AA 2022-23

Università degli Studi di Cagliari

Object Oriented Programming with Python

Data base and data analytics

Corso di Laurea IADA

Informatica Applicata e Data Analytics

Prof. Giuliano Armano

Summary of the lesson

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

OOP in Python

OOP

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Short Introduction to OOP

- Instead of starting with ako formal introduction to OOP, let us consider the following example

  Suppose you have to implement geometric shapes, like triangles, squares, and rectangles

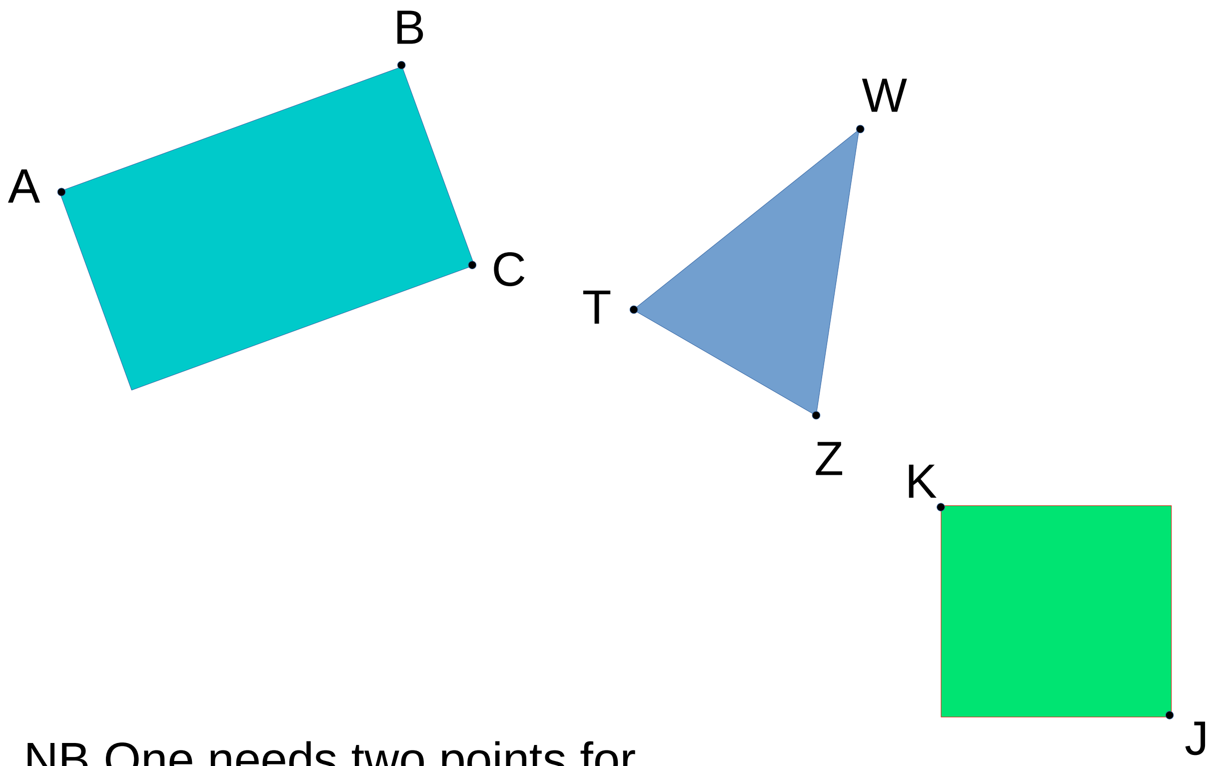
  You must find a way to draw each shape in a canvas (e.g., an arbitrary window over which any shape must be drawn)
- Two solutions

  Procedural

  Object-Oriented

Short Introduction to OOP

- Figures in a canvas … (example)



NB One needs two points for squares and three points for rectangles and triangles ...

