2.3 Class Definitions

A class serves as the primary means for abstraction in object-oriented programming. In Python, every piece of data is represented as an instance of some class. A class provides a set of behaviors in the form of ***member functions*** (also known as ***methods***), with implementations that are common to all instances of that class. A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of ***attributes*** (also known as ***fields***, ***instance variables***, or ***data members***).

2.3.1 Example: CreditCard Class

As a first example, we provide an implementation of a CreditCard class based on the design we introduced in Figure 2.3 of Section 2.2.1. The instances defined by the CreditCard class provide a simple model for traditional credit cards. They have identifying information about the customer, bank, account number, credit limit, and current balance. The class restricts charges that would cause a card's balance to go over its spending limit, but it does not charge interest or late payments (we revisit such themes in Section 2.4.1).

Our code begins in Code Fragment 2.1 and continues in Code Fragment 2.2. The construct begins with the keyword, **class**, followed by the name of the class, a colon, and then an indented block of code that serves as the body of the class. The body includes definitions for all methods of the class. These methods are defined as functions, using techniques introduced in Section 1.5, yet with a special parameter, named **self**, that serves to identify the particular instance upon which a member is invoked.

The self Identifier

In Python, the self identifier plays a key role. In the context of the CreditCard class, there can presumably be many different CreditCard instances, and each must maintain its own balance, its own credit limit, and so on. Therefore, each instance stores its own instance variables to reflect its current state.

Syntactically, self identifies the instance upon which a method is invoked. For example, assume that a user of our class has a variable, my_card, that identifies an instance of the CreditCard class. When the user calls my_card.get_balance( ), identifier self, within the definition of the get_balance method, refers to the card known as my_card by the caller. The expression, self._balance refers to an instance variable, named _balance, stored as part of that particular credit card's state.

```
1  class CreditCard:
2    """A consumer credit card."""
3
4    def __init__(self, customer, bank, acnt, limit):
5      """Create a new credit card instance.
6
7      The initial balance is zero.
8
9      customer  the name of the customer (e.g., 'John Bowman')
10     bank      the name of the bank (e.g., 'California Savings')
11     acnt      the acount identifier (e.g., '5391 0375 9387 5309')
12     limit     credit limit (measured in dollars)
13     """
14     self._customer = customer
15     self._bank = bank
16     self._account = acnt
17     self._limit = limit
18     self._balance = 0
19
20   def get_customer(self):
21     """Return name of the customer."""
22     return self._customer
23
24   def get_bank(self):
25     """Return the bank's name."""
26     return self._bank
27
28   def get_account(self):
29     """Return the card identifying number (typically stored as a string)."""
30     return self._account
31
32   def get_limit(self):
33     """Return current credit limit."""
34     return self._limit
35
36   def get_balance(self):
37     """Return current balance."""
38     return self._balance
```

Code Fragment 2.1: The beginning of the CreditCard class definition (continued in Code Fragment 2.2).

```
39    def charge(self, price):
40      """Charge given price to the card, assuming sufficient credit limit.
41
42      Return True if charge was processed; False if charge was denied.
43      """
44      if price + self._balance > self._limit:      # if charge would exceed limit,
45        return False                               # cannot accept charge
46      else:
47        self._balance += price
48        return True
49
50    def make_payment(self, amount):
51      """Process customer payment that reduces balance."""
52      self._balance -= amount
```

Code Fragment 2.2: The conclusion of the CreditCard class definition (continued from Code Fragment 2.1). These methods are indented within the class definition.

We draw attention to the difference between the method signature as declared within the class versus that used by a caller. For example, from a user's perspective we have seen that the get_balance method takes zero parameters, yet within the class definition, self is an explicit parameter. Likewise, the charge method is declared within the class having two parameters (self and price), even though this method is called with one parameter, for example, as my_card.charge(200). The interpretter automatically binds the instance upon which the method is invoked to the self parameter.

The Constructor

A user can create an instance of the CreditCard class using a syntax as:

```
cc = CreditCard('John Doe, '1st Bank', '5391 0375 9387 5309', 1000)
```

Internally, this results in a call to the specially named __init__ method that serves as the ***constructor*** of the class. Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables. In the case of the CreditCard class, each object maintains five instance variables, which we name: _customer, _bank, _account, _limit, and _balance. The initial values for the first four of those five are provided as explicit parameters that are sent by the user when instantiating the credit card, and assigned within the body of the constructor. For example, the command, self._customer = customer, assigns the instance variable self._customer to the parameter customer; note that because customer is ***unqualified*** on the right-hand side, it refers to the parameter in the local namespace.

