OOP principles (again)

- ***encapsulation***: hiding design details to make the program clearer and more easily modified later
- ***modularity***: the ability to make objects "stand alone" so they can be reused (our modules). Like the math module
- ***inheritance***: create a new object by inheriting (like father to son) many object characteristics while creating or over-riding for this object
- ***polymorphism***: (hard) Allow one message to be sent to any object and have it respond appropriately based on the type of object it is.

There are now three groups in our coding scheme:

- user
- programmer, class user
- programmer, class designer

- The class designer is creating code to be used by other programmers

- In so doing, the class designer is making a kind of library that other programmers can take advantage of

encapsulation

- Hiding the details of what the message entails means that changes can be made to the object and the flow of messages (and their results) can stay the same
- Thus the implementation of the message can change but its intended effect stay the same
- If the class is well designed, then a <u>user</u> of the class need only use the provided methods to use the class instance.
- The <u>class designer</u> is free to place information in the class that is important to the designer, but not the user, of the class.
- The class designer hides details of the implementation so that the program was easier to read and write

- modularity, make an object so that it can be reused in other contexts

- providing an interface (the methods) that are the approved way to deal with the class *Hides details* of the implementation so that the program was easier to read and write

- Provides *modularity* that makes an object so that it can be reused in other contexts

Special class methods

What are magic methods? They're everything in object-oriented Python. They're special methods that you can define to add "magic" to your classes. They're always surrounded by double underscores (e.g.__init__ or __lt__).

Enhances classes with features such as slices, item, calling capability, mathematical operations etc.

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to ***operator overloading***, allowing classes to define their own behavior with respect to language operators.

Class Special Methods

- __init__
- __call__
- __item__
- __add__
- __slice__
- __repr__
- __str__
- __hash__
- __len__
- __getattr__
- __setattr__
- __delattr__
- __delete__
- __getitem__
- __setitem__
- __delitem__
- __iter__
- __reversed__
- __contains__
- __getslice__
- __delslice__
- __add__
- __sub__
- __mod__
- __divmod__
- __pow__
- __and__
- __xor__
- __or__
- __neg__
- __pos__
- __abs__
- __int__
- __float__
- __lt__
- __le__
- __eq__
- __ne__
- __gt__
- __ge__
- many more

Math-like Operators		
Expression	**Method name**	**Description**
x + y	`__add__()`	Addition
x − y	`__sub__()`	Subtraction
x * y	`__mul__()`	Multiplication
x / y	`__div__()`	Division
x == y	`__eq__()`	Equality
x > y	`__gt__()`	Greater than
x >= y	`__ge__()`	Greater than or equal
x < y	`__lt__()`	Less than
x <= y	`__le__()`	Less than or equal
x != y	`__ne__()`	Not equal
Sequence Operators		
len(x)	`__len__()`	Length of the sequence
x in y	`__contains__()`	Does the sequence y contain x?
x[key]	`__getitem__()`	Access element *key* of sequence x
x[key]=y	`__setitem__()`	Set element *key* of sequence x to value y
General Class Operations		
x=myClass()	`__init__()`	Constructor
print (x), str(x)	`__str__()`	Convert to a readable string
	`__repr__()`	Print a Representation of x
	`__del__()`	Finalizer, called when x is garbage collected

TABLE 12.1 Python Special Method Names

Operator overloading

- the plus operator is ***overloaded***

- that is, the operator can do/mean different things (have multiple/overloaded meanings) depending on the types involved

- if python does not recognize the operation and that combination of types, you get an error

What does `var1+var2` do?

- with two strings, we get concatenation
- with two integers, we get addition
- with an integer and a string we get:

```
Traceback (most recent call last):
 File "<pyshell#9>", line 1, in <module>
   1+'a'
TypeError: unsupported operand type(s) for
  +: 'int' and 'str'
```

Python overload ops

- Python provides a set of operators that can be overloaded. You can't overload all the operators, but you can for many

- Like all the special class operations, they use the two underlines before and after.

- They come in three general classes:
  - numeric type operations (+,-,<,>,print etc.)
  - container operations ([ ], iterate,len, etc.)
  - general operations (printing, construction)

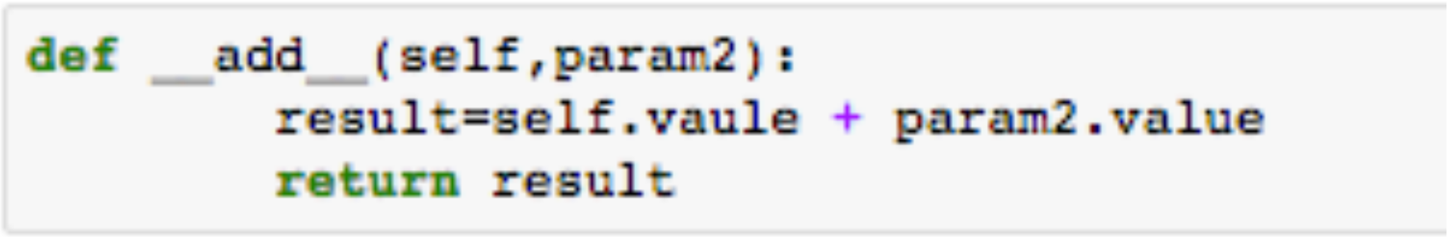
how does v1+v2 map to __add__

`v1 + v2`

is turned, by Python, into

`v1.__add__(v2)`

- These are ***exactly equivalent expressions***. It means that the first variable calls the __add__ method with the second variable passed as an argument



```
def __add__(self,param2):
        result=self.vaule + param2.value
        return result
```

v1 is bound to `self`, v2 bound to `param2`

Example:
-creation of a specialized vector class that supports addition, subtraction, etc.

- vector should be able to hold any type that supports addition, subtraction, etc.

