Object-Oriented Programming in Python

Videogames Technology
Escuela Politécnica Superior

Departamento de Automática

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

1. Programming paradigms
   - Understanding concepts
   - Programming paradigms types
2. Object-Oriented Programming
   - Objectives
   - Basic concepts
   - Characteristics
3. Classes in Python
   - Sintax
   - Class objects
   - Constructors
   - More about methods
   - Solved exercises
   - Approach to a final problem

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 [1].

Programming paradigms types (I)

Declarative programming

Describe 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. (Prolog)
  - Example: Anne's father is Raul, Raul's mother is Agnes. Who is Ana's grandmother
- **Functional**: it is based on the evaluation of functions (like maths) recursively (Lisp y Haskell).
  - Example: the factorial from 0 and 1 is 1 and n is the factorial from n * factorial (n-1). What is the factorial from 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 (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 nesting, loops, conditionals and subroutines. `GOTO` command is forbidden (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

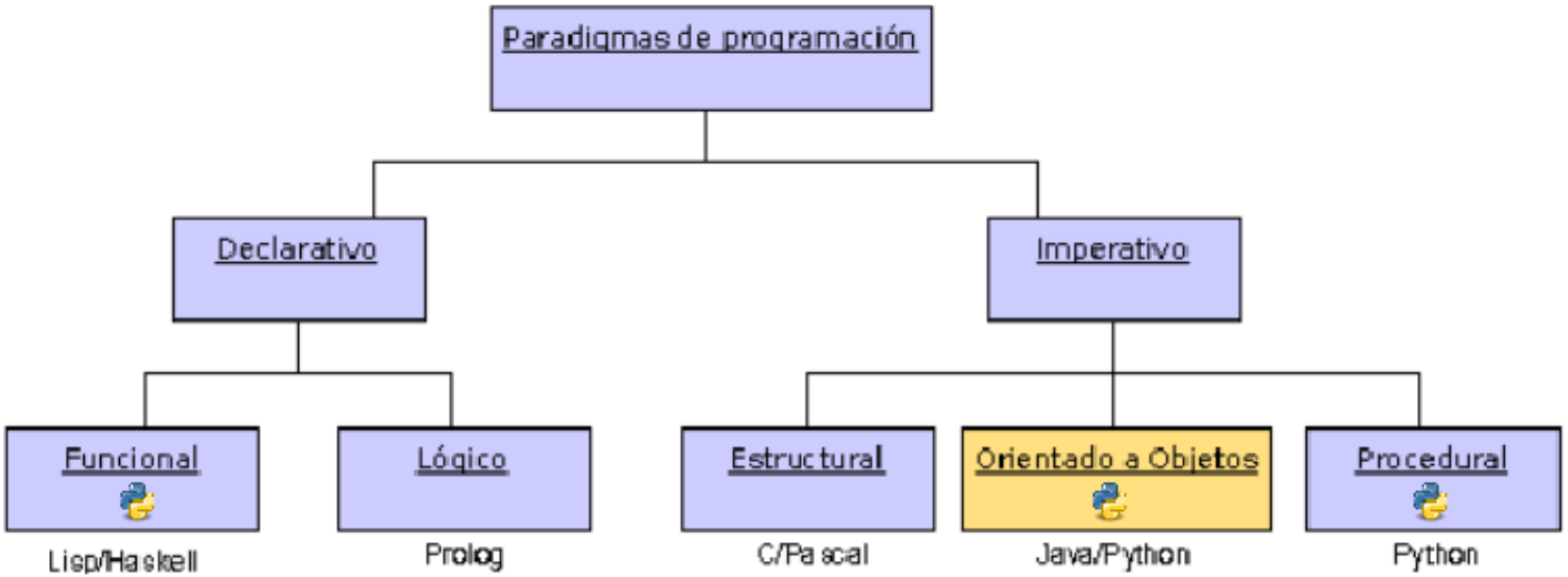
Evolves from imperative programming. It is based on objects that allow express the characteristics and behavior in a closer way to real life (Java, Python, C++).

- **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**: Ease of using the tool.
- **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 attributes and functions

Atribute

Individual characteristics that determine the qualities of an object

Method

Function responsible for performing operations

Object-Oriented Programming

Concepts (IV)

Object or instance

Specific representation of a class, namely, a class member with their corresponding attributes.