Encapsulation

By the conventions described in Section 2.2.3, a single leading underscore in the name of a data member, such as _balance, implies that it is intended as ***nonpublic***. Users of a class should not directly access such members.

As a general rule, we will treat all data members as nonpublic. This allows us to better enforce a consistent state for all instances. We can provide accessors, such as get_balance, to provide a user of our class read-only access to a trait. If we wish to allow the user to change the state, we can provide appropriate update methods. In the context of data structures, encapsulating the internal representation allows us greater flexibility to redesign the way a class works, perhaps to improve the efficiency of the structure.

Additional Methods

The most interesting behaviors in our class are charge and make_payment. The charge function typically adds the given price to the credit card balance, to reflect a purchase of said price by the customer. However, before accepting the charge, our implementation verifies that the new purchase would not cause the balance to exceed the credit limit. The make_payment charge reflects the customer sending payment to the bank for the given amount, thereby reducing the balance on the card. We note that in the command, self._balance $-=$ amount, the expression self._balance is qualified with the self identifier because it represents an instance variable of the card, while the unqualified amount represents the local parameter.

Error Checking

Our implementation of the CreditCard class is not particularly robust. First, we note that we did not explicitly check the types of the parameters to charge and make_payment, nor any of the parameters to the constructor. If a user were to make a call such as visa.charge('candy'), our code would presumably crash when attempting to add that parameter to the current balance. If this class were to be widely used in a library, we might use more rigorous techniques to raise a TypeError when facing such misuse (see Section 1.7).

Beyond the obvious type errors, our implementation may be susceptible to logical errors. For example, if a user were allowed to charge a negative price, such as visa.charge(-300), that would serve to *lower* the customer's balance. This provides a loophole for lowering a balance without making a payment. Of course, this might be considered valid usage if modeling the credit received when a customer returns merchandise to a store. We will explore some such issues with the CreditCard class in the end-of-chapter exercises.

Testing the Class

In Code Fragment 2.3, we demonstrate some basic usage of the CreditCard class, inserting three cards into a list named wallet. We use loops to make some charges and payments, and use various accessors to print results to the console.

These tests are enclosed within a conditional, **if** `__name__ == '__main__'`:, so that they can be embedded in the source code with the class definition. Using the terminology of Section 2.2.4, these tests provide ***method coverage***, as each of the methods is called at least once, but it does not provide ***statement coverage***, as there is never a case in which a charge is rejected due to the credit limit. This is not a particular advanced from of testing as the output of the given tests must be manually audited in order to determine whether the class behaved as expected. Python has tools for more formal testing (see discussion of the unittest module in Section 2.2.4), so that resulting values can be automatically compared to the predicted outcomes, with output generated only when an error is detected.

```
53 if __name__ == '__main__':
54   wallet = [ ]
55   wallet.append(CreditCard('John Bowman', 'California Savings',
56                            '5391 0375 9387 5309', 2500) )
57   wallet.append(CreditCard('John Bowman', 'California Federal',
58                            '3485 0399 3395 1954', 3500) )
59   wallet.append(CreditCard('John Bowman', 'California Finance',
60                            '5391 0375 9387 5309', 5000) )
61
62   for val in range(1, 17):
63     wallet[0].charge(val)
64     wallet[1].charge(2*val)
65     wallet[2].charge(3*val)
66
67   for c in range(3):
68     print('Customer =', wallet[c].get_customer( ))
69     print('Bank =', wallet[c].get_bank( ))
70     print('Account =', wallet[c].get_account( ))
71     print('Limit =', wallet[c].get_limit( ))
72     print('Balance =', wallet[c].get_balance( ))
73     while wallet[c].get_balance( ) > 100:
74       wallet[c].make_payment(100)
75       print('New balance =', wallet[c].get_balance( ))
76     print( )
```

Code Fragment 2.3: Testing the CreditCard class.

2.3.2 Operator Overloading and Python's Special Methods

Python's built-in classes provide natural semantics for many operators. For example, the syntax a + b invokes addition for numeric types, yet concatenation for sequence types. When defining a new class, we must consider whether a syntax like a + b should be defined when a or b is an instance of that class.

By default, the + operator is undefined for a new class. However, the author of a class may provide a definition using a technique known as ***operator overloading***. This is done by implementing a specially named method. In particular, the + operator is overloaded by implementing a method named __add__, which takes the right-hand operand as a parameter and which returns the result of the expression. That is, the syntax, a + b, is converted to a method call on object a of the form, a.__add__(b). Similar specially named methods exist for other operators. Table 2.1 provides a comprehensive list of such methods.