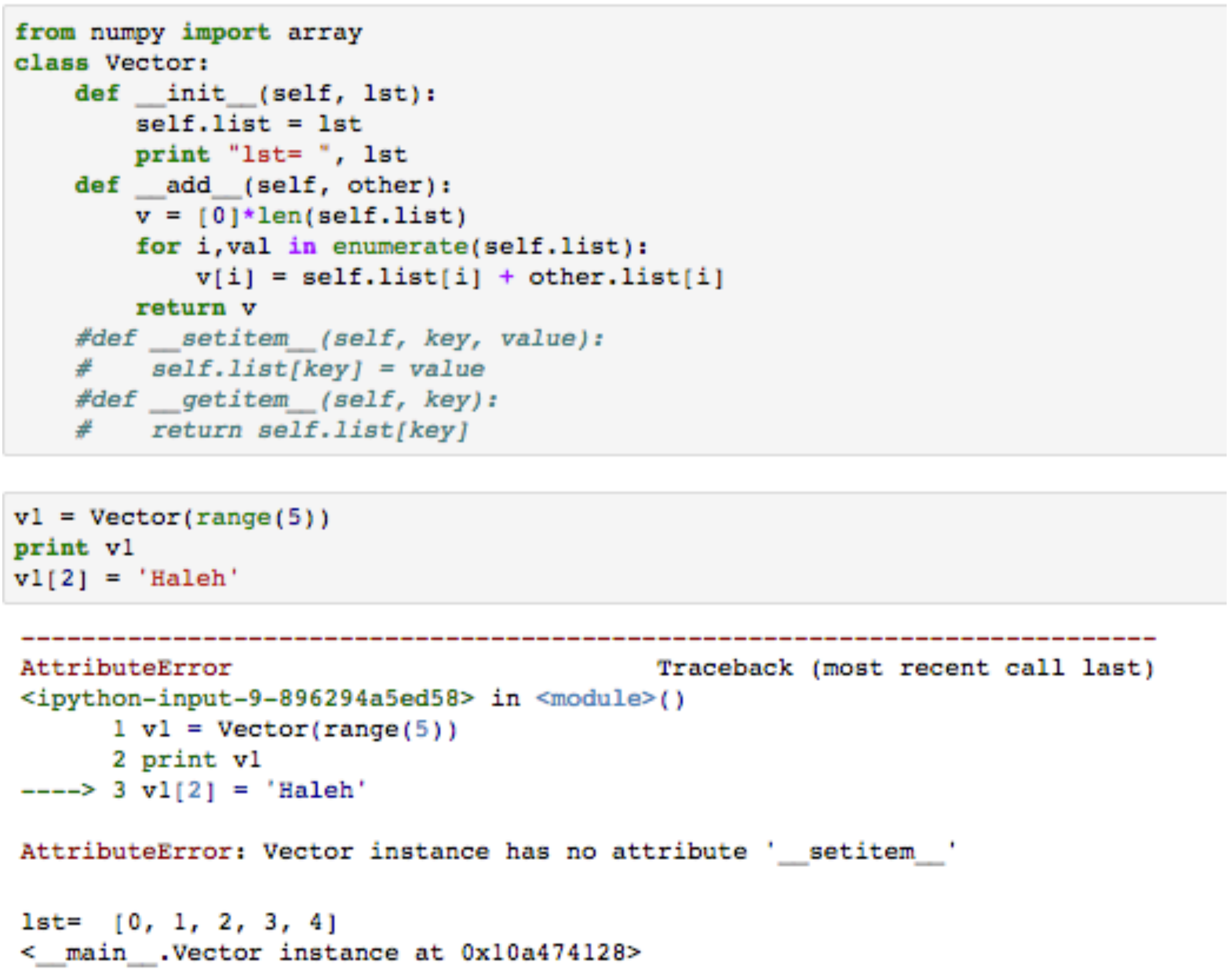
Additions of lists
```
a = [1,2,'string']
b = [1,3,'cat']
```

Output
```
[1, 2, 'string', 1, 3, 'cat']
```

Desired output
```
[2, 5, 'stringcat']
```

Using __setitem__ and __getitem__ for this example.

object.__setitem__(self, key, value)
Called to implement assignment to self[key]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper key values as for the __getitem__() method.



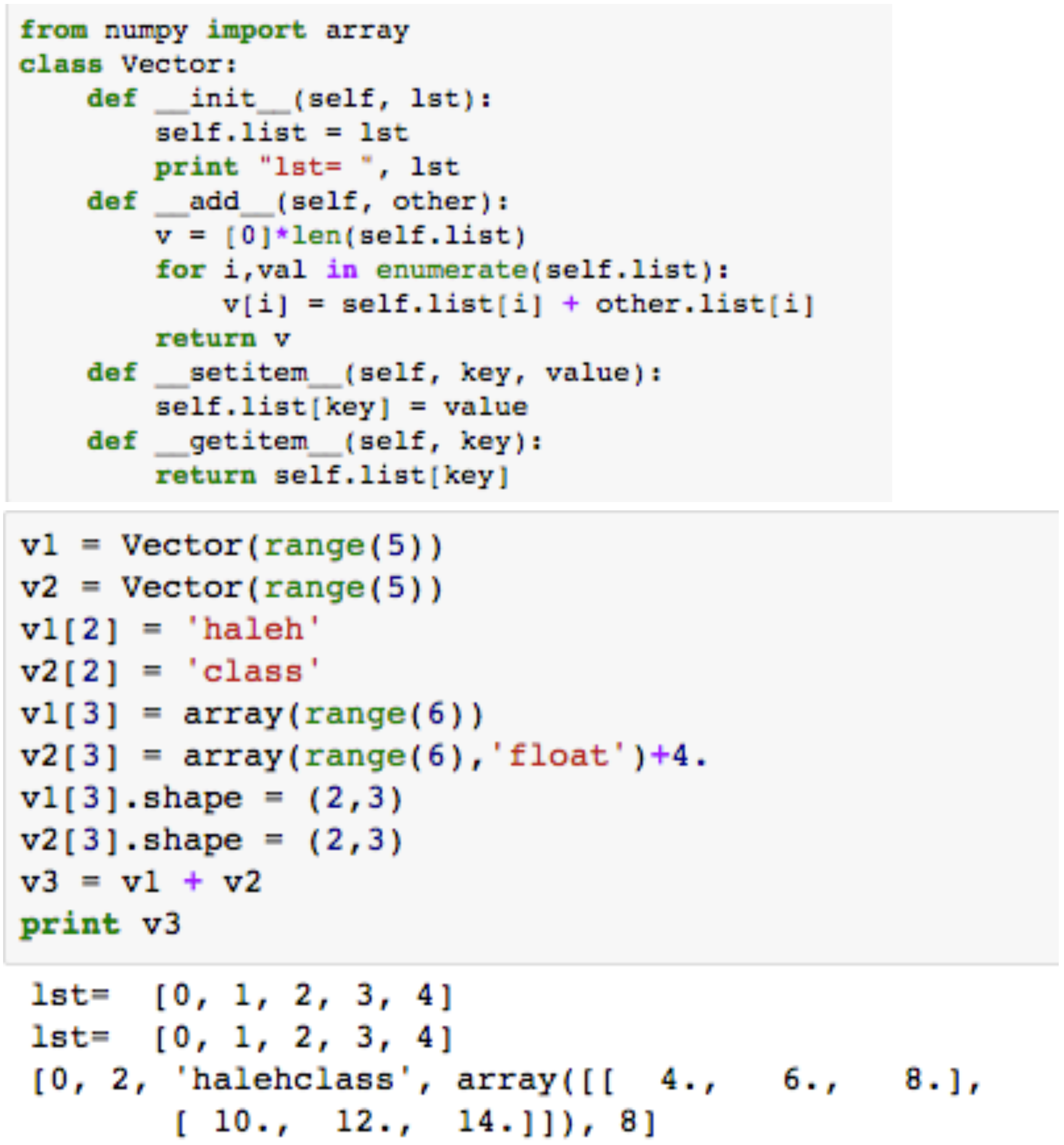
```
from numpy import array
class Vector:
    def __init__(self, lst):
        self.list = lst
        print "lst= ", lst
    def __add__(self, other):
        v = [0]*len(self.list)
        for i,val in enumerate(self.list):
            v[i] = self.list[i] + other.list[i]
        return v
    #def __setitem__(self, key, value):
    #    self.list[key] = value
    #def __getitem__(self, key):
    #    return self.list[key]
```

```
vl = Vector(range(5))
print vl
vl[2] = 'Haleh'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-896294a5ed58> in <module>()
      1 vl = Vector(range(5))
      2 print vl
----> 3 vl[2] = 'Haleh'

AttributeError: Vector instance has no attribute '__setitem__'


lst=  [0, 1, 2, 3, 4]
<__main__.Vector instance at 0x10a474128>
```



```
from numpy import array
class Vector:
    def __init__(self, lst):
        self.list = lst
        print "lst= ", lst
    def __add__(self, other):
        v = [0]*len(self.list)
        for i,val in enumerate(self.list):
            v[i] = self.list[i] + other.list[i]
        return v
    def __setitem__(self, key, value):
        self.list[key] = value
    def __getitem__(self, key):
        return self.list[key]
```

```
v1 = Vector(range(5))
v2 = Vector(range(5))
v1[2] = 'haleh'
v2[2] = 'class'
v1[3] = array(range(6))
v2[3] = array(range(6),'float')+4.
v1[3].shape = (2,3)
v2[3].shape = (2,3)
v3 = v1 + v2
print v3
```

```
lst=  [0, 1, 2, 3, 4]
lst=  [0, 1, 2, 3, 4]
[0, 2, 'halehclass', array([[  4.,   6.,   8.],
       [ 10.,  12.,  14.]]), 8]
```

