Object-Oriented Programming in Python

Inteligencia Artificial en los Sistemas de Control Autónomo
Máster en Ciencia y Tecnología desde el Espacio

Departamento de Automática

Universidad de Alcalá
ISG

Objectives

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

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 [?].

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.

- **Structural**: is based on nesting, loops, conditionals and subroutines. `GOTO` command is forbidden (C, Pascal, Python).
  - Example: *reviewing products of a shopping list and add the item X to the shopping if it is available.*
- **Object-Oriented Programming**

Programming paradigms types (III)

Object-Oriented Programming

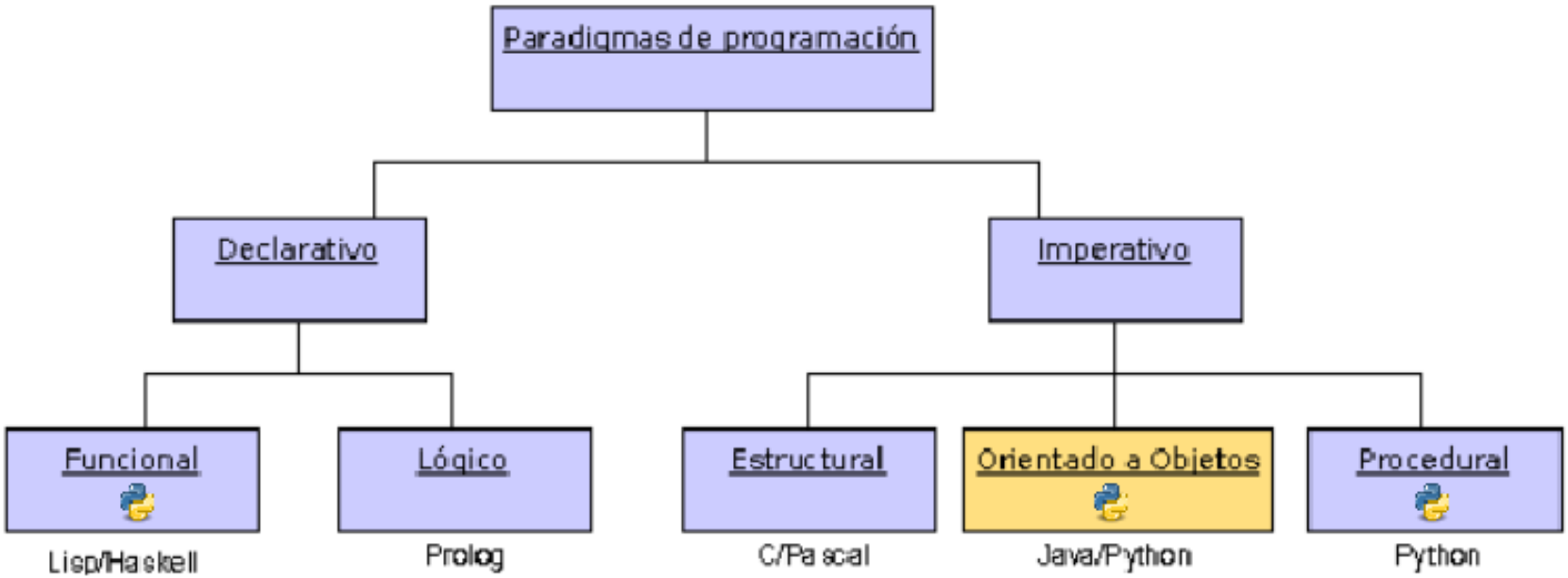
Evolves from imperative programming. It is based on objects that allow express the characteristics and behavior in a closer way to real life (Java, Python, C++).

