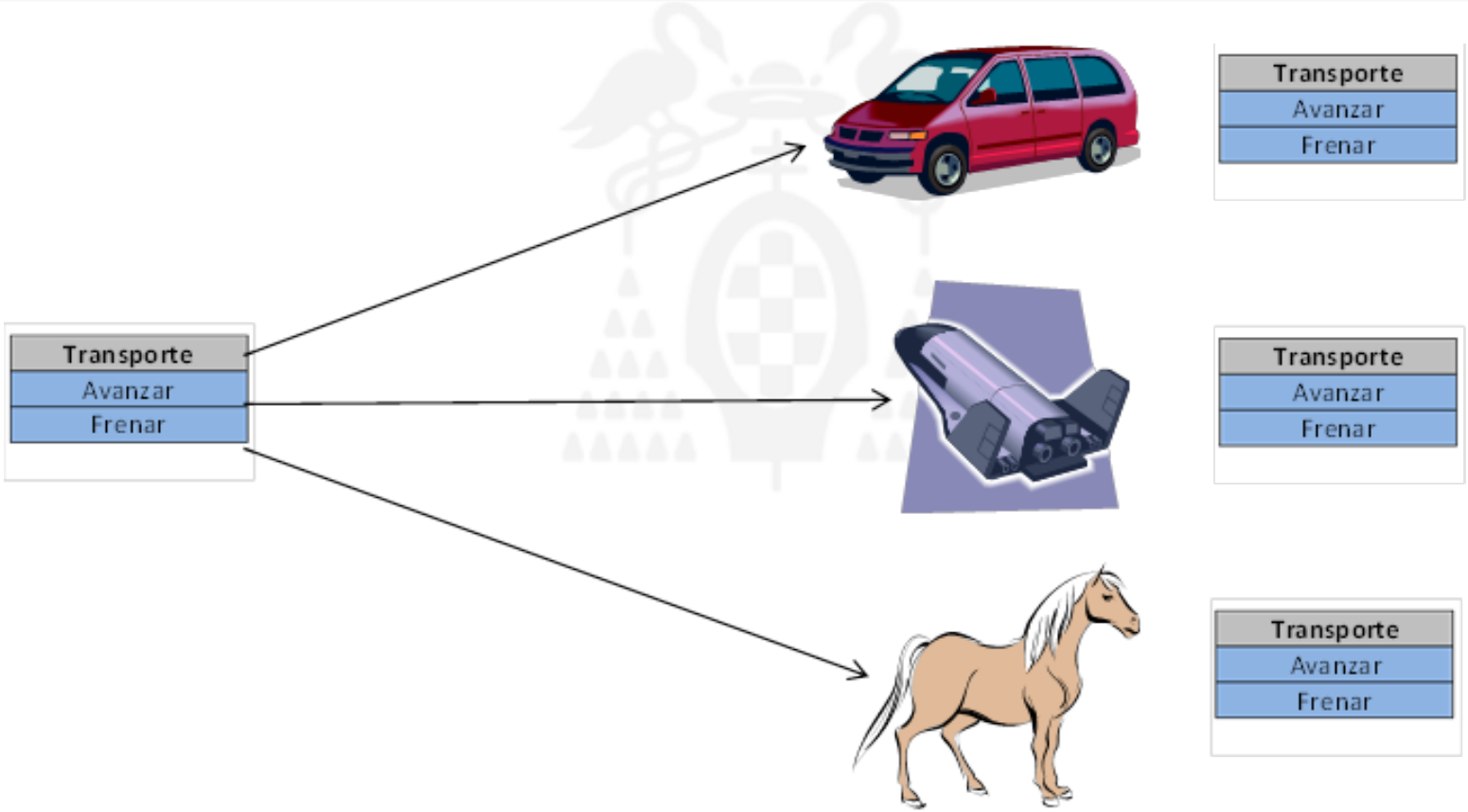


Figura: Example of polymorphism. Obtained from: `http://blogjavacartagena.blogspot.com/2014/03/clase-abstracta-en-java.html`

Characteristics

Example of polymorphism



Source: http://contadores-acumuladores-java.blogspot.com/2016/05/herencia-en-programacion-es-permitir-la.html

Characteristics

Abstraction and encapsulation (I)

Abstraction

Mechanism that allows the isolation of the not relevant information to a level of knowledge.

