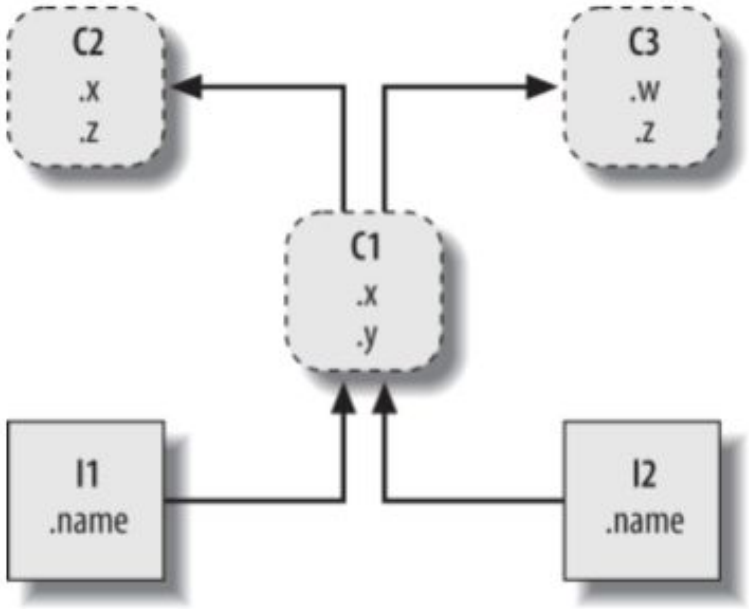
Classes are objects

Attribute Inheritance Search

The OOP story in Python boils down to this expression: ***object.attribute*** It means exactly the following: find the first occurrence of attribute by looking in object, then in all classes above it, from bottom to top and left to right.



For *I2.w* the path is *I2, C1, C2, C3.*

What are the paths for I1.x, I2.x, I1.y, I2.y, I1.z, I2.z, I2.name ?

Main concepts behind Python classes

- Classes and instances are almost identical—each type's main purpose is to serve as another kind of namespace—a package of variables.
- 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 automatically linked to their superclasses according to the way we list them in parentheses in a class header line; the left-to-right order there gives the order in the tree.

class C2: ... class C3: ... class C1(C2, C3): ... I1 = C1() I2 = C1()

Examples

```
class C1:
    def setname(self, who):
            self.name = who
I1 = C1()
I2 = C1()
I1.setname('bob')
I2.setname('sue')
```

```
class C1:
    def __init__(self, who):
            self.name = who
I1 = C1('bob')
I2 = C1('sue')
```

```
class Employee:
      def    computeSalary(self): ...
      def giveRaise(self): ...
      def promote(self): ...
     def retire(self): ...

class Engineer(Employee):
         def computeSalary(self): ...
```

```
bob = Employee()
sue = Employee()
tom = Engineer()

company = [bob, sue, tom]
for emp in company:
print(emp.computeSalary())
```

Class Objects

- The class statement creates a class object and assigns it a name. Just like the function def statement, the Python class statement is an executable statement. When reached and run, it generates a new class object and assigns it to the name in the class header. Also, like defs, class statements typically run when the files they are coded in are first imported.
- Each instance object inherits class attributes and gets its own namespace. Instance objects created from classes are new namespaces; they start out empty but inherit attributes that live in the class objects from which they were generated.
- Assignments to attributes of self in methods make per-instance attributes. Inside a class's method functions, the first argument (called self by convention) references the instance object being processed; assignments to attributes of self create or change data in the instance, not the class.
- Each *object.attribute* reference invokes a new, independent search .