- **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

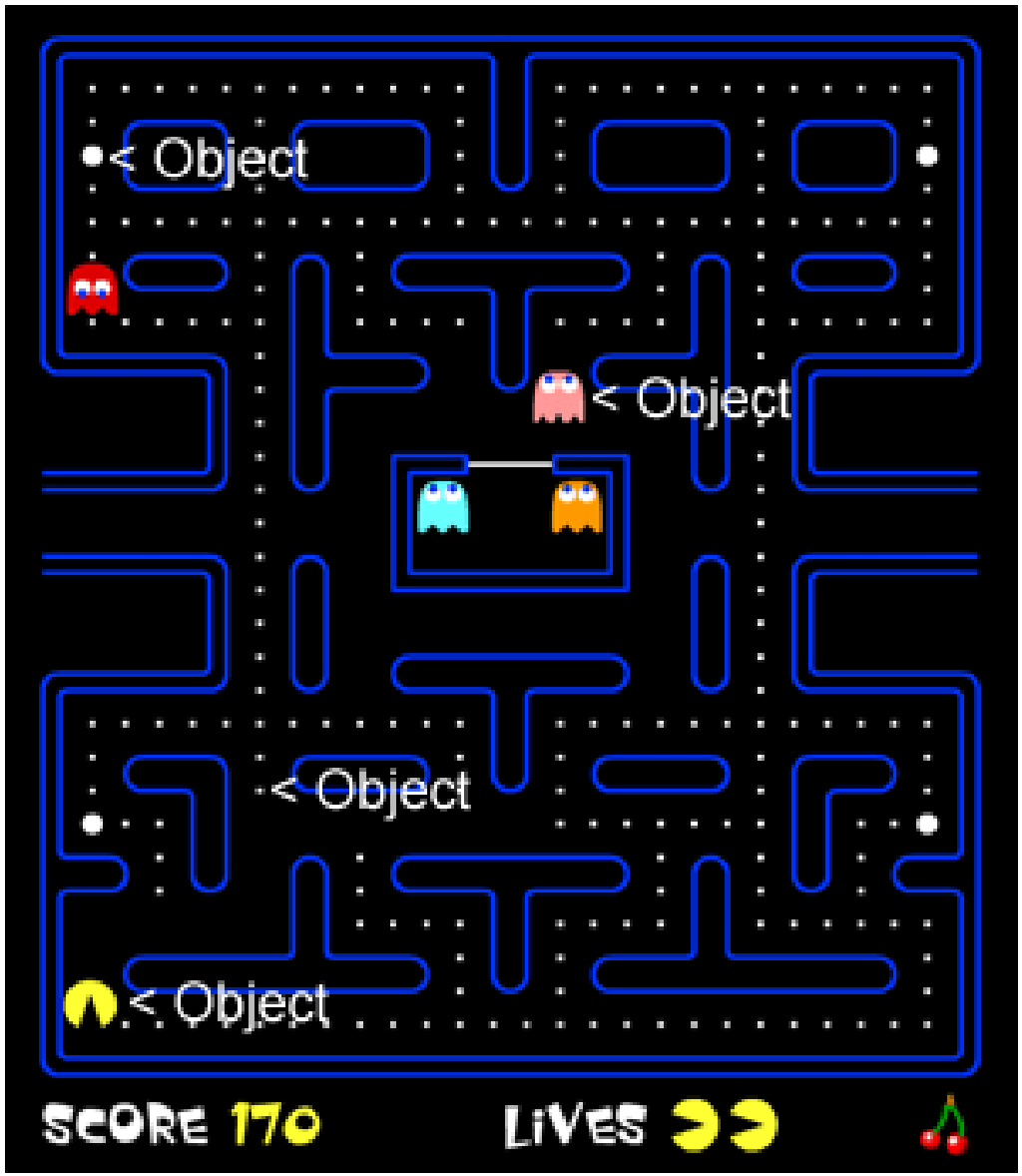
Concepts (IV)

Object or instance

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

Object-Oriented Programming

Concepts (V)



(Source)

Object-Oriented Programming

Concepts (VI)

Two operations on classes

Instantiation

Creates a new object
Standard functional notation

```
x = MyClass()
```

Example

```
time = Time()
```

Attribute references

Accesses an attribute value
Standard dot syntax

```
obj.name
```

Example

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

Object-Oriented Programming

Constructors (I)