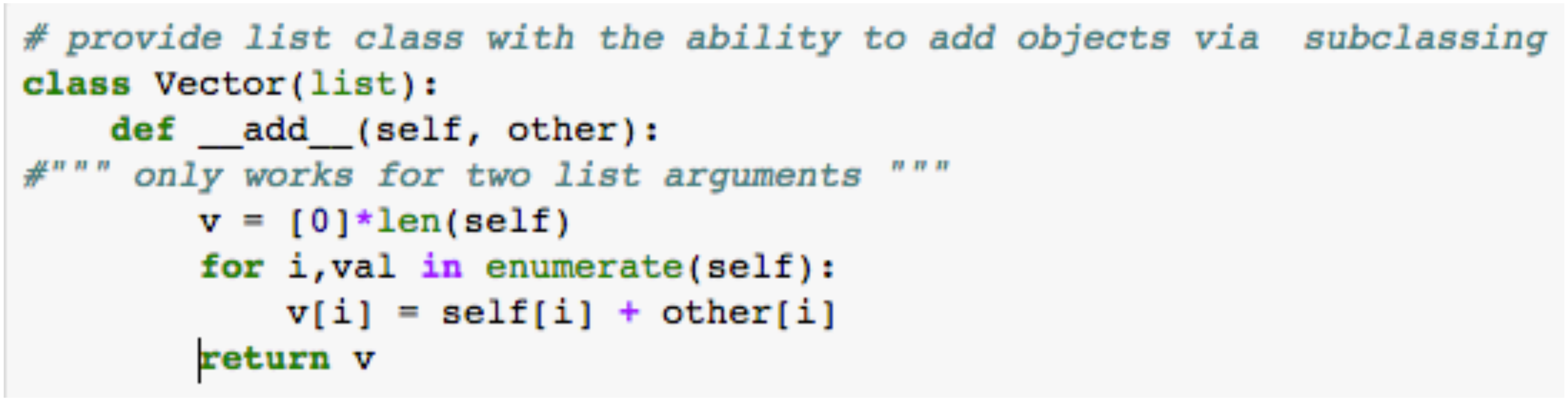
Better approaches

- Instead of creating a new class, take advantage of existing classes

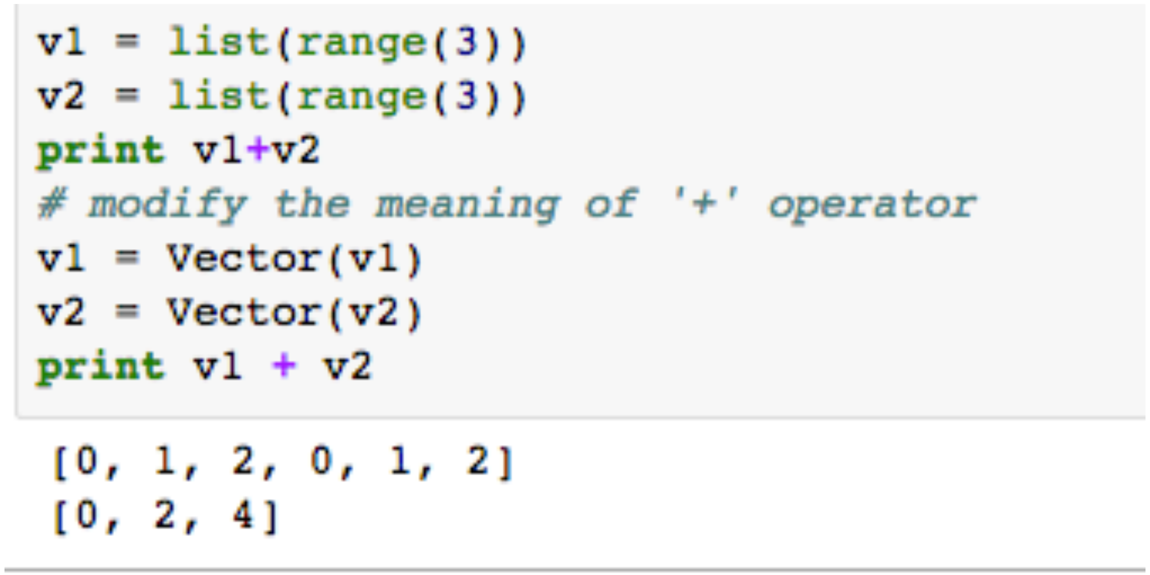
- Two possibilities
  - subclass the list class
  - extend list

