COE 302 Object Oriented Programming with Python II

Akinola Oluwole

soakinola@abuad.edu.ng

Content

- Inheritance and class hierarchies
- …

Inheritance and class hierarchies

- A family of classes is known as a class hierarchy.
- Child classes can inherit data and methods from parent classes
- They can modify these data and methods, and they can add their own data and methods.
- other parts of the code do not need to distinguish whether an object is the parent or the child - all generations in a family tree can be treated as a unified object.
- A parent class is usually called *base class* or *superclass*, while the child class is known as a *subclass* or *derived class*.

A class for straight lines

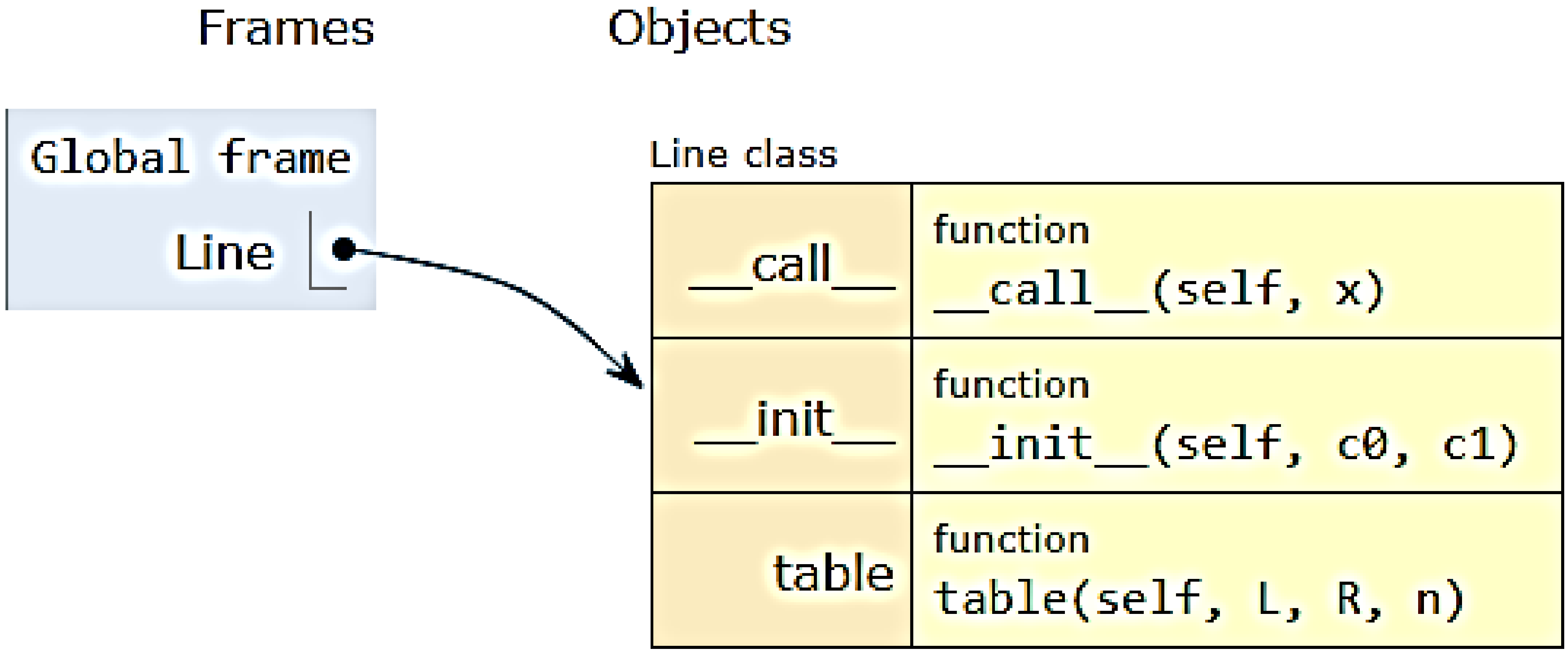
- Assume that we have written a class for straight lines,

  $y = c_0 + c_1 x$:
- The constructor `__init__` initializes the coefficients `c0` and `c1` in the expression for the straight line: $y = c_0 + c_1 x$:.
- The call operator `__call__` evaluates the function $c_0 + c_1 x$, while the `table` method samples the function at `n` points and creates a table of `x` and `y` values.
- A function is a concrete example of a Callable class. Any object can be made callable by simply giving it a `__call__` method that accepts the required arguments.

```python
class Line(object):
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```


Frames
Objects
Global frame
Line
Line class
__call__
function
__call__(self, x)
__init__
function
__init__(self, c0, c1)
table
function
table(self, L, R, n)

Class for parabolas

- A parabola $y = c_0 + c_1 x + c_2 x^2$ contains a straight line as a special case $(c_2=0)$.
- A class for parabolas will therefore be similar to a class for straight lines. Adding the new term c_2x^2 in the function evaluation and store c_2 in the constructor:
- The table method from class Line is added without any modifications

```
class Parabola(object):
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

A class for parabolas using inheritance

- Class Parabola does not need to repeat the code already written in class Line.
- We can specify that class Parabola inherits all code from class Line by adding (Line) in the class headline:

```
class Parabola(Line):
    pass
```

- To explicitly demonstrate the validity of the assertion, The concept of inheritance is applied.
- Class Parabola is derived from class Line, or equivalently, that class **Parabola** is a subclass of its superclass **Line.**
- Class Parabola should not be identical to class Line: it needs to add data in the constructor (for the new term) and to modify the call operator (because of the new term), but the table method can be inherited as it is.

