PYTHON – CLASS

CLASS

- At a fundamental level, classes are really just packages of functions and other attributes. very much like modules.
- However, the automatic attribute search of classes (aka inheritance) is far more powerful than modules or functions.
- Classes provide a natural structure for code that localizes logic and names and aids debugging.

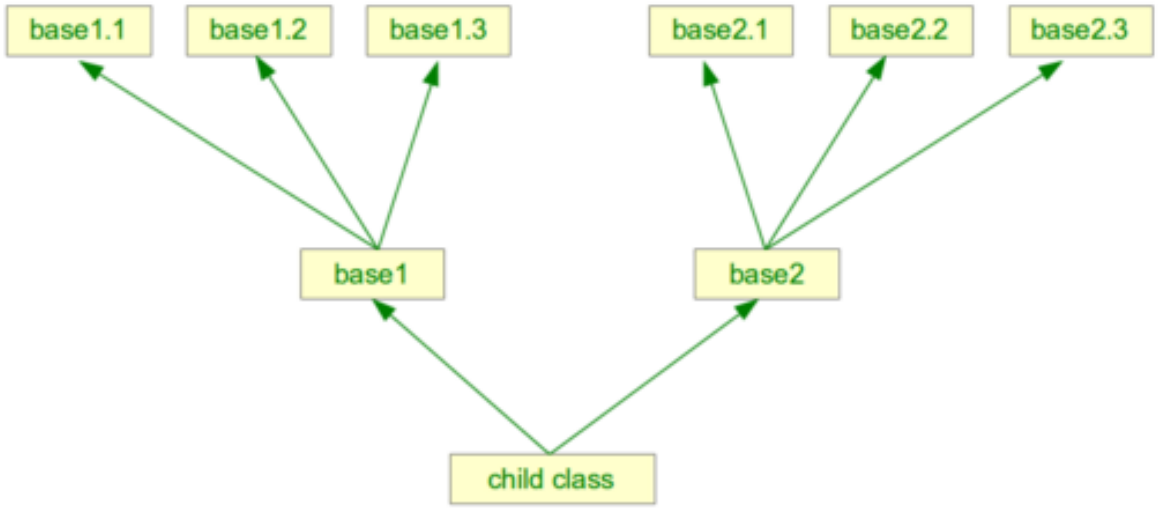
Python Classes

- Classes are designed to create and manage new objects, and to support inheritance.
  - FYI: Python OOP is entirely optional, you can get a lot done with just functions and modules.
- When used well, classes will cut development time radically.
- Classes are used in major Python tools (eg. sockets & tkinter) so a working knowledge is very useful.

Why Classes?

1. We can “instantiate” multiple instances, one for each entity.
2. Rich inheritance of legacy and customized attributes.
3. Operator overloading means that we can customize operators to work as we please on objects.

Python allows multiple class inheritance

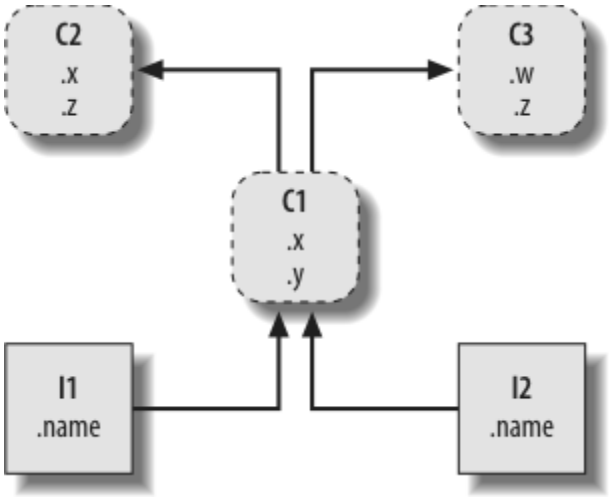


Multiple object instances

- Classes are object generators.
- Calling a class generates a new object of the class type with a distinct namespace.
- Each object generated from a class has access to the class's attributes (data, methods) and gets a namespace of its own for data that varies per object.
- Data is stored with the object instance. Methods are found by relating back to the class.

Customization via inheritance

- Classes support the OO notion of inheritance; we can extend a class by redefining its attributes in a subclass.
- More generally: classes can build up namespace hierarchies, which define names to be used by objects created from classes in the hierarchy.