    not really possible because list is built-in

Subclass List class



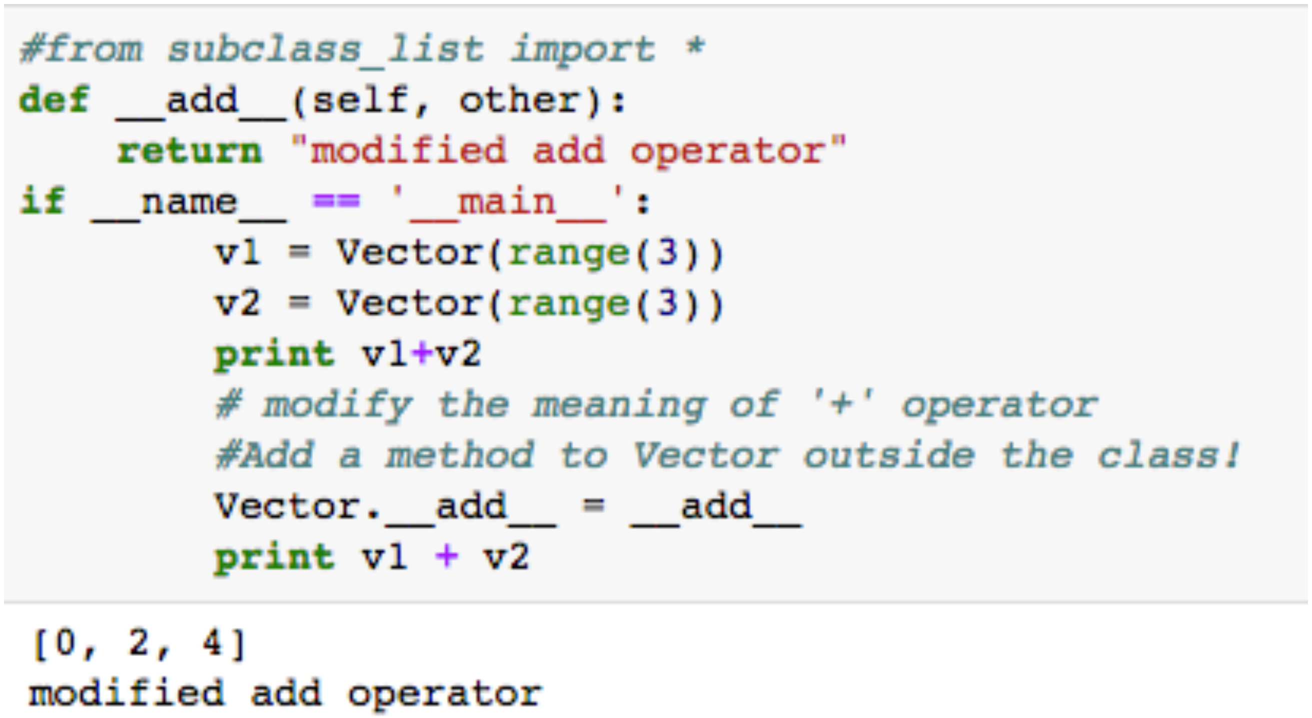
```
# provide list class with the ability to add objects via  subclassing
class Vector(list):
    def __add__(self, other):
#""" only works for two list arguments """
        v = [0]*len(self)
        for i,val in enumerate(self):
            v[i] = self[i] + other[i]
        return v
```



```
v1 = list(range(3))
v2 = list(range(3))
print v1+v2
# modify the meaning of '+' operator
v1 = Vector(v1)
v2 = Vector(v2)
print v1 + v2
```

```
[0, 1, 2, 0, 1, 2]
[0, 2, 4]
```

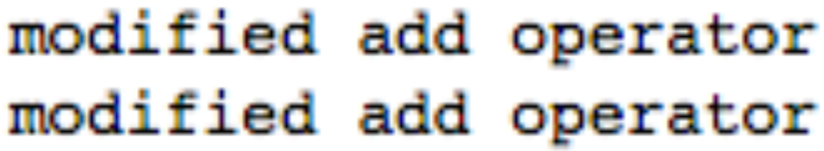
Extend a user class



```
#from subclass_list import *
def __add__(self, other):
    return "modified add operator"
if __name__ == '__main__':
        v1 = Vector(range(3))
        v2 = Vector(range(3))
        print v1+v2
        # modify the meaning of '+' operator
        #Add a method to Vector outside the class!
        Vector.__add__ = __add__
        print v1 + v2
```

```
[0, 2, 4]
modified add operator
```

If you run it again:

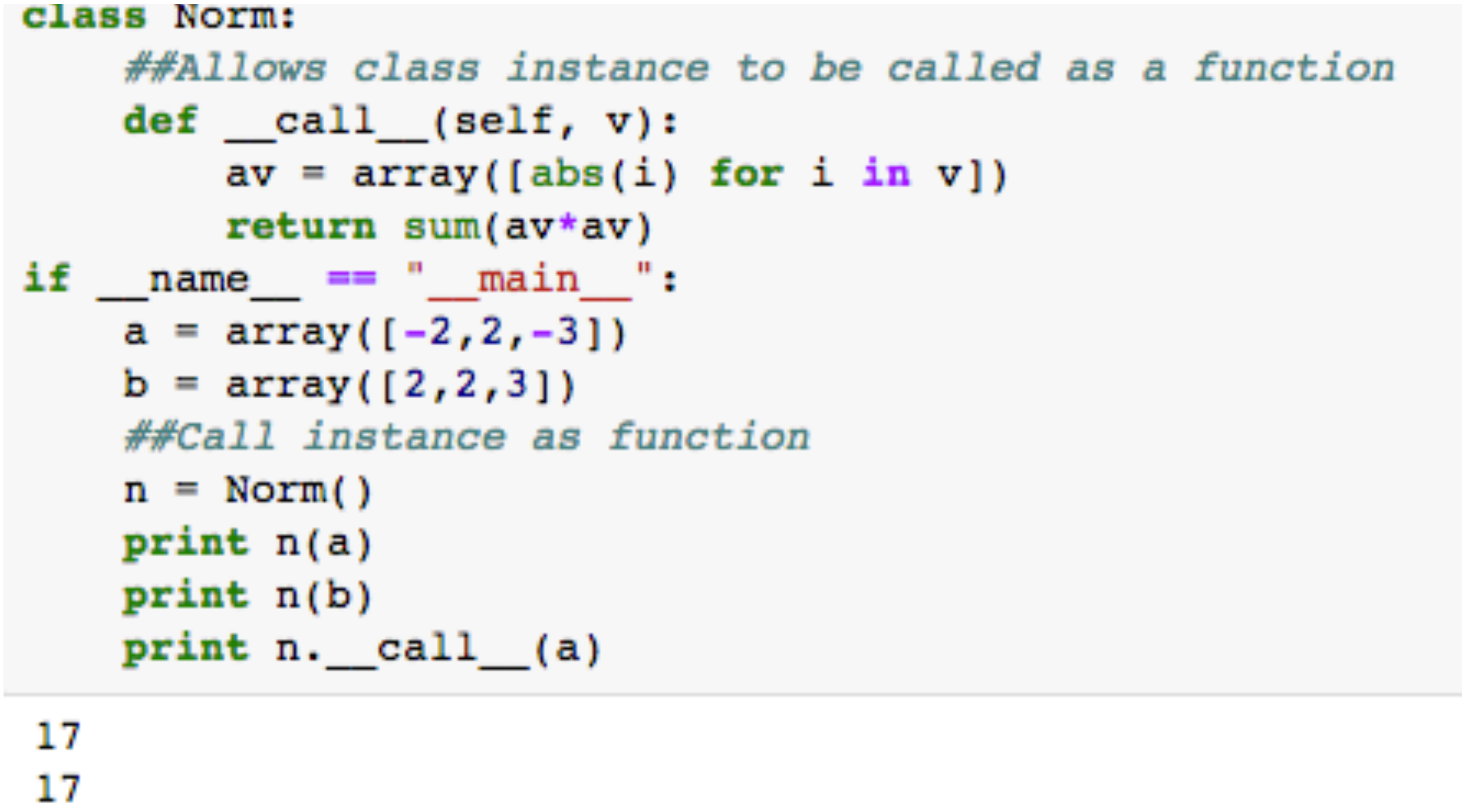


```
modified add operator
modified add operator
```

