Lesson 6

ILAI (M1) @ LAAI I.C. @ LM AI

10 October 2024

Michael Lodi

Department of Computer Science and Engineering

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

These slides draw very heavily from Simone Martini's slides.

One-slide recap of lesson 5

recursive function: call itself in her body. Need a base case and a recursive simpler (moving towards base case) case

recursion is a form of repetition

`dict` is a mutable mapping between immutable objects (keys) and objects (values)

not sequences, we consider them to be unordered. Quick access to values through keys.
For details on hash, cryprography, python dictionaries, sets:
https://tenthousandmeters.com/blog/python-behind-the-scenes-10-how-python-dictionaries-work/

Python searches for names in the LEGB order:

- Local (function body: parameters and names used left of assignment)
- Enclosing: functions enclosing (non-local non-global), from innermost to outermost
- Global: names introduced at the main level of the file
- Builtin: names initialised when we start the machine (e.g. `len`, `print`…)

We (in theory) can use `global` and `nonlocal` keywords to modify the scope

We can create lists and dictionaries using a compact notation:

```
[<expression> for <name> in <sequence> <series of for/if>]
{<key:value> for <name> in <sequence> <series of for/if>}
```

Who to contact

For questions on exercises, on Python, tutoring/mentoring

1. *contact* **federico.ruggeri6@unibo.it**
   or
   mohammadrez.hossein3@unibo.it
2. *if you still have questions (eg. Theoretical), contact me: michael.lodi@unibo.it*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Next lectures for ILAI

Monday, 14 October 2024 15:00 - 18:00 (Regular, room 0.5)

Thursday 17 Oct: NO LECTURE (prof. Lodi at a conference)

ALL UPDATES ALREADY
ON THE COURSE WEBSITE

Use this time for trying exercises on Virtuale ;-)

Early exam - 8th November, 15:00 - Lab 4 + …

- a @studio.unibo.it account - mandatory
- an account on the Ingegneria cluster: https://remo.ing.unibo.it/app/student/infoy (you need the step 1 account to generate the ing account)
- **At the latest 1 week before** access (with the step 1 account) the website https://eol.unibo.it/

Register for the exam on the Unibo app or Almaesami - **PREFERRED**

- https://almaesami.unibo.it/almaesami/welcome.htm

Only if you are not able to register on app/Almaesami

https://forms.office.com/e/RaYAWQMBP2 (using your @studio account only)

If you have already filled the form but become able to register on AlmaEsami, do so.

Between 7 October and 4 November (included). Late requests will not be accepted.

Essential that you don't skip other lectures of other courses to study for this

classes and objects

We already know that...

Object:

value encapsulated into an *identity*

methods on the capsule

available methods depend on the *type*

Type (~ Class)

collection of objects which share

structure

(operations)

methods

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

From the point of view of the class:

A class (~ a type):

define the structure of its objects

the available operations and methods

The objects of the class

are the "values" of the class – the *instances*

they share the structure

and the methods defined in the class

So (1,2,3) is an instance (a value) of `tuple`

[1,2,3] is an instance of `list`

Python is a democratic language…

Classes and instances may be defined by the user

User-defined classes and their instances

are first-class citizens

same rights of any other value!

We may bind them to names, insert them in other objects, pass them to functions, functions may return them

An example: a class for cartesian points

A point

two informations: x and y

any point is created at the origin 0,0

Two operations/methods

whoareyou()   returns the pair x,y

move(delta) modifies the point:

from x,y

to x+delta,y+delta

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

An example: a class for cartesian points

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
          self.x+=delta
          self.y+=delta
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

An example: a class for cartesian points

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    def whoareyou (self):
        return self.x, self.y
    def move(self, delta):
          self.x+=delta
          self.y+=delta
```

Things to be discussed:

- instances?
- who is this "`self`" parameter?
- the funny double underscore of method `__init__` ?

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Creating instances

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    . . .
```

To create instances:

use the name of the class as a function

it returns an instance

its structure defined in method __init__

```
p=Point()
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Instances are mutable

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    . . .
```

We may access the *attributes of* (the data stored in) an instance: use the dot notation

```
p=Point()
print(p.x)     #prints 0
```

Attributes are mutable

```
p.x+=1
print(p.x)  #prints 1
```

Methods may be called on instances

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    def whoareyou (self):
        return self.x, self.y
    def move(self, delta):
          self.x+=delta
          self.y+=delta
p=Point()
print(p.whoareyou())    # prints (0,0)
p.move(2)
print(p.whoareyou())    # prints (2,2)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

self

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    def whoareyou (self):
        return self.x, self.y
    . . .
```

The *dot notation* ensures that

- the first parameter of a method, when called
- is a reference to the object receiving the method

```
p.whoareyou()
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

self

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    def whoareyou (self):
        return self.x, self.y
    . . .
```

The *dot notation* ensures that the first parameter of a method is bound to the receiver

- by convention, for the first parameter we use `self`

– `self` must be used to access the attributes of the instance

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The __init__ method

The `__init__` method is called behind the scenes when an instance is created (through the name of the class)

We use it to initialize the instances of the class

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

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

note: creation of attributes