A class for parabolas using inheritance Contd.

- If we implement the constructor and the call operator in class **Parabola**, these will override the inherited versions from class **Line**.
- If we do not implement a table method, the one inherited from class **Line** is available as if it were coded visibly in class **Parabola**.
- Class **Parabola** must first have the statements from the class **Line** methods `__call__` and `__init__`, and then add extra code in these methods.
- **Avoiding repeating code** therefore call up functionality in class **Line** instead of copying statements from class **Line** methods to **Parabola** methods.

A class for parabolas using inheritance Contd.

- Any method in the superclass **Line** can be called using the syntax

```
Line.methodname(self, arg1, arg2, ...)
```

- # or

```
super(Parabola,   self).methodname(arg1,   arg2,
...)
```

- The Class show how to write class **Parabola** as a subclass of class **Line**, and implement just the new additional code and that is not already written in the **superclass**

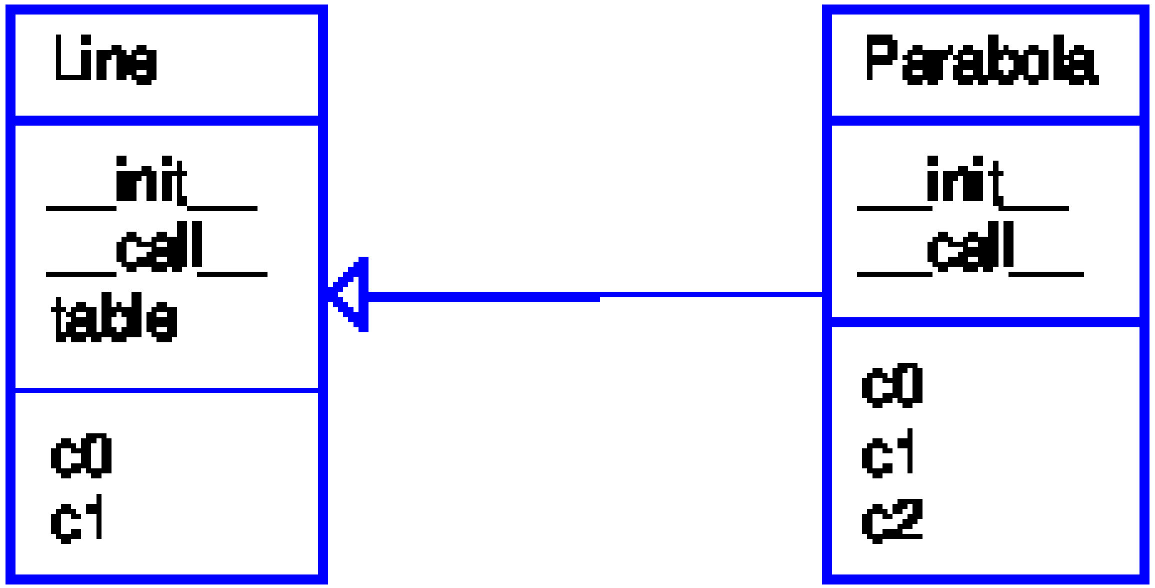
A class for parabolas using inheritance Contd.

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)  # let Line store c0 and 
c1
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

A class for parabolas using inheritance Contd.

- This short implementation of class Parabola provides exactly the same functionality as the first version of class Parabola
- The figure shows the class hierarchy in UML fashion.
- The arrow from one class to another indicates inheritance.



Program flow

- The program flow can be somewhat complicated when we work with class hierarchies. Consider the code segment

- `p = Parabola(1, -1, 2)`
- `p1 = p(x=2.5)`

A class for parabolas using inheritance Contd.

- Calling Parabola(1, -1, 2) leads to a call to the constructor method __init__,
- where the arguments c0, c1, and c2 in this case are int objects with values 1, -1, and 2.
- The self argument in the constructor is the object that will be returned and referred to by the variable p.
- Inside the constructor in class **Parabola** we call the constructor in class **Line**. In this latter method, we create two data attributes in the self object.
- Printing out dir(self) will explicitly demonstrate what self contains so far in the construction process.
- Back in class Parabola's constructor, we add a third attribute c2 to the same self object. Then the self object is invisibly returned and referred to by p.

A class for parabolas using inheritance Contd.

- The other statement, *p1 = p(x=2.5)*, has a similar program flow.
- First we enter the *p.__call__* method with self as p and x as a float object with value 2.5.
- The program flow jumps to the *__call__* method in class Line for evaluating the linear part $c_0 + c_1x$ of the expression for the parabola, and then the flow jumps back to the *__call__* method in class Parabola where we add the new quadratic term.

