Object-Oriented Python — An Introduction

Lectures Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 19th April, 2022 00:09

The Reason for This Material at the Outset

A large majority of people who play with deep learning algorithms operate in the zombie mode — meaning that they simply run canned programs downloaded from the internet with the expectation that a combination of the downloaded software and their own dataset would lead to results that would somehow pave their way to fame and fortune. This, unfortunately, is no way for a student to prepare himself or herself for the future.

The goal of our deep learning class is to help you become more genuine in how you utilize your deep learning skills.

I'll therefore be starting my part of this class with a focus on object-oriented (OO) Python since that's what many of today's software tools for deep learning are based on.

Regarding this lecture, in what follows, I'll start with the main concepts of OO programming in general and then devote the rest of the material to Python OO.

Outline

Outline

Some Examples of PyTorch Syntax

If you are not already well-schooled in the syntax of object-oriented Python, you might find the following examples somewhat befuddling:

```
import torchvision.transforms as tvt
xform      =  tvt.Compose( [tvt.Grayscale(num_output_channels = 1), tvt.Resize((64,64)) ] )
out_image  =  xform( input_image_pil )
```

The statement in the third line appears to indicate that we are using **xform** as a function which is being returned by the statement in the second line. Does that mean functions in Python return functions?

To fully understand what's going on here you have to know what's meant by an object being **callable**. Python makes a distinction between **function objects** and **callables**. While all function objects are callables, not all callables are function objects.

Some Examples of PyTorch Syntax (contd.)

Now consider the following example:

```
class EncoderRNN(torch.nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        === the rest of the definition ===
```

We are obviously trying to define a new class named **EncoderRNN** as a subclass of **torch.nn.Module** and the method **__init__()** is there to initialize an instance object constructed from this class.

But why are we making the call **super(EncoderRNN, self).__init__()** and supplying the name of the subclass again to this method? To understand this syntax, you have to know how you can ask a method to get part of the work done by a method defined for one of its superclasses. How that works is different for a single-inheritance class hierarchy and for a multiple-inheritance class hierarchy.

Some Examples of PyTorch Syntax (contd.)

For another example, the two layers in the following neural network (from a PyTorch tutorial) are declared in lines (A) and (B). And how the network is connected is declared in forward() in line (C). We push data through the network by calling model(x) in line (D). But we never call **forward()**. How is one supposed to understand this?

```
class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        torch.nn.Module.__init__( self )
        self.linear1 = torch.nn.Linear(D_in, H)                                    ## (A)
        self.linear2 = torch.nn.Linear(H, D_out)                                   ## (B)
    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)  # using clamp() for nn.ReLU         ## (C)
        y_pred = self.linear2(h_relu)
        return y_pred


N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)               # N is batch size
y = torch.randn(N, D_out)
model = TwoLayerNet(D_in, H, D_out)
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(10000):
    y_pred = model(x)                                                              ## (D)
    loss = criterion(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Some Examples of PyTorch Syntax (contd.)

For another example that may confuse a beginning Python programmer, consider the following syntax for constructing a data loader in a PyTorch script:

```
training_data = torchvision.datasets.CIFAR10(
        root=self.dataroot, train=True, download=True, transform=transform_train)

train_data_loader = torch.utils.data.DataLoader(
        training_data, batch_size=self.batch_size,shuffle=True, num_workers=2)
```

Subsequently, you may see the following sorts of calls:

```
dataiter = iter(train_data_loader)
images, labels = dataiter.next()
```

or calls like

```
for data in train_data_loader:
    inputs,labels = data
    ...
    outputs = model(inputs)
    ...
```

(continued on next slide)

Some Examples of PyTorch Syntax (contd.)

(continued from previous slide)

For a novice Python programmer, a construct like

```
for x in something:
    ...
```

to make sense, "somethinng" is likely to be one of the typical storage containers, like a list, tuple, set, etc. But 'train_data_loader' does not look like any of those storage containers. So what's going on here?

Outline

The Main OO Concepts

The following fundamental notions of object-oriented programming in general apply to object-oriented Python also:

- Class, Instances, and Attributes
- Encapsulation
- Inheritance
- Polymorphism

What are Classes and Instances?

- At a high level of conceptualization, a class can be thought of as a category. We may think of "Cat" as a class.
- A specific cat would then be an instance of this class.
- For the purpose of writing code, a class is a data structure with attributes.
- An instance constructed from a class will have specific values for the attributes.
- To endow instances with behaviors, a class can be provided with methods.

Attributes: Methods, Instance Variables, and Class Variables

- A method is a function you invoke on an instance of the class or the class itself.
- A method that is invoked on an instance is sometimes called an instance method.
- You can also invoke a method directly on a class, in which case it is called a class method or a static method.
- Attributes that take data values on a per-instance basis are frequently referred to as instance variables.
- Attributes that take on values on a per-class basis are called class attributes or static attributes or class variables.

Encapsulation

- Hiding or controlling access to the implementation-related attributes and the methods of a class is called encapsulation.
- With appropriate data encapsulation, a class will present a well-defined public interface for its clients, the users of the class.
- A client should only access those data attributes and invoke those methods that are in the public interface.

Inheritance and Polymorphism

- Inheritance in object-oriented code allows a subclass to inherit some or all of the attributes and methods of its superclass(es).

- Polymorphism basically means that a given category of objects can exhibit multiple identities at the same time, in the sense that a Cat instance is not only of type Cat, but also of type FourLegged and Animal, all at the same time.

Polymorphism (contd.)

- As an example of polymorphism, suppose we a declare a list