Short Introduction to OOP

- A procedural solution
  - Define the procedure draw_triangle
  - Define the procedure draw_square
  - Define the procedure draw_rectangle
- Each procedure will have its own parameters, e.g. two coordinates for squares and three (x,y) coordinates for rectangles and triangles
- Let us suppose that foreground and background color can be specified as well

Short Introduction to OOP

- Code for drawing a triangle

```
def draw_triangle(p1, p2, p3,
                  fg_color='blue', bg_color='green'):
  "Draw a triangle (fg an bg color may be specified)"
  # p1, p2, p3 must be (x,y) coordinates …
  # code for drawing the triangle …
  # the canvas is supposed to be known …
  # actual code for drawing the triangle goes HERE
  return
```

- The same approach **must** be followed to implement
  - draw_square
  - draw_rectangle

Short Introduction to OOP

- Using draw_triangle, draw_square and draw_rectangle ...

```
def foo():
  draw_triangle((0,10),(0,25),(11,30), bg_color= 'red')
  draw_square((35,12),(14,22))
  draw_rectangle((0,15), (20,20), (6,10), 'red', 'grey')
```

- Here each shape requires a specific drawing procedure and may have different parameters ...

Short Introduction to OOP

- A viable alternative for drawing a triangle, a square, or a rectangle using a single procedure ...

```
def draw_shape(shape, points,
                  fg_color='blue', bg_color='green'):
  "Draw a shape (fg an bg color may be specified)"
  # points must be an iterable of (x,y) coordinates …
  assert shape in ('triangle', 'square', 'rectangle')
  kwargs = {'fg_color': fg_color, 'bg_color': bg_color}
  if shape == 'triangle':
    draw_triangle(*points, **kwargs)
  if shape == 'square':
    draw_square(*points, **kwargs)
  if shape == 'rectangle':
    draw_rectangle(*points, **kwargs)
```

Short Introduction to OOP

- Using draw_shape ...

```
def foo():
    draw_shape('triangle', ((0,10),(0,25),(11,30)),
               bg_color= 'red')
    draw_shape('square', ((35,12),(14,22))
    draw_shape('rectangle', ((0,15), (20,20), (6,10)),
               'red', 'grey')
```

- Now the “how-to” is embedded into the procedure draw_shape, which **however** must decide (using conditional statements) which procedure has to be called
- Moreover, the “concepts” of triangle, square and rectangles are spread along the source code instead of being put apart according to the figure to be drawn ...

Short Introduction to OOP

- The object-oriented solution (actually object-based)

```
class Triangle(object):
  def __init__(self, p1, p2, p3):
    "Init the triangle"
    self.p1, self.p2, self.p3 = p1, p2, p3
  def draw(self, fg_color= 'blue', bg_color='green'):
    "Draw the triangle (fg and bg color may be specified)"
    # actual code for drawing the triangle goes HERE
    return
```

- Similar classes can be specified for
  - Squares --> `class Square`
  - Rectangles --> `class Rectangle`

Short Introduction to OOP

- Using draw the object-oriented way ...

```
def foo():
  t = Triangle((0,10),(0,25),(11,30))
  s = Square((35,12),(14,22))
  r = Triangle((0,10),(0,25),(11,30))
  t.draw(bg_color= 'red')
  s.draw()
  r.draw('red', 'grey')
```

- Here each shape **knows** how to draw itself and the same name (i.e., draw) is used to denote the same conceptual operation
- Moreover, all information about a triangle, a square, or a rectangle is embedded into the corresponding object

Short Introduction to OOP

- What is the difference between an object-based and an object-oriented solution? Let’s see ...

```
class Triangle(object):
  'A triangle'
  # same as before

class Rectangle(object):
  'A rectangle'
  # same as before

class Square(Rectangle):
  'A square'
  def __init__(self, p1, p2):
    'Init the square'
    x1, y1 = p1 ; x2, y2 = p2
    super().__init__(p1, (x2,y1), p2)
```

NB The code aside works for the simple case in which the square is not rotated along the x axis. A simple strategy for finding the third point in the general case involves rotations and translations.

Short Introduction to OOP

- Now Square is a subclass of Rectangle
  - In fact, the third point (for the rectangle) can be found from those that define the square
- Note that now Square in fact uses Rectangle::draw to draw itself! This is an example of inheritance (in particular Square specializes Rectangle)
- No change whatsoever occurs at the client side –i.e., the code that creates Triangles, Squares and Rectangles, as well as the code that calls draw, remains the same

---

NB Depending on the number of shared operations, the class hierarchy here could have been also deeper. For instance, one may define Shape as superclass of all geometric shapes, Triangle and Rectangle as subclasses of Shape, and finally Square as subclass of Rectangle.