When a binary operator is applied to two instances of different types, as in 3 * 'love me', Python gives deference to the class of the *left* operand. In this example, it would effectively check if the int class provides a sufficient definition for how to multiply an instance by a string, via the __mul__ method. However, if that class does not implement such a behavior, Python checks the class definition for the right-hand operand, in the form of a special method named __rmul__ (i.e., "right multiply"). This provides a way for a new user-defined class to support mixed operations that involve an instance of an existing class (given that the existing class would presumably not have defined a behavior involving this new class). The distinction between __mul__ and __rmul__ also allows a class to define different semantics in cases, such as matrix multiplication, in which an operation is noncommutative (that is, A * x may differ from x * A).

Non-Operator Overloads

In addition to traditional operator overloading, Python relies on specially named methods to control the behavior of various other functionality, when applied to user-defined classes. For example, the syntax, str(foo), is formally a call to the constructor for the string class. Of course, if the parameter is an instance of a user-defined class, the original authors of the string class could not have known how that instance should be portrayed. So the string constructor calls a specially named method, foo.__str__(), that must return an appropriate string representation.

Similar special methods are used to determine how to construct an int, float, or bool based on a parameter from a user-defined class. The conversion to a Boolean value is particularly important, because the syntax, **if** foo:, can be used even when foo is not formally a Boolean value (see Section 1.4.1). For a user-defined class, that condition is evaluated by the special method foo.__bool__().

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b	a.__iadd__(b)	
a −= b	a.__isub__(b)	
a *= b	a.__imul__(b)	
...	...	
+a	a.__pos__()	
−a	a.__neg__()	
~a	a.__invert__()	
abs(a)	a.__abs__()	
a < b	a.__lt__(b)	
a <= b	a.__le__(b)	
a > b	a.__gt__(b)	
a >= b	a.__ge__(b)	
a == b	a.__eq__(b)	
a != b	a.__ne__(b)	
v in a	a.__contains__(v)	
a[k]	a.__getitem__(k)	
a[k] = v	a.__setitem__(k,v)	
del a[k]	a.__delitem__(k)	
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)	
len(a)	a.__len__()	
hash(a)	a.__hash__()	
iter(a)	a.__iter__()	
next(a)	a.__next__()	
bool(a)	a.__bool__()	
float(a)	a.__float__()	
int(a)	a.__int__()	
repr(a)	a.__repr__()	
reversed(a)	a.__reversed__()	
str(a)	a.__str__()	

Table 2.1: Overloaded operations, implemented with Python's special methods.

Several other top-level functions rely on calling specially named methods. For example, the standard way to determine the size of a container type is by calling the top-level len function. Note well that the calling syntax, len(foo), is not the traditional method-calling syntax with the dot operator. However, in the case of a user-defined class, the top-level len function relies on a call to a specially named __len__ method of that class. That is, the call len(foo) is evaluated through a method call, foo.__len__(). When developing data structures, we will routinely define the __len__ method to return a measure of the size of the structure.

Implied Methods

As a general rule, if a particular special method is not implemented in a user-defined class, the standard syntax that relies upon that method will raise an exception. For example, evaluating the expression, a + b, for instances of a user-defined class without __add__ or __radd__ will raise an error.

However, there are some operators that have default definitions provided by Python, in the absence of special methods, and there are some operators whose definitions are derived from others. For example, the __bool__ method, which supports the syntax **if** foo:, has default semantics so that every object other than None is evaluated as True. However, for container types, the __len__ method is typically defined to return the size of the container. If such a method exists, then the evaluation of bool(foo) is interpreted by default to be True for instances with nonzero length, and False for instances with zero length, allowing a syntax such as **if** waitlist: to be used to test whether there are one or more entries in the waitlist.

In Section 2.3.4, we will discuss Python's mechanism for providing iterators for collections via the special method, __iter__. With that said, if a container class provides implementations for both __len__ and __getitem__, a default iteration is provided automatically (using means we describe in Section 2.3.4). Furthermore, once an iterator is defined, default functionality of __contains__ is provided.