```
animals = ['kitty', 'fido', 'tabby', 'quacker', 'spot']
```

of cats, dots, and a duck — instances made from different classes in some Animal hierarchy — and if we were to invoke a method calculateIQ() on this list of animals in the following fashion

```
for item in animals:
    item.calculateIQ()
```

polymorphism would cause the correct implementation code for `calculateIQ()` to be automatically invoked for each of the animals.

Regarding the Previous Example on Polymorphism

- In many object-oriented languages, a method such as calculateIQ() would need to be declared for the root class Animal for the control loop shown on the previous slide to work properly.
- All of the public methods and attributes defined for the root class would constitute the public interface of the class hierarchy and each class in the hierarchy would be free to provide its own implementation for the methods declared in the root class.
- Polymorphism in a nutshell allows us to manipulate instances belonging to the different classes of a hierarchy through a common interface defined for the root class.

Outline

Attributes: Pre-Defined vs. Programmer-Supplied

- A class in Python comes with certain pre-defined attributes.
- The pre-defined attributes of a class are not to be confused with the programmer-supplied attributes such as the class and instance variables and the programmer-supplied methods.
- By the same token, an instance constructed from a class is an object with certain pre-defined attributes that again are not be confused with the programmer-supplied instance and class variables associated with the instance and the programmer-supplied methods that can be invoked on the instance.

Attributes: Pre-Defined vs. Programmer-Supplied (contd.)

- Note that in Python the word **attribute** is used to describe any property, variable or method that can be invoked with the dot operator on either the class or an instance constructed from a class.

- What I have mentioned above is worthy of note in case you have previously run into an object-oriented language in which a distinction was made between *attributes* and *methods*.

- In Python, the attributes available for a class include the programmer-supplied class and instance variables and methods. This usage of attribute makes it all encompassing, in the sense that it includes the pre-defined data attributes and methods, the programmer-supplied class and instance variables, and, of course, the programmer-supplied methods. Basically, as mentioned above, an attribute is anything that can be invoked using the dot operator on either a class or an instance.

Pre-Defined and Programmer-Supplied Methods for a Class

- To define it formally, a **method** is a function that can be invoked on an object using the object-oriented call syntax that for Python is of the form **obj.method()**, where **obj** may either be an instance of a class or the class itself.

- Therefore, the **pre-defined** functions that can be invoked on either the class itself or on a class instance using the object-oriented syntax are also methods.

- The pre-defined attributes, both variables and methods, employ a special naming convention: *the names begin and end with two underscores*.

- Python makes a distinction between **function objects** and **callables**. While all function objects are callables, not all callables are function objects.

Outline

Function Objects vs. Callables

- **A function object** can only be created with a **def** statement.
- On the other hand, a **callable** is any object that can be called like a function.
- For example, a class name can be called directly to yield an instance of a class. Therefore, a class name is a callable.
- An instance object can also be called directly; what that yields depends on whether or not the underlying class provides a definition for the **system-supplied __call__()** method.

In Python, '()' is an Operator — the Function Call Operator

- You will see objects that may be called with or without the '()' operator and, when they are called with '()', there may or may not exist any arguments inside the parentheses.

- For a class **X** with method **foo**, calling just **X.foo** returns a result different from what is returned by **X.foo()**. The former returns the method object itself that **X.foo** stands for and the latter will cause execution of the function object associated with the method call.

An Example of a Class with Callable Instances

```
import random
random.seed(0)
#---------------------------  class X  -----------------------------
class X:
    def __init__( self, arr ) :
        self.arr = arr
    def get_num(self, i):
        return self.arr[i]
    def __call__(self):
        return self.arr
#----------------------- end of class definition  ---------------------

xobj = X( random.sample(range(1,10), 5) )
print(xobj.get_num(2))                                    #  1
print(xobj())                                             #  [7, 9, 1, 3, 5]
```

If you execute this code, you will see the output shown in the commented out portions of the last two lines.

In the last line of the code, note how we are calling the function call operator '()' on the instance constructed from the class X.

The Same Example But With No Def for __call__

```
import random
random.seed(0)
#----------------------------  class X  ---------------------------
class X:
    def __init__( self, arr ) :
        self.arr = arr
    def get_num(self, i):
        return self.arr[i]
#    def __call__(self):
#        return self.arr
#------------------------- end of class definition  ----------------
xobj = X( random.sample(range(1,10), 5) )
print(xobj.get_num(2))                                  #  1
print(xobj())                                           #  Traceback (most recent call last)
                                                        #    File "UsingCall2.py", line 15, in <module>
                                                        #        print(xobj())
                                                        #  TypeError: 'X' object is not callable
```

Outline

Defining a Class in Python

- I'll present the full definition of a Python class in stages.

- I'll start with a very simple example of a class to make the reader familiar with the *pre-defined* **__init__()** method whose role is to initialize the instance returned by a call to the constructor.

- First, here is the simplest possible definition of a class in Python:

```
class SimpleClass:
    pass
```

  An instance of this class may be constructed by invoking its pre-defined default constructor:

```
x = SimpleClass()
```

Defining a Class in Python (contd.)

- Here is a class with a *user-supplied* **__init__()**. This method is automatically invoked to initialize the state of the instance returned by a call to **Person()**:

```
#------------- class Person --------------
class Person:
    def __init__(self, a_name, an_age ):
        self.name = a_name
        self.age = an_age
#--------- end of class definition --------