- *A driver does not need to know how the carburetor works.*
- *To talk on the phone does not need to know how the voice is transferred.*
- *To use a computer do not need to know the internal composition of their materials.*

Characteristics

Abstraction and encapsulation (II)

Encapsulation

Mechanism used to provide an *access level* to methods and attributes for avoiding unexpected state changes. It is applied to limit the visibility of the attributes and to create methods controlling them (`set()` y `get()`).

The most common access levels are:

- **public**: visible for everyone [default level in Python].
- **private**: visible for the creator class [start with a double underscore (but it does not end in the same manner)].
- **protected**: visible for the creator class and its descendents. Not exist in Python.

Characteristics

Examples of abstraction and encapsulation (I)

Encapsulation:

For the mathematical equation, shown in the figure assume that complex functions are required - >But I the end we obtain result for it

Calculator shows the result of equation but hides the implementation (calculating the result) involved.

Abstraction:

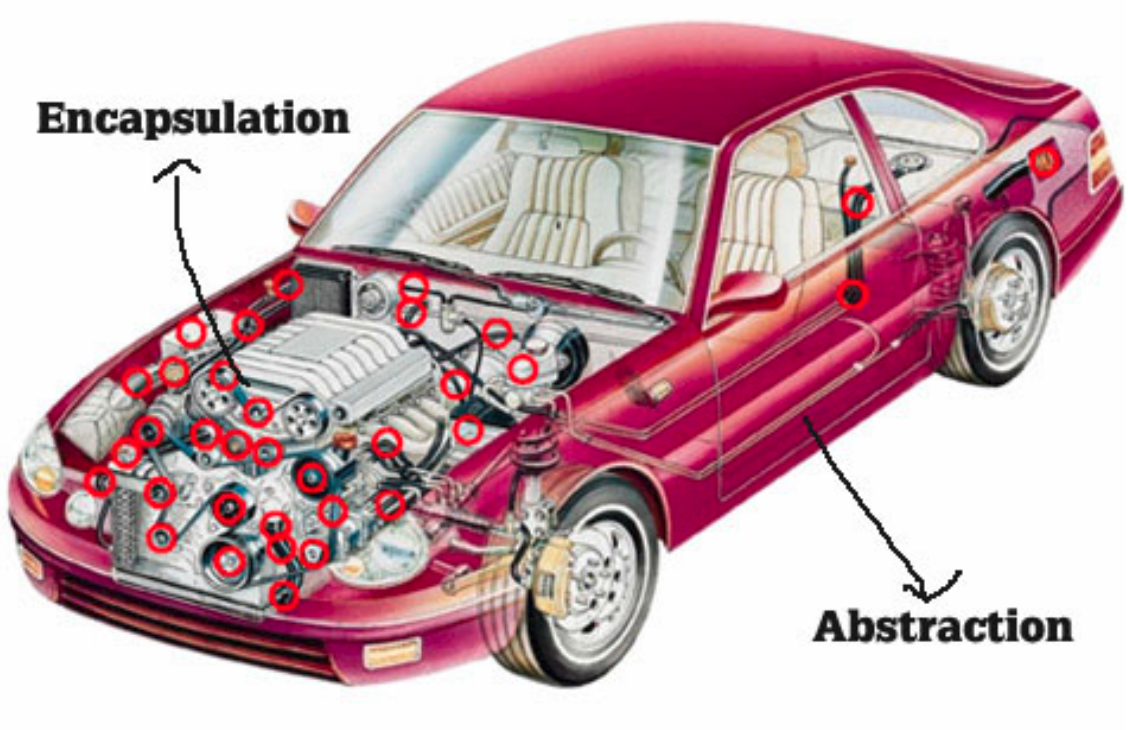
The calculator shown in the figure has to be powered by a battery source. How the battery module works for the calculator is not necessary to know for the user who uses the calculator.

