Object-Oriented Programming in Python

Videogames Technology
Escuela Politécnica Superior

Departamento de Automática

Universidad
de Alcalá

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: Arcade

Table of Contents

1. Programming paradigms
   - Understanding concepts
   - Programming paradigms types
2. Object-Oriented Programming
   - Objectives
   - Basic concepts
   - Constructors
   - OO game example
3. Inheritance
   - Definition
   - Examples
   - Types of inheritance
4. Concepts of OOP
   - Polymorphism
   - Encapsulation
   - Encapsulation: The “pythonic” way
   - More about methods
   - Overriding methods
5. Arcade
6. Exercises
   - Exercise 1: Asteroids
   - Exercise 2: Tetris
   - Exercise 3: Pac-Man

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.

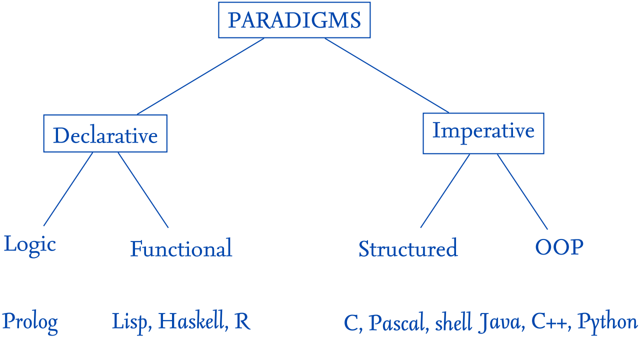
- **Structural**: is based on nesting, loops, conditionals and subroutines. (C, Pascal, Python).
  - Example: *reviewing products of a shopping list and add the item X to the shopping if it is available.*
- **Object-Oriented Programming**: is based on objects and classes (C++, Java, Python)

Arcade supports both paradigms

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

Programming paradigms types (IV)

Classification



PARADIGMS

Declarative

Imperative

Logic

Functional

Structured

OOP

Prolog

Lisp, Haskell, R

C, Pascal, shell Java, C++, Python

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

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.

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

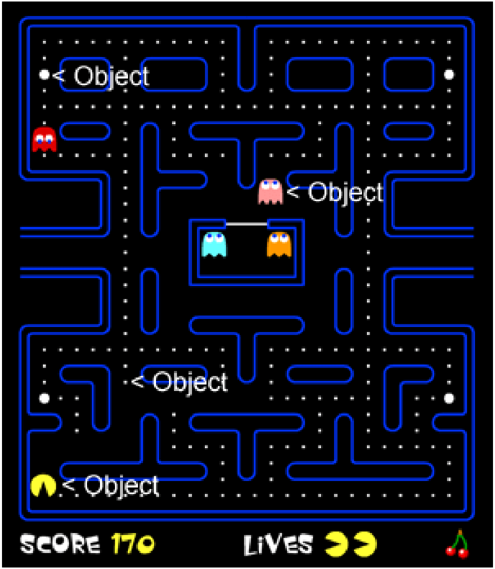
Concepts (IV)

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

```
>>> snoopy = Dog()
>>> laika = Dog("Laika")
```

Attribute references

Accesses an attribute value

Standard dot syntax

```
obj.name
```

Example

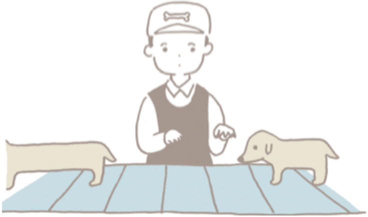
```
>>> snoopy.name = "Snoopy"
>>> print(snoopy.name)
>>> name = snoopy.name
```

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

dog.py

```python
class Dog:
    def __init__(self, name="Unknown", age=0):
        # Constructor
        self.name = name      # Attribute
        self.age = age        # 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("Laika")
    # snoopy and laika are objects

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

    snoopy.bit()
    snoopy.describe()

    print() # Print empty line
    laika.age = 10
    laika.describe()
```

Output

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

Name:  Laika
Age:  10
```

(Source code)

dog.py

```python
class Dog:
    def __init__(self, name="Unknown", age=0):
        # Constructor
        self.name = name       # Attribute
        self.age = age         # 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("Laika")
    # snoopy and laika are objects

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

    snoopy.bit()
    snoopy.describe()

    print() # Print empty line
    laika.age = 10
    laika.describe()
```

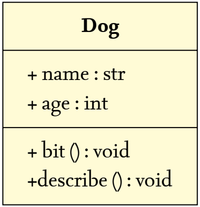
Output

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

Name:  Laika
Age:  10
```