Object-Oriented Programming

Concepts (V)

Constructor

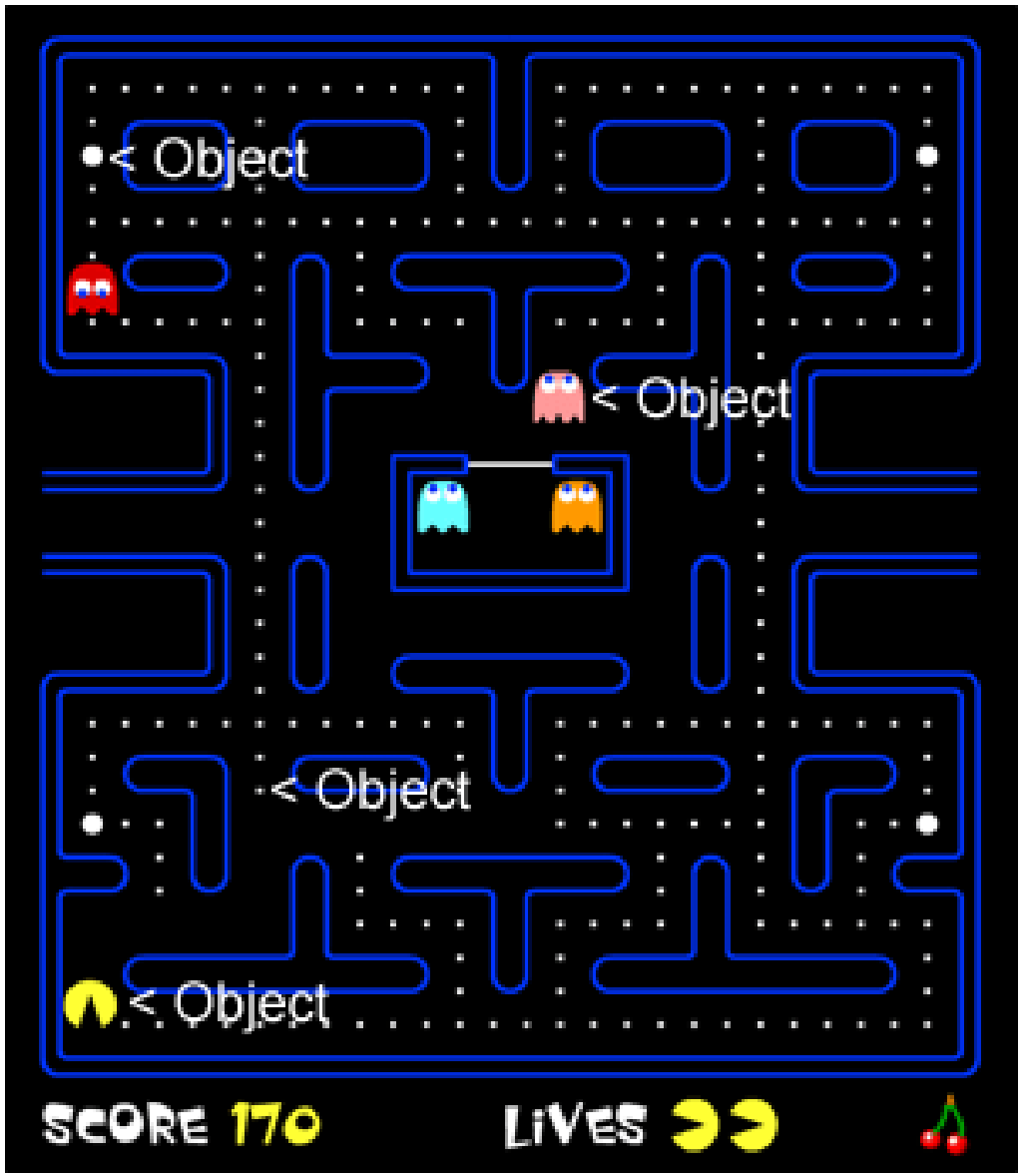
Method called when an object is created. It allows the initialization of attributes.

