Programação e Algoritmia
--x--
Programming and Algorithmics

1 – Object-Oriented Programming

Problem

- Create a program in Python to:
  - Calculate the distance between 2 points in a 2D/3D space

First (basic) solution

- # read / define point A
- # read / define point B
- # calculate distance (eventually in a function)
- # display result

Analyzing the solution

```
import math
# read / define point A
point1_x = 0
point1_y = 0

# read / define point B
point2_x = 10
point2_y = 10

# calculate  distance (in a function)
distance = math.sqrt(math.pow(point2_x-point1_x,2)
+ math.pow(point2_y-point1_y,2) )

# display result
print(distance)
```

- What if we want not 2 but 100 points ?

Thinking in terms of objects …

- pointA = Point(10,20)
- pointB = Point(20,30)
- distance= pointA.distance(pointB)
- print(f“The distance between {pointA} and {pointB} is {distance}”)

Another problem

- Simulation of a lamp
  - Turn on
  - Turn off
  - Get state
  - Switch state

- How to solve it **adopting object(s)** ?

Lamp class

```python
lamp1 = Lamp(1)
lamp2 = Lamp(2)

lamp1.turn_on()
print(lamp1.status())

lamp1.turn_off()
print(lamp1.status())
```

Example of use of the class Lamp

```python
# create several lamps (instances)
lamps=[]
n_lamps = 5
for i  in range(1,n_lamps):
    lamps.append(Lamp(i))
print(lamps[1])
print(lamps[1].status())

# turn on all lamps
for lamp in lamps:
    lamp.turn_on()

# swith some
for i  in range(1,n_lamps,2):
    lamps[i].switch()

# print status
for lamp in lamps:
    print(lamp.status())
```

```
<__main__.Lamp object at 0x0000025A21819308>
Light 2 is off
Light 1 is on
Light 2 is on
Light 3 is on
Light 4 is on
Lamp 2 now is off
Lamp 4 now is off
Light 1 is on
Light 2 is off
Light 3 is on
Light 4 is off
```

Classes (and Objects)

Why objects (and classes) ?

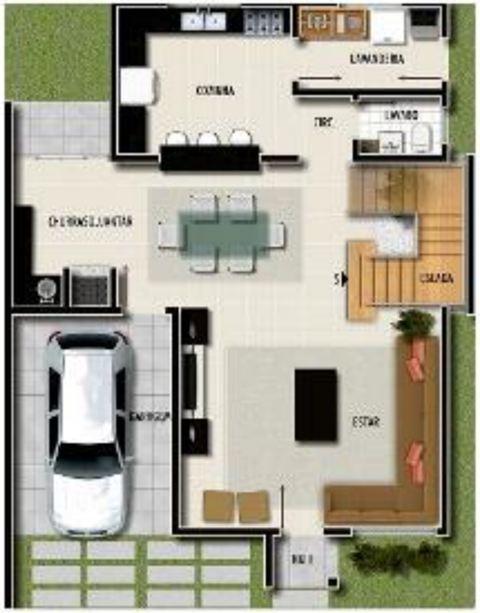
- As demands for new and more powerful software are increasing, building software quickly, correctly and economically is important.
- Objects, or more precisely, the classes objects come from, are **essentially reusable software components**.

- Software-development groups can use a modular, object-oriented design-and-implementation approach **to be much more productive** than with earlier popular techniques like “structured programming.”

- Object-oriented programs are often easier to understand, correct and modify.

