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 ˆ 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).

```
class Vector:
  """Represent a vector in a multidimensional space."""

  def __init__(self, d):
    """Create d-dimensional vector of zeros."""
    self._coords = [0] * d

  def __len__(self):
    """Return the dimension of the vector."""
    return len(self._coords)

  def __getitem__(self, j):
    """Return jth coordinate of vector."""
    return self._coords[j]

  def __setitem__(self, j, val):
    """Set jth coordinate of vector to given value."""
    self._coords[j] = val

  def __add__(self, other):
    """Return sum of two vectors."""
    if len(self) != len(other):                 # relies on __len__ method
      raise ValueError('dimensions must agree')
    result = Vector(len(self))                  # start with vector of zeros
    for j in range(len(self)):
      result[j] = self[j] + other[j]
    return result

  def __eq__(self, other):
    """Return True if vector has same coordinates as other."""
    return self._coords == other._coords

  def __ne__(self, other):
    """Return True if vector differs from other."""
    return not self == other                    # rely on existing __eq__ definition

  def __str__(self):
    """Produce string representation of vector."""
    return '<' + str(self._coords)[1:-1] + '>'  # adapt list representation
```

Code Fragment 2.4: Definition of a simple Vector class.

2.3.4 Iterators

Iteration is an important concept in the design of data structures. We introduced Python's mechanism for iteration in Section 1.8. In short, an ***iterator*** for a collection provides one key behavior: It supports a special method named __next__ that returns the next element of the collection, if any, or raises a StopIteration exception to indicate that there are no further elements.

Fortunately, it is rare to have to directly implement an iterator class. Our preferred approach is the use of the ***generator*** syntax (also described in Section 1.8), which automatically produces an iterator of yielded values.

Python also helps by providing an automatic iterator implementation for any class that defines both __len__ and __getitem__. To provide an instructive example of a low-level iterator, Code Fragment 2.5 demonstrates just such an iterator class that works on any collection that supports both __len__ and __getitem__. This class can be instantiated as SequenceIterator(data). It operates by keeping an internal reference to the data sequence, as well as a current index into the sequence. Each time __next__ is called, the index is incremented, until reaching the end of the sequence.

```
class SequenceIterator:
  """An iterator for any of Python's sequence types."""

  def __init__(self, sequence):
    """Create an iterator for the given sequence."""
    self._seq = sequence             # keep a reference to the underlying data
    self._k = −1                     # will increment to 0 on first call to next

  def __next__(self):
    """Return the next element, or else raise StopIteration error."""
    self._k += 1                     # advance to next index
    if self._k < len(self._seq):
      return(self._seq[self._k])     # return the data element
    else:
      raise StopIteration( )         # there are no more elements

  def __iter__(self):
    """By convention, an iterator must return itself as an iterator."""
    return self
```

Code Fragment 2.5: An iterator class for any sequence type.

2.3.5 Example: Range Class

As the final example for this section, we develop our own implementation of a class that mimics Python's built-in range class. Before introducing our class, we discuss the history of the built-in version. Prior to Python 3 being released, range was implemented as a function, and it returned a list instance with elements in the specified range. For example, range(2, 10, 2) returned the list [2, 4, 6, 8]. However, a typical use of the function was to support a for-loop syntax, such as **for** k **in** range(10000000). Unfortunately, this caused the instantiation and initialization of a list with the range of numbers. That was an unnecessarily expensive step, in terms of both time and memory usage.

The mechanism used to support ranges in Python 3 is entirely different (to be fair, the "new" behavior existed in Python 2 under the name xrange). It uses a strategy known as ***lazy evaluation***. Rather than creating a new list instance, range is a class that can effectively represent the desired range of elements without ever storing them explicitly in memory. To better explore the built-in range class, we recommend that you create an instance as r = range(8, 140, 5). The result is a relatively lightweight object, an instance of the range class, that has only a few behaviors. The syntax len(r) will report the number of elements that are in the given range (27, in our example). A range also supports the __getitem__ method, so that syntax r[15] reports the sixteenth element in the range (as r[0] is the first element). Because the class supports both __len__ and __getitem__, it inherits automatic support for iteration (see Section 2.3.4), which is why it is possible to execute a for loop over a range.

At this point, we are ready to demonstrate our own version of such a class. Code Fragment 2.6 provides a class we name Range (so as to clearly differentiate it from built-in range). The biggest challenge in the implementation is properly computing the number of elements that belong in the range, given the parameters sent by the caller when constructing a range. By computing that value in the constructor, and storing it as self._length, it becomes trivial to return it from the __len__ method. To properly implement a call to __getitem__(k), we simply take the starting value of the range plus k times the step size (i.e., for k=0, we return the start value). There are a few subtleties worth examining in the code:

- To properly support optional parameters, we rely on the technique described on page 27, when discussing a functional version of range.
- We compute the number of elements in the range as
  max(0, (stop − start + step − 1) // step)
  It is worth testing this formula for both positive and negative step sizes.
- The __getitem__ method properly supports negative indices by converting an index −k to len(self)−k before computing the result.

```
class Range:
  """A class that mimic's the built-in range class."""

  def __init__(self, start, stop=None, step=1):
    """Initialize a Range instance.

    Semantics is similar to built-in range class.
    """
    if step == 0:
      raise ValueError('step cannot be 0')

    if stop is None:                    # special case of range(n)
      start, stop = 0, start            # should be treated as if range(0,n)

    # calculate the effective length once
    self._length = max(0, (stop − start + step − 1) // step)

    # need knowledge of start and step (but not stop) to support __getitem__
    self._start = start
    self._step = step

  def __len__(self):
    """Return number of entries in the range."""
    return self._length

  def __getitem__(self, k):
    """Return entry at index k (using standard interpretation if negative)."""
    if k < 0:
      k += len(self)                    # attempt to convert negative index

    if not 0 <= k < self._length:
      raise IndexError('index out of range')

    return self._start + k * self._step
```

Code Fragment 2.6: Our own implementation of a Range class.