Object Oriented Programming **OOP**

Using python.

Iván Andrés Trujilllo Abella

Facultad de Ingenieria
Pontificia Universidad Javeriana

trujilloiv@javeriana.edu.co

Write code in python

PEP8

- four spaces for indentation
-

Object Oriented Programming

OOP

There are important concepts classes and objects.

E

ach object is a instance of a class.

Blueprint - class

The classes are blueprint.

OOP

features

- Inheritance
- Cohesion
- Abstraction
- Polymorphism
- Encapsulation

Encapsulation

Hide code

polymorphism

pep8

important recomendations

- 4 spaces to indent
- a length of maximum 79 characters per line

break before binary operators

pep8

```
result = first_value
+ second_value
+ third value
```

import statement

always head of sheet, in the following order.

```
standard library imports
third import
local
```

we need add a blank line separating each one.

```
import os
import sys

import numpy
import pandas

import module_statistics
```

pep8

```
lista = [
        1,2,3,
]
```

Classes

```
class ClassName:
    pass
```

another

PEP8

```
i = i + 1
update += 1
result = first_value + second_value
```

dont use space in parameters by default.

```
def function(parameter_one, parameter_two=0.1)
```

classes

The name of classes uses CamelWord

```
import pandas as pd
pd.__doc__
```

Built in classes

The following are examples of **built-in classes**.

- int
- float
- str
- Bool
- list

__init__

initializer method

Also denominated as **constructor**

Cluster project

We can uses class to make a project about cluster, adding and removing features.

Methods

A method is similar to a function but belong to a instance of a class.

```
data =[]
data.append('value')
```

Special Methods

__init__ __str__ notice that have the adjective of method. Why?

Override Methods

suppose that you have two points Q and P, if we apply $+$ this will be produce a error then we need override it: this inside of the class definition of point

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __str__(self):
    return "Point({},{})".format(self.x,self.y)

  def __add__(self,another):
    return Point(self.x + another.x , self.y + another.y)

point_one = Point(10,11)
print(point_one)
print(point_one + point_one)
```

define a class for cluster. define a origin define a belonging to cluster define a linkage method.

Parent class

change attributes value

```
class Point:
    def __init__(self, atributteOne,atributteTwo):
    self.atributteOne= atributteOne
    self.atributteTwo= atributteTWo
```

Without the __init__ method we have change attributes values using dot notation each time that we create a instance. **self** made reference to the id or name variable of the instance. **self** no it is a compulsory or keyword only is a agreement. You can think in **self** as a placeholder.

Attributes in the instances are known as instance variables.

Docstring

```
class Point:
    """ This could be a message to know
     what is the purpose of this class."""
    def __init__(self, atributteOne,atributteTwo):
    self.atributteOne= atributteOne
    self.atributteTwo= atributteTWo
```

Method

Methods are as functions, it allow us interact with the object.

```
class Point(object):
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def distance(self, sPoint):
    return (((self.x - sPoint.x)**2)
         + ((self.y - sPoint.y)**2))**(1/2)
```

Then the **distance methods** allow us to know the euclidean distance with another point.

```
pointOne= Point(0,0)
pointOne.distance(Point(0,0))
```

Inheritance

```
class metric(point):
    pass
```

In this sense in this case Point is object and inherits attributes. Then **point** it is superclass of metric.

Point class

Now we can create some instances of the **Point class**.

```
x = Point(10,4)
y = Point(11,12)
```

__str__ method

When you type:

```
Point(0,0)
```

this will return something like *<__main__ . Point object at 0x7fa9>* when the user invoked a **print()** statement over the object then call the **__str__ method**

Instance variables

live in a instance.

```
class Point():
    pass
point_one=Point()
point_one.x=1
point_one.y=-3
```

unlikewise to functions a method have at least one parameter.