#test code:
a_person = Person( "Zaphod", 114 )
print(a_person.name)                    # Zaphod
print(a_person.age)                     # 114
```

Pre-Defined Attributes for a Class

- Being an object in its own right, every Python class comes equipped with the following pre-defined attributes:

__name__ : string name of the class

__doc__ : documentation string for the class

__bases__ : tuple of parent classes of the class

__dict__ : dictionary whose keys are the names of the class variables and the methods of the class and whose values are the corresponding bindings

__module__ : module in which the class is defined

Pre-Defined Attributes for an Instance

- Since every class instance is also an object in its own right, it also comes equipped with certain pre-defined attributes. We will be particularly interested in the following two:

  __class__ : string name of the class from which the instance was constructed

  __dict__ : dictionary whose keys are the names of the instance variables

- It is important to realize that the namespace as represented by the dictionary **__dict__** for a class object is not the same as the namespace as represented by the dictionary **__dict__** for an instance object constructed from the class.

__dict__ vs. dir()

- As an alternative to invoking **__dict__** on a class name, one can also use the built-in global **dir()**, as in

```
dir( MyClass )
```

  that returns a list of all the attribute names, for variables and for methods, for the class (both directly defined for the class and inherited from a class's superclasses).

- If we had called "`print( dir(Person) )`" for the `Person` class defined on Slide 30, the system would have returned:

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
    '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
    '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
    '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Illustrating the Values for System-Supplied Attributes

```
#------ class Person --------
class Person:
    "A very simple class"
    def __init__(self,nam,yy ):
        self.name = nam
        self.age = yy

#-- end of class definition --

#test code:
a_person = Person("Zaphod",114)
print(a_person.name)                    # Zaphod
print(a_person.age)                     # 114

# class attributes:
print(Person.__name__)                  # Person
print(Person.__doc__)                   # A very simple class
print(Person.__module__)                # main
print(Person.__bases__)                 # ()
print(Person.__dict__)                  # {‘__module__’ : ‘__main__’, ‘__doc__’ : ‘A very simp..’,
                                        # ‘__init__’:<function __init..’,
# instance attributes:
print(a_person.__class__)               # __main__.Person
print(a_person.__dict__ )               # {‘age’:114, ‘name’:’Zaphod’}
```

Class Definition: More General Syntax

```
class MyClass :
    ‘optional documentation string’
    class_var1
    class_var2 = var2

    def __init__( self, var3 = default3 ):
        ‘optional documentation string’
        attribute3 = var3
        rest_of_construction_init_suite

    def some_method( self, some_parameters ):
        ‘optional documentation string’
        method_code

    ...
    ...
```

Class Definition: More General Syntax (contd.)

- Regarding the syntax shown on the previous slide, note the class variables **class_var1** and **class_var2**. Such variables exist on a per-class basis, meaning that they are static.

- A class variable can be given a value in a class definition, as shown for **class_var2**.

- In general, the header of **__init__()** may look like:

```
def __init__(self, var1, var2, var3 = default3):
    body_of_init
```

  This constructor initializer could be for a class with three instance variables, with the last default initialized as shown. The first parameter, typically named `self`, is set implicitly to the instance under construction.

Class Definition: More General Syntax (cond.)

- If you do not provide a class with its own **__init__()**, the system will provide the class with a default **__init__()**. You override the default definition by providing your own implementation for **__init__()**.

- The syntax for a user-defined method for a class is the same as for stand-alone Python functions, except for the special significance accorded the first parameter, typically named `self`. It is meant to be bound to a reference to the instance on which the method is invoked.

The Root Class `object`

- All classes are subclassed, either directly or indirectly from the root class **object**.
- The **object** class defines a set of methods with default implementations that are inherited by all classes derived from object.
- The list of attributes defined for the **object** class can be seen by printing out the list returned by the built-in **dir()** function:

  ```
  print( dir( object ) )
  ```

  This call returns

  ```
  ['__class__','__delattr__','__doc__','__getattribute__','__hash__','__init__','__new__',__reduce__',
          '__reduce_ex__','__repr__','__setattr__','__str__']
  ```

- We can also examine the attribute list available for the **object** class by printing out the contents of its __**dict**__ attribute by

  ```
  print( object.__dict__ )
  ```

  This will print out both the attribute names and their bindings.

Outline

How Python Creates an Instance from a Class

Python uses the following two-step procedure for constructing an instance from a class:

STEP 1:

- The call to the constructor creates what may be referred to as a generic instance from the class definition.
- The generic instance's memory allocation is customized with the code in the method `__new__()` of the class. This method may either be defined directly for the class or the class may inherit it from one of its parent classes.

Creating an Instance from a Class (contd.)

- The method `__new__()` is implicitly considered by Python to be a static method. Its first parameter is meant to be set equal to the name of the class whose instance is desired and it must return the instance created.
- If a class does not provide its own definition for `__new__()`, a search is conducted for this method in the parent classes of the class.

STEP 2:

- Then the instance method `__init__()` of the class is invoked to initialize the instance returned by `__new__()`.

Example Showing __new__() and __init__() Working Together for Instance Creation

- The script shown on Slide 43 defines a class `X` and provides it with a *static method* `__new__()` and an *instance method* `__init__()`.

- We do not need any special declaration for `__new__()` to be recognized as static because this method is special-cased by Python.

- Note the contents of the namespace dictionary `__dict__` created for class `X` as printed out by `X.__dict__`. This dictionary shows the names created specifically for class `X`. On the other hand, `dir(X)` also shows the names inherited by `X`.

Instance Construction Example (contd.)

- In the script on the next slide, also note that the namespace dictionary `xobj.__dict__` created at runtime for the instance `xobj` is empty — for obvious reasons.

- As stated earlier, when `dir()` is called on a class, it returns a list of all the attributes that can be invoked on a class and on the instances made from that class. The returned list also includes the attributes inherited from the class's parents.

- When called on a instance, as in `dir( xobj )`, the returned list is the same as above plus any instance variables defined for the class.

Instance Construction Example (contd.)

