ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 7

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

14 October 2024

Michael Lodi

Department of Computer Science and Engineering

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

One-slide recap of lesson 6

Class («type») defines:

- structure of objects (**attributes**)
- available operations (**methods**)

Objects are the values (better, the **instances**)

`__init__(self, …)` method is called when an object is created. I can use the name of the class as a function to instantiate an object of that class

Every method has `self` as the first parameter, which is bound (automatically) to the instance on which the method is called

Attributes are created using the dot notation (e.g. `self.x`) on the LHS of an assignment.

«Dunder methods» / «Magic methods» in the form `__<name>__` are special purposes. Used eg to determine the behaviour of operators on objets (e.g. `a.__add__(b)` ➔ `a+b` and depends on the type of `a`)

«Private» attributes can be created using `__` just before the name. However, they can be accessed anyway because they are just mangled as `_<ClassName>__<attribute>`

Class attributes are defined at top level inside the class (outside methods) and can be accessed using the class name before the dot. Their value is shared between every instance

There is no scope nesting for class definitions

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

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

Next week: still subject to small probability of change, but…:

Monday, 21 October 2024	15:00 - 18:00 Room 0.5 (regular)
Tuesday, 22 October 2024	**09:15 - 12:00 Room 0.5 (Covering prof Chesani)**
Thursday, 24 October 2024	09:15 - 11:30 Room 2.8 (regular)

ALL UPDATES ALREADY
ON THE COURSE WEBSITE

Use this time to try exercises on Virtuale ;-)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

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

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

subclasses

inheritance

dynamic method lookup

Extending a class

```
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
```

We want to define a new class:

same data, more operations:

A method to move the point to the origin

Extending a class

```
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
```

A new class: same data, more operations.
A method to move the point to the origin

```
class NewPoint:
    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 origin(self):
        self.x,self.y=0,0
```

Subclasses

Any `NewPoint` *could be seen* as a `Point`

More precisely:

any operation (method) on `Point` is defined

(it makes sense) also on a `NewPoint`

But *for Python*, there is no relation between a `Point` and a `NewPoint`

Moreover, *code is replicated* in `Point` and `NewPoint`

Subclasses

Any `NewPoint` could be seen as a `Point`

But *for Python*, there is no relation between a `Point` and a `NewPoint`

We want to make this relation explicit:

define `NewPoint` is a such way that

any instance of `NewPoint` is an instance of `Point`

define `NewPoint` to be a subclass of `Point`

Subclasses are defined by "extension"

```
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

class NewPoint (Point):
    def origin(self):
        self.x,self.y=0,0
```

Subclasses are defined by " extension "

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

Equivalent terminology:

`NewPoint` is a subclass of `Point`

`Point` is the (immediate) superclass of `NewPoint`

`NewPoint` is based on `Point`, is derived from `Point`

`Point` is the base class of `NewPoint`

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Subtyping

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

Instances of `NewPoint` are *also* instances of `Point`

Type of a value: the class in which it has been created
But any object of a class is also an instance of its superclasses

Function `isinstance(`*`object`*`,`*`class`*`)`
generalises the function `type(ob)`

Subtyping: class `object`

`object`: it is a predifined class, top of the hierarchy of classes

any class is a subclass of `object`

```
isinstance(p, object) - True
isinstance(3, object) - True
```

Inheritance

```
class Point:
    def __init__(self,xx,yy):
        self.x=xx
        self.y=yy
    def move (self,delta):
        self.x,self.y= self.x+delta,self.y+delta
class NewPoint(Point):
    def origin(self):
        self.x,self.y=0,0
```

`NewPoint` *inherits* from `Point` attributes and methods

`np=NewPoint(1,2)` where is `__init__` for `NewPoint`?

`np.move(3)` where is `move()` for `NewPoint`?

`np.origin()`

`p=Point(5,6)`

`p.origin()` WRONG!