```
class Point():
    getX(self):
        return self.x
point_one =  Point()
point_one.x=10
print(point_one.getX())
```

init method

hello

Searching instance variables and methods.

Ina hierarchical way, first search in the instance and after in the class.

returning instances

We can return a instance defining this in the class:

```
class Point:
    def __init__(self,...)
    """docstring"""
    def method(self,..):
    return Point(...)
```

sorting instances

For understand better this it is necessary check the concept of lambda, and the method key in list.


Pontificia Universidad
JAVERIANA
Bogotá

side effects

REST API

Inheritance

it is a useful concept to inherit attributes and methods from another class, for instance felines share mainly properties with cats therefore felines is a **super class** and cats is a **child class** or **subclass**.

```
class Feline:
    def __init__(self):
        self.tail=1
        self.fangs=1
    def hunting(self):
        print('i am hunter')
```

After all, every cat it is a feline:

```
class Cat(Feline):
  def __init__(self):
    Feline.__init__(self)
    self.domestic=1

gato = Cat()
```

overriding the parent method

Extending class

inherit a method of a parent class, but change its behavior. only need specify the behavior of method in subclass to override the parent class method.

test.testEqual function

```
import test
test.testEqual(function(...), expected_value)
```

we can uses

```
assert exp_one == exp__two
```

when both variables have different values, then a

side effects

Development

Testing classes

we can test the _ _ init _ _ method.

exceptions

StopIterationException

Errors

Syntax errors and exceptions(error in execution) the last are syntactically correct and the first not.
Exception is a object, with a description a traceback (where problem ocurred).

Kind of erors

Built-in Exceptions

- ZeroDivisionError
- NameError
- TypeError
- ValueError
- KeyError (to dictionary)
- IndexError (to list)

syntax

```
try:
#statement_to_try.

except:
#if statement_to_try fail then run statement_except.

else:
#if try is finished python search the else clause and excute
    else statement.

finally:
#python always executed this statement.
```

it is important to know that could appear several except blocks.

example

```
num = 1
den = 0
try:
  div= num/den
  num + den + rest
except:
  pass
```

example

```
num = 1
den = 0
try:
  div= num/den
except ZeroDivisionError:
pass
```

Now suppose that num is a **string** then another

Catch specific exceptions are better

Take into account that is better catch specific exceptions and not general to not hide other problems.

```
num = 1
for den in [1,'string',30,0]:
  try:
    result = num/den
    print(result)
  except TypeError:
    print('data contain string')
  except ZeroDivisionError:
    print('data contain a zero in denominator')
```

raise exception

```
if exp is not str:
    raise Exception('Not is a string')
```

Assert

This function could be useful to debugging code.

```
assert(condition), "If not met dislpay this message"
```

if the condition not is met this will produce a AssertionError and the program crash.

Encapsulation

Sometimes we need refer some attributes or methods that can not invoke outside of the self class, this privacy is guaranted with dundesercore