Constructors (II)

```
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()
```

Object-Oriented Programming

Synthesizing OOP terminology



(Source)

dogs.py

```
class Dog:
    def __init__(self): # Constructor
        self.name = "Unknown" # Attribute
        self.age = 10         # Attribute

    def bit(self):            # Method
        print(self.name + " has bitten")

    def describe(self):       # Method
        print("Name: ", self.name)
        print("Age: ", self.age)

if __name__ == '__main__':
    snoopy = Dog() # Instanciate class Dog ...
    laika = Dog()  # snoopy and laika are objtects

    snoopy.name = "Snoopy"
    snoopy.age = 4

    laika.name = "Laika"

    snoopy.bit()

    snoopy.describe()
    print()
    laika.describe()
```

Output

```
Snoopy has bitten
Name:  Snoopy
Age:  4

Name:  Laika
Age:  10
```

(Source code)

dogs.py

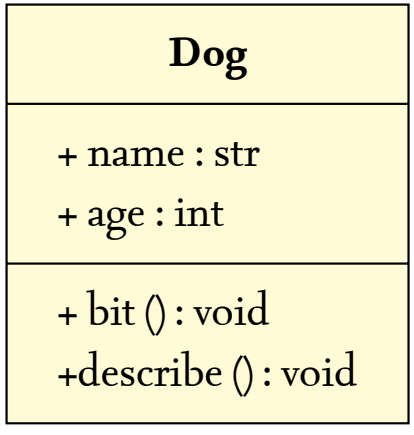
```
class Dog:
    def __init__(self): # Constructor
        self.name = "Unknown" # Attribute
        self.age = 10         # Attribute

    def bit(self):            # Method
        print(self.name + " has bitten")

    def describe(self):       # Method
        print("Name: ", self.name)
        print("Age: ", self.age)

if __name__ == '__main__':
    snoopy = Dog() # Instanciate class Dog ...
    laika = Dog()  # snoopy and laika are objtects

    snoopy.name = "Snoopy"
    snoopy.age = 4

    laika.name = "Laika"

    snoopy.bit()

    snoopy.describe()
    print()
    laika.describe()
```

Output

```
Snoopy has bitten
Name:  Snoopy
Age:  4

Name:  Laika
Age:  10
```

(Source code)

UML class diagram



Object-Oriented Programming

Game example

Characteristics

Inheritance

Inheritance

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 attributes and methods from its superclass
- *Class hierarchy*: A set of classes related by inheritance

Characteristics

Inheritance (II)

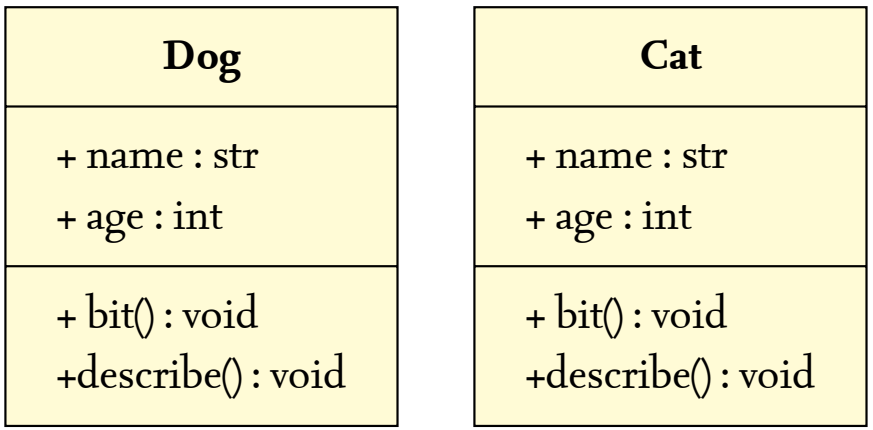
Types of inheritance

- 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.