Callable objects

- Inside a class, define a method __call__
- The instance of a class can now be used as

a function



```
class Norm:
    ##Allows class instance to be called as a function
    def __call__(self, v):
        av = array([abs(i) for i in v])
        return sum(av*av)
if __name__ == "__main__":
    a = array([-2,2,-3])
    b = array([2,2,3])
    ##Call instance as function
    n = Norm()
    print n(a)
    print n(b)
    print n.__call__(a)
```

```
17
17
```

class namespaces are dicts

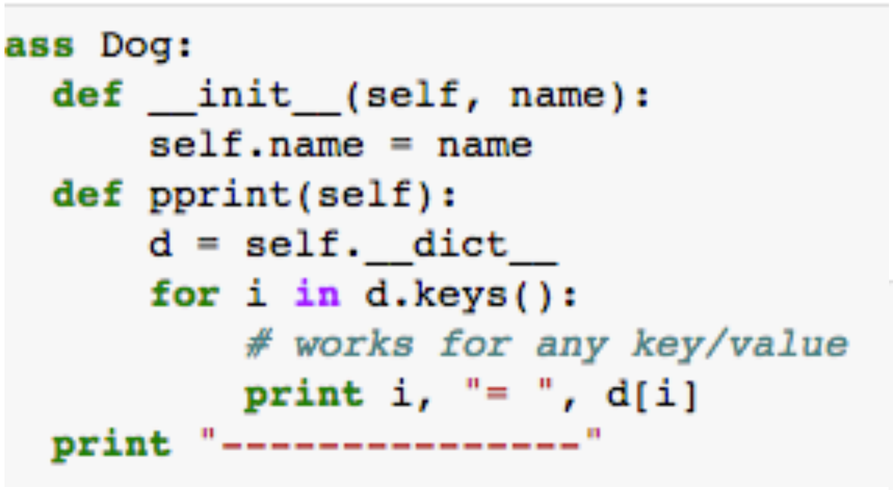
- the namespaces in every object and module is indeed a namespace
- that dictionary is bound to the special variable `__dict__`
- it lists all the local attributes (variables, functions) in the object
- Almost everything in Python works through dictionaries
- Class attributes (variables) are stored in a dictionary associated with the class
- • The __dict__ dictionary is not shared between Attributes

- Example:

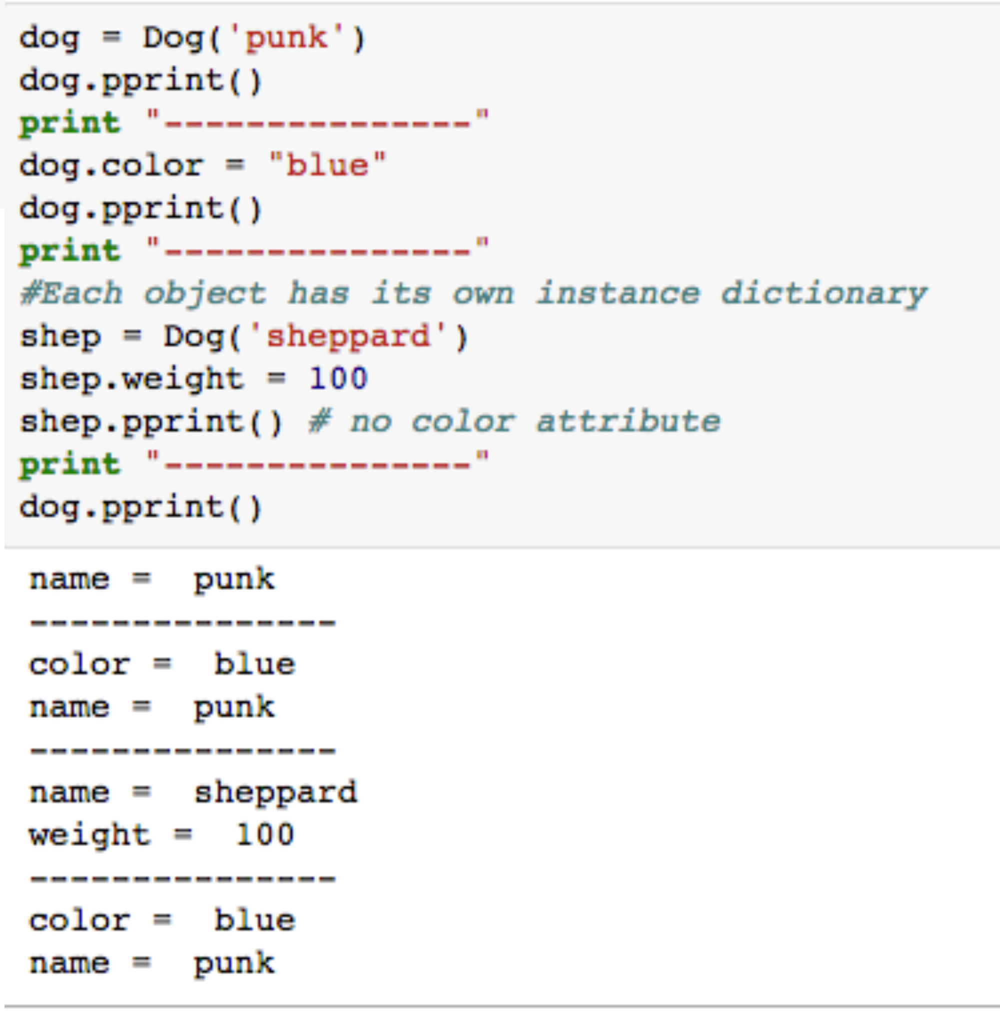
• Dog dog

dog.weight is equivalent to

dog.__dict__[‘weight’]



```
ass Dog:
  def __init__(self, name):
      self.name = name
  def pprint(self):
      d = self.__dict__
      for i in d.keys():
          # works for any key/value
          print i, "= ", d[i]
  print "---------------"
```

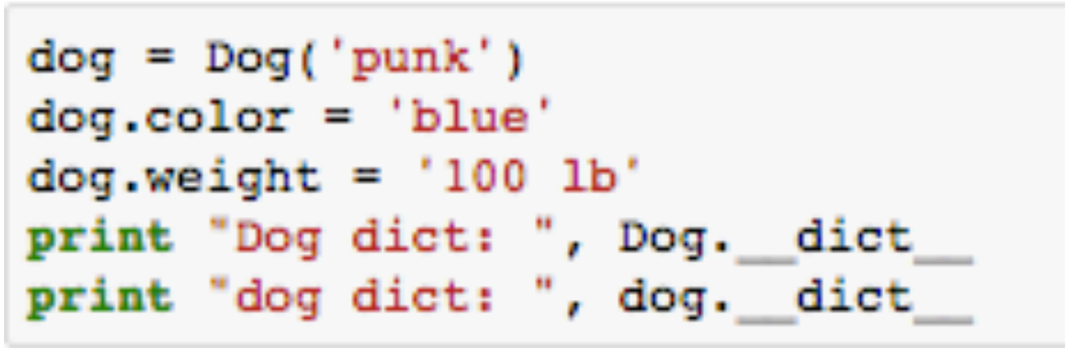


```
dog = Dog('punk')
dog.pprint()
print "---------------"
dog.color = "blue"
dog.pprint()
print "---------------"
#Each object has its own instance dictionary
shep = Dog('sheppard')
shep.weight = 100
shep.pprint() # no color attribute
print "---------------"
dog.pprint()
```

```
name =  punk
---------------
color =  blue
name =  punk
---------------
name =  sheppard
weight =  100
---------------
color =  blue
name =  punk
```

Dictionaries

Class and Class instances have separate dictionaries



```python
dog = Dog('punk')
dog.color = 'blue'
dog.weight = '100 lb'
print "Dog dict: ", Dog.__dict__
print "dog dict: ", dog.__dict__
```

Dog dict: {'__module__': '__main__', 'pprint': <function pprint at 0x10a472ed8>, '__doc__': None, '__init__': <function __init__ at 0x10a472aa0>}

dog dict: {'color': 'blue', 'name': 'punk', 'weight': '100 lb'}

Inheritance

- Inheritance allows the class designer to <u>utilize the design of an existing class to create a new class</u>.