```
class information:
  def __init__(self,data, language):
    self.__info1=info1
    self.info2=info2
data=information('structured','SQL')
print(data.info2)
print(data.info1)
```

the last line will arise a exception due the attribute **info1** only must be accessed by the class self.

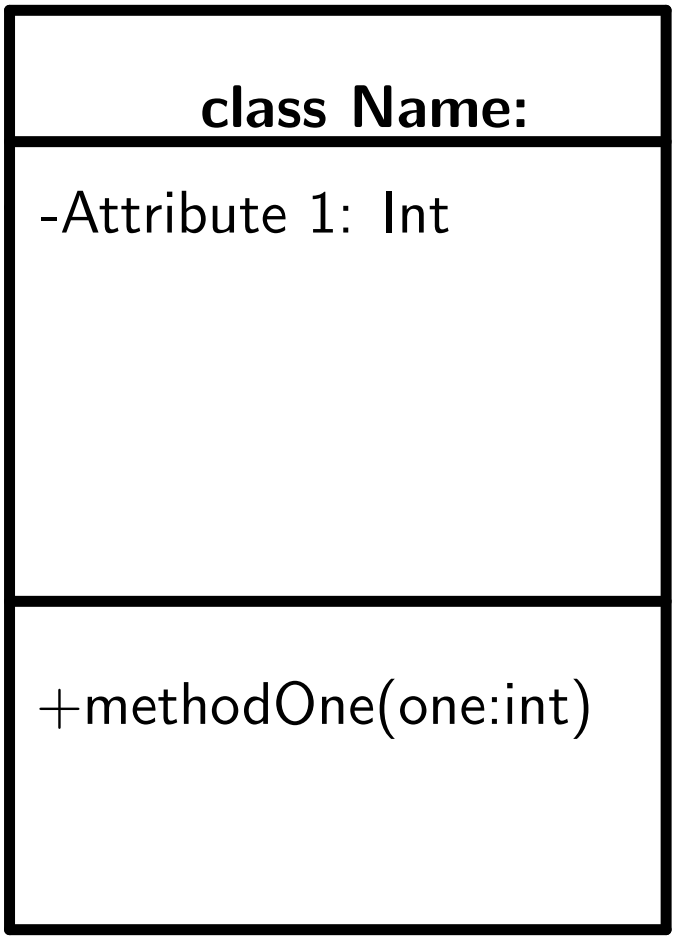
polymorphism

Think in that you can run over the items of a iterable object or sequence, list, tuple and dictionary. This means that a method is shared by two or more objects.
think for isntance that the + operator act over **integer, float, and both.**

Important things

- object based-modular structures
- data abstraction
- automatic memory management
- classes
- inheritance
- polymorphism
- dynamic binding
- multiple and repeat inhertiance

UML class




```
class Name:
    def __init__(self,attribute1):
    self.attribute1 = attribute1

    def methodOne(one):
        statements
```

builtin module

This module is loaded automatically for python and contain the classes for *int*, *str*, *float* and thus..

```
help(class)
```

Variables and functions

```
def count_vowels(iterable):
  vowel=0
  for i in iterable:
    if i in ['a','e','i','o','u']:
      vowel += 1
  return vowel

print(count_vowels('abcdefg'))
vowel= count_vowels
vowel('abcdefg')
```

Notice that, we save the function **count_ vowels** in the variable **vowel**.

nested functions

We must remember that we can define functions inside functions, that are not could called outside of the main function, for instance:

```
def main():
  def hello_world():
    print('hello world')
  hello_world()
main()
```

put if we try call this functions outside of main, this raise a exception (NameError).

returning functions

Remember that $\sum_{i=1}^{n} x_i = \frac{n(n+1)}{2}$ and $\sum_{i=1}^{n} x_i^2 = \frac{n(n+1)(2n+1)}{6}$ We can return functions with another function:

```
def suma(n,degree=1):
  print(f'the sum of the numbers in {degree}')
  def gauss():
    return n*(n+1)/2
  def square():
    return n*(n+1)*(n*2+1)/6
  if degree==1:
    return gauss
  else:
    return square
suma(10,2)()
suma(10,1)()
```

decoratores

used to add feature to a function, then we have:

```
def decorator(function_to_add_feature):
    def auxiliar function():
        statements
        function_to_add_feature():
    return auxiliar_function
```

put

What happen if we dont put super.__init__(self, attributeOne, attributeTwo). My idea is that the syntax: class Child _class(Parent_ class): only refer the class, but when you create a instance and require data for instance:

```
class Parent:
  information='this is for all parent classes'
  def __init__(self,data1,data2):
    self.data1=data1
    self.data2=data2
```

put

```
class Child(Parent):
  child_data = 'data_created in child'
```

Note that if you dont define a __init__ method automatically the instance objeto_child require the parameters defined in Parent class.