Using Battery module along with other modules we use calculator -> Thus using Abstraction encapsulation is performed

Source: https://binalparekh.wordpress.com

Characteristics(III)

Examples of abstraction and encapsulation (II)



Source:
https://www.onlinebuff.com/article_understand-object-oriented-programming-oops-concepts-in-php_46.html

All entities are objects in Python

```
import math
print(type(math))
a = 2
print(type(a))
b = 2.3
print(type(b))
c = lambda x: x+2
print(type(c))
lista = [2, "hola"]
print(type(lista))
lista.append(4)
print(lista)
tupla = (2, "hola")
print(type(tupla))
cadena = "hola"
print(type(cadena))
print("Y el resto...")
```

Classes in Python

Introduction (II)

All entities are objects in Python: output

```
<class 'module'>
<class 'int'>
<class 'float'>
<class 'function'>
<class 'list'>
[2, 'hola', 4]
<class 'tuple'>
<class 'str'>
Y el resto...
```

Classes in Python

Introduction (III)

All entities are objects in Python: use of `dir()` and `help()`

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', ... ,
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> help(list)
Help on class list in module builtins:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |       ...
```

Classes in Python

Syntax (I)

- **Class**: Start with the word **class** followed by class name written in **capital letter** and a colon (substantives).
- **Attributes**: A lowercase noun.
  - There is no need to declare attributes.
- **Inherited class**: Similar to a class but the class name followed by the class father in brackets.
- **Instance**: Object in lower case followed by the class assignment.

Classes in Python

Syntax (II)

- **Method**: Start with the word **def**, and later the method name, a verb, in lower case is written. Next, the parameters in brackets and a colon (`print_name()`).
  - Methods receive automatically a reference to the object (usually named `self`).
- **Constructor**: Method whose name is `__init__()`. Its first argument is `self` and then, the rest ones are used to initialize the **instance attributes**.
- All methods and attributes are public.
  - By convention, private members begin with double underscore (`__varName`, `__method_name()`)

Classes in Python

Syntax (III). Example 1

coche.py

```
class Vehicle(object):
  def __init__(self, wheels):
    self.wheel = wheels
class Car(Vehicle):
  def __init__(self, wheels, model):
    Vehicle.__init__(self, wheels)
    self.model = model
ford = Car(4, "Mondeo")
```

Classes in Python

Syntax (IV). Example 2

main: *Key* function defined with `def main()`. In it, the wished commands are specified and, finally, an exit condition is created (the `sys` module is required to be imported at the beginning).

main.py

```
import sys

def main():
  print("Hello World")

if __name__ == "__main__":
  sys.exit(main()) # Exit the program after main
      ()

print("Adios mundo") # Will never be read
```

Classes in Python

Syntax (V). Example 3

bicicleta.py

```
class Bicycle: # Class
  speed = 2 # Class attribute and assignment

  def __init__(self, speedB): # Constructor
    self.speed = speedB

  def decreaseSpeed(self): # Own method
    self.speed = self.speed - 1

  def printSpeed(self):
    print(self.speed)

if __name__ == '__main__': # Main
  a = Bicycle(4) # Instance
  a.decreaseSpeed()
  a.printSpeed()
```

Classes in Python

Syntax (VI). Example 4

example_time.py

```
class Time:
    """Represents the time of day

    attributes: hour, minute, second
    """

    def print_time(self):
      print('{0:}:{1:}:{2:}'.format(self.hour, self.minute,
          self.second))

time = Time()
time.hour = 11
time.minute = 59
time.second = 33
time.print_time()
```

Classes in Python

Class objects

Two operations on classes

Attribute references

Accesses an attribute value

Standard dot syntax

```
obj.name
```

```
time.hour = 4
print(time.hour)
hour = time.hour
```

Instantiation

Creates a new object

Standard functional notation

