2.6 Implementing Classes and Objects

When working in the object-oriented programming paradigm, we use the object metaphor to guide the organization of our programs. Most logic about how to represent and manipulate data is expressed within class declarations. In this section, we see that classes and objects can themselves be represented using just functions and dictionaries. The purpose of implementing an object system in this way is to illustrate that using the object metaphor does not require a special programming language. Programs can be object-oriented, even in programming languages that do not have a built-in object system.

In order to implement objects, we will abandon dot notation (which does require built-in language support), but create dispatch dictionaries that behave in much the same way as the elements of the built-in object system. We have already seen how to implement message-passing behavior through dispatch dictionaries. To implement an object system in full, we send messages between instances, classes, and base classes, all of which are dictionaries that contain attributes.

We will not implement the entire Python object system, which includes features that we have not covered in this text (e.g., meta-classes and static methods). We will focus instead on user-defined classes without multiple inheritance and without introspective behavior (such as returning the class of an instance). Our implementation is not meant to follow the precise specification of the Python type system. Instead, it is designed to implement the core functionality that enables the object metaphor.

2.6.1 Instances

We begin with instances. An instance has named attributes, such as the balance of an account, which can be set and retrieved. We implement an instance using a dispatch dictionary that responds to messages that "get" and "set" attribute values. Attributes themselves are stored in a local dictionary called `attributes`.

As we have seen previously in this chapter, dictionaries themselves are abstract data types. We implemented dictionaries with lists, we implemented lists with pairs, and we implemented pairs with functions. As we implement an object system in terms of dictionaries, keep in mind that we could just as well be implementing objects using functions alone.

To begin our implementation, we assume that we have a class implementation that can look up any names that are not part of the instance. We pass in a class to `make_instance` as the parameter `cls`.

```
>>> def make_instance(cls):
        """Return a new object instance, which is a dispatch dictionary."""
        def get_value(name):
            if name in attributes:
                return attributes[name]
            else:
                value = cls['get'](name)
                return bind_method(value, instance)
        def set_value(name, value):
            attributes[name] = value
        attributes = {}
        instance = {'get': get_value, 'set': set_value}
        return instance
```

The `instance` is a dispatch dictionary that responds to the messages `get` and `set`. The `set` message corresponds to attribute assignment in Python's object system: all assigned attributes are stored directly within the object's local attribute dictionary. In `get`, if `name` does not appear in the local `attributes` dictionary, then it is looked up in the class. If the `value` returned by `cls` is a function, it must be bound to the instance.

Bound method values. The `get_value` function in `make_instance` finds a named attribute in its class with `get`, then calls `bind_method`. Binding a method only applies to function values, and it creates a bound method value from a function value by inserting the instance as the first argument:

```
>>> def bind_method(value, instance):
        """Return a bound method if value is callable, or value otherwise."""
        if callable(value):
            def method(*args):
                return value(instance, *args)
            return method
        else:
            return value
```

When a method is called, the first parameter `self` will be bound to the value of `instance` by this definition.

2.6.2 Classes

A class is also an object, both in Python's object system and the system we are implementing here. For simplicity, we say that classes do not themselves have a class. (In Python, classes do have classes; almost all classes share the same class, called `type`.) A class can respond to `get` and `set` messages, as well as the `new` message:

```
>>> def make_class(attributes, base_class=None):
        """Return a new class, which is a dispatch dictionary."""
        def get_value(name):
            if name in attributes:
                return attributes[name]
            elif base_class is not None:
                return base_class['get'](name)
        def set_value(name, value):
            attributes[name] = value
        def new(*args):
            return init_instance(cls, *args)
        cls = {'get': get_value, 'set': set_value, 'new': new}
        return cls
```

Unlike an instance, the `get` function for classes does not query its class when an attribute is not found, but instead queries its `base_class`. No method binding is required for classes.