```
class Child(Parent):
  def __init__(self, argumento1, argumento2):
    self.argumento1= argumento1
    self.argumento2=argumento2
```

note that the child class have as input another variable names as parent and therefore this override the attributes for one.

Super.__ init__(self,..)

Part one

```
class Parent:
  attribute_One='loaded automatically to be loaded'
  tricky_attribute='this a tricky attribute'
  def __init__(self,parameter_One,parameter_Two):
    self.parameter_One= parameter_One
    self.parameter_Two= parameter_Two

class Child(Parent):
  def __init__(self,parameter_One,parameter_Two,
      parameter_Three):
    pass
    # Note that here we need to specify again
        self.parameter_One=paramter_One
    # Note that here we need to specify again self.parameter_Two
        = parameter_Two
```

Super.__ init__(self,..)

Part two

```
    #To avoid this we need only indicate that initialize the
        attributes in Parent class with
        Parent.__init__(self,parameter_One, Parameter_Two)

class Child(Parent):
  def __init__(self,parameter_One, parameter_Two,
      parameter_Three):
    self.parameter_Three= parameter_Three
    Parent.__init__(self,parameter_One,parameter_Two)

try1=Child('informationOne','informationTwo','InformationThree')
try1.parameter_Three
```

Super()

to invoke a function inside another subclass. the word **super()** indicate to python look up in the parent class. This also could be useful to add some features to another subclass functions.

Super()

```
class Dog:
  def __init__(self, pone,ptwo):
    self.pone=pone
    self.ptwo=ptwo
  def bark(self):
    print('wauu!')

class pinsher(Dog):
  def __init__(self, pone, ptwo, age):
    self.age=age
    dog.__init__(self, pone, ptwo)

  def bark_pinsher(self):
    print('-----')
    super().bark()
```

super().__init__()

```
class Animal:
  def __init__(self,var1, var2):
    self.var1=var1
    self.var2=var2
  def bark(self):
    print('auf')

class Dog(Animal):
  def __init__(self,var1,var2,var3):
    self.var3= var3
    super().__init__(var1,var2)
```

Note here that this not include the word **self**. Remember that in this case we are invoking no declaring. Note that class_name. could be useful to multiple inheritance.

Questions

why a class never is used as a glocal scope.

Hierarchical inheritance

isistance and issubclass

```
isinstance(doberman,Dog)
issubclass(Dog, Animal)
```

both functions return a Boolean expression.

Multiple inheritance

```
class Animal:
  def __init__(self,var1, var2):
    self.var1=var1
    self.var2=var2
class Domestic:
  def __init__(self, domestic):
    self.domestic = domestic
    self.kind = kind
class Dog(Animal, Domestic):
  pass
```

if we try to create a instance with the paremeters **var1,var2,domestic** this will arise a exception due, the dog class does not what init method uses.

Multiple inherintace

Overriding __ init __