Constructor

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

Concepts of OOP

Constructors (II)

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

Object-Oriented Programming

Constructors (III)

```
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()
```

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()
```

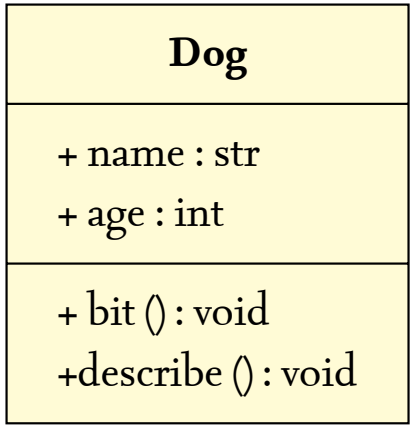
Output

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

Name:  Laika
Age:  10
```

(Source code)

UML class diagram



Object-Oriented Programming

Game example

Inheritance

Definition

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

Inheritance

Types of inheritance

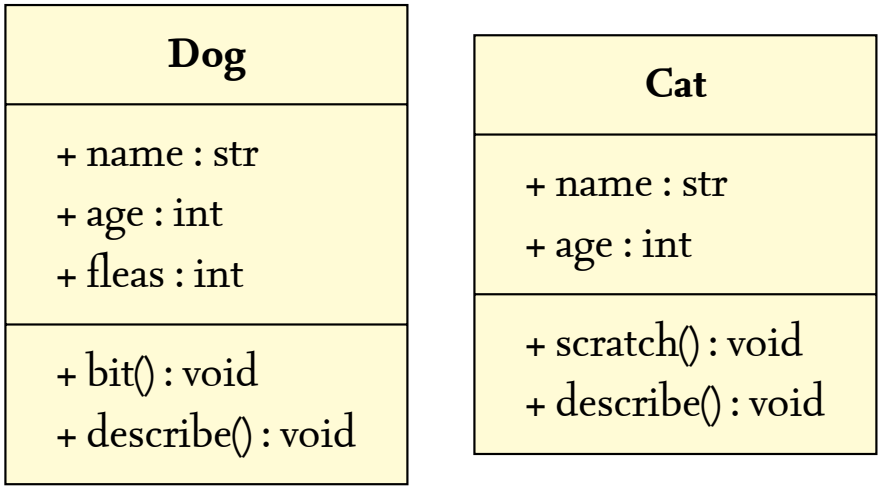
Types of inheritance

- 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.