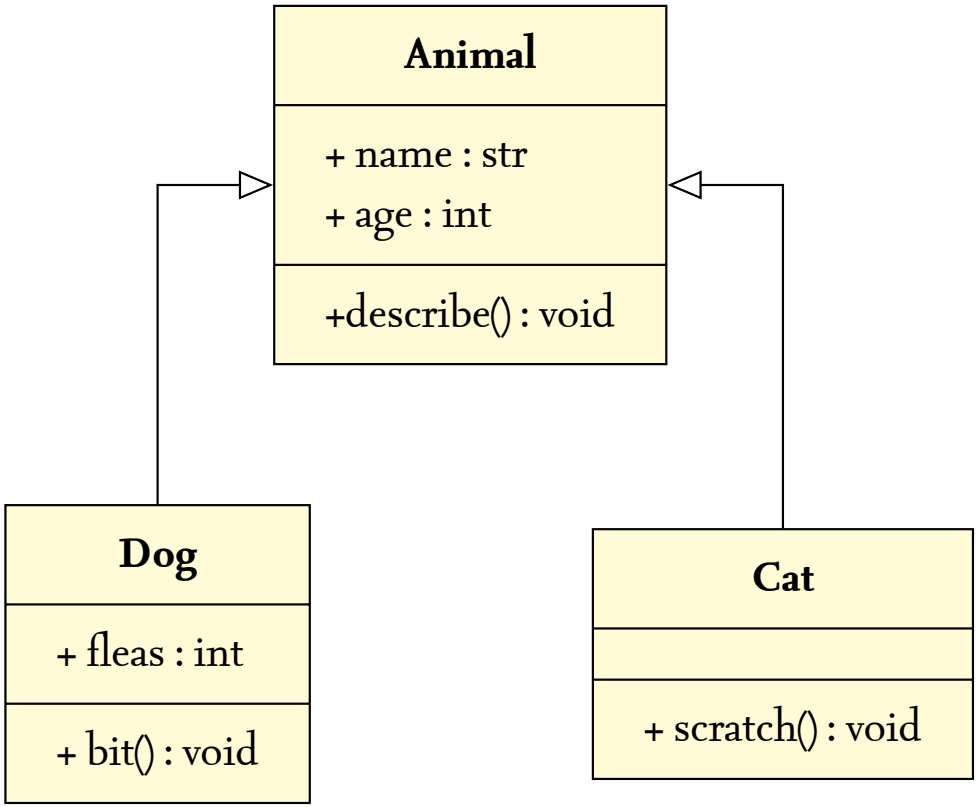
Characteristics

Examples of simple inheritance (I)



Characteristics

Examples of simple inheritance (II)



```
class Animal:
    def __init__(self):
        self.name = "Unknown"
        self.age = 10

    def describe(self):
        print("Name: ", self.name)
        print("Age: ", self.age)

class Dog(Animal):
    def bit(self):
        print(self.name + " has bitten")

class Cat(Animal):
    def scratch(self):
        print(self.name + " has scratched")

if __name__ == '__main__':
    snoopy = Dog()
    garfield = Cat()

    snoopy.name = "Snoopy"
    garfield.name = "Garfield"

    snoopy.bit()
    garfield.scratch()

    garfield.bit() # Error!
```

(Source code)

Characteristics

Examples of simple inheritance (II)