OOP in Python

OOP

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Classes, Methods and Instances

```
>>> from math import sqrt
>>> class Point(object):
...     def __init__(self,x=0,y=0):
...         self.x, self.y = x,y
...     def distance(self,p):
...         d2 = (self.x-p.x)**2 + (self.y-p.y)**2
...         return sqrt(d2)
...

>>> p1 = Point()
>>> print(p1.x,p1.y)
0 0
>>> p1.distance(Point(1,1))
```

a class

a method

a reference to an object

Classes, Methods, and Instances

- Encapsulation (= class construct) YES
- Information hiding ~NO

Classes, Methods and Instances

Information hiding: private and public slots

```
>>> class Blob(object):
...   def __init__(self):
...     self.public = 'I am public'
...     self.__private = 'I am private'
...

>>> b = Blob()
>>> b.public
'I am public'
>>> b.__private

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in -toplevel- b.__private
AttributeError: Blob instance has no attribute '__private'
>>>
```

This slot is "private" ...

OOP in Python

OOP

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Method Dispatching and Binding

- Method dispatching (single vs. multiple) SINGLE
- Method binding (static vs. dynamic) DYNAMIC

Method Dispatching

```
>>> class Point(object):
...   def __init__(self,x=0,y=0):
...     self.x = x
...     self.y = y
...   def distance(self,p):
...     return sqrt( (self.x-p.x)**2 + (self.y-p.y)**2 )
...

>>> p1 = Point(1,2)
>>> p2 = Point(10,20)
>>> p1.distance(p2)
20.12461179749810
>>> Point.distance(p1,p2)
20.12461179749810
>>>
```

Method Binding

```
>>> class Point(object):
...   def __init__(self,x=0,y=0):
...     self.x, self.y = x,y
...   def distance(self,p):
...     return sqrt((self.x-p.x)**2+(self.y-p.y)**2)
...

>>> class CPoint(Point):
...   def __init__(self,x=0,y=0,color=0):
...     Point.__init__(self,x,y)
...     self.color = color
...
```

Method Binding

```
>>> from math import *
>>> p1 = CPoint()
>>> p2 = Cpoint(2,2)
>>>
>>> print p1.distance(p2)
2.82842712475
>>>
>>> CPoint.distance(p1,p2)
2.82842712475
>>>
>>> Point.distance(p1,p2)
2.82842712475
```

Method Binding

```
>>> class Blob(object):
...   def foo(self):
...     print('This is Blob')
...

>>> class BlobOne(Blob):
...   def foo(self):
...     print('This is BlobOne')
...
```

Method Binding

```
>>> def oops(x):
...     x.foo()
...

>>> a = Blob()
>>> b = BlobOne()
>>>
>>> oops(a)
This is Blob
>>>
>>> oops(b)
This is BlobOne
>>>
```

OOP in Python

OOP

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Inheritance

- Interfaces ~NO
- Constructors inheritance NO
- Multiple inheritance YES

NB A way to simulate interfaces is to make use of abstract base classes (see the abc library)