- A class tree, with two instances, a defining class and two superclasses above that.
- All of these objects are namespaces (packages of attributes)
- The inheritance search goes from bottom to top looking for the lowest occurrence of an attribute name.

`object.attribute`

- Attribute fetches are simply tree searches.
- Objects lower in a tree inherit attributes attached to objects higher in that tree.
- The search proceeds from the bottom up and stops at the first match.
- The object has access to the union of all the attributes defined all the way up the tree.

Instance Factories

- Classes serve as instance factories or “cloners”.
- Class attributes provide behaviour—data and functions—that is inherited by all the instances generated from them.
- Instances
  - Represent the concrete items in a program’s domain.
  - They are objects.
  - Their attributes record data unique to each specific object instance
- An instance inherits attributes from its class, and a class inherits attributes from all classes above it

Operator overloading

- By redefining methods and operators, class objects can respond to evolving needs.

- Python provides hooks that classes can use to intercept and implement any built-in type operation (aka overloading)

Creating a class

So what happens when you instantiate a class into an object?

X = myClass()

STEPS

1. You create an object of the type and you store this in memory along with associated data
2. You relate the object to its class via its type name
3. You affiliate class methods, data and operator overrides with the object via “inheritance”

Q: if you define a class with methods and instantiate an object of that class are the methods stored with the instantiated object? What about data? Why?

Creating a class

```
class myClass(baseClass):
    def setName(self,name):
        self.name=name

Me = myClass()
Me.setName('alice')
```

Class Object vs. Instance Object

- Each class statement generates a new **class object**.
- Each time a class is called, it generates a new **instance object**.
- Instances are automatically linked to the classes from which they are created.
- Classes are linked to their superclasses by listing them in parentheses in a class header line; the left-to-right order there gives the order in the tree.

Inheritance

```
class C1():
 x=3

class C2():
 pass

class C3(C1,C2):
 z=4
```

Exercise

1. Create a simple class call C1 which does nothing but a pass.
2. Instantiate an object of that class and then determine its type.
3. Now prove that your object instantiation has its own name space. (hint, where have you seen namespaces before?)
4. Write a second class C2 that inherits from the first and defines a variable c2v='c2c2c2'
5. Now instantiate an object of C2 and access (print) c2v via the object namespace.
6. Create a class C3 that inherits from C2 and defines a function F that simple prints what is passed to it.
7. Create an object X of type C3 and use F to print the value c2v in X name space.

Classes vs. Modules

- Technically different but almost identical
- Importing mods lets us access their namespaces.
- But classes
  - have automatic search links to other (super) namespace objects
- Classes correspond to attributes, not entire files.
- Classes allow multiple instantiations..
  - And each instantiation has its own namespace for “personal” data
- Object namespace covers all inherited name space

Method Call

Example: The method calc_bw() can be called through either an instance

```
udp_port53.calc_bw()
```

or a class

```
traFFic.calc_bw(udp_port53)
```

Constructor

```
class C1(C2, C3):                    # Make and link class C1
  def setname(self, who):            # Method to name the iteration
    self.name = who
```

```
I1 = C1()                            # Create two instances of object C1
I2 = C1()
I1.setname('bob')                    # Sets object name to 'bob'
I2.setname('mel')
print(I1.name)
```

Constructor

```
class C1(C2, C3):
 def __init__(self, who):        # Constructor
      self.name = who


I1 = C1('bob')
I2 = C1('mel')
```

Constructor

```
class C1(C2, C3):
  def __init__(self, who):       # Constructor
       self.name = who


I1 = C1('bob')
```

- Remember: self is always passed into ALL method calls
- I1 is passed in to the self argument of __init__
- Any values listed in parentheses in the class call go to arguments numbered two and beyond.
- The effect here is to initialize instances with some data when they are made, without requiring extra method calls.

Operator Overloading

- The __init__ method is known as the constructor.
- It's the most commonly used representative of a larger class of methods called operator overloading methods.
- Such methods are inherited in class trees as usual and have double underscores (start and end) to make them distinct.
- Python runs them automatically when instances that support them appear in the corresponding operations

Try It

```
class c1():
     def __init__(self):
          return

x=c1()
```

Exercise

Create a class called `myTally` which uses the constructor.

myTally has a methods called `Add(x)` and `Sub(x)`. These will add or subtract x from total value (`Tally`) associated with each instance. Each time a value is Add(ed) or Sub(tracted) the Tally is printed.

```
t=myTally(10)
t.Add(20)
```

Create a subclass of `myTally` called `myTally2` that will create two more methods called `mult(y)` and `div(y)` that will multiply and divide the tally by y. It will also overload the sub(n) method by changing its operation to always subtract twice the value of n.