```
#------------------ class X --------------------

class X (object):                          # X derived from root class object
    def __new__( cls ):                    # the param 'cls' set to the name of the class
        print("__new__ invoked")
        return object.__new__( cls )
    def __init__( self ):
        print("__init__ invoked")


#------------------- Test Code -----------------

xobj = X()                # __new__ invoked
                          # __init__ invoked
print( X.__dict__ )       #{'__module__': '__main__', '__new__': <static method ..>,
                          # .......}

print( dir(X) )           #['__class__','__delattr__', '__getattribute__', '__hash__',
                          # '__init__', '__module__', '__new__', ..........]

print( dir( xobj ) )      #['__class__', '__delattr__', '__getattribute__', '__hash__',
                          # '__init__', '__module__', '__new__', .........]
```

Instance Construction Example (contd.)

```
#------------------ class X --------------------
class X:                                     # X is still derived from the root class object
    def __new__( cls ):                      # The param 'cls' set to the name of the class
        print("__new__ invoked")
        return object.__new__( cls )
    def __init__( self ):
        print("__init__ invoked")

#------------------- Test Code -----------------

xobj = X()              # __new__ invoked
                        # __init__ invoked
print( X.__dict__ )     # {'__module__': '__main__', '__new__': <static method ..>,
                        #   ....... }
print( dir(X) )         # ['__class__','__delattr__', '__getattribute__', '__hash__',
                        #  '__init__', '__module__', '__new__', ........]
print( dir( xobj ) )    # ['__class__', '__delattr__', '__getattribute__', '__hash__',
                        #  '__init__', '__module__', '__new__',.......]
```

Outline

The Syntax for Defining a Method

- A method defined for a class must have special syntax that reserves the first parameter for the object on which the method is invoked. This parameter is typically named `self` for instance methods, but could be any legal Python identifier.

- In the script shown on the next slide, when we invoke the constructor using the syntax

  ```
  xobj = X( 10 )
  ```

  the parameter `self` in the call to `__init__()` is set implicitly to the instance under construction and the parameter `nn` to the value 10.

- A method may call any other method of a class, but such a call must always use class-qualified syntax, as shown by the definition of `bar()` on the next slide.

Defining a Method (contd.)

In the definition shown below, one would think that a function like `baz()` in the script below could be called using the syntax `X.baz()`, but that does not work. (We will see later how to define a class method in Python).

```
#---------- class X ------------
class X:
    def __init__(self, nn):              #  the param 'self' is bound to the instance
        self.n = nn
    def getn(self):                      #  instance method
        return self.n
    def foo(self,arg1,arg2,arg3=1000):   #  instance method
        self.n = arg1 + arg2 + arg3
    def bar( self ):                     #  instance method
        self.foo( 7, 8, 9 )
    def baz():                           #  this is NOT how you define a class method
        pass
#--- End of Class Definition ----

xobj = X(10)
print( xobj.getn() )                # 10
xobj.foo(20,30)
print( xobj.getn() )                # 1050
xobj.bar()
print( xobj.getn() )                # 24
# X.baz()                           # ERROR
```

A Method Can be Defined Outside a Class

- It is not necessary for the body of a method to be enclosed by a class.

- A function object created outside a class can be assigned to a name inside the class. The name will acquire the function object as its binding. Subsequently, that name can be used in a method call as if the method had been defined inside the class.

- In the script shown on the next slide, the important thing to note is that is that the assignment to foo gives X an attribute that is a function object. As shown, this object can then serve as an instance method.

Method Defined Outside a Class (contd.)

```
def bar(self,arg1,arg2, arg3=1000):
    self.n = arg1 + arg2 + arg3

#---------- class X ------------
class X:
    foo = bar
    def __init__(self, nn):
        self.n = nn
    def getn(self):
        return self.n
#--- End of Class Definition ----

xobj = X(10)
print( xobj.getn() )                    # 10
xobj.foo( 20, 30 )
print( xobj.getn() )                    # 1050
```

Only One Method for a Given Name Rule

- When the Python compiler digests a method definition, it creates a function binding for the name of the method.

- For example, for the following code fragment

```
class X:
    def foo(self, arg1, arg2):
        implemention_of_foo
    rest_of_class_X
```

  the compiler will introduce the name foo as a key in the namespace dictionary for class X. The value entered for this key will be the function object corresponding to the body of the method definition.

Only One Method Per Name Rule (contd.)

- So if you examine the attribute **X.__dict__** after the class is compiled, you will see the following sort of entry in the namespace dictionary for X:

```
'foo' : <function foo at 0x805a5e4>
```

- Since all the method names are stored as keys in the namespace dictionary and since the dictionary keys must be unique, this implies that there can exist only one function object for a given method name.

- If after seeing the code snippet shown on the previous slide, the compiler saw another definition for a method named for the same class, then **regardless of the parameter structure of the function**, the new function object will replace the old for the value entry for the method name. This is unlike what happens in C++ and Java where function overloading plays an important role.

Method Names can be Usurped by Data Attribute Names

- We just talked about how there can only be one method of a given name in a class — regardless of the number of arguments taken by the method definitions.
- As a more general case of the same property, a class can have only one attribute of a given name.
- What that means is that if a class definition contains a class variable of a given name after a method attribute of the same name has been defined, the binding stored for the name in the namespace dictionary will correspond to the definition that came later.

Destruction of Instance Objects

- Python comes with an automatic garbage collector. Each object created is kept track of through reference counting. Each time an object is assigned to a variable, its reference count goes up by one, signifying the fact that there is one more variable holding a reference to the object.
- And each time a variable whose referent object either goes out of scope or is changed, the reference count associated with the object is decreased by one. When the reference count associated with an object goes to zero, it becomes a candidate for garbage collection.
- Python provides us with **__del__ ()** for cleaning up beyond what is done by automatic garbage collection.

Encapsulation Issues for Classes

- Encapsulation is one of the cornerstones of OO. How does it work in Python?
- As opposed to OO in C++ and Java, all of the attributes defined for a class are available to all in Python.
- So the language depends on programmer cooperation if software requirements, such as those imposed by code maintenance and code extension considerations, dictate that the class and instance variables be accessed only through get and set methods.
- A Python class and a Python instance object are so open that they can be modified after the objects are brought into existence.

Defining Static Attributes for a Class

- A class definition usually includes two different kinds of attributes: those that exist on a per-instance basis and those that exist on a per-class basis. The latter are commonly referred to as being static.
- A variable becomes **static** if it is declared outside of any method in a class definition.
- For a method to become static, it needs the **staticmethod()** wrapper.
- Shown on the next slide is a class with a **class variable** (meaning a static data attribute) **next_serial_num**

Static Attributes (contd.)

```
#---------- class Robot ------------
class Robot:
    next_serial_num = 1
    def __init__(self, an_owner):
        self.owner = an_owner
        self.idNum = self.get_next_idNum()
    def get_next_idNum( self ):
        new_idNum = Robot.next_serial_num
        Robot.next_serial_num += 1
        return new_idNum
    def get_owner(self):
        return self.owner
    def get_idNum(self):
        return self.idNum
#----- End of Class Definition -----

robot1 = Robot("Zaphod")
print( robot1.get_idNum() )                # 1

robot2 = Robot("Trillian")
print( robot2.get_idNum() )                # 2

robot3 = Robot("Betelgeuse")
print( robot3.get_idNum() )                # 3
```

Static Methods

- In the old days, a static method used to be created by supplying a function object to **staticmethod()** as its argument. For example, to make a method called **foo()** static, we'd do the following

```
def foo():
    print(``foo called'')
foo = staticmethod( foo )
```

  The function object returned by **staticmethod()** is static.

- In the above example, when foo is subsequently called directly on the class using the function call operator '()', it is the callable object bound to foo in the last statement above that gets executed.

- The modern approach to achieving the same effect is through the use of the staticmethod decorator, as in

```
@staticmethod
def foo():
    print ``foo called''
```

Outline

Extending a Class — Using `super()` in Method Definitions

- Method extension for the case of single-inheritance is illustrated in the `Employee-Manager` class hierarchy shown on Slide 61. Note how the derived-class `promote()` calls the base-class `promote()`, and how the derived-class `myprint()` calls the base-class `myprint()`.

- As you will see later, extending methods in **multiple inheritance** hierarchies requires calling `super()`. To illustrate, suppose we wish for a method `foo()` in a derived class `Z` to call on `foo()` in `Z`'s superclasses to do part of the work:

```
class Z( A, B, C, D):
    def foo( self ):
        ....do something....
        super( Z, self).foo()
```

- In the next couple of slides, we will first consider the case of single inheritance, that is, when each child class has a single parent class.

Method Definitions in a Derived Class Calling a Superclass Definition Directly

- The next two slides show a single inheritance hierarchy, with `Employee` as the base class and the `Manager` as the derived class.
- We want both the `promote()` and the `myprint()` methods of the derived class to call on the method of the same names in the base class to do a part of the job.
- As shown in lines (A) and (B), we accomplish this by calling the methods `promote()` and the `myprint()` directly on the class name `Employee`.

Calling a Superclass Method Definition Directly

```
#-------------- base class Employee -----------------
class Employee:
    def __init__(self, nam, pos):
        self.name = nam
        self.position = pos
    promotion_table = {
                         'shop_floor' : 'staff',
                         'staff'      : 'manager',
                         'manager'    : 'executuve'
    }
    def promote(self):
        self.position = Employee.promotion_table[self.position]

    def myprint(self):
        print( self.name, "%s" % self.position, end=" ")

#------------- derived class Manager --------------
class Manager( Employee ):
    def __init__(self, nam, pos, dept):
        Employee.__init__(self,nam,pos)
        self.dept = dept

    def promote(self):              ## this promote() calls parent's promote() for part of the work
        if self.position == 'executive':
            print( "not possible" )
            return
        Employee.promote( self )                                                  ## (A)

    def myprint(self):              ## this print() calls parent's print() for part of the work
        Employee.myprint(self)                                                    ## (B)
        print( self.dept )
```

Calling a Superclass Method Definition Directly (contd.)

(continued from previous slide)

```
#------------------ Test Code --------------------
emp = Employee("Orpheus", "staff")
emp.myprint()                                  # Orpheus staff
emp.promote()
print( emp.position )                          # manager
emp.myprint()                                  # Orpheus manager

man = Manager("Zaphod", "manager", "sales")
man.myprint()                                  # Zaphod manager sales
man.promote()
print( man.position )                          # executive

print(isinstance(man, Employee))               # True
print(isinstance(emp, Manager))                # False
print(isinstance(man, object))                 # True
print(isinstance(emp, object))                 # True
```

Method Definition in a Derived Class Calling `super()` for the Parent Class's Method

- Shown on the next slide is the same class single-inheritance class hierarchy you saw on the previous slides and the methods of the derived class still want a part of the job to be done by the methods of the parent class.
- But now the methods of the derived class invoke the built-in `super()` for calling on the methods of the parent class.

Method Definition in a Derived Class Calling `super()` for the Parent Class's Method (contd.)