Method overriding

If we don't like an inherited method:

override its definition

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

class NewPoint(Point):
    def origin(self):
        self.x,self.y=0,0
    def move(self,delta):
        self.x,self.y= self.x+2*delta,self.y+2*delta
```

We inherit what we don't re-define (override)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Modify methods in subclass by adding something

A colored point
(add attribute color, change whoareyou)

```
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

class ColPoint(Point):
```

How can we initialize it?

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Method `super()`

```
class ColPoint(Point):

    def __init__(self,xx,yy,cc):
        self.x = xx
        self.y = yy
        self.color = cc

    def whoareyou (self):
        return (self.x, self.y) + (self.color, )
```

We don't like much this:
we copied __init__ from Point, plus something.

much better: initialize *first* as a `Point`, *then* do more

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Method `super()`

```
class ColPoint(Point):
    def __init__(self,xx,yy,cc):
        super().__init__(xx, yy)
        self.color = cc

    def whoareyou (self):
        return super().whoareyou() + (self.color, )
```

`super()` returns the instance where it is called *viewed in its superclass*

It is a kind of *self*, moved into its superclass

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Remember:

Any instance of a subclass

is also an instance of any of its superclasses

Be careful:

instances of subclasses have *more information which becomes inaccessible when "viewed" from the superclass*

Method `super()`

super() can only be used inside a class definition

General form of `super` (we don't require this at the exam):

usable everywhere

`super(Class,Obj)`

`Obj` must be an instance of `Class`

returns `Obj` viewed in the superclass of `Class`

More of less

`super()` is `super(type(self),self)`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dynamic method lookup

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

class NewPoint(Point):
    . . .
    def move(self,delta):
        self.x,self.y= self.x+2*delta,self.y+2*delta
```

Now we call the method `move(10)` on a name

```
who.move(10)
```

Which of the two "`move`" is actually called?

It depends…
From what?

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dynamic method lookup

Now we call the method `move(10)` on a name

```
who.move(10)
```

Which of the two "`move`" is actually called?

It depends…
From what?

We already saw something similar:

S1+S2

which + is executed? (addition, concatenation, etc.)
It depends…
From what?

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dynamic method lookup

When a method is called on an instance
is the class of the receiving instance to determine *which* definition will be used

The choice cannot happen on the basis of the text only of the program (i.e., statically)

The choice depends from the execution (it is a dynamic selection)

It depends from the "type of self"

Methods (when called via the dot notation) behave differently from functions!

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Example (simple)

```
class A:
    def __init__(self,x):
        self.aa=x
    def m1(self):
        return self.aa
class B(A):
    def m1(self):
        return super().m1()+1
    def m2(self,x):
        return self.aa+x
class D(B):
    def m2(self,z):
        return self.aa+2*z

a=A(10)
print(a.m1())
b=B(10)
print(b.m1())
d=D(20)
print(d.m1())
print(d.m2(3))
```

A fundamental example: *late binding of self*

```
class E:
    def __init__(self,y):
        self.ee=y
    def f(self):
        return self.ee
    def g(self):
        return self.f()
class H(E):
    def f(self):
        return 100

e=E(10)
print(e.g())
h=H(20)
print(h.g())
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

dynamic method

lookup (or dispatch, or selection)

is the heart of object oriented programming

Let's remove f() from E in our example

```
class E:
    def __init__(self,y):
        self.ee=y
    def f(self):
        return self.ee
    def g(self):
        return self.f()
class H(E):
    def f(self):
        return 100

e=E(10)
print(e.g())
h=H(20)
print(h.g())
```

Let's remove f() from E in our example

```
class E:
    def __init__(self,y):
        self.ee=y
```

Correct Python!

```
    def g(self):
        return self.f()
class H(E):
    def f(self):
        return 100

e=E(10)
print(e.g())
h=H(20)
print(h.g())
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Let's remove f() from E in our example

```
class E:
    def __init__(self,y):
        self.ee=y


    def g(self):
        return self.f()
class H(E):
    def f(self):
        return 100

e=E(10)
print(e.g())
h=H(20)
print(h.g())
```

Correct Python!

We say that

E *delegates* f

or: g in E delegates f

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Let's remove f() from E in our example

```
class E:
    def __init__(self,y):
        self.ee=y

    def g(self):
        return self.f()
class H(E):
    def f(self):
        return 100

~~e=E(10)~~
~~print(e.g())~~
h=H(20)
print(h.g())
```

Correct Python!

g cannot be called on E instances

E *must be subclassed* before use

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Let's remove f() from E in our example

```
class E:
    def __init__(self,y):
        self.ee=y


    def g(self):
        return self.f()
class H(E):
    def f(self):
        return 100

e=E(10)
print(e.g())
h=H(20)
print(h.g())
```

Correct Python!

