DMI - Università degli Studi di Cagliari

OOP and Scripting in Python

Part 2 - OOP Features

Giuliano Armano – DMI Univ. di Cagliari

Part 2 – OOP Features

Python: OOP Features

- Classes, Methods, and Instances
- Methods Dispatching and Binding
- Inheritance
- Polymorphism
- Operators Handling
- Exception handling

Classes, Methods, and Instances

Part 2 – OOP Features: Classes, Methods, and Instances

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" ...

Methods Dispatching and Binding

Part 2 – OOP Features: Methods

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 8
>>> Point.distance(p1,p2)
20.12461179749810 8
>>>
```

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
>>>
```

Inheritance

Part 2 – OOP Features: Inheritance

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

Inheritance (MRO)

- Python new-style subclassing resorts to linearization
  - 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

MRO = Method Resolution Order

Inheritance (MRO)

- Be C a class
- Be B_1, B_2, ..., B_n superclasses of C
- A 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:
  - $L[C(B_1, ..., B_N)] =$

    $[C] + merge(L[B_1], ... ,L[B_N], [B_1, ... , B_N])$

  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 W list of items, e.g. W = [a, b, c, d, e]

- head(W) = a
- tail(W) = [b, c, d, e]

Inheritance (MRO)

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

  Then, we need to define the concept of “good head”

  With W list of items (assume that each item is in fact a list), e.g. W = [A, B, C, D, E]

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

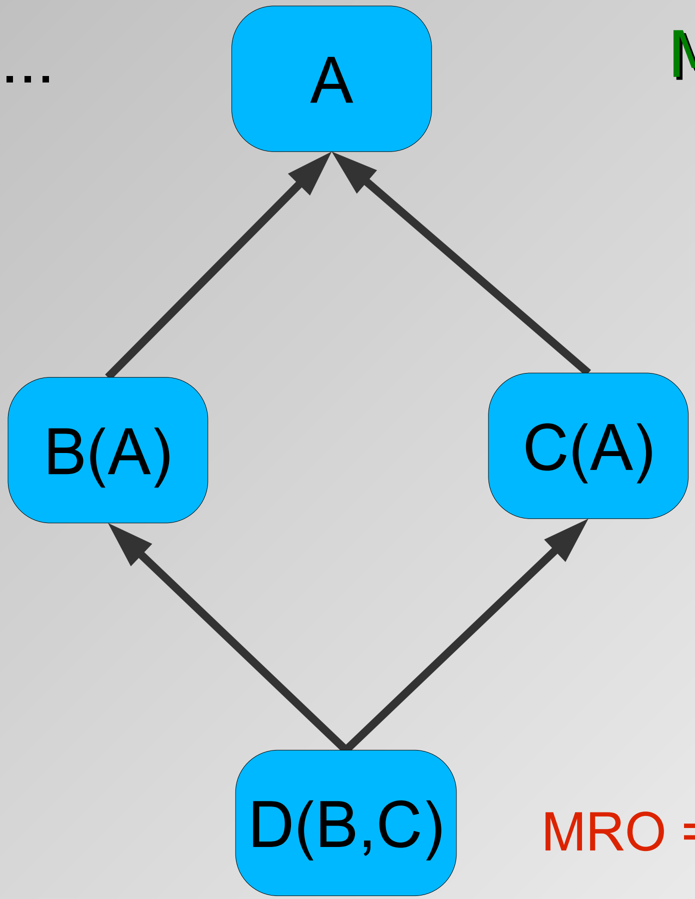
Inheritance (MRO)

- 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

An example ...

MRO: [D, B, C, A]



MRO = method resolution order

Inheritance (MRO)

- Beyond formalizations and algorithms …
  - The previous implementation of class inheritance handling 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 follows a breadth first approach

    For instance, in the previous example, the MRO would be: [D, B, C, A]

Inheritance (MRO)

- Let us solve the MRO problem ..

(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

Polymorphism

Part 2 – OOP Features: Polymorphism

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
>>>
```

Operators Handling

Part 2 – OOP Features: Exceptions 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
```