```
x = MyClass()
```

```
time = Time()
```

Constructors

Concept

Instantiation creates empty objects

- We usually need to initialize attributes
- Initialization operations

Constructor: Method called when an object is created

- In Python, it is the `__init__()`
- A constructor method **never** returns a value.

Time.py with constructor

```
class Time:
    """Represents the time of day

    attributes: hour, minute, second
    """
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def print_time(self):
        print('{0:}:{1:}:{2:}'.format(self.hour, self.minute
            , self.second))

time1 = Time()
time1.print_time()
time2 = Time(11, 40, 23)
time2.print_time()
```

Other special methods

In addition to special method `__init__`, there are several others such as:

- `__str__(self)` It should return a string with `self` information. When `print()` is invoked with the object, if the method `__str__()` is defined, Python shows the result of running this method on the object.
- `__len__(self)` It should return the length or "size" of object (number of elements if is a *set* or *queue*).
- `__add__(self, otro_obj)` It allows to apply the addition operator (+) to objects of the class in which it is defined.
- `__mul__(self, otro_obj)` It allows to apply the multiplication operator (*) to objects of the class in which it is defined.
- `__comp__(self, otro_obj)` It allows to apply the comparison operators (<, >, <=, >=, ==, !=) to objects of the class in which it is defined. It should return 0 if they are equal, -1 if `self` is smaller than `other_obj` and 1 if `self` is greater than `other_obj`.

Other special methods

Example

vector.py

```
class Vector:
    """Represents a vector

    attributes: a y b (init and final)
    """

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return('Vector: (%d, %d)' % (self.a, self.b))

    def __add__(self, other):
        return(Vector(self.a + other.a, self.b + other.b))

vector1 = Vector(1, 9)
vector2 = Vector(6, -1)
print(vector1 + vector2)
print(vector1)
print(vector2)
```

Overriding methods (I)

Overriding: Often we need to adapt an inheritanced method.

Example of overriding

```
class A:
    def hello(self):
        print("A says hello")

class B(A):
    def hello(self):
        print("B says hello")

b = B()
b.hello()
```

Overriding methods (II)

Still, it is possible to get superclass' method with `super()`

Example of `super()`

```
class A:
    def hello(self):
        print("A says hello")

class B(A):
    def hello(self):
        print("B says hello")
        super().hello()

b = B()
b.hello()
```

Simple exercise 1: statement

`Animal` class

1. Create the `Animal` class.
2. Create the constructor. The class will have the attributes `type` and `paws`.
3. Create the get methods from both attributes which receive like own parameter the animal through `self` and return respectively the `type` and `paws`.
4. Create two instances of animals using the constructor.
5. Print the attributes of both instances.

Solved exercise 1

Animal class

animales.py

```
class Animal:
  # Class constructor
  def __init__(self, type, paws):
    self.type = type
    self.paws = paws

  # get() methods of the Animal class
  def getType(self):
    return self.type

  def getPaws(self):
    return self.paws

# Animal instances
snoopy = Animal('Dog', 4)
commonCat = Animal("Cat", 4)

# Screen printing
print snoopy.getType()
print commonCat.getPaws()
```

Extending the exercise 1 (I)

`Animal` class

1. Create a `Cat` class in the same file which inherits from the `Animal` class.
2. Create the constructor and add the `sound` attribute.
3. Create the method `meow` which prints the sound MIAU.
4. Create a instance and check the methods.

Extending the exercise 1 (II)

Class Animals

animales.py

```
# Cat inherits from animal
class Cat(Animal):
  # Constructor calls the constructor of Animal
      class
  def __init__(self, paws):
    Animal.__init__(self, "Cat", paws)
    self.sound = 'miau'