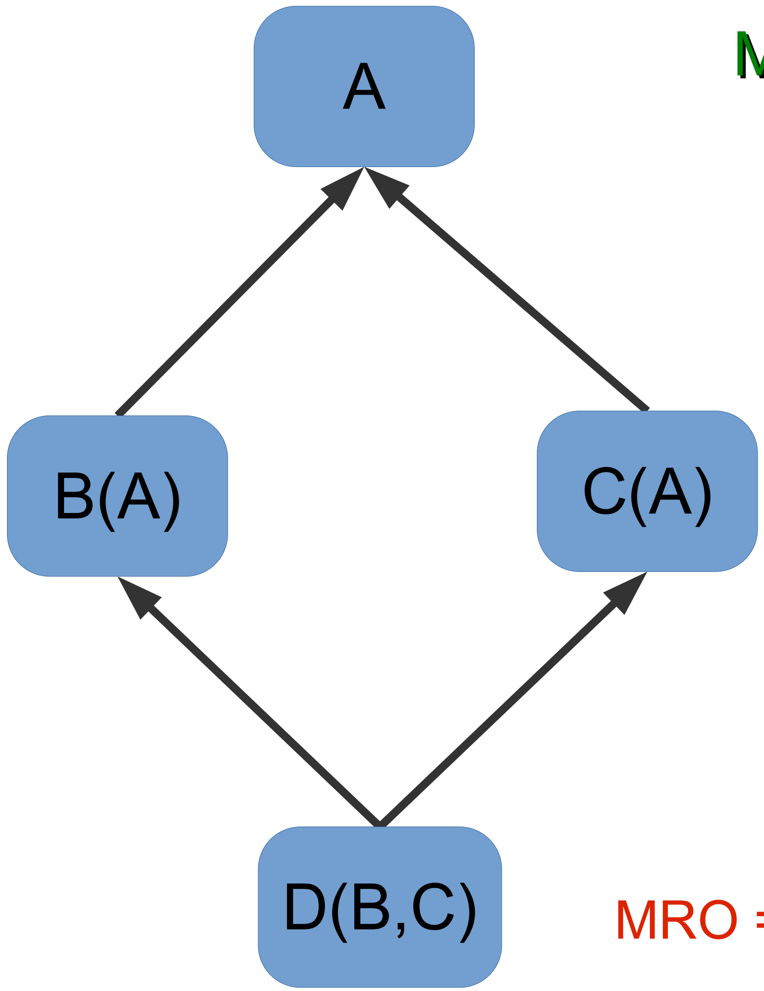
Inheritance

- The Python new programming style requires that a class is directly or indirectly derived from the class named “object”
- Thus, “object” becomes the root of the whole hierarchy of classes
- To find out the order to be followed for searching a method within a hierarchy of classes, the hierachical DAG must be linearized, giving rise to the MRO (Method Resolution Order)

NB DAG = Directed Acyclic Graph

Inheritance – MRO

An example ...



MRO: [D, B, C, A]

NB If A is not object, then further superclasses should be taken into account until object is reached ...

MRO = method resolution order

Inheritance (MRO)

- How the MRO is calculated (abstract view)
  - The MRO algorithm merges the local precedence order of a class with the **linearization** of its direct superclasses
  - When there are several possible choices for the next element of the linearization, the class that has a direct subclass closest to the end of the output sequence is selected

Inheritance (MRO)

- Be C a class
- Be B_1, B_2, ..., B_n superclasses of C
- We want the MRO be monotonic
- An MRO is monotonic when the following is true
  - if B_k precedes B_h in the linearization of C, then B_k precedes B_h in the linearization of any subclass of C

Inheritance (MRO)

- Under the assumption of monotonicity, the linearization of C, say L[C], is obtained by appending to C the result of merging the linearization performed over the parents with the list of parents

Inheritance (MRO)

- In symbols:
  - $L[C(B_1, \ldots, B_N)] = [C] + merge(L[B_1], \ldots, L[B_N], [B_1, \ldots, B_N])$

where

- L[object] = [object]   (root of the hierarchy)
- merge(L[x],[x]) = L[x]   (single inheritance)
- merge(X, Y, ...,, Z) ?   (recursive step)

Inheritance (MRO)

- What about merge(X, Y, ... , Z) ?

  First, we need to define the concepts of head and tail …

- With L = [x, y, z, ...] list of items:

  head(L) = x

  tail(L) = [y,z, ...]

Inheritance (MRO)

- What about merge(X, Y, ... , Z) ?

  First, we need to define the concept of good head

- With W = [A, B, C, D, E] and assuming that each item in W is in fact a list:

  h = head(A) is a good head if it is not in the tail of any of the other lists ...