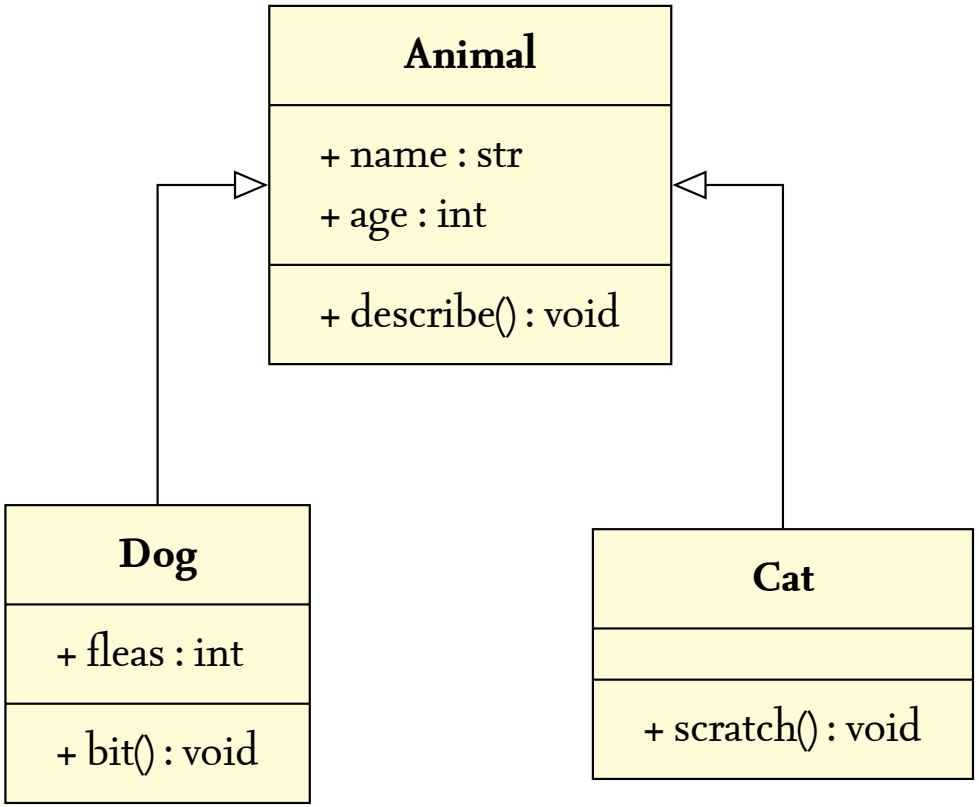
Inheritance

Examples of simple inheritance (I)



Inheritance

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)

Inheritance

Examples of simple inheritance (III)

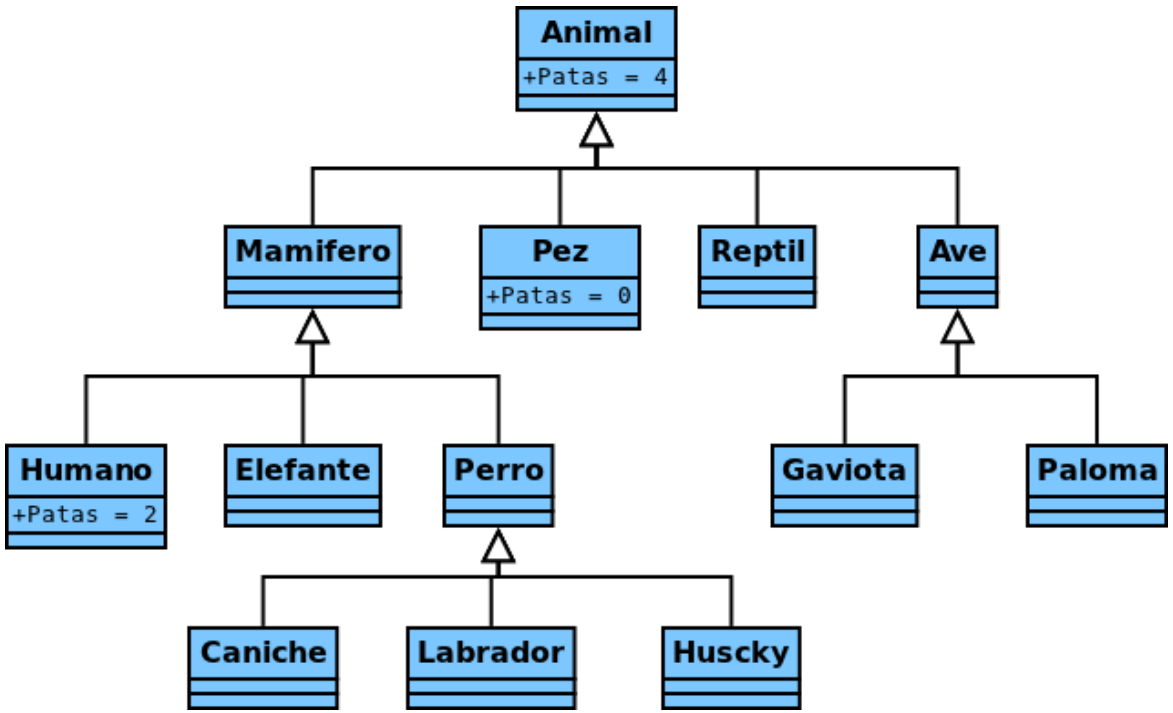


Figura 1: Example of simple Inheritance in OOP. Obtained from: `http://android.scenebeta.com`