```
#------------- base class Employee -----------------
class Employee:
    def __init__(self, nam, pos):
        self.name = nam
        self.position = pos
    promotion_table = {
                        'shop_floor' : 'staff',
                        'staff'      : 'manager',
                        'manager'    : 'executuve'
    }
    def promote(self):
        self.position = Employee.promotion_table[self.position]
    def myprint(self):
        print( self.name, "%s" % self.position, end=" ")

#------------ derived class Manager --------------
class Manager( Employee ):
    def __init__(self, nam, pos, dept):
        Employee.__init__(self,nam,pos)
        self.dept = dept

    def promote(self):
        if self.position == 'executive':
            print( "not possible" )
            return
        super(Manager, self).promote()                                 ## (A)

    def myprint(self):
        super(Manager, self).myprint()                                 ## (B)
        print( self.dept )
```

Derived Class Constructor Calling Base Class's Constructor Directly

- The point of this slide is to emphasize what you might have noticed already in the `Employee-Manager` class hierarchy: The first thing that the constructor of a derived class must do is to call on the constructor of the base class for the initializations that are in the latter's `__init__()`.

- In the code shown on the next slide, this call is made in line (A). As you can see, the derived class is calling `__init__()` directly on the name of the base class.

Derived Class Constructor Calls Base Class Constructor Directly (contd.)

```
#-------------- base class Employee -----------------
class Employee:
    def __init__(self, nam, pos):
        self.name = nam
        self.position = pos
    promotion_table = {
                        'shop_floor' : 'staff',
                        'staff'      : 'manager',
                        'manager'    : 'executuve'
    }
    def promote(self):
        self.position = Employee.promotion_table[self.position]

    def myprint(self):
        print( self.name, "%s" % self.position, end=" ")

#------------- derived class Manager --------------
class Manager( Employee ):
    def __init__(self, nam, pos, dept):
        Employee.__init__(self, nam, pos)                              ## (A)
        self.dept = dept

    def promote(self):
        if self.position == 'executive':
            print( "not possible" )
            return
        super(Manager, self).promote()

    def myprint(self):
        super(Manager, self).myprint()
        print( self.dept )

#------------------ Test Code --------------------
emp = Employee("Orpheus", "staff")
emp.myprint()                                # Orpheus staff
print("\n")
emp.promote()
print( emp.position )                        # manager
emp.myprint()                                # Orpheus manager
man = Manager("Zaphod", "manager", "sales")
man.myprint()                                # Zaphod manager sales
man.promote()
print("\n")
print( man.position )                        # executive

print(isinstance(man, Employee))             # True
print(isinstance(emp, Manager))              # False
print(isinstance(man, object))               # True
print(isinstance(emp, object))               # True
```

Derived Class Constructor Calling Base Class Constructor Through `super()`

- As shown on the next slide, another way for a derived class constructor to call on the constructor of the base class is using the built-in `super()` as shown on the next slide.
- Note in Line (A) the following syntax for the call to `super()`:

```
super(Manager, self).__init__(nam,pos)
```

- The call to `super()` MUST mention the name of the derived class in which `super()` is called.

Derived Class Constructor Calling Base Class Constructor Through `super()` (contd.)

```
#-------------- base class Employee -----------------
class Employee:
    def __init__(self, nam, pos):
        self.name = nam
        self.position = pos
    promotion_table = {
                        'shop_floor' : 'staff',
                        'staff'      : 'manager',
                        'manager'    : 'executuve'
    }
    def promote(self):
        self.position = Employee.promotion_table[self.position]

    def myprint(self):
        print( self.name, "%s" % self.position, end=" ")

#------------- derived class Manager --------------
class Manager( Employee ):
    def __init__(self, nam, pos, dept):
        super(Manager, self).__init__(nam,pos)                    ## (A)
        self.dept = dept

    def promote(self):
        if self.position == 'executive':
            print( "not possible" )
            return
        super(Manager, self).promote()
    def myprint(self):
        super(Manager, self).myprint()
        print( self.dept )

#------------------ Test Code --------------------
emp = Employee("Orpheus", "staff")
emp.myprint()                                 # Orpheus staff
print("\n")
emp.promote()
print( emp.position )                         # manager
emp.myprint()                                 # Orpheus manager
man = Manager("Zaphod", "manager", "sales")
man.myprint()                                 # Zaphod manager sales
man.promote()
print("\n")
print( man.position )                         # executive

print(isinstance(man, Employee))              # True
print(isinstance(emp, Manager))               # False
print(isinstance(man, object))                # True
print(isinstance(emp, object))                # True
```

New Style Syntax for Calling `super()` in Derived-Class Constructor

- However, Python3 allows a derived class's constructor to call `super()` without using the derived class's name as its argument.

- Shown on the next slide is the same hierarchy that you saw on the previous slide, but with the following new-style syntax for the call to `super()` in line (A):

```
super().__init__(nam, pos)
```

New Style Syntax for Calling super() in Derived-Class Constructor (contd.)

