The power of OOP

- We can bundle together objects that share common attributes with procedures or functions that operate on those attributes
- We can use abstraction to isolate the use of objects from the details of how they are constructed
- We can build layers of object abstractions that inherit behaviors from associated classes of objects
- We can create our own classes of objects on top of Python's basic classes

Defining new types

- In Python, the `class` statement is used to define a new type

```
class Coordinate(object):
    … define attributes here …
```

- As with `def`, indentation used to indicate which statements are part of the class definition

Defining new types

- In Python, the `class` statement is used to define a new type

```
class Coordinate(object):
    … define attributes here …
```

- As with `def`, indentation used to indicate which statements are part of the class definition
- Classes can inherit attributes from other classes, in this case `Coordinate` inherits from the object classs. `Coordinate` is said to be a **subclass** of `object`, `object` is a **superclass** of `Coordinate`. One can override an inherited attribute with a new definition in the class statement.

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Method is another name for a procedural attribute, or a procedure that "belongs" to this class

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator is used to access an attribute of an object. So the `__init__` method above is defining two attributes for the new `Coordinate` object: `x` and `y`.

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator is used to access an attribute of an object. So the `__init__` method above is defining two attributes for the new `Coordinate` object: `x` and `y`.

When accessing an attribute of an instance, start by looking within the class definition, then move up to the definition of a superclass, then move to the global environment

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator is used to access an attribute of an object. So the `__init__` method above is defining two attributes for the new `Coordinate` object: `x` and `y`.

Data attributes of an instance are often call **instance variables.**

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

c = Coordinate(3,4)
origin = Coordinate(0,0)
print c.x, origin.x
```

The expression `classname(values…)` creates a new object of type classname and then calls its `__init__` method with the new object and `values…` as the arguments. When the method is finished executing, Python returns the initialized object as the value.

Note that don't provide argument for self, Python does this automatically

Visualizing this idea