Object Oriented Programming in Python

Classes and objects

CentraleSupélec

Outline

Outline

Object-oriented programming (OOP)

OOP: a programming paradigm for directly mapping real-life problems into a program

- it is based on the notion of class (a user-defined data type)
- and objects (instances of a given class)

an object is a data structure that contains:

- data: in form of variables called attributes or fields
- behaviour: in form of procedures called methods

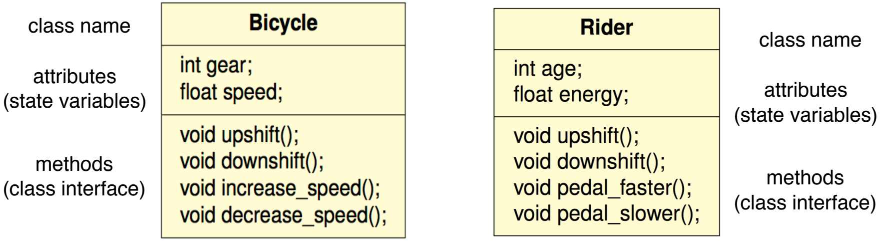
Real-world objects

real-world objects share two characteristics: they all have a state and a behaviour

examples of real-world objects

- Dog:
  - state: name, color, breed, hungry, ...
  - behaviour: barking, fetching, wagging tail, eating, ...
- Bicycle:
  - state: current gear, current pedal cadence, current speed, ...
  - behaviour: changing gear, changing pedal cadence, applying brakes, ...

Example: class “Bicycle” and class “Rider”



What is a (software) class ?

class: the *blueprint* characterising a category of objects

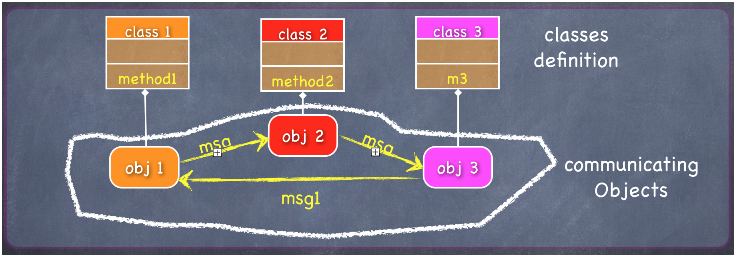
- defines the attributes representing the state of objects
- defines the methods representing the behaviour of objects

several objects can be instantiated from a given class

What an Object-Oriented program looks like?

an Object-Oriented program consists of:

- a collection of classes definitions
- a collection of objects' instances



computation: instantiated objects perform the desired computation by invoking each other methods (i.e. by exchanging messages)

Outline

Classes in Python

- Class: bundle together *data* and *functionalities*

- defining a Class defines a *new data type* allowing new *instances* (objects) of that data type to be created

Class definition syntax

in Python a class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example:

```
class MyClass:
   # this is a comment

    # this is an attribute
    i = 12345

    # this is a method
    def f(self):
        return 'hello world'
```

defines a class called `MyClass` with one attribute named `i` and one method named `f`

Object instantiation and reference to object members

```
class MyClass:
    i = 12345

    def f(self):
        return 'hello world'
```

instantiation of an object of a class: uses *function call notation*

```
x = MyClass() # creates an object of type MyClass and associates it to variable x
```

reference to an object's attributes and methods: through the `.` operator

```
x = MyClass()
x.i # refers to attribute i
x.f() # refers to method f()
```

Object initialisation: method `__init__()`

To initialise objects in a specific manner a class must define a special method called `__init__()`

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart # declares and initialise an attribute named r
        self.i = imagpart # declares and initialise an attribute named i
    def display(self):
        print('(', self.r, ',' ,self.i, ')') # displays the value if r and i in between brackets
```

create a `Complex` object with given initial value and display its values

```
c = Complex(3,-7)
c.display()
>> ( 3 , -7 )
```

Use of self in a class method declaration

remark: a method declared in a class must have at least one attribute named self

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart # declares and initialise an attribute named r
        self.i = imagpart # declares and initialise an attribute named i
    def display(self):
        print('(', self.r, ',' ,self.i, ')') # displays the value if r and i in between brackets
```

what self stands for ?

- it represents an instance object of the class the method belongs to
- an invocation of a method on an instance of the class replaces self with the invoking object

```
c = Complex(3,−7) # corresponds to invoking Complex.__init__(c,3,−7)
c.display() # corresponds to invoking Complex.display(c)
```

Class and instance variables

variables of a class may be:

- instance variables: storing data unique to an object
- class variables: storing data shared by all objects instances of the class

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name # instance variable unique to each instance
```

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind # shared by all dogs
'canine'
>>> e.kind # shared by all dogs
'canine'
>>> d.name # unique to d
'Fido'
>>> e.name # unique to e
'Buddy'
```

Inheritance: classes and subclasses

Inheritance: define a class as a subclass of another class

```
class DerivedClassName(BaseClassName):
    <statement−1>
    .
    .
    .
    <statement−N>
```

Example: a class for representing pets

```
class Pet(object): #class Pet inherits from class object
  def __init__(self,name,species):
    self.name = name
    self.species =species

  def getName(self):
    return self.name

  def getSpecies(self):
    return self.species

  def __string__(self):
    return "%s is a %s" % (self.name , self.species)
```

Example of subclasses: Dog and Cats

```
class Dog(Pet): #class Dog inherits from class Pet
  def __init__(self,name,chases_cats):
    Pet.i__init__(self,name,"Dog")
    self.chases_cats = chases_cats

  def getChasesCats(self):
    return self.chases_cats

class Cat(Pet): #class Cat inherits from class Pet
  def __init__(self,name,hates_dogs):
    Pet.__init__(self,name,"Cat")
    self.hates_dogs = hates_dogs

  def getHatesDog(self):
    return self.hates_dogs
```

```
>> ginger = Cat("Ginger",True)
>> clifford = Dog("Clifford",False)
>> barnaby = Pet("Barnaby","Parrot")
>> holly = Pet("Holly","Dog")
>> clifford.getName()
>> Clifford
>> ginger.getSpecies()
>> Cat
>> holly.getSpecies()
>> Pet
```

isinstance(): checking if an object is an instance of a class

`isinstance('object','class')`: returns True is object is an instance of class

```
>> ginger = Cat("Ginger",True)
>> clifford = Dog("Clifford",False)
>> barnaby = Pet("Barnaby","Parrot")
>> holly = Pet("Holly","Dog")
>> isinstance(clifford, Dog)
>> True
>> isinstance(holly, Dog)
>> False
>> isinstance(holly, Pet)
>> True
```