Inheritance

Example of multiple inheritance

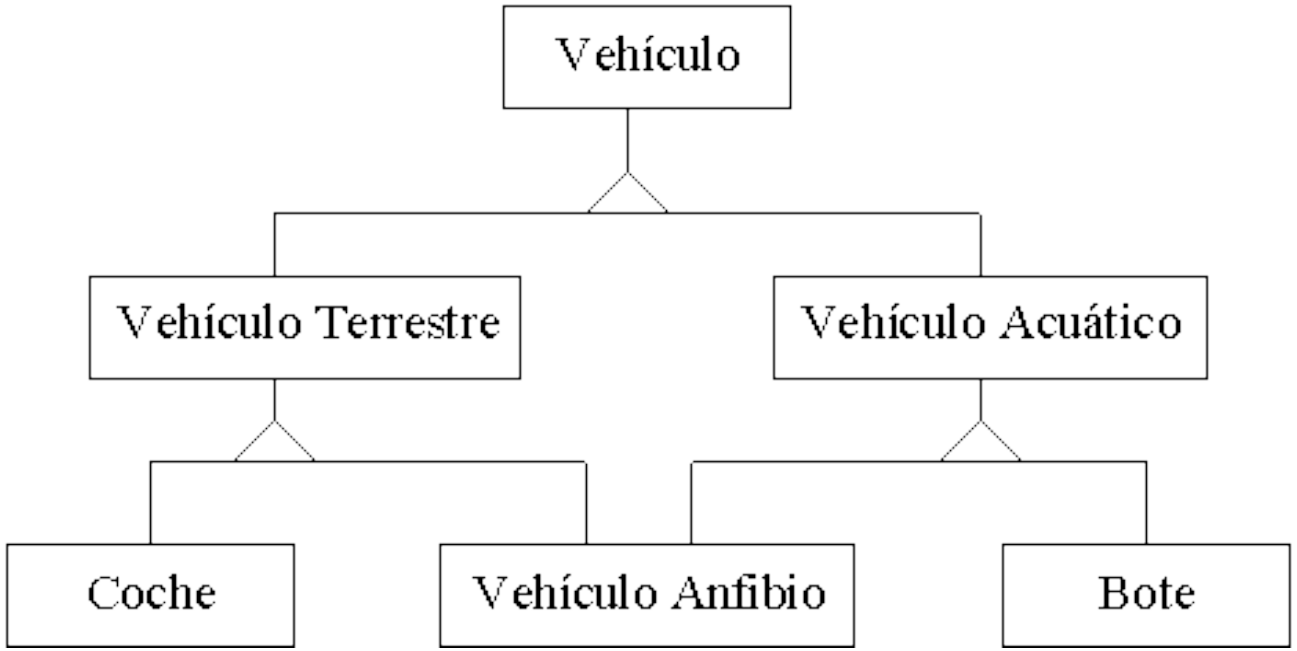


Figura 2: Example of multiple Inheritance in OOP. Obtained from: `http://www.avizora.com`

Concepts of OOP

Polymorphism (I)