(Source code)

UML class diagram



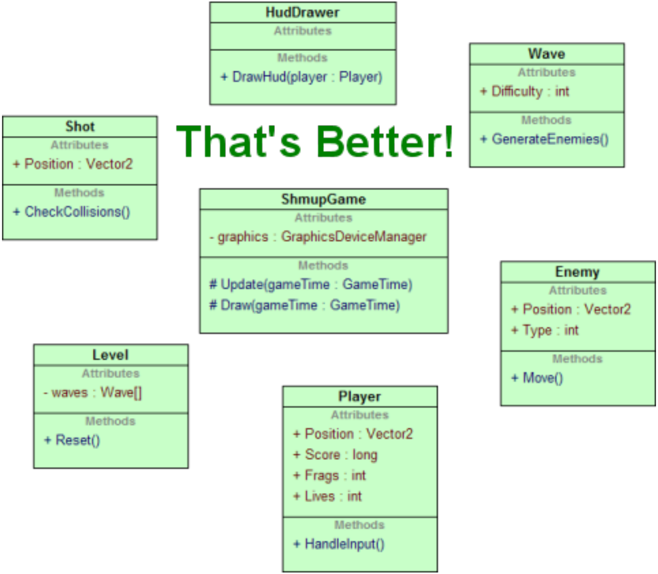
Dog

+ name : str
+ age : int

+ bit () : void
+describe () : void

Object-Oriented Programming

OO game example



That's Better!

HudDrawer
Attributes
Methods
+ DrawHud(player : Player)

Wave
Attributes
+ Difficulty : int
Methods
+ GenerateEnemies()

Shot
Attributes
+ Position : Vector2
Methods
+ CheckCollisions()

ShmupGame
Attributes
- graphics : GraphicsDeviceManager
Methods
Update(gameTime : GameTime)
Draw(gameTime : GameTime)

Enemy
Attributes
+ Position : Vector2
+ Type : int
Methods
+ Move()

Level
Attributes
- waves : Wave[]
Methods
+ Reset()

Player
Attributes
+ Position : Vector2
+ Score : long
+ Frags : int
+ Lives : int
Methods
+ HandleInput()

(Source)

Inheritance

Definition

Inheritance

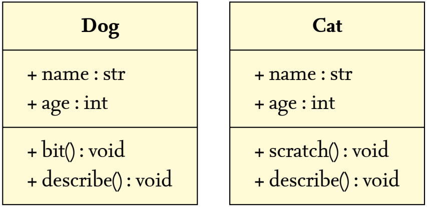
Mechanism of reusing code in OOP. Consists of generating child classes from other existing (super-class) allowing the use and adaptation of the attributes and methods of the parent class to the child class

A subclass inherits all the attributes and methods from its superclass

- *Superclass*: "Father" of a class
- *Subclass*: "Child" of a class

Inheritance

Examples of simple inheritance (I)



Dog

+ name : str
+ age : int

+ bit() : void
+ describe() : void

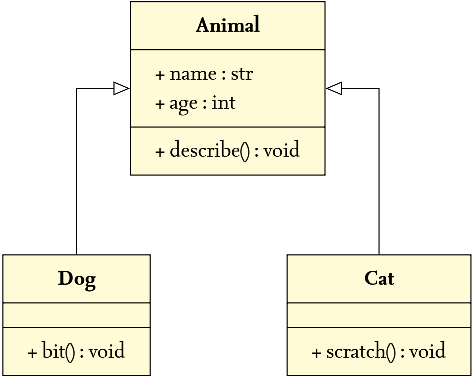
Cat

+ name : str
+ age : int

+ scratch() : void
+ describe() : void

Inheritance

Examples of simple inheritance (II)