E *must be subclassed* before use

Can we express *inside the language* this "must be subclassed before use" ?

Abstract classes: just a glimpse (not in the exam)

```
from abc import ABC,abstractmethod

class E(ABC):
    def __init__(self,y):
        self.ee=y
    @abstractmethod    #this is a decorator
    def f(self):
        pass
    def g(self):
        return self.f()
class H(E):
    def f(self):
        return 100

e=E(10) # ERROR:can't instantiate abstract class
h=H(20)
print(h.g())
```

Abstract base classes: just a glimpse

Classes where some methods
are defined without (complete) implementation

Complete body for the abstract method should be provided in *subclasses*

Abstract base classes *cannot be instantiated*

They are *interfaces*: *names* of operations
vs implementation of those names

Abstract base classes: are blueprints, interfaces

```
from abc import ABC,abstractmethod

class MyAbstract(ABC):
    @abstractmethod
    def method1(self):
        pass
    @abstractmethod
    def method2(self):
        pass

class MyConcrete(MyAbstract):
    def method1(self):
        real implementation
    def method2(self):
        real implementation
```

Abstract base classes: are blueprints, interfaces

```
class MyAbstract(ABC): #«complete abstract base class»
    @abstractmethod
    def method1(self):
        pass
    @abstractmethod
    def method2(self):
        pass

class MyConcrete(MyAbstract):
    def method1(self):
        real implementation
    def method2(self):
        real implementation
```

We deal with a contract:
MyConcrete says:
"I will respect the interface
MyAbstract"

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Abstract base classes: just a glimpse

ABCs are *interfaces*: *names* of operations
vs implementation of those names

In Java
Interfaces are an independent concept
lists of names (and types) of methods

A Java class may
- *extends* a (super)class
- *implements* an interface
provides implementations for the methods

Single inheritance

Any class is a subclass of a *single immediate* superclass

That is:

in any class definition

```
class A(superclass):
    <body>
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Multiple inheritance

A class definition may list more than one superclass:

```
class A(B,C):
    <body>
```

Hence A is subclass of both B and C (which in general are not related by a super/subclass relation between them)

Multiple inheritance

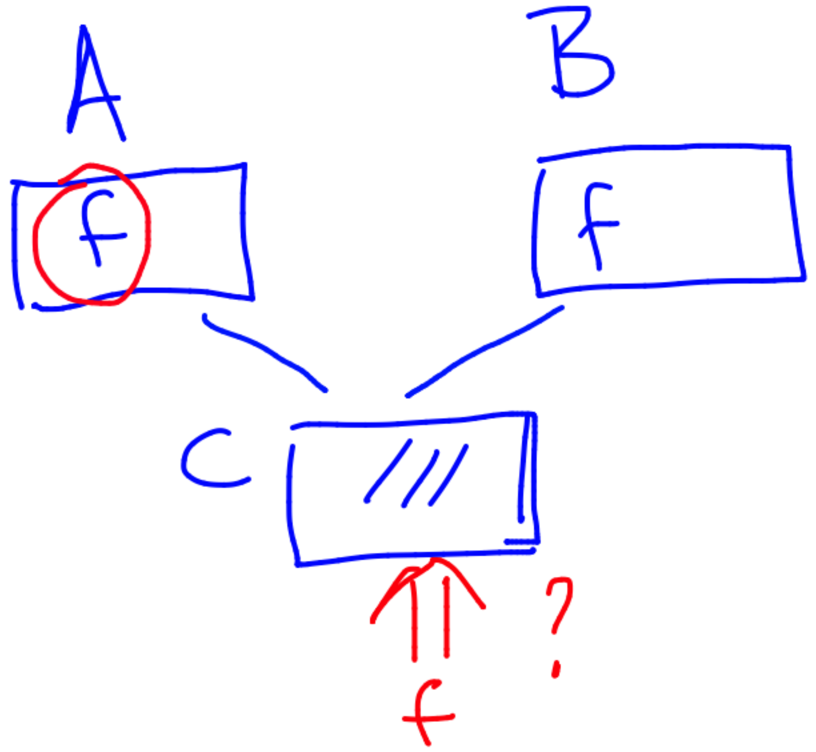
a class inherits from more than one "immediate" superclass

Conceptually simple, but raises problems with non trivial solutions

- who is super() if we have more than one superclass?
- which method is inherited if defined in more than one immediate superclass?
- etc.

Look in the documentation for MRO: Method Resolution Order

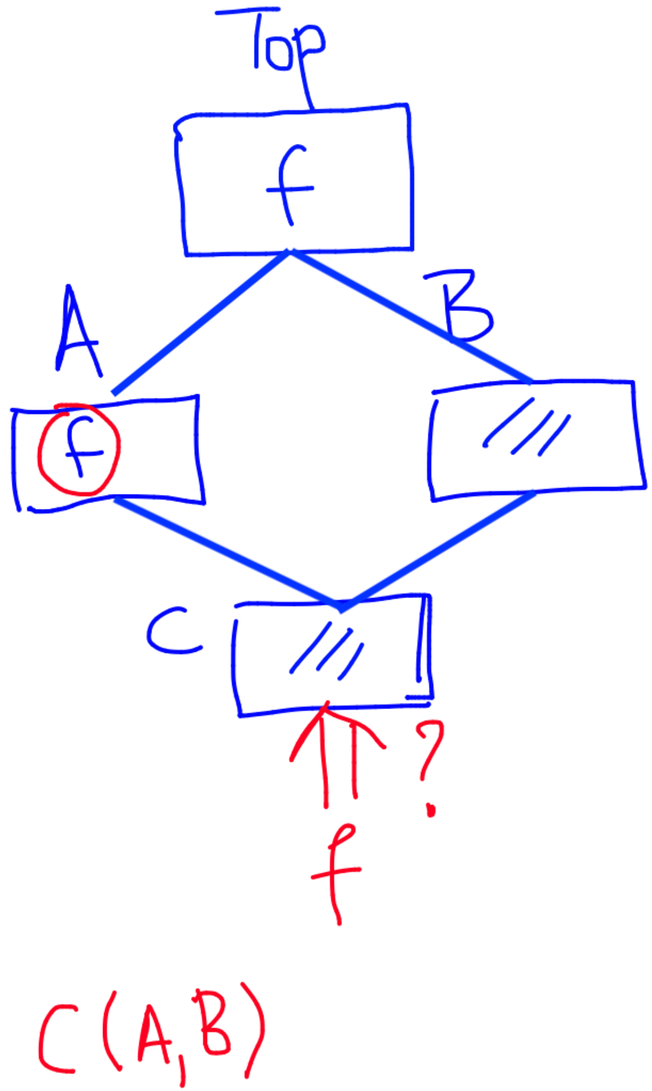
Multiple inheritance: simple examples



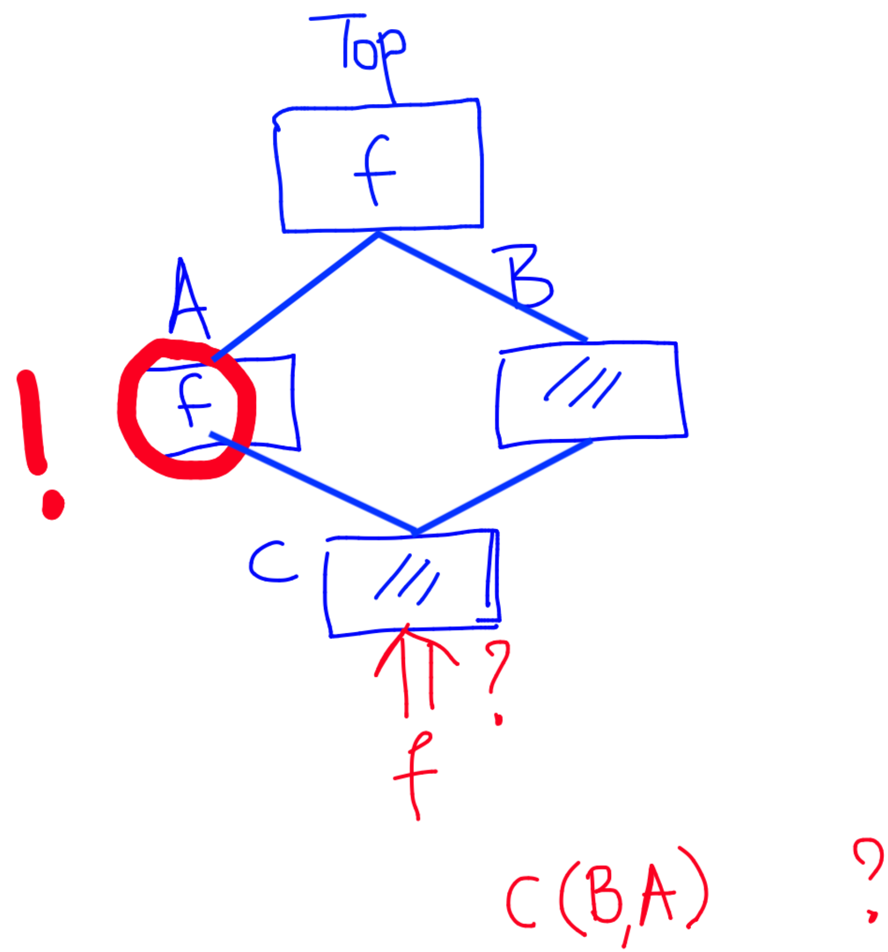
C(A,B)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Multiple inheritance: simple examples



Multiple inheritance: simple examples



Multiple inheritance

- who is super() if we have more than one superclass?

- which method is inherited if defined in more than one immediate superclass?

- etc.

The language define an official

Method Resolution Order

Look in the documentation for MRO

Class method: `class.mro()`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Multiple inheritance

There are cases where the MRO is not defined
Hence they are not legal programs