- That is, we can create a new class that specializes an existing class by utilizing the existing class's attributes, specializing only those attributes that distinguish the new class.

- In this way, classes can share common elements and change only those attributes that distinguish the new class.

Class-Instance relations

- Remember the relationship between a class and its instances
  - a class can have many instances, each made initially from the constructor of the class
  - the methods that an instance can call are initially shared by all instances of a class
  - When referencing a value in an attribute, Python first looks in the instance for the attribute, and, if not found there, it then looks in the class the instance was derived from. In this way, attributes stored in the class are available to every instance derived from the class.

Class-Class relations

- Classes can also have a separate relationship with other classes
- the relationships forms a hierarchy
  - ***hierarchy***: A body of persons or things ranked in grades, orders or classes, one above another

Classes related by a hierarchy

- when we create a class, which is itself another object, we can state how it is related to other classes
- the relationship we can indicate is the class that is 'above' it in the hierarchy
- Every class maintains at least one parent class.

class statement

name of the class above
this class in the hierarchy

```
class MyClass (SuperClass):
    pass
```

- The top class in Python is called `object`.
- it is predefined by Python, always exists
- use `object` when you have no superclass

```
class MyClass (object):
    pass


class Child1Class (MyClass):
    pass


class Child2Class (MyClass):
    pass
```

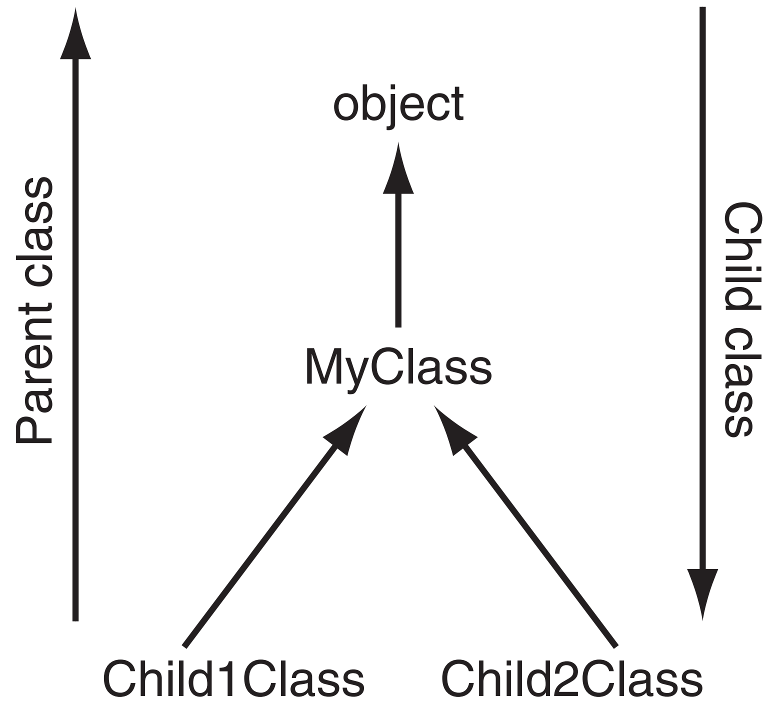


FIGURE 12.1 A simple class hierarchy.

- The is-a relationship is one way, much as the instance-of relationship.
- The instance remember who its class is, but the class does not track its instance.
- A class remember who its parent class is, but the parent class does not track its child classes.

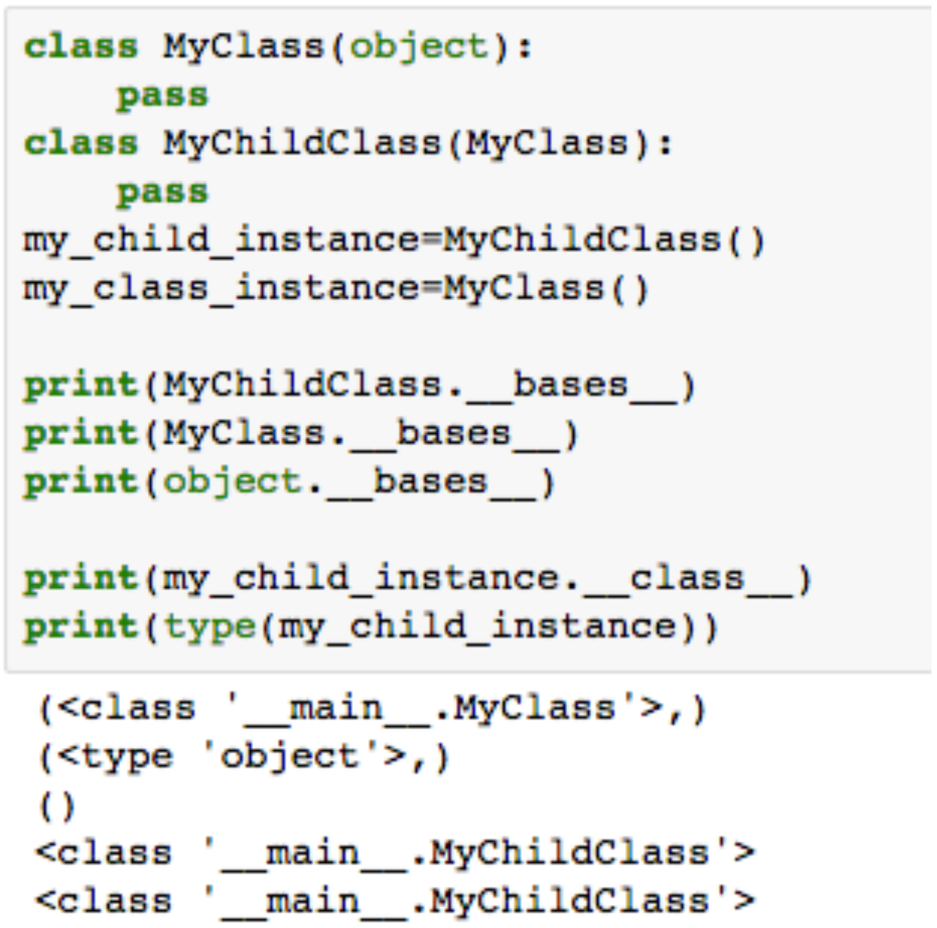
Subclass Example

- Polygon
  - object with n unequal sides
- Regular polygon: special Polygon
  - object with n sides, all equal length
- Triangle: special polygon (3 sides)
- Square: special regular polygon

Class Hierarchy

- class Polygon:
- class regPolygon(Polygon):
- class Triangle(Polygon):
- class Square(regPolygon):
- class equiTriangle(Triangle) or

class equiTriangle(regPolygon)



```python
class MyClass(object):
    pass
class MyChildClass(MyClass):
    pass
my_child_instance=MyChildClass()
my_class_instance=MyClass()

print(MyChildClass.__bases__)
print(MyClass.__bases__)
print(object.__bases__)

print(my_child_instance.__class__)
print(type(my_child_instance))
```

```
(<class '__main__.MyClass'>,)
(<type 'object'>,)
()
<class '__main__.MyChildClass'>
<class '__main__.MyChildClass'>
```

- Each Python class indicates specifically in its class definition who its parent is.
- This relationship is recorded in the __bases__ attribute of each class.

is-a, super and sub class

- the class hierarchy imposes an ***is-a*** relationship between classes
  - MyChildClass ***is-a*** (or is a kind of) MyClass
  - MyClass ***is-a*** (or is a kind of) object
  - object has as a subclass MyClass
  - MyChildClass has as a superclass MyClass
- An example: if you create a Car class, then a Ford class *is a* Car, but a more particular kind of Car. Subsequently, a Mustang class *is a* Ford class, but a more particular kind of Ford.