```
#-------------- base class Employee -----------------
class Employee:
    def __init__(self, nam, pos):
        self.name = nam
        self.position = pos
    promotion_table = {
                        'shop_floor' : 'staff',
                        'staff'      : 'manager',
                        'manager'    : 'executuve'
    }
    def promote(self):
        self.position = Employee.promotion_table[self.position]

    def myprint(self):
        print( self.name, "%s" % self.position, end=" ")

#------------- derived class Manager --------------
class Manager( Employee ):
    def __init__(self, nam, pos, dept):
        super().__init__(nam, pos)                                 ## (A)
        self.dept = dept

    def promote(self):
        if self.position == 'executive':
            print( "not possible" )
            return
        super(Manager, self).promote()

    def myprint(self):
        super(Manager, self).myprint()
        print( self.dept )

#------------------ Test Code --------------------
emp = Employee("Orpheus", "staff")
emp.myprint()                                 # Orpheus staff
print("\n")
emp.promote()
print( emp.position )                         # manager
emp.myprint()                                 # Orpheus manager
print()
man = Manager("Zaphod", "manager", "sales")
man.myprint()                                 # Zaphod manager sales
man.promote()
print("\n")
print( man.position )                         # executive

print(isinstance(man, Employee))              # True
print(isinstance(emp, Manager))               # False
print(isinstance(man, object))                # True
print(isinstance(emp, object))                # True
```

Outline

An Example of Diamond Inheritance

- Python allows a class to be derived from multiple base classes. In general, the header of such a derived class would look like

```
class derived_class( base1, base2, base3, .... ):
    ....body of the derived class....
```

- The example shown on the next slide contains a diamond hierarchy for illustrating multiple inheritance. The class D inherits from two parents, B and C. Both those classes inherit from A, thus forming an example of diamond inheritance.

- Such hierarchies lead to following questions: Suppose we invoke a method on a derived-class instance and the method is not defined directly in the derived class, in what order will the base classes be searched for an implementation of the method? The order in which the class and its bases are searched for the implementation code is commonly referred to as the **Method Resolution Order** (MRO) in Python.

An Example of Diamond Inheritance (contd.)

```
#--------------  Classes of A, B, C, D of the hierarchy  ---------------
class A (object):
    def __init__(self):
        print("called A's init")

class B( A ):
    def __init__(self):
        print("called B's init")
        super().__init__()

class C( A ):
    def __init__(self):
        print("called C's init")
        super().__init__()

class D(B,C):
    def __init__(self):
        print("called D's init")
        super().__init__()

#--------------------------  Test Code  -------------------------------
dobj = D()
#              called D's init
#              called B's init
#              called C's init
#              called A's init
#

print(D.__mro__)
#                    (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <t
```

On the Importance of Using `super()` in `__init__()`

- In order to illustrate the importance of using `super()`, I will present GOOD-practice and BAD-practice scenarios when defining a multiple-inheritance hierarchy.

- The example shown on the previous slide represented a GOOD-practice scenario in which we called the parent class's `__init__()` in the definition of `__super__()` for all the derived classes.

- To see why what the programmer of the ABC-hierarchy did on the previous slide amounted to a good practice: Let's say that a client of the original ABC hierarchy decides to extend that hierarchy because he/she wants to endow the original classes B and C with additional behavior that is defined in the class `Mixin` defined on the next slide. The "mixed-in" versions of the original classes B and C are called P and Q, respectively.

Importance of Using `super()` in `__init__()` (contd.)

```
#-----------------  Classes of A, B, C of the original hierarchy  ----------------
class A (object):
    def __init__(self):
        print("called A's init")

class B( A ):
    def __init__(self):
        print("called B's init")
        super(B,self).__init__()

class C( A ):
    def __init__(self):
        print("called C's init")
        super(C,self).__init__()

#----- A client wants to mixin additional behaviors with the classes B and C ---
class Mixin(A):
    def __init__(self):
        print("called Mixin's init")
        super(Mixin, self).__init__()

class P(B, Mixin):
    def __init__(self):
        print("called P's init")
        super(P, self).__init__()

class Q(C, Mixin):
    def __init__(self):
        print("called Q's init")
        super(Q, self).__init__()

#---------------------------  Test Code  ------------------------------

print("\nP  instance being constructed:\n")
Pobj = P()
#                    called P's init
#                    called B's init
#                    called Mixin's init
#                    called A's init

print("\nQ  instance being constructed:\n")
Qobj = Q()
#                    called Q's init
#                    called C's init
#                    called Mixin's init
#                    called A's init

print(P.__mro__)
#                   (<class '__main__.P'>, <class '__main__.B'>, <class '__main__.Mixin'>, <class '__main__.A'>, <class 'object'>)

print(Q.__mro__)
#                   (<class '__main__.Q'>, <class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.A'>, <class 'object'>)
```

New-Style Version of the Good-Practice Example of Using `super` in `__init__()`

- The example shown on the next slide is basically the same as what was shown in the previous slide: A GOOD-practice scenario in which the original programmer of the `ABC` hierarchy has used the new-style `super()` to call the parent class's `__init__()` in the `__init__()` for all the derived classes.
- The only difference from the previous example is that the calls like

```
super(B,self).__init__()
```

  in the `__init__()` of the derived classes have been replaced by the more compact

```
super().__init__()
```

- As a result, when a client of the ABC-hierarchy decides to extend the original hierarchy in order to create mixin versions of those classes as shown in the part of the code that begins with the class definition for the `Mixin` class, there is no problem.

New-Style Version of the Good Practice Example (contd.)