```
class CC:
    pass
class DD:
    pass
class AA(CC,DD):
    pass
class BB(DD,CC):     #consistent if we swap
  DD,CC
    pass
class EE(AA,BB):
    pass
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

If you want more [Gabbrielli Martini, p.300]

Method Resolution Order in Python

The changes in the definition of the MRO across various versions of Python are a good example of the subtleties of multiple inheritance. Let us recall that an MRO algorithm computes, for any class definition, the order in which its superclasses are visited to decide from which superclass a particular method is inherited. It is, therefore, a way to visit (a portion of) the graph describing the subclass hierarchy, to obtain a linear order which starts at the given class and ends in one maximal class.

In Python 2.2 the MRO algorithm performs a deep-first visit of the graph, respecting the order specified in the class definitions. It was noted, however, that the implemented algorithm was *not* monotonic—there were cases in which the MRO of a subclass was not an extension without re-ordering of the MROs of its superclasses—a requirement which seems instead fundamental. It was thus decided to adopt an algorithm that (i) it is *monotonic*: If `C1` precedes `C2` in the MRO of `C`, then `C1` precedes `C2` in the MRO of any subclass of `C`; and (ii) it *preserves the local precedence ordering*: If we have a definition `A(B,C,D)`, then in `A`'s MRO, `B` precedes `C` which precedes `D`. The result is the MRO algorithm used by default starting from Python 3.0 (and which is the one that was used before for the Dylan and CLOS programming languages.)

Sometimes, however, the two constraints (i) and (ii) above cannot be simultaneously enforced. In this case, the MRO algorithm fails—a program which otherwise would be legal is instead *incorrect*, only because a consistent MRO cannot be computed. One of the simplest such programs is the following.

```
class CC: pass
class DD: pass
class AA(CC,DD): pass
class BB(DD,CC): pass    #consistent if we defined BB(CC,DD)
class EE(AA,BB): pass
```

Indeed, the MRO of `AA` is trivial—local precedence gives `{CC, DD, object}`, in this order, and the same happens for the MRO of `BB`: `{DD, CC, object}`. However, trying to compute the MRO for `EE` we see that monotonicity would impose that both `CC` precedes `DD` (from the MRO of `AA`) *and* `DD` precedes `CC` (from the MRO of `BB`.)

And even more [S. Martini, personal communication]

```
class Top:
    def f(self):
        return 'Top'

class A(Top):
    def f(self):
        return 'A'

class B(Top):
    pass


class C(B,A):
    pass

c=C()
print(c.f())
```

```
class Top:
    def f(self):
        return 'Top'

class A(Top):
    def f(self):
        return 'A'

class B(Top):
    def f(self):
        return super().f()

class C(B,A):
    pass

c=C()
print(c.f())
```

Sets: just a glimpse

Unordered collection with no duplicate elements

Useful to avoid duplicates

Fast operations like on dictionaries

`e = set()` #do not use {}, which is empty dictionary

```
set("abracadabra")
>>> {'d', 'c', 'a', 'r', 'b’}
```