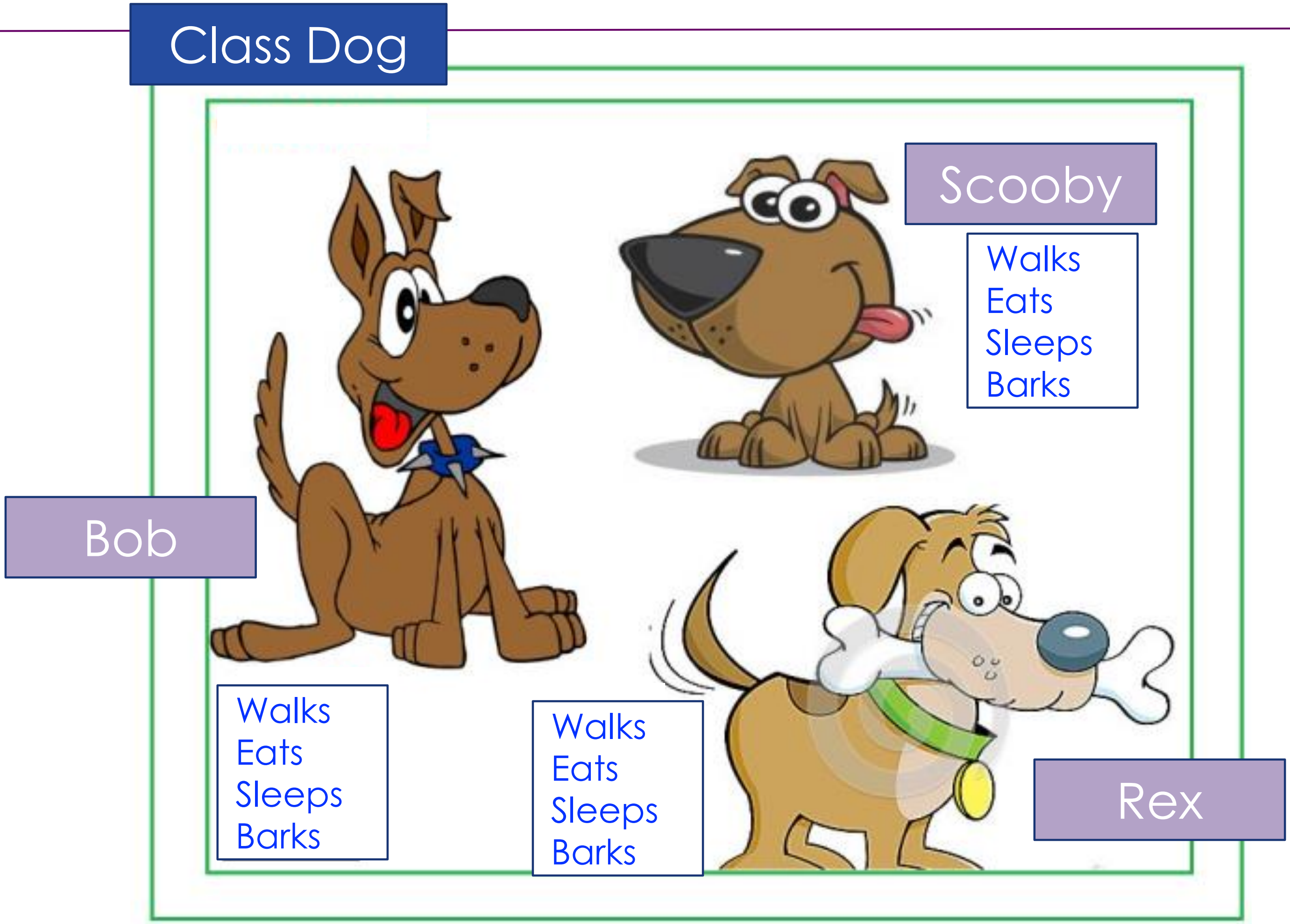
Class vs Objects

= OBJECT



= CLASS

Class vs Objects



Class Dog

Scooby

Walks
Eats
Sleeps
Barks

Bob

Walks
Eats
Sleeps
Barks

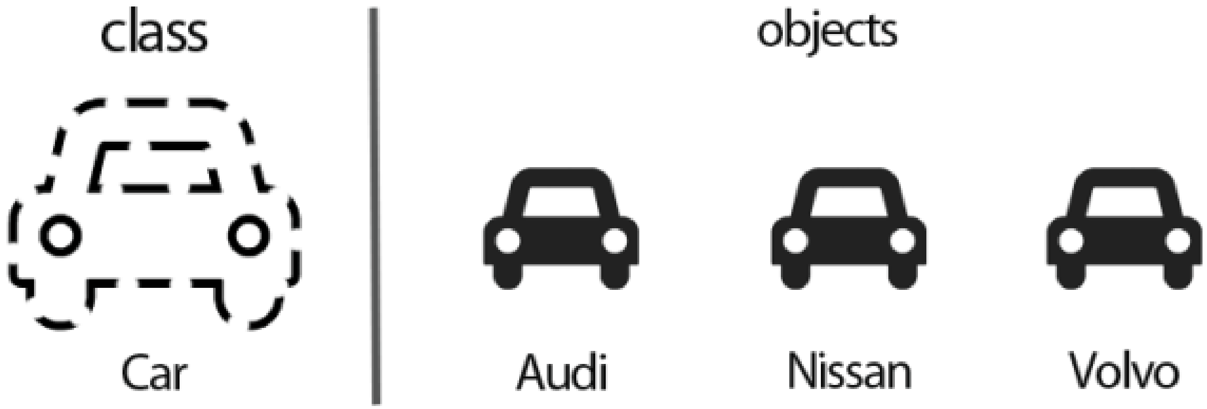
Walks
Eats
Sleeps
Barks

Rex

Class vs Objects

- **Class** is a user-defined prototype from which **Objects** are created

- Class = **Data** + Associated **Methods**



Classes

- A class is a structure that abstracts a set of objects with similar characteristics.

- A class defines the behavior of its objects ...
  - through methods
- ... and the possible states of these objects
  - through attributes.

- In other words, a class describes the services offered by its objects and what information they can store.
- A class **serves as a template for creating objects** (designated by instances of the class)

Objects

- An object, in real life, is anything we can name

- An object, in object-oriented programming, is **an instance** (that is, **a copy**) **of a class**
  - Portugal is an example of country, or, Portugal is an instance of Country.
  - There are many possible instances of the class Book

```
oneBook = Book("Python Programming")
otherBook = Book("Machine Learning in Python")
book3 = Book("Numpy")
```

Objects

- Objects are manipulated by references

```
name1 = Person("Manuel")
name2 = name1
```

- An object can store states through its attributes and react to messages sent to it, as well as send messages to other objects

Defining New Classes

Creating/Defining a class

- Creation of classes and instances in Python is **a simple process**

```
# definition of a class
class MyClass1():
    pass # meaning that there is no more specification (members)

# definition of another class
class MyClass2():
    pass


# instances (objects) of the 2 classes
x = MyClass1()
print(x)

y = MyClass1()
w = MyClass2()
print(w)
```

Output:

```
<__main__.MyClass1 object at 0x00000157421CDF48>
<__main__.MyClass2 object at 0x00000157421DA408>
```

Creating instances

```
# instances (objects) of
the 2 classes

x = MyClass1()
print(x)

y = MyClass1()
w = MyClass2()
print(w)
```

- Code at left instantiates 3 objects (x, y, w) of 2 classes (MyClass1 and MyClass2)
- Creating an instance (object) in Python has a very simple syntax:
  - Associate the name of the object we want to create to the class name
  - Ex: y = MyClass1()