Inheritance (MRO)

- Merge algorithm
  - Be h the head of the first list found (otherwise stop)
  - If h is not a good head then try to find a good head on the next list and so on until a good head is found (otherwise stop)
  - Add the good head found to the linearization of C and remove it from the lists in the merge
  - Repeat the operations above until all lists are removed or it is impossible to find good heads
  - If it is impossible to construct the merge, Python will refuse to create the class C and will raise an exception

Inheritance (MRO)

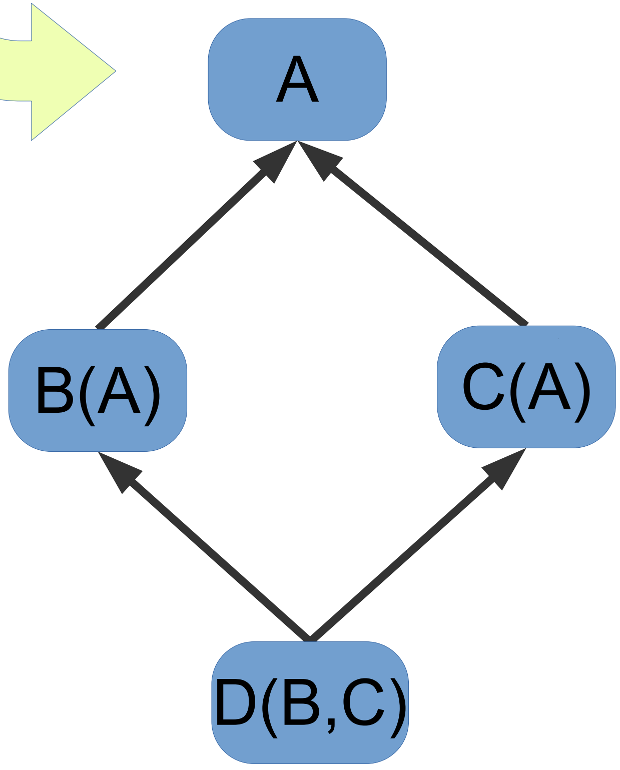
- Let us solve the MRO problem for
  (now going forward)

  L[D(B,C)]
  - L[D(B,C)] = [D] + merge(L[B],L[C],[B,C])

  L[B] = L[B(A)]
  - L[B(A)] = [B] + merge(L[A],[A])

  L[C] = L[C(A)]
  - L[C(A)] = [C] + merge(L[A],[A])

  L[A]
  - L[A] = [A]



Inheritance (MRO)

- Solving the MRO problem ...

  (now going backwards)

  L[A]

  - L[A] = [A]

  L[B(A)]

  - L[B(A)] = [B] + merge(L[A],[A])

    = [B] + merge([A],[A]) = [B,A]

  L[C(A)]

  - L[C(A)] = [C] + merge(L[A],[A])

    = [C] + merge([A],[A]) = [C,A]

Inheritance (MRO)

- Solving the MRO problem ...

  (still going backwards)

  L[D(B,C)]

  - L[D(B,C)] = [D] + merge(L[B],L[C],[B,C])
    = [D] + merge([B,A],[C,A],[B,C])

  B is a good head, hence select it:

  - L[D(B,C)] = [D,B] + merge([A],[C,A],[C])

Inheritance (MRO)

- Solving the MRO problem

  (still going backwards)

  L[D(B,C)]

  - L[D(B,C)] = [D,B] + merge([A],[C,A],[C])

  A is NOT a good head, hence try with another head.

  C is a good head, hence select it:

  - L[D(B,C)] = [D,B,C] + merge([A],[A],[])

  A is NOW a good head, hence select it:

  - L[D(B,C)] = [D,B,C,A] + merge([],[],[]) = [D,B,C,A]

See also: http://en.wikipedia.org/wiki/C3_linearization

Inheritance (MRO)

- Beyond formalizations and algorithms …
  - The previous implementation of class inheritance handling (until Python 2.3) was following a depth first approach

    For instance, in the previous example, the MRO would be:

    [D, B, A, C]
  - The current implementation of class inheritance handling (from Python 2.3) follows a breadth first approach

    For instance, in the previous example, the MRO would be:

    [D, B, C, A]

Inheritance (MRO)