dog.py

```python
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)

Class hierarchy: A set of classes related by inheritance

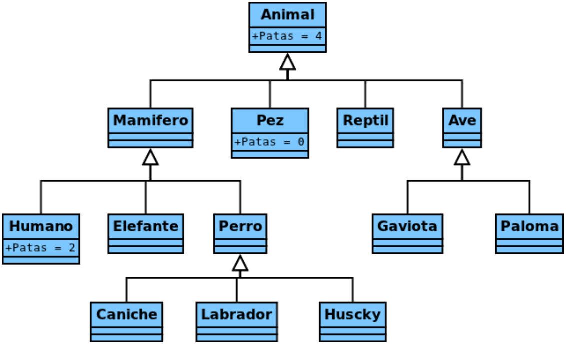
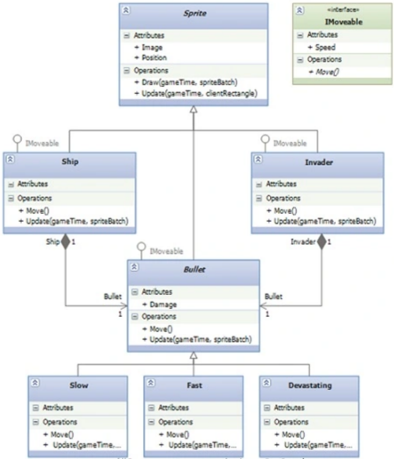


Figura 1: Example of simple Inheritance in OOP. Obtained from: `http://android.scenebeta.com`

Inheritance

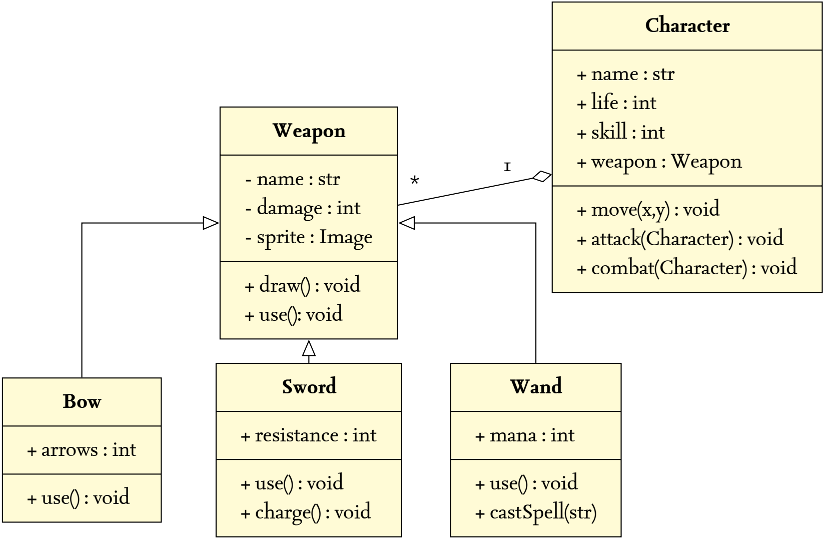
Examples of simple inheritance (IV)



("Comments on my design v3 - Stack Overflow," 2011)

Inheritance

Examples of simple inheritance (V)



Character

+ name : str
+ life : int
+ skill : int
+ weapon : Weapon

+ move(x,y) : void
+ attack(Character) : void
+ combat(Character) : void

Weapon

- name : str
- damage : int
- sprite : Image

+ draw() : void
+ use(): void

Bow

+ arrows : int

+ use() : void

Sword

+ resistance : int

+ use() : void
+ charge() : void

Wand

+ mana : int

+ use() : void
+ castSpell(str)

Inheritance

Types of inheritance (I)

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

Types of inheritance (II)

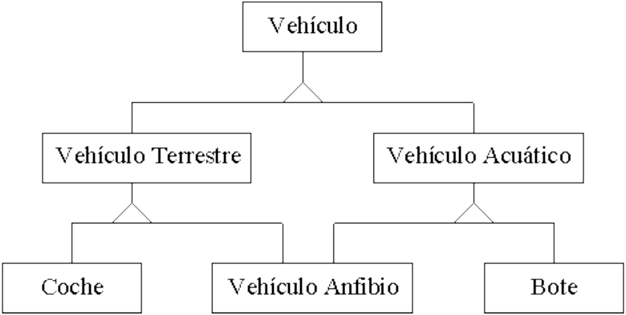


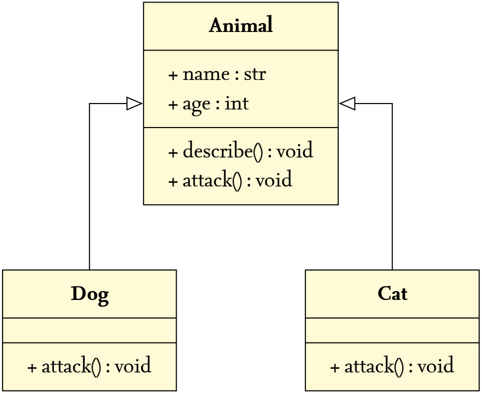
Figura 2: Example of multiple Inheritance in OOP. Obtained from: `http://www.avizora.com`

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.