Using classes

- A class Python has associated to it a set of names (namespace)
  - As modules
- After definition, its **members (attributes and methods)** are used and manipulated by programs using the syntax class.member
- To invoque a method of a class from outside the class it is necessary to specify class name and method name

```
# definition of class student
class Student:
    # class methods

    # self represent an instance of the class
    def setNameCourse(self, name_value, course_value):
        self.name = name_value          # attrib 1 - Name
        self.course = course_value      # attrib 2 - Course

    # print data of an instance
    def displayNameCourse(self):
        print(f"{self.name} : {self.course}" )

# main program
# create instances (objects)
s1 = Student()
s2 = Student()

# add data to attributes
s1.setNameCourse("Ana Maria Martins","Lic. Matemática")
s2.setNameCourse("Lucas Guedes","Lic. Enga Gestão Industrial")

# visualizar dados
s1.displayNameCourse()
s2.displayNameCourse()
```

```
OUTPUT:
Ana Maria Martins : Lic. Matemática
Lucas Guedes : Lic. Enga Gestão Industrial
```

Naming a class

- Specification of a class does not require parenthesis after its name
  - They are needed for functions, methods and creation of instances

```
class Student:
```

Naming a class

- Class names should follow standard Python rules regarding variable names

- Rules:
  - Names must start by a letter or an underscore (_)
  - Names may contain letters, numbers or underscore
  - Spaces are not allowed

- Recommendations
  - Use of CamelCase notation

First letter in caps as well start of each word

The variant lowerCamelCase can also be used

Class Attributes

- Attributes are characteristics of an object.
  - the data structure that will represent the class.

- Example:
  - an object of the class "Employee" would have as attributes "name", "address", "phone", "NIF", etc.

- The set of values of the attributes of a given object is called **state**

Attributes and Self

- In attributes definition, the variable must be identified with self
  - ex.: **self.course**

Attribute specification

- Python allows definition of attributes after the creation of a class instance (object), even without their definition inside the class
  - i.e, allows dynamic and arbitrary definition of attributes associated to an instance

- The code example shows an example:
  - a simple class is created without data members or methods, and later, in the main program, attributes are added to created instances.

- The notation used is known as dot notation

  <object>.<attribute> = < value>

```
# definition of a class
class Point2D:   # no need for ()
    pass

# create 2 instances (objects)
p1 = Point2D()
p2 = Point2D()

# define dynamic. attributes (x, y)
p1.x = 43
p1.y = 18
p2.x = -9
p2.y = 15

# output
print(f' p1 = ({p1.x},{p1.y})')
print(f' p2 = ({p2.x},{p2.y})’)
```

```
OUTPUT:
 p1 = (43,18)
 p2 = (-9,15)
```

Behaviors and Methods

- After attributes definition (the data) how objects behave needs to be defined
- Object actions and interactions among them define the behavior of instances (objects) of a class
- Actions define what happens to the state of attributes and the general behavior of the class

- The method is the fundamental element to define the behavior of a class
- Specification of a method is like the specification of a function
  - Syntax:
  - def followed by name, parenthesis, parameters, “:”
  - Ex:

```
def setNameCourse(self, name_value, course_value):
```

Defining a Class Method

```
def meth_name(self, parameter,…, parameter):
        statements
```

- `self`
  - must be the first parameter
  - reference to the current object
  - provides access to the object attributes

Methods

- A method may contain regular Python instructions

- It may use the parameters (as variables)

- The major difference for a function is the requirement of at least one parameter (a reference for the instance being processed)
  - Usually designated by **self**, by convention

Example

```
import math

# definition of a class
class Point2D:   # no need for ()
    # methods
    def set_xy(self, x, y):
        self.x = x
        self.y = y

    def reset(self):  # initialize coords
        self.set_xy(0,0)

    def distance(self,otherPoint):
        return math.sqrt((self.x - otherPoint.x)**
2  + (self.x - otherPoint.x)** 2 )

# create 2 instances (objects)
p1 = Point2D()
p2 = Point2D()

# define dinamically attributes (x and y)
p1.set_xy(43, 18)
p2.set_xy(-9,15)

# distances
print(f' dist(p1,p2) = {p1.distance(p2):5.2f}')
print(f' dist(p2,p1) = {p2.distance(p1):5.2f}')
```

```
dist(p1,p2) = 73.54
dist(p2,p1) = 73.54
```

- Class contains 3 methods
- **set_xy**
  - Assigns values to object self
- **reset**
  - Calls set_xy to set attributes to zero
  - Note the use of self.
- **distance**
  - Calculates distance of the point defined by the instance (self) to other point (passed as parameter)

Methods invocation

- A Method in Pythons is interpreted as a function defined and specified in the context of a class

- It can be invoked in the exterior of a class

```
Example (class Student):
s1.setNameCourse("Ana Martins","Lic. Matemática")
```

- And inside the class

```
Example (class Point2D):
self.set_xy(0,0)
```

The constructor method __init__

- As many of the languages supporting object-oriented programming, Python implements the constructor concept

- A constructor is a specific **method to create and initialize an instance of the class**

- The constructor of Python class is defined by the method __init__
  - The double underscores mark the method as special for the Python interpreter

Example - constructor for Point2D class

- In the example at right, the constructor initializes the attributes values (x, y)
- And outputs a message
- Class definition also includes another method, reset()
- What is your opinion regarding the solution adopted to print information on p1 and p2 ?

```
# class Point2D with constructor
class Point2D:
    # data members
    x = None
    y = None

    # constructor
    def __init__(self, x, y):
        self.x = x
        self.y = y
        print("... instance of Point2D created")

    def reset(self):
        self.x = None
        self.y = None

# create instances (objects)
p1 = Point2D(3,5)
p2 = Point2D(-7,11)
print(p1.x, p1.y)
print(p2.x, p2.y)
p1.reset()
print(p1.y)
```

```
OUTPUT:
... instance of Point2D created
... instance of Point2D created
3 5
-7 11
None
```

How to represent an object as a string?