The simplest: try it

```
class rec: pass
rec.name = 'Bob'
rec.age  = 40

 x = rec()
 y = rec()
```

```
>>>  x.name, y.name
>>> x.name = 'Sue'
>>> rec.name, x.name, y.name        # ('Bob', 'Sue', 'Bob')
```

```
>>> list(rec.__dict__.keys())
['age', '__module__', '__qualname__', '__weakref__', 'name', '__dict__', '__doc__']
>>> list(name for name in rec.__dict__ if not name.startswith('__'))     #  ['age', 'name']
>>> list(x.__dict__.keys())      #  ['name']
>>> list(y.__dict__.keys())      # []
>>> x.name, x.__dict__['name']   #('Sue', 'Sue')
>>> x.age          # 40
>>> x.__dict__['age']     #  KeyError
>>> x.__class__              #  <class '__main__.rec'>
>>> rec.__bases__        #  (<class 'object'>,)
```

Classes Versus Dictionaries: compare

```
>>> rec = ('Bob', 40.5, ['dev', 'mgr'])    # Tuple

>>> rec = {}
>>> rec['name'] = 'Bob'      #Dictionary
>>> rec['age']  = 40.5
>>> rec['jobs'] = ['dev', 'mgr']
```

```
>>> class rec: pass
>>> rec.name = 'Bob'          # Class
>>> rec.age  = 40.5
>>> rec.jobs = ['dev', 'mgr']

>>> class rec: pass
>>> pers1 = rec()
>>> pers1.name = 'Bob'
>>> pers1.jobs = ['dev', 'mgr']
>>> pers1.age  = 40.5
>>>
>>> pers2 = rec()
>>> pers2.name = 'Sue'
>>> pers2.jobs = ['dev', 'cto']
>>>
>>> pers1.name, pers2.name   #('Bob', 'Sue')
```

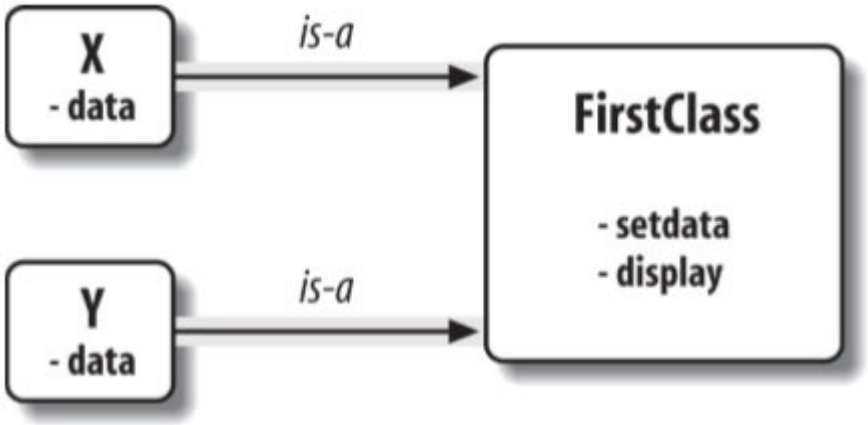
Classes Versus Dictionaries: compare

```
>>> class Person:
        def __init__(self, name, jobs, age=None):      # class = data + logic
            self.name = name
            self.jobs = jobs
            self.age  = age
        def info(self):
            return (self.name, self.jobs)
>>> rec1 = Person('Bob', ['dev', 'mgr'], 40.5)       # Construction calls
>>> rec2 = Person('Sue', ['dev', 'cto'])
>>> rec1.jobs, rec2.info()                          # Attributes + methods
                                                  #(['dev', 'mgr'], ('Sue', ['dev', 'cto']))

>>> rec = dict(name='Bob', age=40.5, jobs=['dev', 'mgr'])      # Dictionaries
>>> rec = {'name': 'Bob', 'age': 40.5, 'jobs': ['dev', 'mgr']}
>>> rec = Rec('Bob', 40.5, ['dev', 'mgr'])                    # Named tuples
```

Try it:

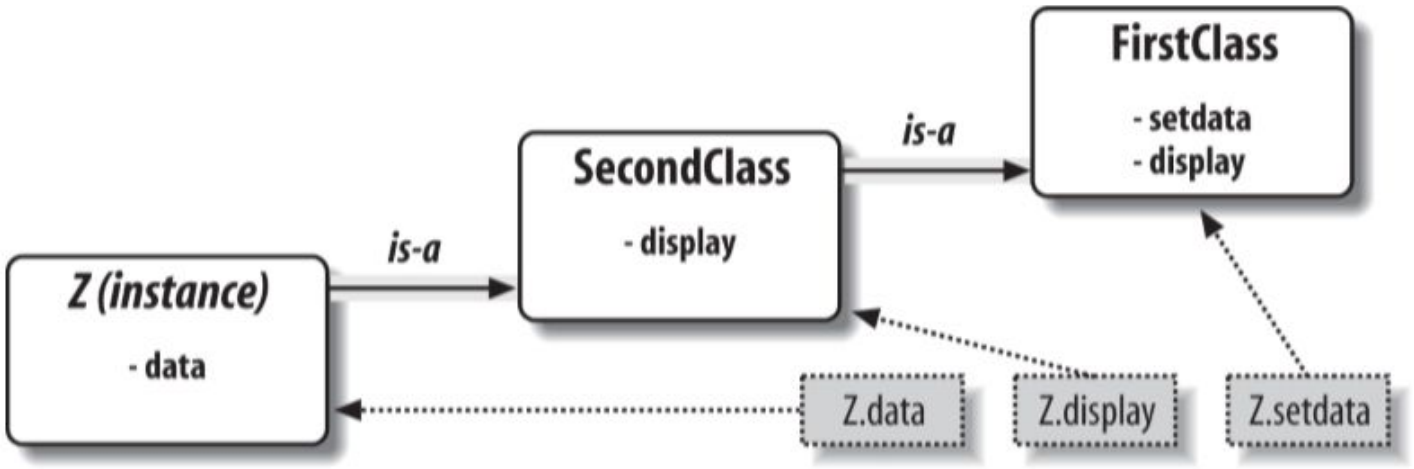
```
class FirstClass:
      def setdata(self, value):
            self.data = value
      def display(self):
            print(self.data)
```




```
x = FirstClass()
y = FirstClass()
x.data = "New value"
x.display()
x.setdata("King Arthur")    x.display()
y.setdata(3.14159)          x.display()
x.data = "New value"        x.display()
x.anothername = "spam"
```

Try it:

```
class SecondClass(FirstClass):
    def display(self):
        print('Current value = "%s"' % self.data)
z = SecondClass()
z.setdata(42)
z.display()
 x.display()
```



Calling superclass methods

```
class Super:
    def __init__(self, x):
            ...default code...

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)
            ...custom code...
I = Sub(1, 2)
```

Classes Are Attributes in Modules

```
from modulename import FirstClass       # Copy name into my scope class
SecondClass(FirstClass):
        def display(self): ...


import modulename                       # Access the whole module
class SecondClass(modulename.FirstClass):
        def display(self): ...
```

General form

```
class name(superclass,...):          # Assign to name
    attr = value                     # Shared class data
    def method(self,...):            # Methods
        self.attr = value            # Per-instance data
```

*A*ny sort of statement can be nested inside class body—**print**, assignments, **if**, **def**, and so on. All the statements inside the class statement run when the class statement itself runs (not when the class is later called to make an instance). In general any type of name assignment at the top level of a class statement creates a same-named attribute of the **resulting class object**. For example, assignments of simple nonfunction objects to class attributes produce data attributes, shared by all instances.

Try it:

```
class SharedData:
        spam = 42
x = SharedData()
y = SharedData()

x.spam, y.spam          # (42, 42)
SharedData.spam = 99
x.spam, y.spam, SharedData.spam  #(99, 99, 99)
x.spam = 88
x.spam, y.spam, SharedData.spam  #(88, 99, 99)
```

Storing the same name in two places

```
class MixedNames:
    data = 'spam'              # class attribute not instance
    def __init__(self, value):
        self.data = value      # instance attribute not class
     def display(self):
         print(self.data, MixedNames.data)

x = MixedNames(1)
y = MixedNames(2)
x.display(); y.display()       # 1 spam
                               # 2 spam
```

Abstract classes: try it

```
class Super:
    def method(self):
        print('in Super.method')
    def delegate(self):
        self.action()              # Expected to be defined
class Inheritor(Super):            # Inherit method verbatim
    pass
class Replacer(Super):             # Replace method completely
    def method(self):
        print('in Replacer.method')
class Extender(Super):             # Extend method behavior
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')
class Provider(Super):             # Fill in a required method
        def action(self):
        print('in Provider.action')

for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()
print('\nProvider...')
x = Provider(),
x.delegate()
```