Polymorphism

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

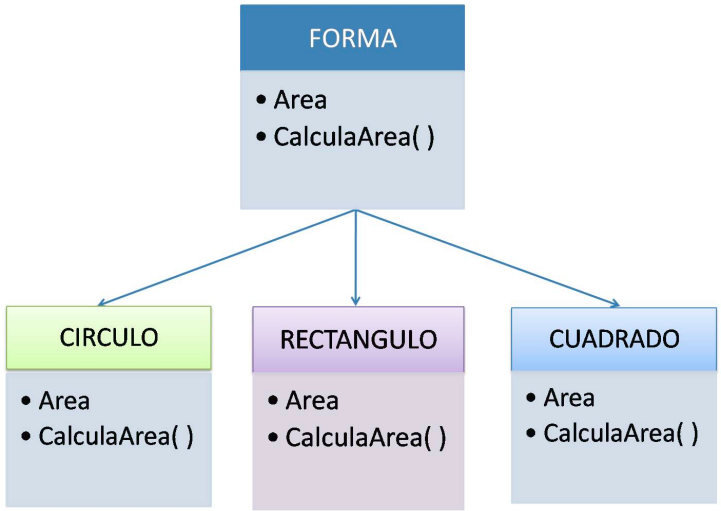


Figura 3: Example of polymorphism. Obtained from: `http://virtual.uaeh.edu.mx`

```
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)

Concepts of OOP

Abstraction

Abstraction

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

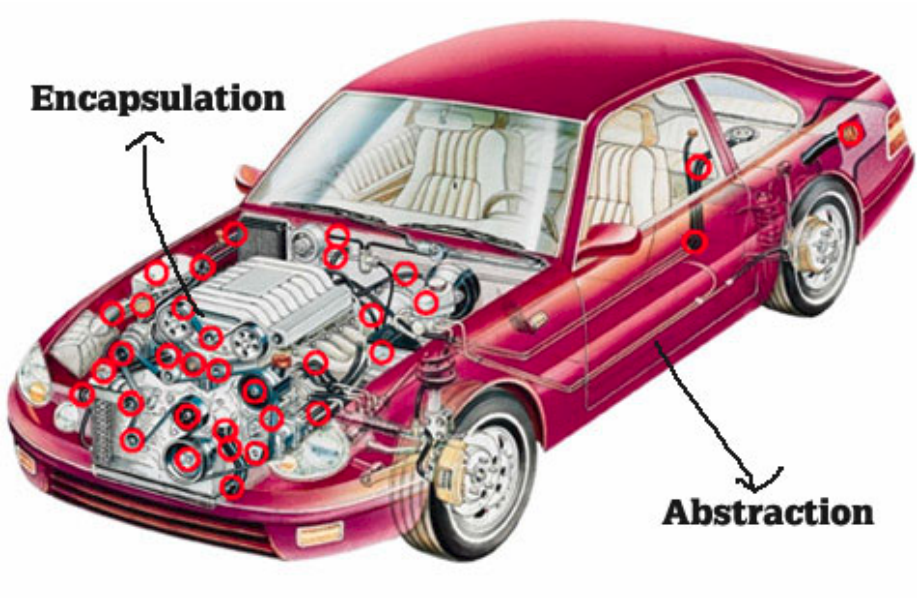
- A driver does not need to know how the carburetor works.
- To talk on the phone does not need to know how the voice is transferred.
- To use a computer do not need to know the internal composition of their materials.

Concepts of OOP

Encapsulation (I)

Encapsulation

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



(Source)

Concepts of OOP

Encapsulation (I)

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 [does 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)

Concepts of OOP

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`.

Concepts of OOP

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()
```

Concepts of OOP

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()
```

```
import arcade

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600

class MyGame(arcade.Window):
    """ Our Custom Window Class"""

    def __init__(self):
        """ Initializer """

        # Call the parent class initializer
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Lab 7 -
            User Control")

    def on_draw(self):
        arcade.start_render()


def main():
    window = MyGame()
    arcade.run()


main()
```

```
import arcade

class MyGame(arcade.Window):
    def __init__(self, width, height, title):
        super().__init__(width, height, title)

        arcade.set_background_color(arcade.color.ASH_GREY)

        self.ball_x = 50
        self.ball_y = 50

    def on_draw(self):
        arcade.start_render()

        arcade.draw_circle_filled(self.ball_x, self.ball_y, 15,
            arcade.color.AUBURN)

    def update(self, delta_time):
        self.ball_x += 1
        self.ball_y += 1

def main():
    window = MyGame(640, 480, "Drawing Example")
    arcade.run()