Class Statement

```
class FirstClass():
   def setdata(self, value):
      self.data = value
   def display(self):
      print(self.data)
```

Q: what is the difference?

```
 x=Firstclass
 y=Firstclass()
```

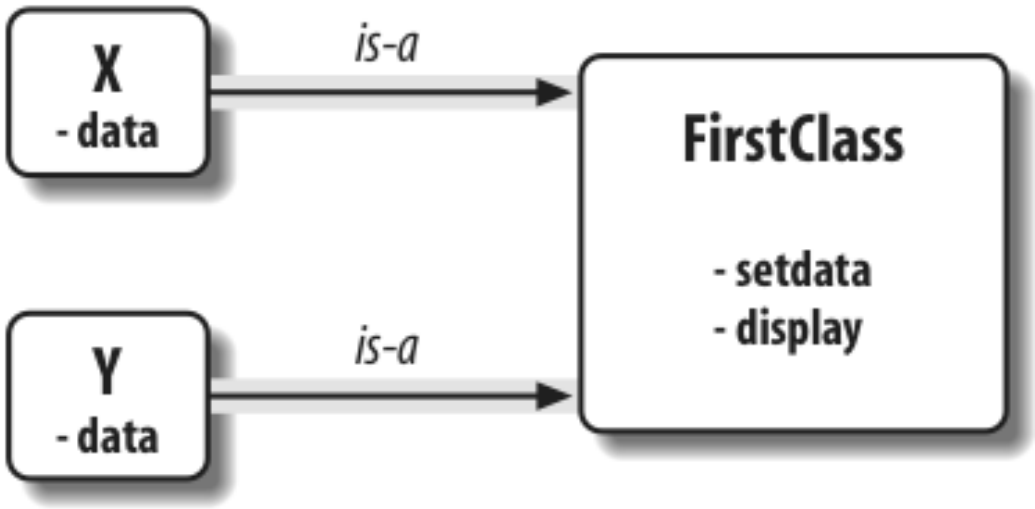
Class Statement

```
>>>
>>> x=Firstclass
>>> y=Firstclass()
>>>
>>> x
<class '__main__.Firstclass'>
>>> y
<__main__.Firstclass object at 0x02CB7E90>
>>>
```

OBJECTS

```
class FirstClass:
  def setdata(self, value):
        self.data = value
  def display(self):
        print(self.data)

x = FirstClass()
y = FirstClass()
```



- At this point, we have three objects in memory:
  - Two instance objects
  - A class object

Class Objects

When we run a class statement, we get a class object.

```
class FirstClass():
  def setdata(self, value):         # Define class methods
        self.data = value           # self is the instance
  def display(self):
        print(self.data)           # self.data: per instance
```

- Just like the function "def" statement, a Python "class" statement is an *executable statement.*
- When reached and run, it generates a new class object and assigns it to the name stated.
  - Distinguish this from "instantiating" an object of type class
- Like defs, class statements typically run when their files are imported.

Class/Object Attributes.

```
class FirstClass:
  def setdata(self, value):  # Define class method
        self.data = value    # self is the instance
```

- Technically, the class statement scope morphs into the attribute namespace of the class object
  - just like a module's global scope.

- After running a class statement, class/object attributes are accessed by name qualification:

```
object.name
class.name
```

```
>>> class MyClass():
        dat=109
        def __init__(self,x):
            self.data=x


>>> var=MyClass(6)
>>> var.data
6
>>> var.dat
109
>>> var.dat=77
>>> var.dat
77
>>>
>>>
>>> MyClass.x=777
>>> var.x
777
>>>
```

Exercise: generate the example yourself. Then create another instance of MyClass called var2 and see how var2.dat relates to var.dat and MyClass.dat.

Exercise

```
class Employee:
 def __init__(self,name):
     self.surname=input("enter surname: ")
     self.firstname=name
 def Name(self):
     print(self.firstname)
     print(self.surname)
```