```
class Point:
    def __init__(self):
        self.x=0
        self.y=0
    . . .
```

Observe that in

```
self.x = 0
```

that `self.x` on the left of = creates an instance attribute

`self.x` is a kind of generalised name, hence when at the left of =, creates new bindings

Pythontutor

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The __init__ method

The `__init__` method may have additional parameters, besides self

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

When an instance is created we must provide values for those parameters:

```
p=Point(0,1)
p=Point() #exception
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The __init__ method

The `__init__` method may have additional parameters, besides self

```
class Point:
    def __init__(self, xx=0, yy=0):
        self.x=xx
        self.y=yy
      . . .
```

When an instance is created we must provide values for those parameters:

```
p=Point(0,1)
p=Point()
```

dunder

"double underscore" is pronounced *dunder*

Methods of the form

`__name__`

are "magic methods", *by convention* reserved for special purposes by the Python machine

We should not fiddle with them, besides their intended purpose

We will see many of them

The command `class`

Introduces a class definition:

`class` *name*`:`

commands

method definitions

Both *commands* and *methods* are optional

The simplest class definition:

```
class A:
    pass
```

with no structure whatsoever

The command `class`

```
class name:
        commands
        method definitions
```

The *class name may be called as a function:*

it create an instance of the class

calling implicitly the method __init__, if there is one

- as many arguments as the ones of __init__ *(but self)*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Method definitions

They are like function definitions

(well: syntactically *they are* function definitions)

A method acts on the object receiving it

A method has always at least one parameter

the first parameter of a method (which we call `self` by convention)

is bound by the dot notation to the receiving object

Initialization

The *class name may be called as a function:*

it create an instance of the class

calling implicitly the method __init__, is there is one

as many arguments as the ones of __init__

If there is no __init__, no initialization is performed

No common structure is given to all the instances

However, we may add attributes to single instances

Homework

(1) Define a class

Student

with attributes:

registration number, name, year (1,2,3,4,5)

and methods

pass(): move the student to their next year; set year to None if year was already 5

show() : prints all the attributes of the instance

(2) Create 10 instances and put them into a list

(3) Sort that list by increasing registration number

Let's start again from

```
class Point:
    def __init__(self,xx,yy):
        self.x=xx
        self.y=yy
    def whoareyou (self):
        return self.x, self.y
    def move(self, delta):
          self.x+=delta
          self.y+=delta
```

Let's define a new method for adding (in a trivial way) two points

A new method

Given two `Point` instances `p` and `q`

define a method `Psum` such that

```
p.Psum(q)
```

returns a new `Point` instance with values

`p.x+q.x`	for x
`p.y+q.y`	for y

A new method

Given two `Point` instances `p` and `q`

define a method `Psum` such that

```
p.Psum(q)
```

returns a new `Point` instance with

```
p.x+q.x
p.y+q.y
```

```
def Psum(self,other):
    return Point(self.x+other.x,
                 self.y+other.y)
```

A new method

```
class Point:
    def __init__(self,xx,yy):
        self.x=xx
        self.y=yy
    def whoareyou (self):
        return self.x, self.y
    def move(self, delta):
         self.x+=delta
         self.y+=delta
    def Psum(self,other):
        return Point(self.x+other.x,
                     self.y+other.y)
```

A new method

```
p=Point(1,2)
q=Point(3,4)
s=p.Psum(q)
print(s.whoareyou())
            # prints (4,6)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

A new method

```
p=Point(1,2)
q=Point(3,4)
s=p.Psum(q)
print(s.whoareyou())
            # prints (4,6)
```

Would'd be nice if we could overload + and write

`s=p+q`

instead of `s=p.Psum(q)` ?

A new special method: __add__

Instead of

```
def Psum(self,other):
    return Point(self.x+other.x,
                 self.y+other.y)
```

define

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

+ calls the special method __add__, if any

Printing an instance

```
p=Point(0,1)
print(p)
```

is not of much help

We could add a method

```
def Pprint(self):
    print(str(p.whoareyou()))
```

We may do *much* better. Define the magic method

```
def __str__(self):
    return
'Point'+str(self.whoareyou())
```

Special method __str__

The magic method

```
__str__(self, optional args)
```

- must return a string
- is automatically called when the instance is argument of print()

__str__ vs. __repr__

```
>>> A = tree(0, tree(1, tree(2), tree(3)), tree())
>>> print(A)
0
.1
..2
..3
```

```
>>> A
tree(0, tree(1, tree(2), tree(3)), tree())
```

invece che

```
>>> A
<__main__.tree object at 0x101d98860>
```

Equality between objects

On instances of user defined classes

`==` coincides with `is`

It cannot be otherwise!

How could the Python machine divine what the user intended to be equality?

Equality between objects

On instances of user defined classes

`==` coincides with `is`

The magic method

`__eq__`

overload the `==` operation!

Equality between objects

The magic method

`__eq__`

overload the `==` operation!

On Point:

```
def __eq__(self,other):
    return self.x==other.x
           and self.y==other.y
```

Some magic methods for comparison

`__ne__(self, other)` defines !=

`__lt__(self, other)` defines <

`__gt__(self, other)` defines >

`__le__(self, other)` defines <=

`__ge__(self, other)` defines >=

Some numerical magic methods

`__add__(self, other)` defines +

`__sub__(self, other)` defines –

`__mul__(self, other)` defines *

`__floordiv__(self, other)` defines //

`__truediv__(self, other)` defines /

`__pow__` defines **

Boolean operations (and,or,not) *cannot* be redefined through special dunder methods `__xxx__`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

More numerical magic methods

`__pos__(self)` defines the unary +

`__neg__(self)` defines the unary -

`__abs__(self)` defines function `abs()`

`__round__(self, n)` defines function `round()`

`__floor__(self)` defines function `math.floor()`

`__ceil__(self)` defines function `math.ceil()`

`__trunc__(self)` defines function `math.trunc()`

Methods for accessing sequences

`__len__(self)` defines `len`

`__getitem__(self, key)` defines `self[key]`

`__setitem__(self, key, value)` defines `self[key]=value`

An example in SetItemExample.py

etc. etc. etc.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Methods for augmented assignment

`__iadd__(self, other)`

Define the RHS of +=

`__isub__(self, other)`

Define the RHS of -=

etc

They should be used if you want += *to modify in place* the object at LHS, like += on lists…).

If you want, instead, that += return a new instance, just define __add__ and += will use that for +.

An example in file Augmented.py

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Copying (cloning) objects

import copy

copy.copy(o1)

returns the shallow copy of o1

copy.deepcopy(o1)

returns the deep copy of o1

Copying (cloning) objects

```
class Point:
    def __init__(self, xx=0, yy=0):
        self.x=xx
        self.y=yy

class TwoPoints:
    def __init__(self, p1, p2):
         self.p1 = p1
         self.p2 = p2

a = Point(1,2)
b = Point(3,4)
t = TwoPoints(a,b)

from copy import copy, deepcopy

c = copy(t)
d = deepcopy(t)
```

"Private" attributes

"Private" attributes

```
from random import randint

class Coin:
    def __init__(self):
        self.__c=randint(0,1)  #private

    def get(self):
        return self.__c

    def toss(self):
        self.__c=randint(0,1)

    def setc(self,value):
        if value == 0 or value ==1:
            self.__c=value
```

"Private" attributes

An attribute which

- starts with dunder
- *does not end* with dunder

is *mangled* by the Python machine, so that it appears "private" to the definition class

"Private" attributes

An attribute

__name

is usable (is seen) "as is" *only in the class in which it is defined (in all its methods)*

It is not visible outside the class

(not even in subclasses:

even if we don't know yet what they are ;-)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

"Private" attributes

An attribute

__name

- is *mangled* by the Python machine
- is usable (is seen) as is *only in the class …*

A __name attribute inside the class CLASSNAME is mangled into

_CLASSNAME__name

The command `class`

Introduces a class definition:

```
class name:
        commands
        method definitions
```

Class attributes vs instance attributes

All instances of a class share an attribute
modifications to the attribute are immediately shared by all instances

"class attribute" (or static attribute, kind of Java terminology)

Class attributes are accessed by dot notation using the class name

```
class Cat:
    legs = 4 #class attribute

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def f(self):
        print("All cats have", Cat.legs, "legs")
```

Class attributes vs instance attributes

```
class Cat:
    legs = 4 # class attribute

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def f(self):
        print("All cats have", Cat.legs, "legs")


print(Cat.legs) #...even without any instance
lulu = Cat("Lulù", "gray") #instance of cat
lulu.f()
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Class attributes vs instance attributes

```
class Cat:
    legs = 4 #class attribute

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def f(self):
        print("All cats have", Cat.legs, "legs")


lulu = Cat("Lulù", "gray")
mimi = Cat("Mimì", "red")
lulu.f()
mimi.f()
Cat.legs = 3
lulu.f()
mimi.f()
```

Class attributes vs instance attributes

```
class Cat:
    legs = 4 # class attribute

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def f(self):
        print("All cats have", Cat.legs, "legs")


lulu = Cat("Lulù", "gray")
print(lulu.legs) #non-standard way of accessing class
                                                 attribute
lulu.legs = 42 #new instance attribute
print(lulu.legs) #instance shadows class attribute
print(Cat.legs) #accessing class attribute
```

Class attributes vs instance attributes

Class attributes are accessed by dot notation using the class name

You cannot use the non-qualified name, even in the class definition

```
class Cat:
    legs = 4 # class attribute

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def f(self):
        print("All cats", legs, "legs") #wrong! use Cat.legs
```

There is no scope nesting for class definitions

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Method definitions: fine points

They are like function definitions

(well: syntactically *they are* function definitions)

they are attributes like any other attribute

To call a method *as a function:* fully qualify its name with the class name: *they are class attributes*

You must provide the instance as first argument

```
p=Point(0,0)
p.move(3)
```

equivalent to `Point.move(p,3)`

Keyword and default parameters

https://www.geeksforgeeks.org/default-arguments-in-python/