- How to find out the MRO for a class

```
>>> class A(object): pass
>>> class B(A): pass
>>> class C(A): pass
>>> class D(B,C): pass
>>> print(D.__mro__)
(<class '__main__.D'>, <class '__main__.B'>,
<class '__main__.C'>, <class '__main__.A'>, <class
'object'>)
```

OOP in Python

OOP

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Polymorphism

- Universal
  - Parametric Class NO
  - By Inclusion YES
- Ad-Hoc
  - Overloading ~NO
  - Coercion ~YES

Inclusion Polymorphism

```
>>> class B(object):
...     def method1(self):
...         print('method1 of B')
...

>>> class D(B):
...     def method1(self):
...         print('method1 of D')
...

>>> d = D()
>>> d.method1()
method1 of D
```

Inclusion Polymorphism

```
>>> class B(object):
...   def method1(self):
...     print('method1 of B')
...

>>> class D(B):
...   def method1(self):
...     print('method1 of D')
...

>>> b = B()
>>> b.method1()
method1 of B
```

Overloading

```
>>> class bop(object):
...   def goo(self):
...     print('This is goo w/out parameters')
...   def goo(self,w,z):
...     print('This is goo with parameters')
...

>>> b = bop()
>>> b.goo(100,200)
This is goo with parameters
>>> o.goo() # NOT WORKING ...
TypeError: goo() missing 2 required positional arguments: 'w'
and 'z'
```

Overloading

```
>>> class bip(object):
...   def foo(self,x,y):
...     print('This is bip.foo, with parameters')
...
>>> class oops(bip):
...   def foo(self):
...     print('This is oops.foo, w/out parameters')
...
>>> o = oops()
>>> bip.foo(o,10,20)
This is bip.foo, with parameters
>>> o.foo(10,20) # NOT WORKING ...
TypeError: foo() takes 1 positional argument but 3 were given
```

Coercion/Conversion

- Conversion:

```
>>> a = 10
>>> b = float(a)
>>> b
10.0
```

- Coercion:

```
>>> x = 1
>>> y = 2.3
>>> print(x+y)
3.3
>>>
```

OOP in Python

OOP

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Comparison Operators

```
__lt__(a, b)     # a < b
__le__(a, b)     # a ≤ b
__eq__(a, b)     # a == b
__ne__(a, b)     # a != b
__ge__(a, b)     # a ≥ b
__gt__(a, b)     # a > b
```

Logical Operators

```
__and__(a, b)    # a and b
__or__(a, b)     # a or b
__xor__(a, b)    # a xor b
__not__(a, b)    # not a
```

Arithmetic Operators

```
__add__(a, b)     # a + b
__sub__(a, b)     # a - b
__mul__(a, b)     # a * b
__div__(a, b)     # a / b
__abs__(a)        # abs(a)
__mod__(a, b)     # a % b
```

Operators Redefinition (an example)

- Many operators can be redefined like C++ does ...

```
>>> class Blob(object):
...   def __init__(self,x=0):
...     self.x = x
...   def __add__(self,y):
...     return self.x + y
...

# continues on next slide ...
```

Operators Redefinition (an example)

- Many operators can be redefined like C++ does ...

```
# now let’s define a Blob object an try the “+” op ...

>>> a = Blob()
>>> print(a.__add__(1))
1
>>> print(a+1)
1
```

Operators Redefinition (an example)

- Some important operators ...
  - __init__ object constructor (in fact, object initializer)
  - __call__ make an object behave as a function
  - __iter__ make an object iterable
  - __getitem__ make an object behave as a dict (getter)
  - __setitem__ make an object behave as a dict (setter)
  - __len__ get the length of an object
  - __str__ turn an object into a string (e.g., for printing)
  - __repr__ return object information as string
  - __lshift__ customize the “<<” operator
  - ... etc ...