Attribute vs inheritance: has-a vs is-a relationship

- Whether to use inheritance or an attribute depends on the problem being solved.
- If it is natural to say that class **Parabola** is a **Line** object, we say that **Parabola** has an is-a relationship with class **Line**. Alternatively, if it is natural to think that class **Parabola** has a **Line** object, we speak about a has-a relationship with class **Line**.
- In the present example, we may argue that technically the expression for the **parabola** is a straight line plus another term and hence claim an is-a relationship, but we can also view a **parabola** as a quantity that has a line plus an extra term, which makes the has-a relationship relevant.

Attribute vs inheritance: has-a vs is-a relationship

- How classes depend on each other is influenced by two factors: sharing of code and logical relations. From a sharing of code perspective, many will say that class Parabola is naturally a subclass of Line, the former adds code to the latter. On the other hand, Line is naturally a subclass of Parabola from the logical relations in mathematics.
- Computational efficiency is a third perspective when we implement mathematics. When Line is a subclass of Parabola we always evaluate the c_2x^2 term in the parabola although this term is zero.

Attribute vs inheritance: has-a vs is-a relationship

- Nevertheless, when Parabola is a subclass of Line, we call Line.__call__ to evaluate the linear part of the second-degree polynomial, and this call is costly in Python. From a pure efficiency point of view, we would reprogram the linear part in Parabola.__call__.
- This little discussion here highlights the many different considerations that come into play when establishing class relations.

Extension and restriction of a superclass.

- In the example where **Parabola** as a subclass of **Line**, we used inheritance to extend the functionality of the superclass. The case where **Line** is a subclass of **Parabola** is an example on restricting the superclass functionality in a subclass.

Superclass for defining an interface

- As another example of class hierarchies, we now want to represent functions by classes, but in addition to the __call__ method, we also want to provide methods for the first and second derivative. The class can be sketched as

Superclass for defining an interface

```
class SomeFunc(object):
    def __init__(self, parameter1, parameter2, ...)
        # Store parameters
    def __call__(self, x):
        # Evaluate function
    def df(self, x):
        # Evaluate the first derivative
    def ddf(self, x):
        # Evaluate the second derivative
```

Superclass for defining an interface

- For a given function, the analytical expressions for first and second derivative must be manually coded. However, we could think of inheriting general functions for computing these derivatives numerically, such that the only thing we must always implement is the function itself. To realize this idea, we create a superclass

```
class FuncWithDerivatives(object):
    def __init__(self, h=1.0E-5):
        self.h = h  # spacing for numerical derivatives
    def __call__(self, x):
        raise NotImplementedError('___call__ missing in class %s' %
self.__class__.__name__)
    def df(self, x):
        """Return the 1st derivative of self.f."""
        # Compute first derivative by a finite difference
        h = self.h
        return (self(x+h) - self(x-h))/(2.0*h)
    def ddf(self, x):
        """Return the 2nd derivative of self.f."""
        # Compute second derivative by a finite difference:
        h = self.h
        return (self(x+h) - 2*self(x) + self(x-h))/(float(h)**2
```

Example II

- For a more complicated function, e.g., $f(x) = \ln|p\tanh(qx\cos rx)|.$, we may skip the analytical derivation of the derivatives, and just code $f(x)$ and rely on the difference approximations inherited from the superclass to compute the derivatives:

```
class MyComplicatedFunc(FuncWithDerivatives):
    def __init__(self, p, q, r, h=1.0E-5):
        FuncWithDerivatives.__init__(self, h)
        self.p, self.q, self.r = p, q, r

    def __call__(self, x):
        return log(abs(self.p*tanh(self.q*x*cos(self.r*x))))
```

- Class MyComplicatedFunc inherits the df and ddf methods from the uperclass FuncWithDerivatives.
- These methods compute the first and second derivatives approximately, provided that we have defined a __call__ method.
- If we fail to define this method, we will inherit __call__ from the superclass, which just raises an exception, saying that the method is not properly implemented in class MyComplicatedFunc.
- The important message in this subsection is that we introduced a super class to mainly define an *interface*, i.e., the operations (in terms of methods) that one can do with a class in this class hierarchy.

- The superclass itself is of no direct use, since it does not implement any function evaluation in the `__call__` method.
- However, it stores a variable common to all subclasses (`h`), and it implements general methods `df` and `ddf` that any subclass can make use of.
- A specific mathematical function must be represented as a subclass, where the programmer can decide whether analytical derivatives are to be used, or if the more lazy approach of inheriting general functionality (`df` and `ddf`) for computing numerical derivatives is satisfactory.
- In object-oriented programming, the superclass very often defines an interface, and instances of the superclass have no applications on their own - only instances of subclasses can do anything useful.

Quadratic Equation Exercise

- $-\frac{b}{2} \pm \frac{\sqrt{b^2-4ac}}{2a}$

- Following the examples in previous sections, write a quadratic equation class. Explain your code line after line. Deadline Monday 27th, April 2020 (My office).
- There will be penalty for late submission

Reference

- Object-oriented programming (hplgit.github.io)
- http://www.pythontutor.com/visualize.html
- Class and Object — Python Numerical Methods (berkeley.edu)