Defining Classes in Python

User-defined classes are created by `class` statements, which consist of a single clause. A class statement defines the class name and a base class (discussed in the section on Inheritance), then includes a suite of statements to define the attributes of the class:

```
class <name>(<base class>):
    <suite>
```

When a class statement is executed, a new class is created and bound to `<name>` in the first frame of the current environment. The suite is then executed. Any names bound within the `<suite>` of a `class` statement, through `def` or assignment statements, create or modify attributes of the class.

Classes are typically organized around manipulating instance attributes, which are the name-value pairs associated not with the class itself, but with each object of that class. The class specifies the instance attributes of its objects by defining a method for initializing new objects. For instance, part of initializing an object of the `Account` class is to assign it a starting balance of 0.

The `<suite>` of a `class` statement contains `def` statements that define new methods for objects of that class. The method that initializes objects has a special name in Python, `__init__` (two underscores on each side of "init"), and is called the *constructor*for the class.

```
>>> class Account(object):
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
```

The `__init__` method for `Account` has two formal parameters. The first one, `self`, is bound to the newly created `Account`object. The second parameter, `account_holder`, is bound to the argument passed to the class when it is called to be instantiated.

The constructor binds the instance attribute name `balance` to `0`. It also binds the attribute name `holder` to the value of the name`account_holder`. The formal parameter `account_holder` is a local name to the `__init__` method. On the other hand, the name `holder` that is bound via the final assignment statement persists, because it is stored as an attribute of `self` using dot notation.

Having defined the `Account` class, we can instantiate it.

```
>>> a = Account('Jim')
```

This "call" to the `Account` class creates a new object that is an instance of `Account`, then calls the constructor function`__init__` with two arguments: the newly created object and the string `'Jim'`. By convention, we use the parameter name `self`for the first argument of a constructor, because it is bound to the object being instantiated. This convention is adopted in virtually all Python code.

Now, we can access the object's `balance` and `holder` using dot notation.

```
>>> a.balance
0
>>> a.holder
'Jim'
```

Identity. Each new account instance has its own balance attribute, the value of which is independent of other objects of the same class.

```
>>> b = Account('Jack')
>>> b.balance = 200
>>> [acc.balance for acc in (a, b)]
[0, 200]
```

To enforce this separation, every object that is an instance of a user-defined class has a unique identity. Object identity is compared using the `is` and `is not` operators.

```
>>> a is a
True
>>> a is not b
True
```

Despite being constructed from identical calls, the objects bound to `a` and `b` are not the same. As usual, binding an object to a new name using assignment does not create a new object.

```
>>> c = a
>>> c is a
True
```

New objects that have user-defined classes are only created when a class (such as `Account`) is instantiated with call expression syntax.

Methods. Object methods are also defined by a `def` statement in the suite of a `class` statement. Below, `deposit` and`withdraw` are both defined as methods on objects of the `Account` class.

```
>>> class Account(object):
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

While method definitions do not differ from function definitions in how they are declared, method definitions do have a different effect. The function value that is created by a `def` statement within a `class` statement is bound to the declared name, but bound locally within the class as an attribute. That value is invoked as a method using dot notation from an instance of the class.

Each method definition again includes a special first parameter `self`, which is bound to the object on which the method is invoked. For example, let us say that `deposit` is invoked on a particular `Account` object and passed a single argument value: the amount deposited. The object itself is bound to `self`, while the argument is bound to `amount`. All

invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

To invoke these methods, we again use dot notation, as illustrated below.

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
>>> tom_account.withdraw(90)
10
>>> tom_account.withdraw(90)
'Insufficient funds'
>>> tom_account.holder
'Tom'
```

When a method is invoked via dot notation, the object itself (bound to `tom_account`, in this case) plays a dual role. First, it determines what the name `withdraw` means; `withdraw` is not a name in the environment, but instead a name that is local to the`Account` class. Second, it is bound to the first parameter `self` when the `withdraw` method is invoked. The details of the procedure for evaluating dot notation follow in the next section.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#defining-classes