- Problem:

```
>>> p1 = Point2D(2,3)
>>> print (p1)
<point.Point object at 0x10dc5f6d0>
```

- Special method:

```
def __str__(self):
      return string
```

- Example (class Point2D):

```
def __str__(self):
      return "Point (" + str(self.x) + "," + str(self.y) + ")”

#usage
>>> print (p1)
Point (2,3)
```

Class attributes / variables

- Class attributes (or class variables) differ from data attributes (variables of instances) as they can be accessed without the creation of an instance (of the class)
  - They are equivalent to static variables in Java

- Are defined outside methods but insider the body and structure of the class
  - Instance variables are defined inside __init__ method and preceded by self
  - Example:

```
class MyClass:

    classAttribute = 5355 # class attribute

    def method1(self):
        return "Hello"
```

- They are **associated to the class** and not to a particular instance
  - There is only 1 shared by class and all instances

Example (class attributes)

```
class House:
    '''A very simple counter class'''

    # class attributes
    count = 0

    def __init__(self):
        self.__class__.count += 1
        print(f"House {self.__class__.count} created!")



# variables (attributes) pre-defined for classes
print(f"Starting with {House.count}.")

# creating some instances
c1 = House()
c2 = House()

print(f"Now we have {House.count}.")
```

Output:

```
Starting with 0.
House 1 created!
House 2 created!
Now we have 2 houses created.
```

Built in class attributes

- Classes in Python have a set of predefined attributed (built in class attributes)
- They can be accessed using dot operator

  <class>.<built_in_attribute>

__dict__	Dictionary with the names (namespace) of the class
__doc__	String with class documentation (None if not defined)
__name__	Class name
__module__	Name of the module in which class was defined In interactive mode is designated by __main__
__bases__	Tuple with base classes (superclasses).

Example

```
class Counter:
    '''A very simple counter class'‘’

    def __init__(self, init=0):
        self.count = init

    def incr(inc =1):
        self.count += inc

# attributes pre-defined for classes
print(Counter.__dict__)
print(Counter.__doc__)
print(Counter.__name__)
print(Counter.__module__)
print(Counter.__bases__)
```

Output:

```
{'__module__': '__main__', '__doc__': 'A
very simple counter class', '__init__':
<function Counter.__init__ at
0x000001730801DCA8>, 'incr':
<function Counter.incr at
0x00000173081 6B558>, '__dict__':
<attribute '__dict__' of 'Counter'
objects>, '__weakref__': <attribute
'__weakref__' of 'Counter' objects>}

A very simple counter class

Counter

__main__

(<class 'object'>,)
```

Static methods

- Static method can be called without creating an object or instance.
  - Simply create the method and call it directly.

- Notice that **play()** does not use self

- This runs directly against the concept of object-oriented programming, but at times it can be useful to have a static method.

```
class Music:
    @staticmethod
    def play():
        print("*playing music*")

    def stop(self):
        print("stop playing")

Music.play()

obj = Music()
obj.stop()
```

Class methods

- A class method is a method that’s shared among all objects.
- Notice the use of **cls** as parameter
- Class methods can be called from instances and from the class itself

```
class Fruit:
    name = 'Fruitas'

    @classmethod
    def printName(cls):
        print('The name is:’,
              cls.name)

Fruit.printName()

apple = Fruit()
berry = Fruit()

apple.printName()
berry.printName()
```

Static vs class methods

- Like a static method, a class method doesn't need an object to be instantiated

- A static method knows nothing about the class or instance.
  - You can just as well use a function call.

- A class method gets the class when the method is called.
  - In a class method, the parameter is always the class itself.
  - It knows abouts the classes attributes and methods.

Deleting Attributes and Objects

- Delete object attribute

```
del obj.attribute

# example
p1 = Point(2,3)
del p1.x             # only affects p1 object
```

- Delete object

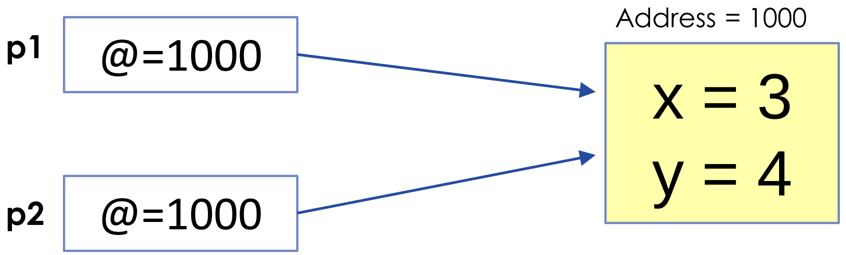
```
del obj

# example
p1 = Point(2,3)
del p1
print (p1)
NameError: name 'p1' is not defined.
```

Objects Copy

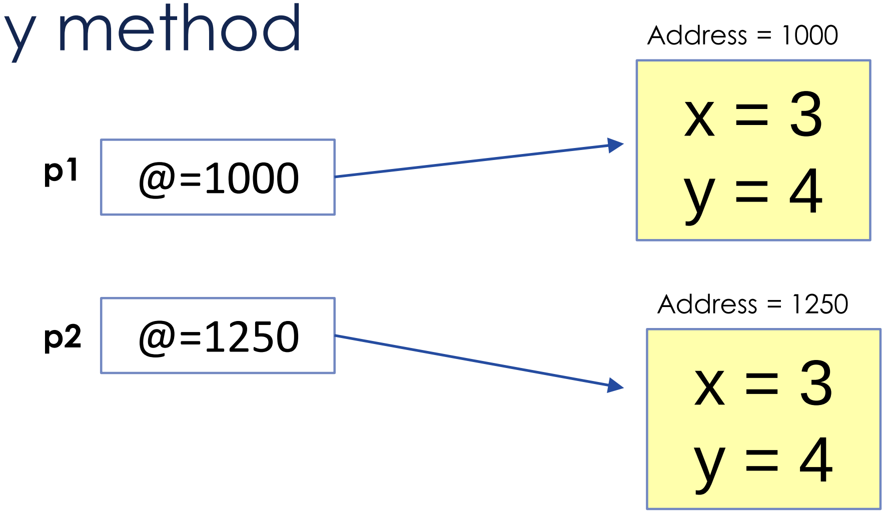
- Assignment – only copies references (aliasing)

```
>>> p1 = Point(3, 4)
>>> p2 = p1
>>> p1 is p2
True
>>> p1 == p2
True
```