Initialization. The `new` function in `make_class` calls `init_instance`, which first makes a new instance, then invokes a method called `__init__`.

```
>>> def init_instance(cls, *args):
        """Return a new object with type cls, initialized with args."""
        instance = make_instance(cls)
        init = cls['get']('__init__')
        if init:
            init(instance, *args)
        return instance
```

This final function completes our object system. We now have instances, which `set` locally but fall back to their classes on `get`. After an instance looks up a name in its class, it binds itself to function values to create methods. Finally, classes can create `new` instances, and they apply their `__init__` constructor function immediately after instance creation.

In this object system, the only function that should be called by the user is `make_class`. All other functionality is enabled through message passing. Similarly, Python's object system is invoked via the `class` statement, and all of its other functionality is enabled through dot expressions and calls to classes.

2.6.3 Using Implemented Objects

We now return to use the bank account example from the previous section. Using our implemented object system, we will create an `Account` class, a `CheckingAccount` subclass, and an instance of each.

The `Account` class is created through a `make_account_class` function, which has structure similar to a `class` statement in Python, but concludes with a call to `make_class`.

```
>>> def make_account_class():
        """Return the Account class, which has deposit and withdraw methods."""
        interest = 0.02
        def __init__(self, account_holder):
            self['set']('holder', account_holder)
            self['set']('balance', 0)
        def deposit(self, amount):
            """Increase the account balance by amount and return the new balance
            new_balance = self['get']('balance') + amount
            self['set']('balance', new_balance)
            return self['get']('balance')
        def withdraw(self, amount):
            """Decrease the account balance by amount and return the new balance
            balance = self['get']('balance')
            if amount > balance:
                return 'Insufficient funds'
            self['set']('balance', balance - amount)
            return self['get']('balance')
        return make_class(locals())
```

The final call to `locals` returns a dictionary with string keys that contains the name-value bindings in the current local frame.

The `Account` class is finally instantiated via assignment.

```
>>> Account = make_account_class()
```

Then, an account instance is created via the `new` message, which requires a name to go with the newly created account.

```
>>> kirk_account = Account['new']('Kirk')
```

Then, `get` messages passed to `kirk_account` retrieve properties and methods. Methods can be called to update the balance of the account.

```
>>> kirk_account['get']('holder')
'Kirk'
>>> kirk_account['get']('interest')
0.02
>>> kirk_account['get']('deposit')(20)
20
>>> kirk_account['get']('withdraw')(5)
15
```

As with the Python object system, setting an attribute of an instance does not change the corresponding attribute of its class.

```
>>> kirk_account['set']('interest', 0.04)
>>> Account['get']('interest')
0.02
```

Inheritance. We can create a subclass `CheckingAccount` by overloading a subset of the class attributes. In this case, we change the `withdraw` method to impose a fee, and we reduce the interest rate.

```
>>> def make_checking_account_class():
        """Return the CheckingAccount class, which imposes a $1 withdrawal fee.'
        interest = 0.01
        withdraw_fee = 1
        def withdraw(self, amount):
            fee = self['get']('withdraw_fee')
            return Account['get']('withdraw')(self, amount + fee)
        return make_class(locals(), Account)
```

In this implementation, we call the `withdraw` function of the base class `Account` from the `withdraw` function of the subclass, as we would in Python's built-in object system. We can create the subclass itself and an instance, as before.

```
>>> CheckingAccount = make_checking_account_class()
>>> jack_acct = CheckingAccount['new']('Spock')
```

Deposits behave identically, as does the constructor function. withdrawals impose the $1 fee from the specialized `withdraw` method, and `interest` has the new lower value from `CheckingAccount`.

```
>>> jack_acct['get']('interest')
0.01
>>> jack_acct['get']('deposit')(20)
20
>>> jack_acct['get']('withdraw')(5)
14
```

Our object system built upon dictionaries is quite similar in implementation to the built-in object system in Python. In Python, an instance of any user-defined class has a special attribute `__dict__` that stores the local instance attributes for that object in a dictionary, much like our `attributes` dictionary. Python differs because it distinguishes certain special methods that interact with built-in functions to ensure that those functions behave correctly for arguments of many different types. Functions that operate on different types are the subject of the next section.

Continue: 2.7 Object Abstraction