In Section 1.3 we drew attention to the distinction between expression a **is** b and expression a == b, with the former evaluating whether identifiers a and b are aliases for the same object, and the latter testing a notion of whether the two identifiers reference *equivalent* values. The notion of "equivalence" depends upon the context of the class, and semantics is defined with the __eq__ method. However, if no implementation is given for __eq__, the syntax a == b is legal with semantics of a **is** b, that is, an instance is equivalent to itself and no others.

We should caution that some natural implications are *not* automatically provided by Python. For example, the __eq__ method supports syntax a == b, but providing that method does not affect the evaluation of syntax a != b. (The __ne__ method should be provided, typically returning **not** (a == b) as a result.) Similarly, providing a __lt__ method supports syntax a < b, and indirectly b > a, but providing both __lt__ and __eq__ does *not* imply semantics for a <= b.

2.3.3 Example: Multidimensional Vector Class

To demonstrate the use of operator overloading via special methods, we provide an implementation of a Vector class, representing the coordinates of a vector in a multidimensional space. For example, in a three-dimensional space, we might wish to represent a vector with coordinates $\langle 5, -2, 3 \rangle$. Although it might be tempting to directly use a Python list to represent those coordinates, a list does not provide an appropriate abstraction for a geometric vector. In particular, if using lists, the expression [5, −2, 3] + [1, 4, 2] results in the list [5, −2, 3, 1, 4, 2]. When working with vectors, if $u = \langle 5, -2, 3 \rangle$ and $v = \langle 1, 4, 2 \rangle$, one would expect the expression, u + v, to return a three-dimensional vector with coordinates $\langle 6, 2, 5 \rangle$.

We therefore define a Vector class, in Code Fragment 2.4, that provides a better abstraction for the notion of a geometric vector. Internally, our vector relies upon an instance of a list, named _coords, as its storage mechanism. By keeping the internal list encapsulated, we can enforce the desired public interface for instances of our class. A demonstration of supported behaviors includes the following:

```
v = Vector(5)                  # construct five-dimensional <0, 0, 0, 0, 0>
v[1] = 23                      # <0, 23, 0, 0, 0> (based on use of __setitem__)
v[-1] = 45                     # <0, 23, 0, 0, 45> (also via __setitem__)
print(v[4])                    # print 45 (via __getitem__)
u = v + v                      # <0, 46, 0, 0, 90> (via __add__)
print(u)                       # print <0, 46, 0, 0, 90>
total = 0
for entry in v:                # implicit iteration via __len__ and __getitem__
  total += entry
```

We implement many of the behaviors by trivially invoking a similar behavior on the underlying list of coordinates. However, our implementation of __add__ is customized. Assuming the two operands are vectors with the same length, this method creates a new vector and sets the coordinates of the new vector to be equal to the respective sum of the operands' elements.

It is interesting to note that the class definition, as given in Code Fragment 2.4, automatically supports the syntax u = v + [5, 3, 10, −2, 1], resulting in a new vector that is the element-by-element "sum" of the first vector and the list instance. This is a result of Python's ***polymorphism***. Literally, "polymorphism" means "many forms." Although it is tempting to think of the other parameter of our __add__ method as another Vector instance, we never declared it as such. Within the body, the only behaviors we rely on for parameter other is that it supports len(other) and access to other[j]. Therefore, our code executes when the right-hand operand is a list of numbers (with matching length).

```
1  class Vector:
2    """Represent a vector in a multidimensional space."""
3
4    def __init__(self, d):
5      """Create d-dimensional vector of zeros."""
6      self._coords = [0] * d
7
8    def __len__(self):
9      """Return the dimension of the vector."""
10     return len(self._coords)
11
12   def __getitem__(self, j):
13     """Return jth coordinate of vector."""
14     return self._coords[j]
15
16   def __setitem__(self, j, val):
17     """Set jth coordinate of vector to given value."""
18     self._coords[j] = val
19
20   def __add__(self, other):
21     """Return sum of two vectors."""
22     if len(self) != len(other):              # relies on __len__ method
23       raise ValueError('dimensions must agree')
24     result = Vector(len(self))               # start with vector of zeros
25     for j in range(len(self)):
26       result[j] = self[j] + other[j]
27     return result
28
29   def __eq__(self, other):
30     """Return True if vector has same coordinates as other."""
31     return self._coords == other._coords
32
33   def __ne__(self, other):
34     """Return True if vector differs from other."""
35     return not self == other                 # rely on existing __eq__ definition
36
37   def __str__(self):
38     """Produce string representation of vector."""
39     return '<' + str(self._coords)[1:-1] + '>'   # adapt list representation
```

Code Fragment 2.4: Definition of a simple Vector class.