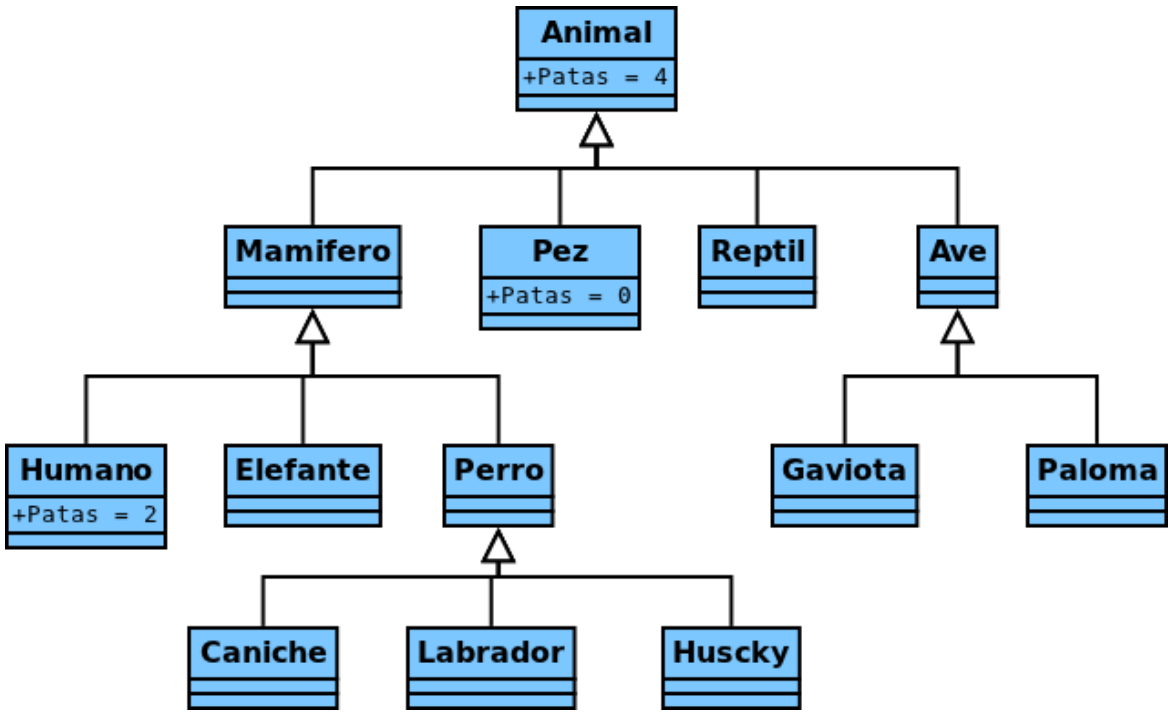


Figura 1: Example of simple Inheritance in OOP. Obtained from: `http://android.scenebeta.com`

Characteristics

Multiple inheritance

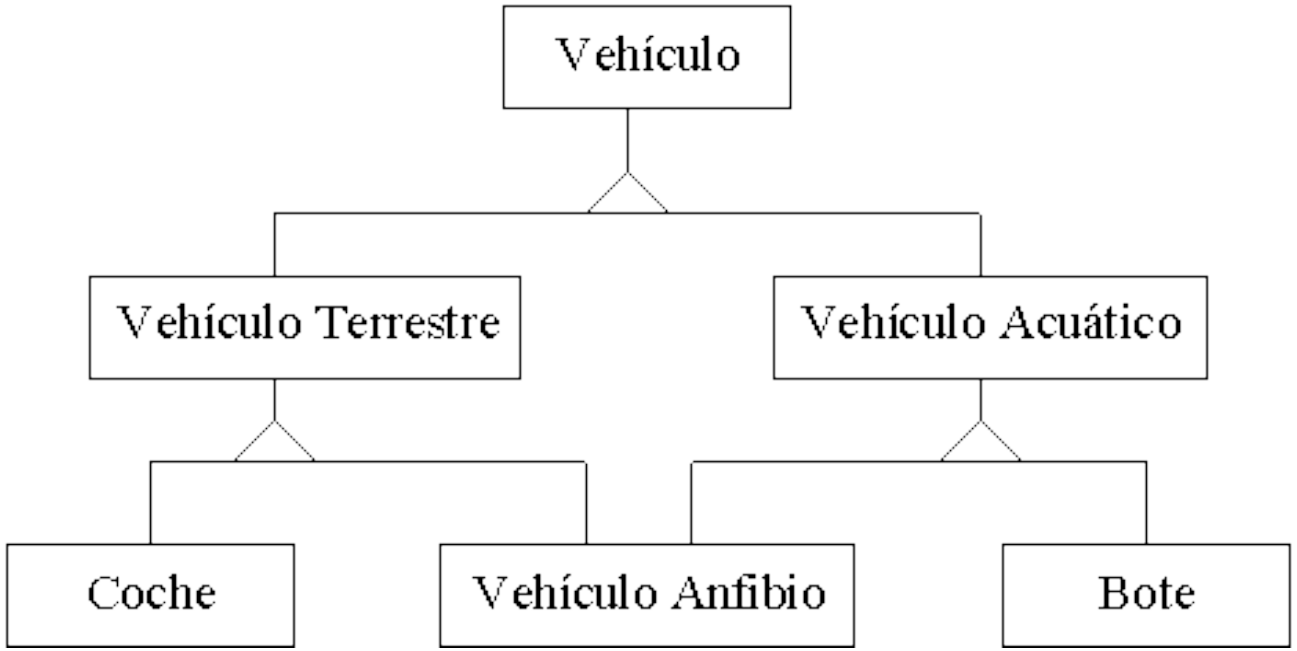


Figura 2: Example of multiple Inheritance in OOP. Obtained from: `http://www.avizora.com`

Characteristics

Polymorphism (I)