- the hope of such an arrangement is the saving/re-use of code. If a new class is created as part of an existing class hierarchy, then the new class can reuse existing code from the hierarchy, specializing only those aspects or attributes that are unique to the new class.

- superclass code contains general code that is applicable to many subclasses. By sharing code from the class hierarchy, the coding of classes can be somewhat standardized.

- subclass uses superclass code (via sharing) but specializes code for itself when necessary

Scope for objects, the full story

1. Look in the object for the attribute
2. If not in the object, look to the object's class for the attribute (up the instance-of creation)
3. If not in the object's class, look up the hierarchy of that class for the attribute (up the is-a relation)
4. If you hit object, then the attribute does not exist

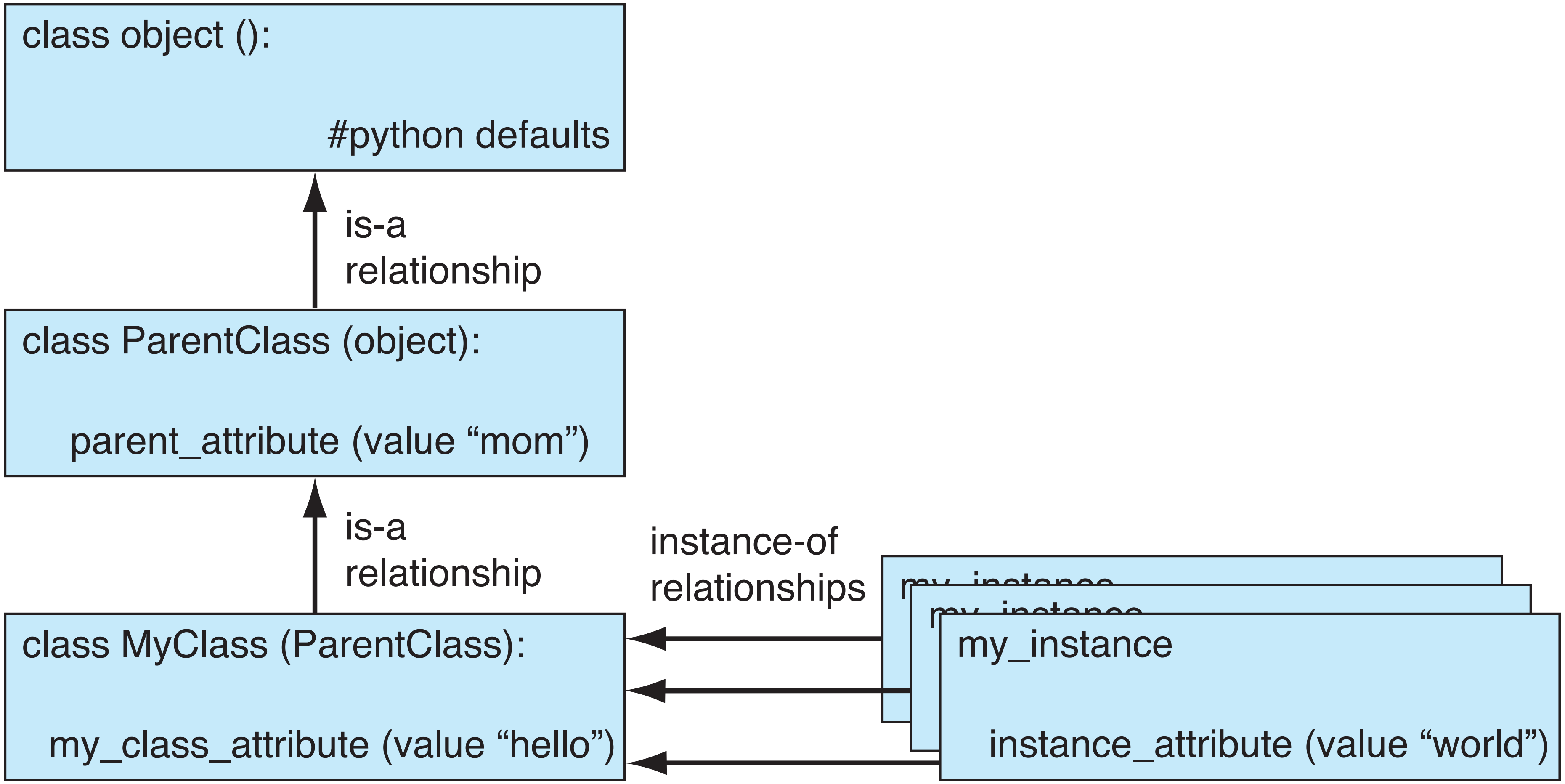

class object ():
#python defaults
is-a relationship
class ParentClass (object):
parent_attribute (value "mom")
is-a relationship
instance-of relationships
class MyClass (ParentClass):
my_class_attribute (value "hello")
my_instance
my_instance
my_instance
instance_attribute (value "world")

FIGURE 12.2 The players in the "find the attribute" game.

Inheritance is powerful but also can be complicated

- many powerful aspects of OOP are revealed through uses of inheritance

- However, some of that is a bit detailed and hard to work with. Definitely worth checking out but a bit beyond the class

builtins are objects too

- One nice way, easy way, to use inheritance is to note that all the builtin types are objects also
- thus you can inherit the properties of builtin types then modify how they get used in your subclass
- you can also use any of the types you pull in as modules

specializing a method

- One technical detail. Normal method calls are called ***bound methods***. Bound methods have an instance in front of the method call and automatically pass self

```
my_inst = MyClass()
my_inst.method(arg1,arg2)
```

- `my_inst` is an instance, so the method is bound

unbound methods

it is also possible to call a method without Python binding `self`. In that case, the user has to do it.

- unbound methods are called as part of the class but `self` passed by the user

```
my_inst = MyClass()
MyClass.method(my_inst, arg2, arg3)
```

`self` is passed **explicitly** (`my_inst` here)!

- Consider an example. We want to specialize a new class as a subclass of list.

```
class MyClass(list):
```

- easy enough, but we want to make sure that we get our new class instances initialized the way they are supposed to, by calling `__init__` of the super class

Why call the super class init?