```
#-----------------  Classes A, B, C of the orignal hierarchy  ----------------
class A (object):
    def __init__(self):
        print("called A's init")

class B( A ):
    def __init__(self):
        print("called B's init")
        super().__init__()

class C( A ):
    def __init__(self):
        print("called C's init")
        super().__init__()

#------- The Mixin class and the mixin versions of the B and C classes -----
class Mixin(A):
    def __init__(self):
        print("called Mixin's init")
        super().__init__()

class P(B, Mixin):
    def __init__(self):
        print("called P's init")
        super().__init__()

class Q(C, Mixin):
    def __init__(self):
        print("called Q's init")
        super().__init__()

#--------------------------  Test Code  ------------------------------

print("\nP  instance being constructed:\n")
Pobj = P()
#                   called P's init
#                   called B's init
#                   called Mixin's init
#                   called A's init

print("\nQ  instance being constructed:\n")
Qobj = Q()
#                   called Q's init
#                   called C's init
#                   called Mixin's init
#                   called A's init

print(P.__mro__)
#                  (<class '__main__.P'>, <class '__main__.B'>, <class '__main__.Mixin'>, <class '__main__.A'>, <class 'object'>)

print(Q.__mro__)
#                  (<class '__main__.Q'>, <class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.A'>, <class 'object'>)
```

Consequences of NOT Using `super()`

- What's shown on the next slide represents a BAD-practice scenario in which the original programmer has explicitly called the parent class's `__init__()` in the definition of `__init__()` for the derived class B.

- Subsequently, a client of the ABCD hierarchy decides to extend the ABC hierarchy because he/she wants to endow the original classes B and C with additional behavior that is defined in the class Mixin as defined below. The "mixed-in" versions of the original classes B and C are called P and Q, respectively.

- As shown in the "test" section, while Q's constructor calls `Mixin`'s constructor, as it should, P's constructor fails to do so.

Consequences of NOT Using `super()` (contd.)

```
#-----------------  Classes of A, B, C of the hierarchy  ----------------
class A (object):
    def __init__(self):
        print("called A's init")

class B( A ):
    def __init__(self):
        print("called B's init")
        A.__init__(self)

class C( A ):
    def __init__(self):
        print("called C's init")
        super(C,self).__init__()

class D(B,C):
    def __init__(self):
        print("called D's init")
        super(D,self).__init__()

#----------------------  User classes Mixin, P, Q -----------------------
class Mixin(A):
    def __init__(self):
        print("called Mixin's init")
        super(Mixin, self).__init__()

class P(B, Mixin):
    def __init__(self):
        print("called P's init")
        super(P, self).__init__()

class Q(C, Mixin):
    def __init__(self):
        print("called Q's init")
        super(Q, self).__init__()

#---------------------------  Test Code  -------------------------------
print("\nP  instance being constructed:\n")
Pobj = P()                                     ## this constructor call fails to call the Mixin constructor
#                    called P's init
#                    called B's init
#                    called A's init

print("\nQ  instance being constructed:\n")
Qobj = Q()
#                    called Q's init
#                    called C's init
#                    called Mixin's init
#                    called A's init

print(P.__mro__)
# (<class '__main__.P'>, <class '__main__.B'>, <class '__main__.Mixin'>, <class '__main__.A'>, <class 'object'>)

print(Q.__mro__)
# (<class '__main__.Q'>, <class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.A'>, <class 'object'>)
```

Outline

Iterable vs. Iterator

- A class instance is iterable if you can loop over the data stored in the instance. The data may be stored in the different attributes and in ways not directly accessible by, say, array like indexing.

- A class must provide for an iterator in order for its instances to be iterable and this iterable must be returned by the definition of `__iter__` for the class.

- Commonly, the iterator defined for a class will itself be a class that provides implementation for a method named `__next__` in Python 3 and `next` in Python 2.

An Example of an Iterable Class

```
import random
random.seed(0)
#----------------------------  class X  -------------------------------
class X:
    def __init__( self, arr ) :
        self.arr = arr
    def get_num(self, i):
        return self.arr[i]
    def __call__(self):
        return self.arr
    def __iter__(self):
        return Xiterator(self)

class Xiterator:
    def __init__(self, xobj):
        self.items = xobj.arr
        self.index = -1
    def __iter__(self):
        return self
    def __next__(self):
        self.index += 1
        if self.index < len(self.items):
            return self.items[self.index]
        else:
            raise StopIteration
    next = __next__
#------------------------ end of class definition  ---------------------
```

(continued on next slide)

An Example of an Iterable Class (contd.)

(continued from previous slide)

```
xobj = X( random.sample(range(1,10), 5) )
print(xobj.get_num(2))                                      # 1
print(xobj())                                               # [7, 9, 1, 3, 5]         because xobj is callable
for item in xobj:
    print(item, end=" ")                                    # 7 9 1 3 5               because xobj is iterable

## Using the built-in iter() to construct a new instance of the iterator over the same instance of X:
it   =  iter(xobj)                                          # iter() is a built-in function that calls __iter__()
print(it.next())                                            # 7
print(it.next())                                            # 9

## Trying the iterator for a new instance of X:
xobj2 = X( random.sample(range(1,10), 5) )
print(xobj2())                                              # [8, 7, 9, 3, 4]        because xobj2 is callable
it2 = iter(xobj2)                                           # iter() returns a new instance of the iterator
print(it2.next())                                           # 8
```