- Use copy module / copy method

```
>>> p1 = Point(3, 4)
>>> import copy
>>> p2 = copy.copy(p1)
>>> p1 is p2
False
>>> p1 == p2
False
```



Modules And Classes

- Class definitions are frequently contained in their own module
  - Convention: module (file) name match the class name

Filename: Xpto.py

```
class Xpto:
     def meth1(self):
        print(“Method 1”)

     def meth2 (self):
        print(“Method 2”)
```

- Usage of class Xpto, from another file module
  - must include an import

```
from <filename> import <class name>
from Xpto import Xpto
```

Example - Lamp Class

```
# class Lamp definition
class Lamp:

    on     = False
    number = None

    # constructor
    def __init__(self, number, on = False):
...
```

Key concepts

Key concepts - object

- An object **represents** a physical or abstract entity of the real world

- An object integrates a **data structure and behavior**
  - of a concept, abstraction, entity, process, activity, place, event or other real or abstract entity

- Can be seem as a software abstraction to model and represent the relevant aspects of a unique (and specific) entity of the domain

Key concepts – methods and classes

- Performing a task in a program requires a method.
  - The method houses the program statements that perform its tasks.
  - The method hides these statements from its user,

  just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster.
- In Python, a program unit called a class houses the set of methods that perform the class's tasks.
  - For example, a class that represents a bank account might contain one method to deposit money to an account, another to withdraw money from an account and a third to inquire what the account's balance is.
- A class is similar in concept to a car's engineering drawings
  - which house the design of an accelerator pedal, steering wheel, and so on.

Key concepts - class

- A class makes possible to represent sets of objects with common structure (attributes) and behavior (operations)

- A class is a pattern for a specific object (of the domain)

- A class represents a set of objects (instances) of the same type

- The definition of a class specifies the structure (attributes) and behavior (operations) for all instances (objects) of the class

Key concepts – class (cont.)

- Almost any noun can be reasonably represented as a software object in terms of

- Data attributes
  - e.g., name, color and size

- and behaviors
  - e.g., calculating, moving and communicating

Key concepts - Instantiation

- Just as someone has to build a car from its engineering drawings before you can drive a car, you must build an object of a class before a program can perform the tasks that the class’s methods define.
- The process of doing this is called instantiation.

- An object is then referred to as an instance of its class.

Key concepts - reuse

- Just as a car’s engineering drawings can be reused many times to build many cars, you can reuse a class many times to build many objects.
- Reuse of existing classes when building new classes and programs saves time and effort.
- Reuse also helps building more reliable and effective systems
  - because existing classes and components often have undergone extensive testing, debugging (that is, finding and removing errors) and performance tuning.
- Reusable classes are crucial to the software revolution
  - just as the notion of interchangeable parts was crucial to the Industrial Revolution
- In Python, you’ll typically use a building-block approach to create your programs. To avoid reinventing the wheel, you’ll use existing high-quality pieces wherever possible.
- Reuse is a key benefit of object-oriented programming

Key concepts – Messages and Method calls

- When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, to go faster.
- Similarly, **you send messages to an object.**

- Each message is implemented as a method call that tells a method of the object to perform its task.
- For example, a program might call a bank-account object's deposit method to increase the account's balance.

Key concepts – Attributes and Instance variables

- A car, besides having capabilities to accomplish tasks, also has attributes
  - such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total kms driven
- Like its capabilities, the car's attributes are represented as part of its design
- As you drive an actual car, these attributes are carried along with the car. Every car maintains its own attributes.
  - For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars.
- An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class.
  - For example, a bank-account object has a balance attribute that represents the amount of money in the account.
- Each bank-account object knows the balance in the account it represents, but not the balances of the other accounts in the bank.
  - Attributes are specified by the class's instance variables.

Key concepts - Inheritance

- A new class of objects can be created conveniently by inheritance
  - the new class (called the subclass) **starts with the characteristics of an existing class** (called the superclass),
  - possibly **customizing them and adding** unique characteristics of its own.

- In our car analogy, an object of class “convertible” certainly is an object of the more general class “automobile,”
  - but more specifically, the roof can be raised or lowered.