Unlike Python 3 dictionaries, still truly unordered

See docs for operations:
https://docs.python.org/3/tutorial/datastructures.html#sets

Set comprehension

You can have set comprehensions

```
{x for x in 'abracadabra' if x not in 'abc'}
>>> {'r', 'd'}
```

Be careful: no comprehension on tuples

There is no comprehension on tuples

Be careful: no comprehension on tuples

There is no comprehension on tuples

Yet expressions which *seem*

"tuple comprehension"

are legal Python code…

Be careful: no comprehension on tuples

There is no comprehension on tuples

Yet expressions which *seem*

"tuple comprehension"

are legal Python code…

```
G=(i*2 for i in range(10))
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Generators: super simplified view

```
G=(i*2 for i in range(3))
```

A generator is a kind of "*potential sequence/tuple*" which is able to *generate* the elements of the sequence, through the predefined function next(…)

`next(G)` *: evaluates to 0*

`next(G)` *: evaluates to 2*

`next(G)` *: evaluates to 4*

`next(G)` *: the generator is exhausted*
an error is raised

Generators: super simplified view

```
G=(i*2 for i in range(3))
```

Important use: a generator may be the <sequence> used in a for:

```
for e in G:
    print(e)
```

for calls next(G) at any iteration (assign the result to e)

NB: a generator is a particular case of *iterator*, the general structure on which next() is defined and on which we may iterate with a for

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Generators: explicit definition

List comprehension is a shorthand for an explicit iteration involving append()

Is there an explicit version of a generator?

Generators: explicit definition

```
G=(i for i in range(10))
```

```
def genstup(n):
    for i in range(n):
        yield i
G=genstup(10)
```

```
Q=(i**2 for i in range(10) if i%2==0)
```

```
def quadperf(n):
    for i in range(n):
        if i%2==0:
            yield i**2
Q=quadperf(10)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

yield

The command

`yield` *expression*

could be used only inside the body of a function

The presence of `yield` make the `def` define a *generator*

`yield` *expression*

returns the value of *expression* *and* suspends the evaluation of the function (the frame is not destroyed, all the values of the local names are preserved, the point where execution is paused is preserved)

Recall: function

Two linguistic appearences:

Definition

`def` *name* (*formal parameters*) :

body

Use / call

name (*actual paramenters*)

- evaluate the actual parameters
- bind these values to the formal parameters
- evaluate the body of <name> until a `return` is found
- evaluation happens in a local frame, which is destroyed at `return`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Generators: a very peculiar form of functions

Definition:

Any `def` whose body contains (at least) one `yield`

Use / call

(1) A call to the name of the generator (together with its parameters) returns a generator object (the body is *not* evaluated yet!)

(2) successive calls to the function next(G) on the the generator object, cause the execution of the body of the generator, until a `yield` is found

(3) `yield` *expression* suspends the execution of the body, returns the value of *expression* as value of the call to next(G);
the frame of the call is not destroyed and it is resumed at the next call to next(G)

Generators

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fibo_numbers = fibonacci_generator()

for n in fibo_numbers: #infinite loop
    print("Next fibo number is", n)
```

Generators, recall

We have seen *generators*

As generator expressions: e.g.

```
G=(2*i for i in range(10))
```

Or through functions with the `yield` *command*

They are implicit ways to define sequences. Elements are "generated" by calling `next(…)` *on the generator*

An important point: we may use generators as sequences in for statements:

```
for g in G:
    print(g)
```

Iterators

generators are special cases of *iterators*

An iterator is any instance of a class defining the two methods

__iter__

and

__next__

Iterators are the most general form of sequences that may appear in a for statement

An iterator

```
class MyIter:  #we iterate on even int in an interval
    def __init__(self,start,stop):
        if start%2!=0:
            start+=1
        self.x=start
        self.stop=stop
    def __iter__(self):  # return the iterator
        return self
    def __next__(self):  # return the "next" object
        if self.x<self.stop:
            curr=self.x
            self.x+=2
            return curr
        else:
            raise StopIteration
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

An iterator

```
F=MyIter(1,6)

for i in F:
    print(i)              # 2, 4
for i in F:               # F is already exhausted
    print('here I am')  # never executed
```

or, maybe better:

```
for i in MyIter(1,6):
    print(i)
```

An iterator

```
F=MyIter(1,6)

for i in F:
    print(i)                    # 2, 4
```