main()
```

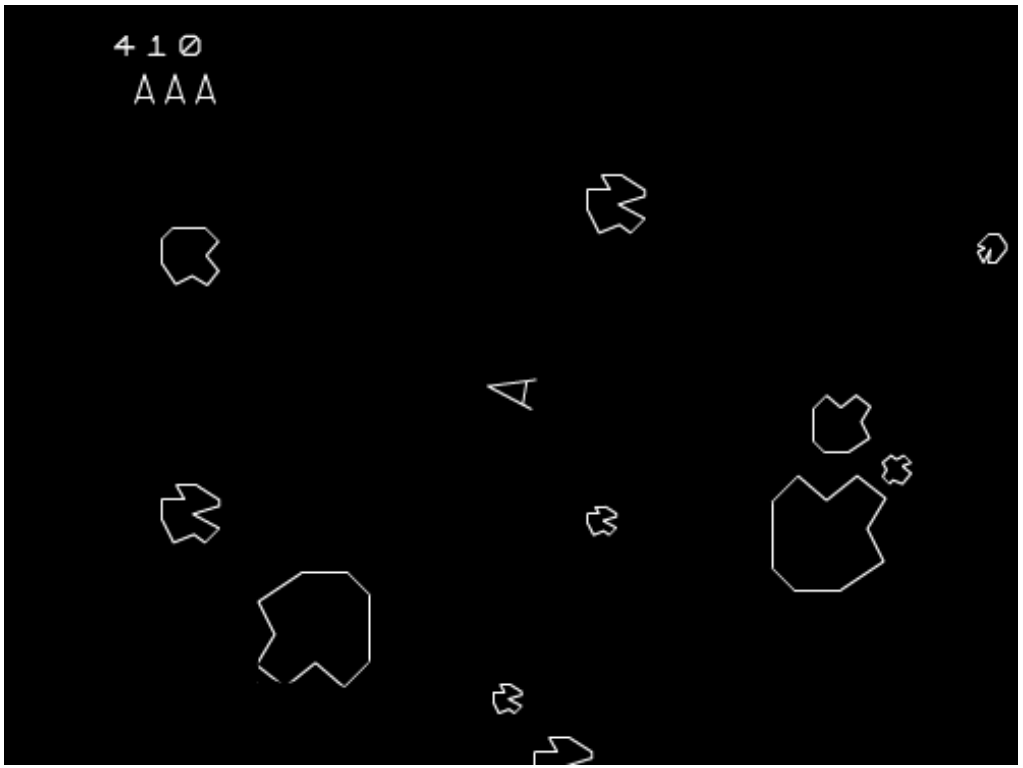
Arcade

The `arcade.Window` class.

- `on_draw()`. Override this function to add your custom drawing code
- `on_update(delta_time: float)`. Move everything. Perform collision checks. Do all the game logic here
- `on_key_release(symbol: int, modifiers: int)`
- `on_mouse_release(x: float, y: float, button: int, modifiers: int)`. Override this function to add mouse button functionality
- `set_viewport(left: float, right: float, bottom: float, top: float)`. Set the coordinates we can see

Check out (reference documentation)

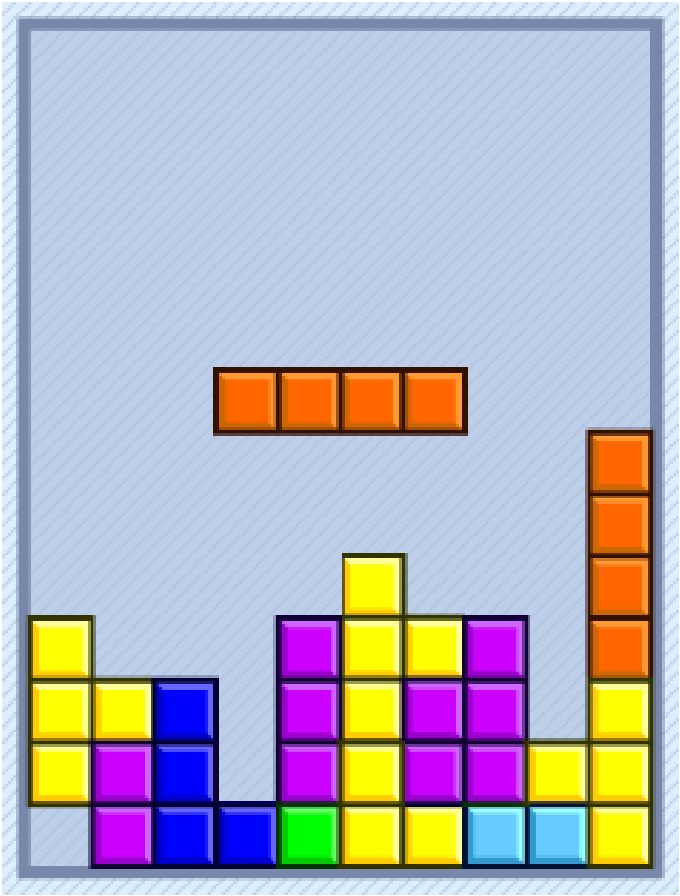