- We will return to this soon

Key concepts - interface

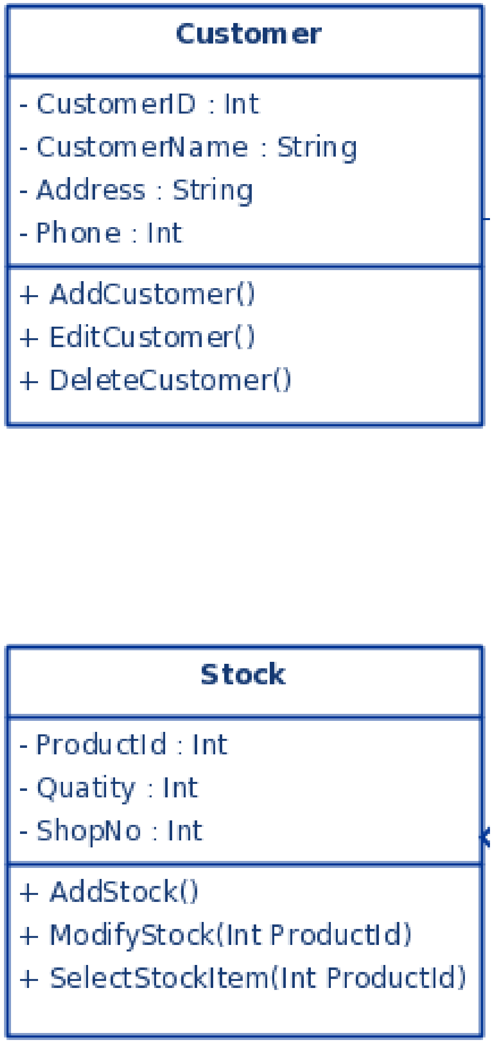
- The interface of a class is determined by the set of attributes and operations (methods) it provides
  - Designated as data members and methods of the class

- The visibility if the data members and methods can be divided into 3 categories:
  1. Public – the interface is visible for all the objects
  2. Protected – the class interface is only visible for the objects of the class or subclasses (by inheritance)
  3. Private – the interface is only seen by the class itself

Visual Representation of Classes

UML

- The Unified Modelling Language (UML) is nowadays the main notation for the modeling of software systems.
- UML is a standard language for the visualization, specification, design and documentation of software components
- The UML representation for a class is as shown at right



Customer

- CustomerID : Int
- CustomerName : String
- Address : String
- Phone : Int

+ AddCustomer()
+ EditCustomer()
+ DeleteCustomer()

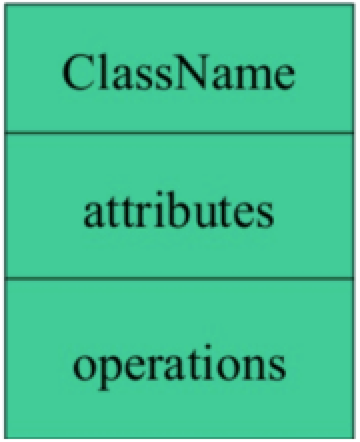
Stock

- ProductId : Int
- Quatity : Int
- ShopNo : Int

+ AddStock()
+ ModifyStock(Int ProductId)
+ SelectStockItem(Int ProductId)

Essential elements of A UML class diagram

- Essential elements of UML class diagram are:
- Class Name
- Attributes
  - is named property of a class which describes the object being modeled. In the class diagram, this component is placed just below the name-compartment.
  - are generally written along with the visibility factor.
  - Public, private, protected are denoted by +, -, #
- Operations

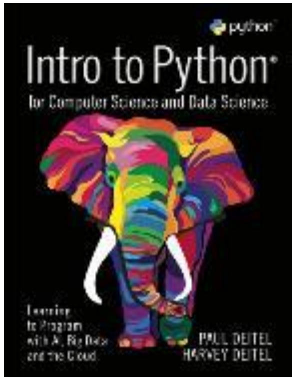


Benefits of Class Diagram

- It provides an overview of how the application is structured before studying the actual code.
  - This can easily reduce the maintenance time
- It helps for better understanding of general schematics of an application.
- Allows drawing detailed charts which highlights code required to be programmed
- Helpful for developers and other stakeholders.

Complete Class Example
Account

Adapted from Deitel & Deitel Book


python
Intro to Python
for Computer Science and Data Science
Learning to Program with AI, Big Data and the Cloud
PAUL DEITEL
HARVEY DEITEL

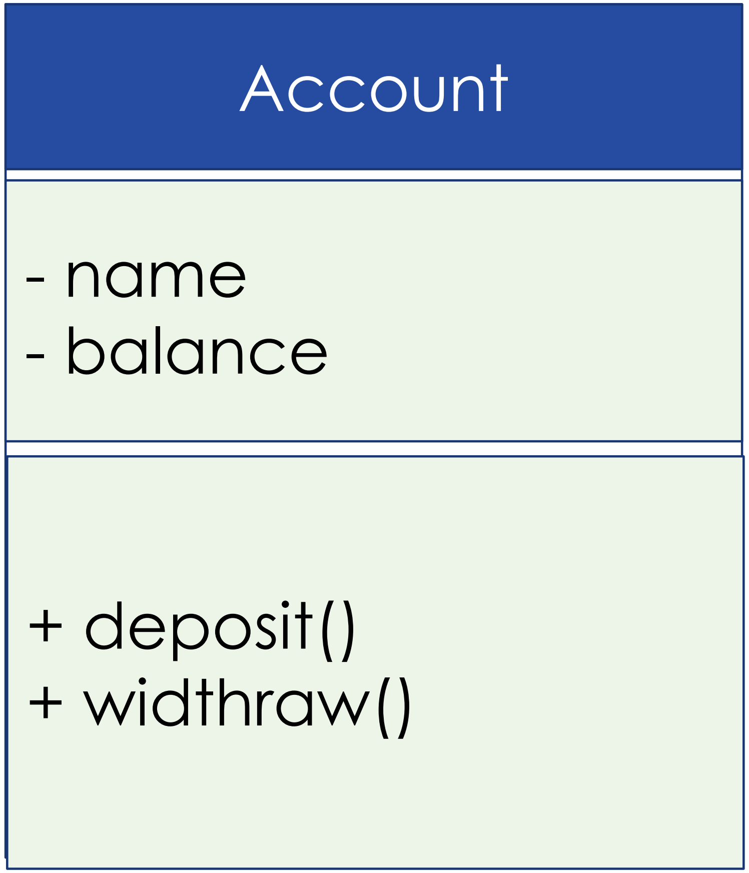
Custom Class Account

- Let’s begin with a bank **Account** class that holds an account holder’s name and balance.