```
class Animal:
  def __init__(self,var1, var2):
    self.var1=var1
    self.var2=var2
class Domestic:
  def __init__(self, domestic):
    self.domestic = domestic
    self.kind = kind
class Dog(Animal, Domestic):
  def __init__(self, var1, var2, domestic):
  Animal.__init__(self, var1,var2)
  Domestic.__init__(self,domestic)
```

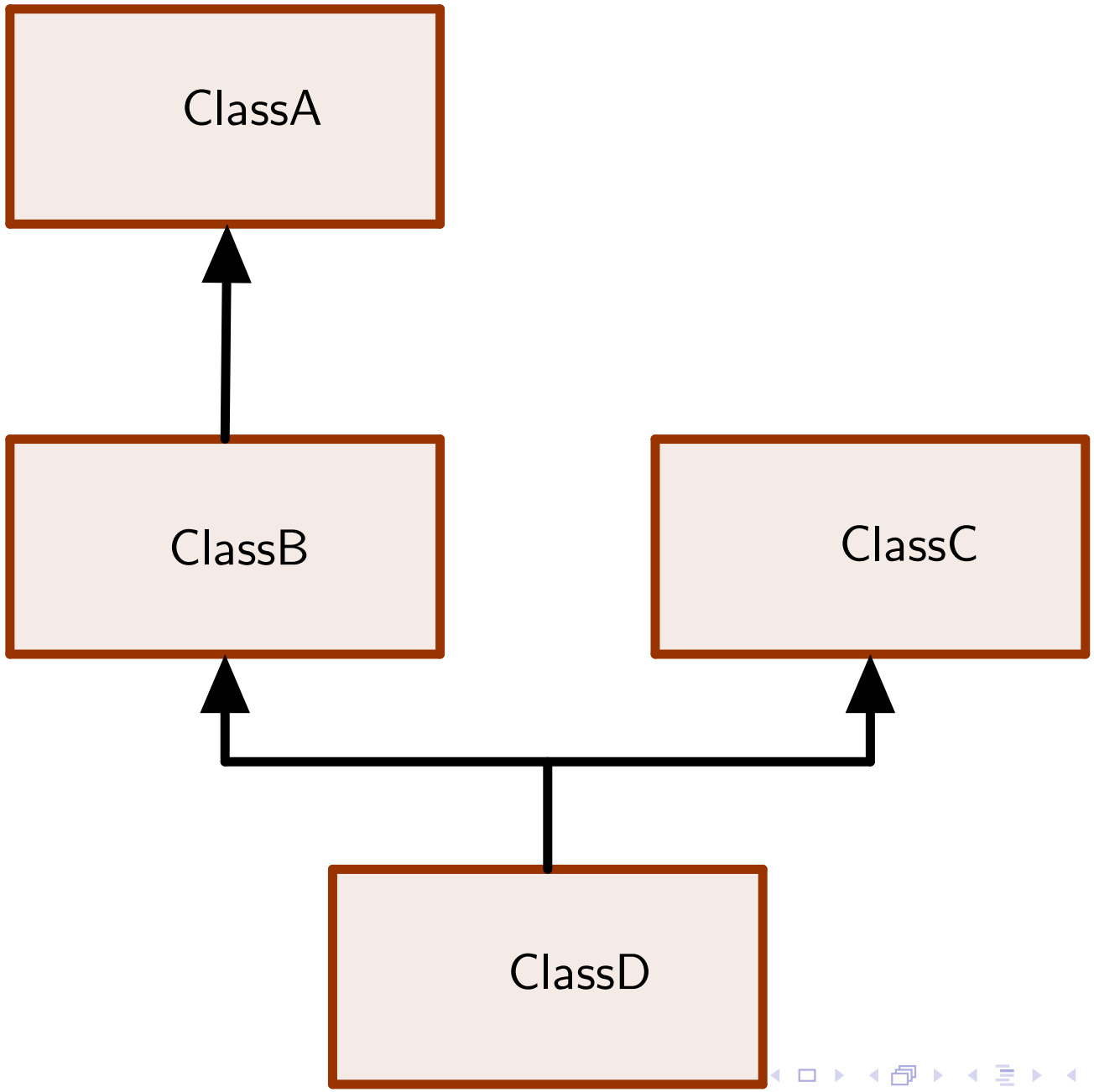
Method Resolution Order

MRO

In some cases when a new class inherit form its parents, each one could be have a same **method name**. following the code of the previous slide we have:

```
print(Dog.mro())
```

this will print out the order in which python look up the methods.


ClassA
ClassB
ClassC
ClassD

Theory

- (+) public, accessible outside of class.
- (−) private, manipulable only inside of class.
- (#) private, but shared by its subclass.

Theory

Message

The message have three properties:

- receiver identity
- Method name
- parameters

to research

four principles:

- Abstracción
- Encapsulamiento
- Modularidad
- Herencia

otros a destacar son:

- polimorphysm
- binding
- concurrence
- persistence

theory insights about encapsulation

encapsulation allow us join in the class attributes and methods.
the key feature of encapsulation is hide trivial information to the user.

Theory insights about polimorphysm

Mean that one method, have multiple implementations. relative methods according to the class.

```
1 + 1
'hello' + ' world'
```

In this case each line contain two objects of different class, but the + operand make a sum up and a concatenation respectively.

Polimorphism

different behaviors or methods associated with different objects could share the same name.

Inheritance

Promote the encapsulation and the polymorphism (why?).

dynamic binding:

Dynamic dispatch

To answer to what behavior of a polyphormic mehthod invoke while the program is running.

dynamic binding or late binding

The messages sent, to invoke a method is in runtime or execution time and not in compilation.