```python
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()
```

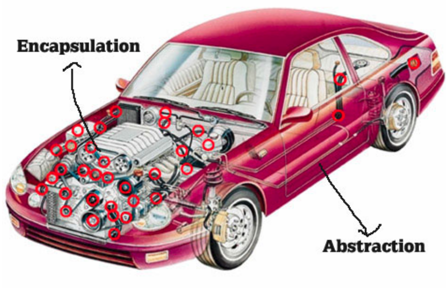
(Source code)

Concepts of OOP

Encapsulation (I)

Encapsulation

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



(Source)

Concepts of OOP

Encapsulation (II)

The most common access levels are:

- **public**: visible for everyone , default in Python
- **private**: visible for the class
  - By *convenion* in Python, starts with one underscore
  - 'Superprivate' starts with a double underscore and does not end in the same manner
- **protected**: visible for the creator class and its descendents [*it does not exist in Python*]

"Getters" and "setters" methods to control the access to attributes

```python
class Dog:
    def __init__(self, name="Unknown", age=10):
        self._name = name
        self.__age = age

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

    def getName(self):
        return self._name

    def setAge(self, age):
        if age > 0:
            self.__age = age

    def getAge(self):
        return self.__age

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

    snoopy._name = "Laika" # No error, but please, DON'T do this

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

Concepts of OOP

Encapsulation: The “pythonic” way (I)

Getters and setters come with some drawbacks

- Verbose and repetitive code
- Linked to the API

A more pythonic way to define getters and setters is using properties

- It is a decorator that transforms methods into getters or setters

  @**property**: Getter

  @**object.setter**: Setter

- Neverless, getters and setters are still used in Python under certain circumstances

Better with an example ...

Concepts of OOP

Encapsulation: The “pythonic” way (II)

```python
class Dog:
    def __init__(self, name="Unknown", age=0):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name=name.upper()


if __name__ == '__main__':
    snoopy = Dog()
    snoopy.name = "Snoopy" # Calls setter

    print(snoopy.name)     # Calls getter
                           # prints 'SNOOPY'
```

Concepts of OOP

Other special methods

Several special methods, including:

- `__str__(self)` It should return a string with information
- `__len__(self)` It should return the length or “size” of object (number of elements if is a set or *queue*)

```
class Inventory:
    def __init__(self, items=[]):
        self._items = items

    def __str__(self):
        return ': '.join(self._items)

    def __len__(self):
        return len(self._items)

if __name__ == '__main__':
    inventory = Inventory(["map", "key"])
    print(inventory)          # Outputs "map: key"
    print(len(inventory))     # Outputs "2"
```

Concepts of OOP

Overriding methods (I)

Often we need to adapt an inheritanced method: Overriding

Overriding example

```python
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()`

Overriding 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()
```

```python
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, "My Game")

    def on_draw(self):
        self.clear()


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


main()
```

```python
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):
        self.clear()

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

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

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

main()
```

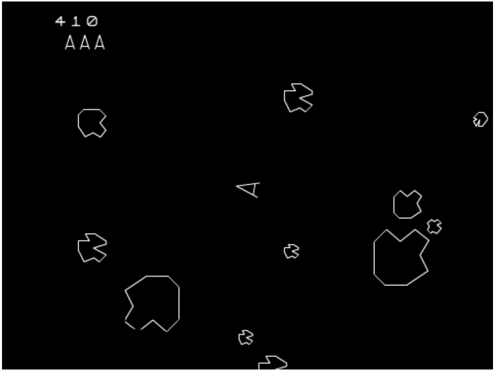
Arcade

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

Check out (reference documentation)

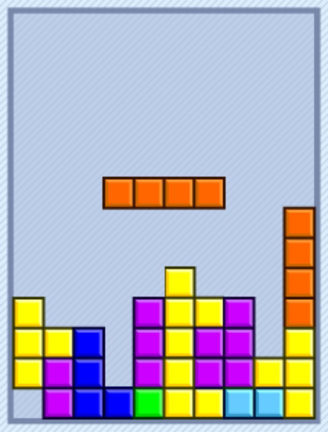
Exercise 1: Asteroids



(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

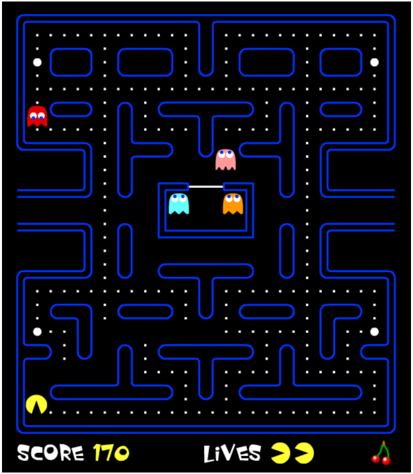
Exercise 2: Tetris



(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