If we don't explicitly say so, our class may inherit stuff from the super class, but we must make sure we call it in the proper context. For example, our `__init__` would be:

```
def __init__(self):
   list.__init__(self)
# do anything else special to MyClass
```

explicit calls to the super

- we explicitly call the super class constructor using an unbound method
- then, after it completes we can do anything special for our new class
- We **specialize** the new class but inherit most of the work from the super.

Gives us a way to organize code

- ***specialization***. A subclass can inherit code from its superclass, but modify anything that is particular to that subclass

- ***over-ride***. change a behavior to be specific to a subclass

- ***reuse-code***. Use code from other classes (without rewriting) to get behavior in our class.

Class for Scientific Computing

Inline lambda functions

- Python offers anonymous inline functions known as lambda function.
- The construction is “lambda <args>: <expression>”

- It is equivalent to a function with <args> as arguments and <expression> as return value:

```
def somefunc(<args>):
    return <expression>
```

- For example, “lambda x, y, z: 3*x+2*y-z” is a short cut for

```
def somefunc(x, y, z)
    return 3*x + 2*y – z
```

Inline lambda functions

- Lambda function can be used in place where we expect variables.
- Say we have a function taking another function as argument:

```
def fill(a,f)
    n = len(a); dx = 1.0/(n-1)
    for i in range(n):
        x = i * dx
     a[i] = f(x)
```

- A lambda function can be used for the f argument:

```
fill(a, lambda x: <expression>)
```

OOP for Integration

Consider other quadrature rules:

- Simpson's rule

$$\int_{-1}^{1} f(x)dx \approx \frac{1}{3} f(-1) + \frac{4}{3} f(0) + \frac{1}{3} f(1)$$

- Two-point Gauss-Legendre rule

$$\int_{-1}^{1} f(x)dx \approx f(-\frac{1}{\sqrt{3}}) + f(\frac{1}{\sqrt{3}})$$

Write a Python class Trapezoidal to evaluate numerical integration using the trapezoidal rule.

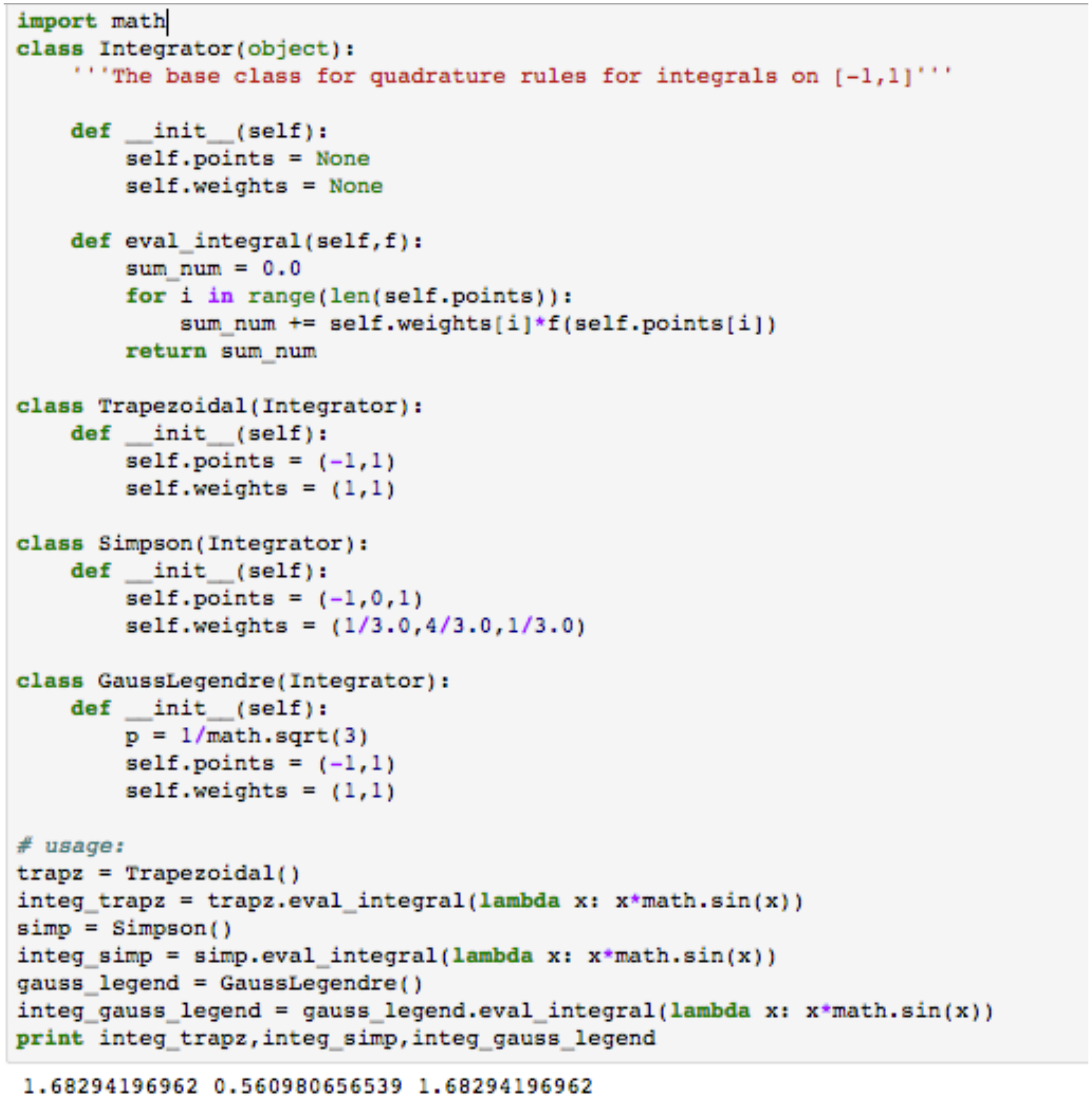
$$\int_{-1}^{1} f(x)dx \approx f(-1) + f(1)$$

It is one of the quadrature rules that can be expressed in the general form:

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

Use the class to compute

$$\int_{-1}^{1} x^3 dx$$



```python
import math
class Integrator(object):
    '''The base class for quadrature rules for integrals on [-1,1]'''

    def __init__(self):
        self.points = None
        self.weights = None

    def eval_integral(self,f):
        sum_num = 0.0
        for i in range(len(self.points)):
            sum_num += self.weights[i]*f(self.points[i])
        return sum_num

class Trapezoidal(Integrator):
    def __init__(self):
        self.points = (-1,1)
        self.weights = (1,1)

class Simpson(Integrator):
    def __init__(self):
        self.points = (-1,0,1)
        self.weights = (1/3.0,4/3.0,1/3.0)

class GaussLegendre(Integrator):
    def __init__(self):
        p = 1/math.sqrt(3)
        self.points = (-1,1)
        self.weights = (1,1)

# usage:
trapz = Trapezoidal()
integ_trapz = trapz.eval_integral(lambda x: x*math.sin(x))
simp = Simpson()
integ_simp = simp.eval_integral(lambda x: x*math.sin(x))
gauss_legend = GaussLegendre()
integ_gauss_legend = gauss_legend.eval_integral(lambda x: x*math.sin(x))
print integ_trapz,integ_simp,integ_gauss_legend
```

```
1.68294196962 0.560980656539 1.68294196962
```