Abstract classes 3.X – special syntax

```
from abc import ABCMeta, abstractmethod
class Super(metaclass=ABCMeta):
    @abstractmethod
    def method(self, ...):       pass

from abc import ABCMeta, abstractmethod
class Super(metaclass=ABCMeta):
    def delegate(self):
        self.action()
    @abstractmethod
    def action(self):          pass
```

Operator overloading

- Methods named with double underscores (__X__) are special hooks (__init__, __add__, __str__, …)
- Such methods are called automatically when instances appear in built-in operations.
- Classes may override most built-in type operations.
- There are no defaults for operator overloading methods, and none are required.
- New-style classes have some defaults, but not for common operations.
- Operators allow classes to integrate with Python's object model.

Introspection Tools

- The built-in **instance.__class__** attribute provides a link from an instance to the class from which it was created. Classes in turn have a **__name__**, just like modules, and a **__bases__** sequence that provides access to superclasses. We can use these here to print the name of the class from which an instance is made rather than one we've hardcoded.
- The built-in **object.__dict__** attribute provides a dictionary with one key/value pair for every attribute attached to a namespace object (including modules, classes, and instances). Because it is a dictionary, we can fetch its keys list, index by key, iterate over its keys, and so on, to process all attributes generically. We can use this here to print every attribute in any instance, not just those we hardcode in custom displays.

Try it with your own classes!!!

Storing Objects: Pickles and Shelves

pickle

Serializes arbitrary Python objects to and from a string of bytes

dbm

Implements an access-by-key filesystem for storing strings

shelve

Uses the other two modules to store Python objects on a file by key

Pickles and Shelves: the whole process

Pickle module is super-general object formatting and deformatting tool: it converts the object (lists, dictionaries, nested combinations class instances…) to a string of bytes, which it can use later to reconstruct (unpickle) the original object in memory.

Shelve module provides an extra layer of structure that allows you to store pickled objects by *key*. **shelve** translates an object to its pickled string with **pickle** and stores that string under a *key* in a **dbm** file; when later loading, **shelve** fetches the pickled string by *key* and re-creates the original object in memory with **pickle**. Your **shelve** of **pickled** objects looks just like a dictionary.

Try it:

```
from person import Person, Manager   # see Mark Lutz, page 844
bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

import shelve
db = shelve.open('persondb')
for obj in (bob, sue, tom):
        db[obj.name] = obj       #the key can be any string
db.close()
```

Try it:

```
>>> import shelve
>>> db = shelve.open('persondb')
>>> len(db)
>>> list(db.keys())
['Sue Jones', 'Tom Jones', 'Bob Smith']
>>> bob = db['Bob Smith']
>>> bob                    # Runs __repr__ from AttrDisplay (see Mark Lutz, page 842)
[Person: job=None, name=Bob Smith, pay=0]
>>> bob.lastName()
'Smith'
>>> for key in db:
        print(key, '=>', db[key])
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
>>> for key in sorted(db):
         print(key, '=>', db[key])
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Real persistence

```
import shelve
db = shelve.open('persondb')
for key in sorted(db):
        print(key, '\t=>', db[key])
sue = db['Sue Jones']
sue.giveRaise(.10)
db['Sue Jones'] = sue
db.close()
```

```
>>> import shelve
>>> db = shelve.open('persondb')
>>> rec = db['Sue Jones']
>>> rec                      # [Person: job=dev, name=Sue Jones, pay=146410]
>>> rec.lastName()           # 'Jones'
>>> rec.pay                  # 146410
```

Problems to solve

1. Think of a several sensible inheritance trees (may be from your future project). Implement first version of them. Use introspection tools to inspect internals.
2. Analyze and implement A Generic Display Tool (see Mark Lutz, page 842). Apply it to your inheritance tree.
3. Provide persistence for your objects . Experiment with storage and explore it.