Defining New Types

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

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

Like with `def`, indentation is used to indicate which statements are part of the definition.

Classes can inherit attributes from other classes, in this case Coordinate inherits from the `object` class. 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'll 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 our fancy name for a procedural attribute

Creating an Instance

Usually when creating an instance of a type, we'll 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 Python programmers use `self` *as the name for 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 called instance variables.

Creating an Instance

Usually when creating an instance of a type, we'll 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
```

This will print "3, 0"

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.*

Print Representation of an Object

Left to its own devices, Python uses a unique but uninformative print presentation for an object.

```
>>> print c
<__main__.Coordinate object at 0x7fa918510488>
```

One can define a `__str__` method for a class, which Python will call when it needs a string to print. This method will be called with the object as the first argument and should return a `str`.

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

  def __str__(self):
    return "<"+self.x+","+self.y+">"
```

```
>>> print c
<3,4>
```

Type of an Object

We can ask for the type of an object

```
>>> print type(c)
<class __main__.Coordinate>
```

This makes sense since

```
>>> print Coordinate, type(Coordinate)
<class __main__.Coordinate> <type 'type'>
```

Use `isinstance()` to check if an object is a Coordinate

```
>>> print isinstance(c,Coordinate)
True
```