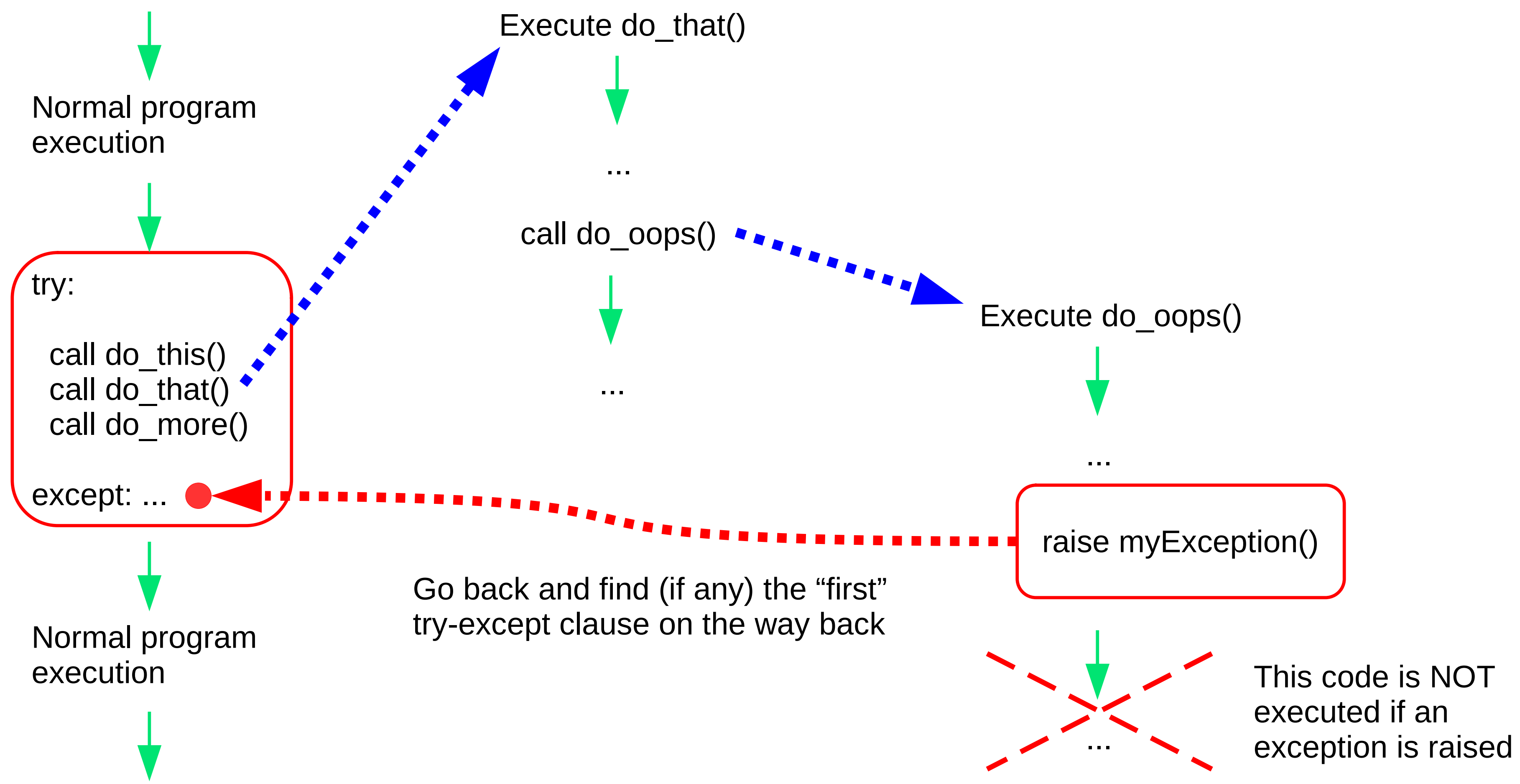
OOP in Python

OOP

- Short introduction to Object Oriented Programming
- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Exception Handling

- Exception handling allows to control the way a program deals with unexpected situations



Exception Handling

- Typical try-except clause in Python

```
…
… (some source code)
…
try:
  # here goes code under check
  # typically more than one call goes here ...
except myException as e:
  # code to execute IF an exception is raised
else:
  # code to execute IF no exceptions are raised
finally:
  # code to execute in any case …
…
… (more source code)
…
```

Exception Handling

- How to raise an exception in Python

```
…
… (some source code)
…
if something_bad_happens():
  raise myException()
…
… (more source code)
…
```

For more information see https://docs.python.org/3/tutorial/errors.html