  # Methods of the Cat class.
  def meow(self):
    print self.sound

# Cat instances
catwithboots = Cat(2)

# Screen printing
catwithboots.meow()
print catwithboots.getType()
```

Exercise 2: statement

Class `Parcel`

1. Create a script, `parcelclass.py`, containing the class `Parcel`.
2. Create the constructor. The class will have the attributes `land_use` and `value`.
3. Create the `assessment` method to calculate the rate associated with the parcel as follows:
   - For single-family residential: *rate = 0.05 * value*
   - For multifamily residential: *rate = 0.04 * value*
   - For all other land uses: *rate = 0.02 * value*
4. Use the class from another *script* named `parceltax.py` which you create una instance of `Parcel` named `parcel1` using the constructor.
5. Print the attribute `land_use` of the instance.
6. Use the method `assessment` of `Parcel` to calculate the assessment of `parcel1`.

Solved exercise 2 (I)

Class Parcel

parcelclass.py

```
class Parcel(object):
    def __init__(self, land_use, value):
        # inicializar objetos de esta clase: constructor
        self.land_use = land_use
        self.value = value

    def assessment(self):
        # single family residence: SFR
        if self.land_use == "SFR":
            rate = 0.05
        # multi-family residence: MFR
        elif self.land_use == "MFR":
            rate = 0.04
        else:
            rate = 0.02
        assessment = self.value * rate
        return assessment
```

Solved exercise 2 (II)

Testing `Parcel`

parceltax.py

```
import parcelclass


parcel1 = parcelclass.Parcel("SFR", 100000)

# once an instance is created, the object's
# properties and methods can be used
print("Land use: ", parcel1.land_use)
tax1 = parcel1.assessment()
print(tax1)
```

Source

Solved exercise 2 (III)

Serializing objects Parcel

tasaparcela_pickle.py

```
import pickle
import parcelclass

parcel1 = parcelclass.Parcel("SFR", 100000)
tax1 = parcel1.assessment()
print (tax1)

print("Serialize the object: \n", parcel1)
fout = open("parcels.db", 'wb')
pickle.dump(parcel1, fout)
fout.close()

fout = open("parcels.db", 'rb')
parcel1out = pickle.load(fout)
fout.close()

print("Object read: \n", parcel1out)
print("Land use: ", parcel1out.land_use)
tax2 = parcel1out.assessment()
print(tax2)
```

Exercise statement

River class

1. Create the River class.
2. Create the constructor and add the name and length attributes.
3. length attribute must be private.
4. Create the setLenght method which receives self and lengthR and allows the set of any value for length.
5. Create the getName method which obtains the name of the river.
6. Create the getLength method which obtains the river length.
7. Create an instance and check the methods.
8. Try to do an assignment of river.name and other assignment with river.length What happen? It is correct to invoke the method named river.getLength() out of the classes? How do you explain that?

Exercise statement

Establishment of hierarchies from `River` class

1. Add to the `River` class the attribute `flow` and the method `divert` which receives two rivers and transfers 5 liters from the first to the second.
2. Create the `Tributary` class which inherits from `River`.
3. Create the method `__init__` of `Tributary` which initializes its `name` and `length` and, also, `tributary_river`, new attribute initialized with the name of the river which the affluent starts.
4. Is there any polymorphism in this sample?
5. Create the main and exit condition and try it. Does the main position affect to the application?
6. Experiment now with conditions and iterative structures limiting when a river can transfer water or try to do some transfer at the same time.

Y más...

Learn more: [2]

Bibliographic references I

- [1] G. van Rossum, Jr. Fred L. Drake.
  Python Tutorial Release 3.10.7, chapter 9.
  Python Software Foundation, 2022.
  https://docs.python.org/3/download.html
- [2] Steven F. Lott, Dusty Phillips
  Python Object-Oriented Programming.
  Packt Publishing, fourth edition, 2021.
- [3] Lenguajes de programacion, capítulo 1.
  Lenguajes de programacion.
  http://rua.ua.es/dspace/bitstream/10045/4030/1/tema01.pdf

Bibliographic references II

[4] Downey, A and Elkner, J and MEYER, C.
Aprenda a Pensar como un Programador con Python, capitulos 14 y 16.
Green Tea Press, 2002.