- An actual bank account class would likely include lots of other information, such as address, birth date, telephone number, account number and more.

- The Account class accepts deposits that increase the balance and withdrawals that decrease the balance.

Class Account in UML



Account

- name
- balance

+ deposit()
+ widthraw()

Defining the Class

```
 # Account.py
"""Account class definition."""
from decimal import Decimal

class Account:
    """Account class for a bank account balance."""
```

- A class definition begins with the keyword class followed by the class’s name and a colon (:)

  This line is called the class header.

- The Style Guide for Python Code recommends that you begin each word in a multi-x

  For example, CommissionEmployee.

- Every statement in a class’s suite is indented.

Initializing Account Objects: Method __init__

```
def __init__(self, name, balance):
    """Initialize an Account object."""
    # if balance is less than 0.00, raise an exception
    if balance < 0.0:
        raise ValueError('Initial balance must be >= to 0.00.')
    self.name = name
    self.balance = balance
```

- Class Account’s **__init__** method initializes an Account object’s name and balance attributes if the balance is valid

-
  ```
  account1 = Account('John Green', 50.0)
  ```

  creates a new object, then initializes its data by calling the **__init__** method.

- Each new class you create can provide an **__init__** method that specifies how to initialize an object’s data attributes.
- Returning a value other than None from __init__ results in a TypeError.
  - Recall that None is returned by any function or method that does not contain a return statement.

def __init__(self, name, balance):

- When you call a method for a specific object, Python implicitly passes a reference to that object as the method's first argument
  - For this reason, all methods of a class must specify at least one parameter
  - By convention most Python programmers call a method's first parameter self
  - A class's methods must use that reference (self) to access the object's attributes and other methods.
- Class Account's __init__ method also specifies parameters for the name and balance

- When an object of class Account is created, it does not yet have any attributes.
  - They're added dynamically via assignments of the form:

```
self.attribute_name = value
```

Method deposit

```
def deposit(self, amount):
    """Deposit money to the account."""

    # if amount is less than 0.0, raise an exception
    if amount < 0.0:
        raise ValueError('amount must be positive.')
    self.balance += amount
```

- The Account class’s deposit method adds a positive amount to the account’s balance attribute.
  - If the amount argument is less than 0.0, the method raises a ValueError, indicating that only positive deposit amounts are allowed.
  - If the amount is valid, amount is added to the object’s balance attribute

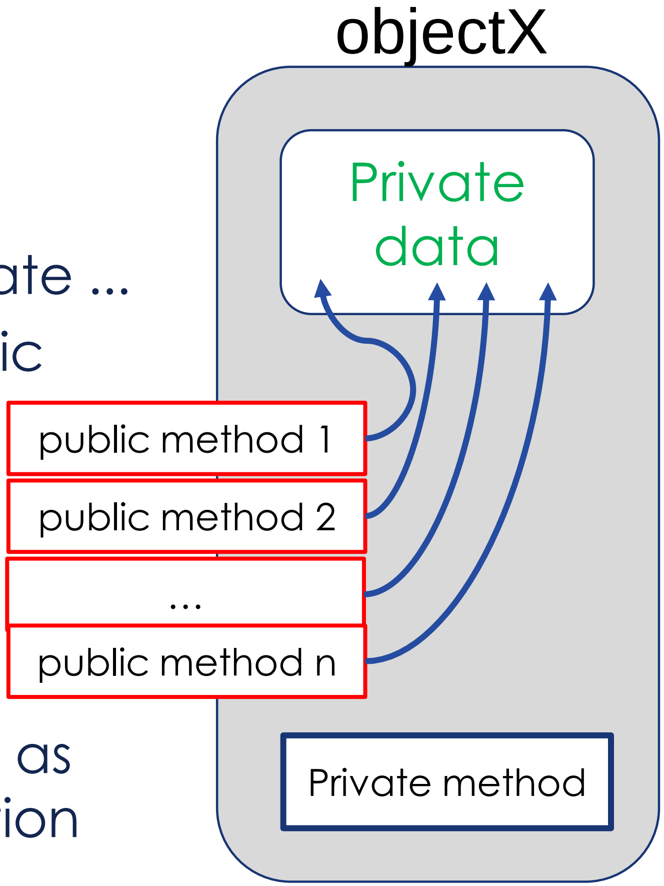
Composition:
Object References as Members of Classes

- An Account **has** a name, and an Account **has** a balance.
- Recall that "everything in Python is an object."
  - This means that an object's attributes are references to objects of other classes.
- For example,
  - an Account object's name attribute is a reference to a string object
- Embedding references to objects of other types is a form of software reusability known as composition and is sometimes referred to as the "**has a" relationship**.

Encapsulation (Data Hiding)

Encapsulation

- Object as capsule
- An object is a closed and autonomous computing unit
  - Able to perform operations on its state ...
  - And return answers whenever public methods are invoked
- Fundamental to achieve **independence of context**
  - to ensure important properties such as modularity, reuse, easy error detection



Encapsulation (Data Hiding)

- Allows hiding implementation details of the class from users.
  - Users don’t need to know implementation details to use a class.
- Implementation can be changed without impacting code using the class
- The objective of encapsulation and creation of private members is to indicate to programmers that they don’t have access rights to class attributes
  - Access must be done through the available methods

Data Hiding

- Encapsulation (Data Hiding) allows **hiding the internal data of an object**
  - But sometimes it is necessary to access this data directly (reading and/or writing).

- Important rules!
  - All attributes must be private.