When the heading of `for` is executed,
it calls `__iter__` on `F`
and stores away ("freezes") this object

At any iteration, `__next__` is called on that object

Iteration stops when the error `StopIteration` is raised

Iterators out of standard sequences (iterables)

```
for i in (1,2,3):
    print(i)
```

When the heading of `for` is executed,
it calls `__iter__` on the sequence,
and stores away ("freezes") this object

```
(1,2,3).__iter__()
```

At any iteration, `__next__` is called on that object

Iteration stops when the error `StopIteration` is raised

Generators are iterators

The language handles *generators* in such a way to define for them the methods

`__iter__` and `__next__`

For `G` generator,

`next(G)` is nothing but `G.__next__()`

Generators are iterators

The language handles *generators* in such a way to define for them the methods

`__iter__` and `__next__`

For `G` generator,

`next(G)` is nothing but `G.__next__()`

The fact that generators are iterators may be expressed *inside* the language through the class hierarchy

(we see this in a moment)

Programming: generators vs iterators

When using

- generators?
- more general iterators?
- simple sequences?

Programming: generators vs iterators

```
class MyIter:
    def __init__(self,start,stop):
        if start%2!=0:
            start+=1
        self.x=start
        self.stop=stop
    def __iter__(self):  # return the iterator
        return self
    def __next__(self):  # return the "next" object
        if self.x<self.stop:
            curr=self.x
            self.x+=2
            return curr
        else:
            raise StopIteration
```

Programming: generators vs iterators

An equivalent generator

```
def MyGen(start,stop):
    if start%2!=0:
        start+=1
    return (i for i in range(start,stop,2))
for k in MyGen(1,6):
    print(k)
```

Or a just forget about generators/iterators and use simple programming…

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Programming: generators vs iterators

Our toy case:

generator is clearer, simpler

even plain sequences would do

Use generators (vs simple sequences) when the "generated" sequence is *complex*

ex: the primes numbers; Fibonacci numbers;
those strings which starts with a letter and …

Use iterators (vs generators) when you need *other methods* (besides __iter__ and __next__) in their class

Programming: generators vs iterators

```
class MySomething: #it is also an iterator
    def __init__(self,start,stop):
        . . .
    def __iter__(self):  # return the iterator
        . . .
    def __next__(self):  # return the "next"
        . . .
    def myothermetod_1(self,x1,x2):
        . . .
    def myothermetod_2(self):
        . . .
```

ABCs and iterators

Express *inside the language* that a class define an iterators.

That is, that it defines __iter__ and __next__ methods

ABCs and iterators

Express *inside the language* that a class define an iterators.

That is, that it defines __iter__ and __next__ methods

Remember: Abstract Base Classes

- some methods *are defined without implementation*
- Implementation is provided in *subclasses*
- Abstract base classes *cannot be instantiated*
- *they are a kind of interfaces*

The ABC for iterators is defined a library

ABCs and iterators

```
class MyIter:
    def __init__(self,start,stop):
        if start%2!=0:
            start+=1
        self.x=start
        self.stop=stop
    def __iter__(self):  # return iterator
        return self
    def __next__(self):  # return the "next"
        if self.x<self.stop:
            curr=self.x
            self.x+=2
            return curr
        else:
            raise StopIteration
```

ABCs and iterators

```
import collections.abc
class MyIter(collections.abc.Iterator):
# class MyIter:
    def __init__(self,start,stop):
        if start%2!=0:
            start+=1
        self.x=start
        self.stop=stop
    def __iter__(self):  # return iterator
        return self
    def __next__(self):  # return the "next"
        if self.x<self.stop:
            curr=self.x
            self.x+=2
            return curr
        else:
            raise StopIteration
```

Built-in types can be subclassed (of course…)

```
class D(list):
    def f(self):
        for e in self: #why it works?
            print(e)

>>>W=D([1,2,3])  # D inherits __init__ from list
>>>W.f()
1
2
3
>>>
```

Built-in types can be subclassed (of course…)

overriding a pre-defined method:

```
class D(list):
    def f(self):
        for e in self:
            print(e)
    def __len__(self):
        return super().__len__() + 10

>>>W=D([1,2,3])
>>>len(W)
13
```