Polymorphism

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

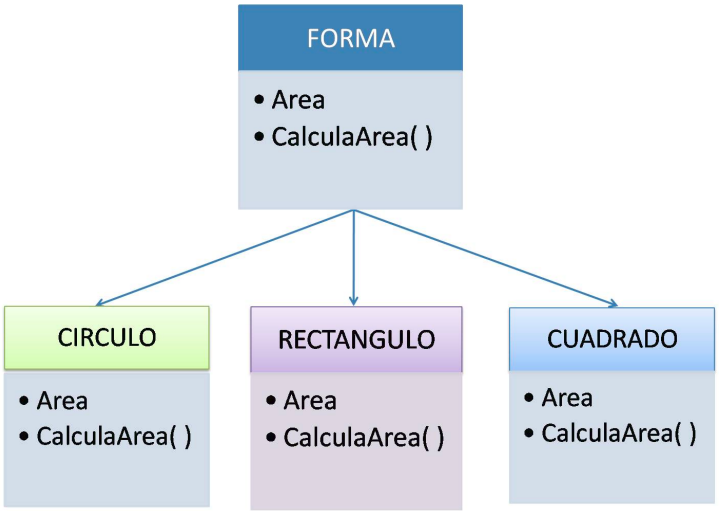


Figura 3: Example of polymorphism. Obtained from: `http://virtual.uaeh.edu.mx`

```
class Animal:
    def __init__(self):
        self.name = "Unknown"
        self.age = 10

    def describe(self):
        print("Name: ", self.name)
        print("Age: ", self.age)

    def attack(self):
        pass

class Dog(Animal):
    def attack(self):
        print(self.name + " has bitten")

class Cat(Animal):
    def attack(self):
        print(self.name + " has scratched")


if __name__ == '__main__':
    snoopy = Dog()
    snoopy.name = "Snoopy"
    garfield = Cat()
    garfield.name = "Garfield"

    for animal in (snoopy, garfield):
        animal.attack()
```

(Source code)

Characteristics

Abstraction and encapsulation (I)

Abstraction

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

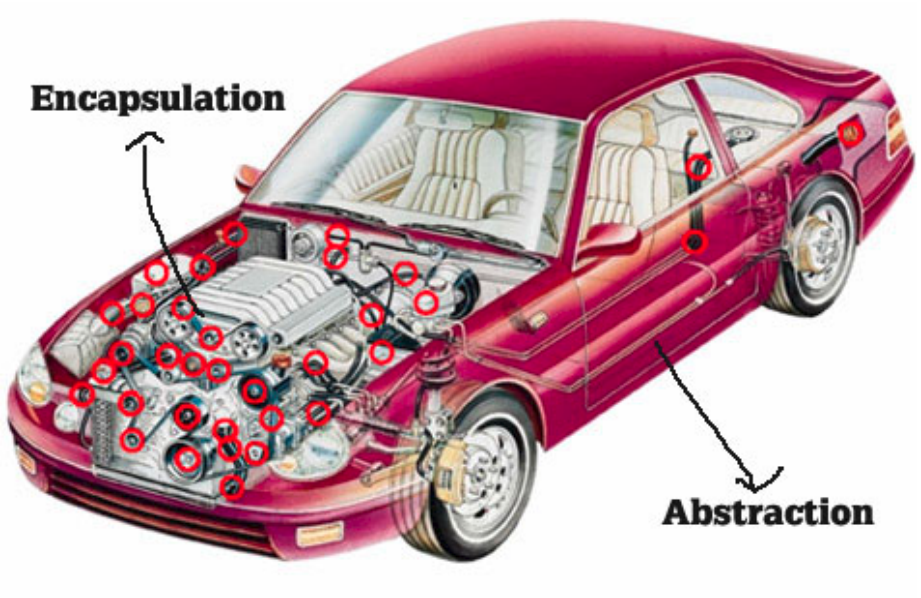
- A driver does not need to know how the carburetor works.
- To talk on the phone does not need to know how the voice is transferred.
- To use a computer do not need to know the internal composition of their materials.

Characteristics

Encapsulation (I)

Encapsulation

Mechanism use to provide an access level to methods and attributes for avoiding unexpected state changes



(Source)

Characteristics

Encapsulation (II)

The most common access levels are:

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

Methods geters and setters to control the access to attributes

```
class Dog:
    def __init__(self):
        self.__name = "Unknown"
        self.__age = 10

    def setName(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    def setAge(self, age):
        if age < 20:
            self.__age = age

    def getAge(self):
        return self.__age

if __name__ == '__main__':
    snoopy = Dog()
    snoopy.setName("Snoopy")
    print(snoopy.getName())

    print(snoopy.__name) # Error!
```

(Source code)

Classes in Python

Class objects

Two operations on classes

Attribute references

Accesses an attribute value
Standard dot syntax

```
obj.name
```

Example

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

Instantiation

Creates a new object
Standard functional notation

```
x = MyClass()
```

Example

```
time = Time()
```

Constructors (I)

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 can get arguments

Other special methods

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

- `__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`.