  - **Access** to the internal information of an object (private part) **must always be carried out through public <u>interface methods</u>**

Getters and Setters

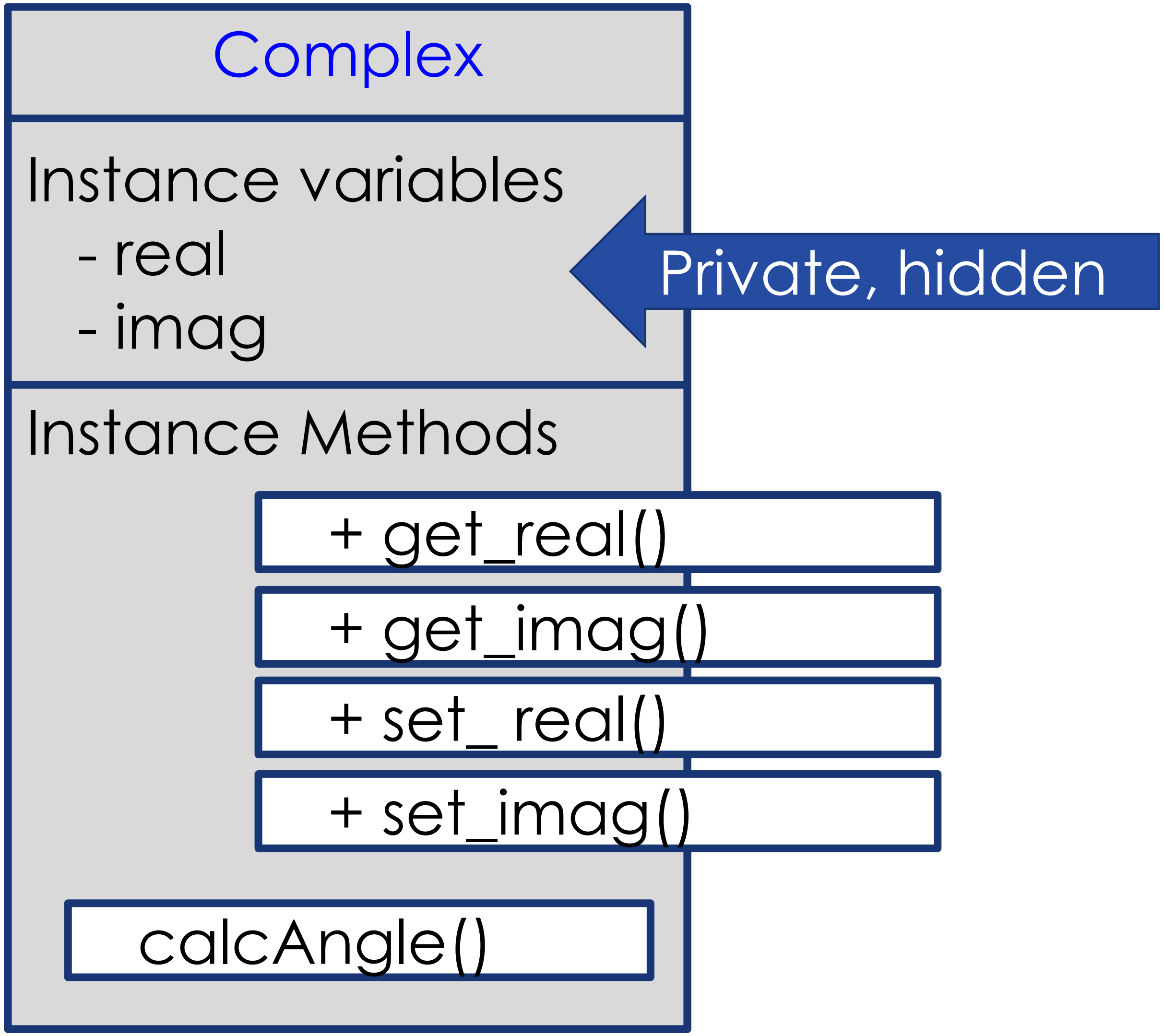
- Important part of the interface methods of the class
- Getter
  - Returns the value of an attribute

```
def getRadius():
    return self.radius
```

- Setter
  - Modifies the state of the object (its attributes)

```
def setRadius(self, newRadius):
    self.radius = newRadius;
```

Example – Class Complex



Complex

Instance variables

- real
- imag

Private, hidden

Instance Methods

+ get_real()

+ get_imag()

+ set_ real()

+ set_imag()

calcAngle()

Private members in Python

- To define private members (attributes and methods) in Python the two underscores prefix is added to their names (as in the example at right)

- Class variable **__cont** is defined as private and must be accessed using method **icount()** only
- The variable name is public and can be accessed directly (**instance.variable**)

```
class MyClassP:
    '''Private vs public members'‘’

    __cont = 0  # private variable
    name = "public class variable"

    def myPublicMethod(self):
        print('... public method')
    def __myPrivateMethod(self):
        print('... private method')

    def icount(self):
        self.__cont += 1
        print(self.__cont)
```

Private members in Python

- Despite being possible to define private members, it is always possible to access members of a class, using built-in variables.

Example (with MyClassP)

```
cl1 = MyClassP()
print(cl1.name)
print(cl1.icount)

print(cl1.myPublicMethod)
print(cl1.myPublicMethod())
# cl1.__myPrivateMethod() is not possible

print(cl1.__class__)
print(cl1._MyClassP__myPrivateMethod())
```

- Encapsulation is **not fully implemented in Python**
  - It is possible to access using built-in class attributes (as in last line of code)
- Some programmers consider that Python does not implement encapsulation

Inheritance

Data Structures

Object Orient Analysis, Design and Programming

Basics