Exercise 1: Asteroids



(Source)

1. Identify the classes in the Asteroids videogame
2. Identify attributes contained in the previous classes
3. Identify methods contained in the previous classes

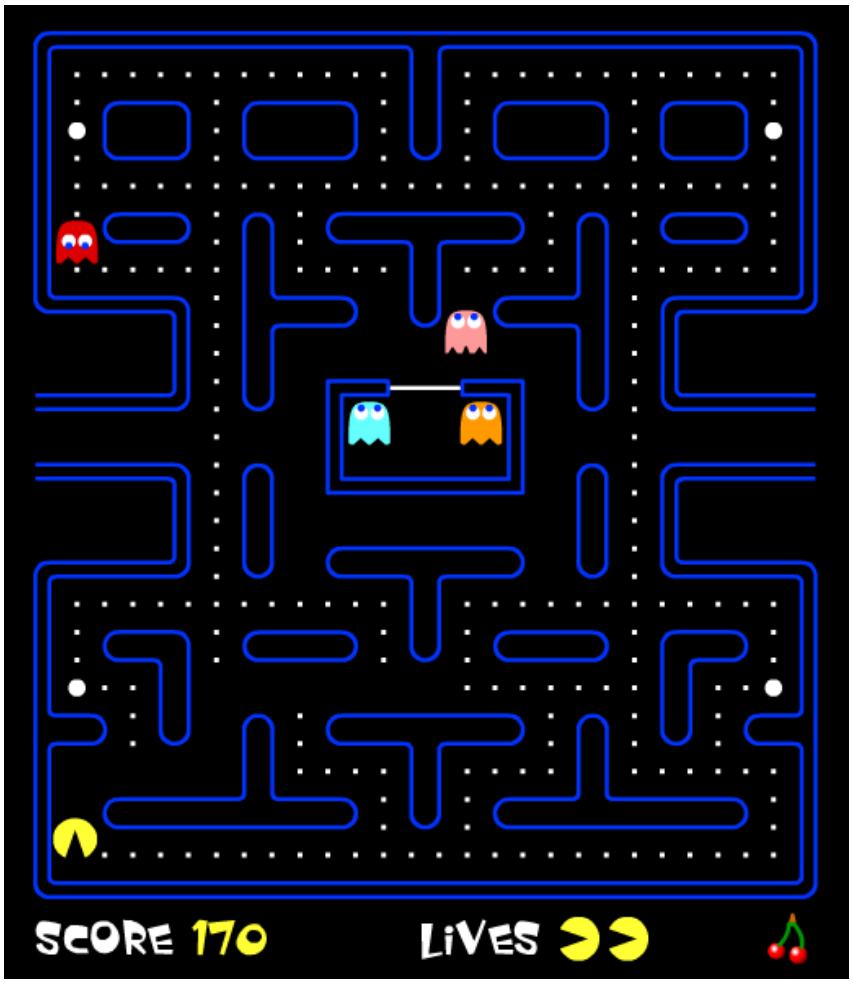
Exercise 2: Tetris



(Source)

1. Identify the classes in the Tetris videogame
2. Identify attributes contained in the previous classes
3. Identify methods contained in the previous classes

Exercise 3: Pac-Man



(Source)

1. Identify the classes in the Pac-Man videogame
2. Identify attributes contained in the previous classes
3. Identify methods contained in the previous classes