Overriding methods (I)

Often we need to adapt an inheritanced method: Overriding

Overriding example

```
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 possible to get superclass' method with `super()`

`super()` example

```
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()
```

Exercise statement

`Animal` class

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

Solved exercise

Animal class

animales.py

```
class Animal:
  #Constructor de la clase.
  def __init__(self,tipo,patas):
    self.tipo = tipo
    self.patas = patas
  #Metodos get de la clase Animal.
  def getTipo(self):
    return self.tipo
  def getPatas(self):
    return self.patas

#Instancias de los animales.
snoopy = Animal('Perro',4)
gatoComun = Animal("Gato",4)
#Impresion por pantalla.
print snoopy.getTipo()
print gatoComun.getPatas()
```

Solved exercise

`Animal` class

1. Create a `gato` class in the same file which inherits from the `animal` class.
2. Create the constructor and add the `sonido` attribute.
3. Create the method `maullar` which prints the sound MIAU.
4. Create a instance and check the methods.

Solved exercise

Class Animals

animales.py

```
#Gato hereda de animal
class Gato(Animal):
  #Constructor de la clase. Llama al constructor de Animal
  def __init__(self,patas):
    Animal.__init__(self,"Gato",patas)
    self.sonido='miau'
  #Metodos propios de la clase gato.
  def maullar(self):
    print self.sonido
#Instancias de los gatos.
gatoConBotas = Gato(2)
#Impresion por pantalla.
gatoConBotas.maullar()
print gatoConBotas.getTipo()
```

Exercise statement

Class `Parcela`

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

Solved exercise

Class Parcela

claseparcela.py

```
# clase a ser utilizada desde otros scripts
class Parcela(object):
    def __init__(self, uso_suelo, valor):
        # inicializar objetos de esta clase: constructor
        self.uso_suelo = uso_suelo
        self.valor = valor

    def valoracion(self):
        # residencia unifamiliar: RU
        if self.uso_suelo == "RU":
            tasa = 0.05
        # residencia multifamiliar: RU
        elif self.uso_suelo == "RM":
            tasa = 0.04
        else:
            tasa = 0.02
        valoracion = self.valor * tasa
        return valoracion
```

Solved exercise

Use of Parcela

tasaparcela.py

```
import claseparcela


miparcela = claseparcela.Parcela("RM", 100000)

# una vez creada una instancia, se pueden usar
# las propiedades y metodos del objeto
print ("Uso del suelo: ", miparcela.uso_suelo)
mitasa = miparcela.valoracion()
print (mitasa)
```

Source

Solved exercise. Serializando objetos `Parcela`

tasaparcela_pickle.py

```
import pickle
import claseparcela

miparcela = claseparcela.Parcela("RM", 100000)
mitasa = miparcela.valoracion()
print (mitasa)

print("Serializamos el objeto: \n", miparcela)
fout = open("parcelas.db", 'wb')
pickle.dump(miparcela, fout)
fout.close()

fout = open("parcelas.db", 'rb')
miparcelaout = pickle.load(fout)
fout.close()

print("Objeto leido: \n", miparcelaout)
print ("Uso del suelo: ", miparcelaout.uso_suelo)
mitasa2 = miparcelaout.valoracion()
print (mitasa2)
```

Exercise statement

`Rio` class

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

Exercise statement

Establishment of hierarchies from `Rio` class

1. Add to the `Rio` class the attribute `caudal` and the method `trasvasar` which receives two rivers and transfers 5 liters from the first to the second.
2. Create the `Afluente` class which inherits from `Rio`.
3. Create the method `__init__` of `Afluente` which initializes its `nombre` and `longitud` and, also, `afluenteDeRio`, 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...

Aprende más: [4]

Bibliographic references I


[1] Lenguajes de programacion, capítulo 1.
Lenguajes de programacion.
`http://rua.ua.es/dspace/bitstream/10045/4030/1/tema01.pdf`


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


[3] G. van Rossum, Jr. Fred L. Drake.
Python Tutorial Release 3.2.3, chapter 9.
Python Software Foundation, 2012.


[4] Dusty Phillips
Python 3 Object Oriented Programming.
Packt Publishing, 2010.