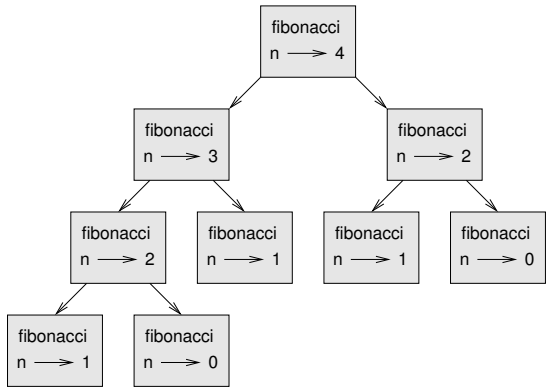
To understand why, consider this **call graph** for `fibonacci` with `n=4`:



A call graph shows a set function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.

A good solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **hint**. Here is an implementation of `fibonacci` using hints:

```
previous = {0:1, 1:1}

def fibonacci(n):
  if previous.has_key(n):
    return previous[n]
  else:
    newValue = fibonacci(n-1) + fibonacci(n-2)
    previous[n] = newValue
    return newValue
```

The dictionary named `previous` keeps track of the Fibonacci numbers we already know. We start with only two pairs: 0 maps to 1; and 1 maps to 1.

Whenever `fibonacci` is called, it checks the dictionary to determine if it contains the result. If it's there, the function can return immediately without making any more recursive calls. If not, it has to compute the new value. The new value is added to the dictionary before the function returns.

Using this version of `fibonacci`, our machines can compute `fibonacci(40)` in an eyeblink. But when we try to compute `fibonacci(50)`, we see the following:

```
>>> fibonacci(50)
20365011074L
```

The `L` at the end of the result indicates that the answer +(20,365,011,074) is too big to fit into a Python integer. Python has automatically converted the result to a long integer.

10.6 Long integers

Python provides a type called `long` that can handle any size integer. There are two ways to create a `long` value. One is to write an integer with a capital `L` at the end:

```
>>> type(1L)
<type 'long'>
```

The other is to use the `long` function to convert a value to a `long`. `long` can accept any numerical type and even strings of digits:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

All of the math operations work on `long`s, so in general any code that works with integers will also work with long integers. Any time the result of a computation is too big to be represented with an integer, Python detects the overflow and returns the result as a long integer. For example:

```
>>> 1000 * 1000
1000000
>>> 100000 * 100000
10000000000L
```

In the first case the result has type `int`; in the second case it is `long`.

10.7 Counting letters

In Chapter 7, we wrote a function that counted the number of occurrences of a letter in a string. A more general version of this problem is to form a histogram

of the letters in the string, that is, how many times each letter appears.

Such a histogram might be useful for compressing a text file. Because different letters appear with different frequencies, we can compress a file by using shorter codes for common letters and longer codes for letters that appear less frequently.

Dictionaries provide an elegant way to generate a histogram:

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...   letterCounts[letter] = letterCounts.get (letter, 0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

We start with an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.

It might be more appealing to display the histogram in alphabetical order. We can do that with the `items` and `sort` methods:

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

You have seen the `items` method before, but `sort` is the first method you have encountered that applies to lists. There are several other list methods, including `append`, `extend`, and `reverse`. Consult the Python documentation for details.

10.8 Glossary

dictionary: A collection of key-value pairs that maps from keys to values. The keys can be any immutable type, and the values can be any type.

key: A value that is used to look up an entry in a dictionary.

key-value pair: One of the items in a dictionary.

method: A kind of function that is called with a different syntax and invoked "on" an object.

invoke: To call a method.

hint: Temporary storage of a precomputed value to avoid redundant computation.

overflow: A numerical result that is too large to be represented in a numerical format.

Chapter 11

Files and exceptions

While a program is running, its data is in memory. When the program ends, or the computer shuts down, data in memory disappears. To store data permanently, you have to put it in a **file**. Files are usually stored on a hard drive, floppy drive, or CD-ROM.

When there are a large number of files, they are often organized into **directories** (also called "folders"). Each file is identified by a unique name, or a combination of a file name and a directory name.

By reading and writing files, programs can exchange information with each other and generate printable formats like PDF.

Working with files is a lot like working with books. To use a book, you have to open it. When you're done, you have to close it. While the book is open, you can either write in it or read from it. In either case, you know where you are in the book. Most of the time, you read the whole book in its natural order, but you can also skip around.

All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.

Opening a file creates a file object. In this example, the variable `f` refers to the new file object.

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

The open function takes two arguments. The first is the name of the file, and the second is the mode. Mode `"w"` means that we are opening the file for writing.

If there is no file named `test.dat`, it will be created. If there already is one, it will be replaced by the file we are writing.

When we print the file object, we see the name of the file, the mode, and the location of the object.

To put data in the file we invoke the `write` method on the file object:

```
>>> f.write("Now is the time")
>>> f.write("to close the file")
```

Closing the file tells the system that we are done writing and makes the file available for reading:

```
>>> f.close()
```

Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is `"r"` for reading:

```
>>> f = open("test.dat","r")
```

If we try to open a file that doesn't exist, we get an error:

```
>>> f = open("test.cat","r")
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Not surprisingly, the `read` method reads data from the file. With no arguments, it reads the entire contents of the file:

```
>>> text = f.read()
>>> print text
Now is the timeto close the file
```

There is no space between "time" and "to" because we did not write a space between the strings.

`read` can also take an argument that indicates how many characters to read:

```
>>> f = open("test.dat","r")
>>> print f.read(5)
Now i
```

If not enough characters are left in the file, `read` returns the remaining characters. When we get to the end of the file, `read` returns the empty string:

```
>>> print f.read(1000006)
s the timeto close the file
>>> print f.read()

>>>
```

The following function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

```
def copyFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.read(50)
    if text == "":
      break
    f2.write(text)
  f1.close()
  f2.close()
  return
```

The `break` statement is new. Executing it breaks out of the loop; the flow of execution moves to the first statement after the loop.

In this example, the `while` loop is infinite because the value `True` is always true. The *only* way to get out of the loop is to execute `break`, which happens when `text` is the empty string, which happens when we get to the end of the file.

11.1 Text files

A **text file** is a file that contains printable characters and whitespace, organized into lines separated by newline characters. Since Python is specifically designed to process text files, it provides methods that make the job easy.

To demonstrate, we'll create a text file with three lines of text separated by newlines:

```
>>> f = open("test.dat","w")
>>> f.write("line one\nline two\nline three\n")
>>> f.close()
```

The `readline` method reads all the characters up to and including the next newline character:

```
>>> f = open("test.dat","r")
>>> print f.readline()
line one

>>>
```

`readlines` returns all of the remaining lines as a list of strings:

```
>>> print f.readlines()
['line two\012', 'line three\012']
```

In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence `012`.

At the end of the file, `readline` returns the empty string and `readlines` returns the empty list:

```
>>> print f.readline()

>>> print f.readlines()
[]
```

The following is an example of a line-processing program. `filterFile` makes a copy of `oldFile`, omitting any lines that begin with `#`:

```
def filterFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.readline()
    if text == "":
      break
    if text[0] == '#':
      continue
    f2.write(text)
  f1.close()
  f2.close()
  return
```

The `continue` statement ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the condition, and proceeds accordingly.

Thus, if `text` is the empty string, the loop exits. If the first character of `text` is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy `text` into the new file.

11.2 Writing variables

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings first. The easiest way to do that is with

the `str` function:

```
>>> x = 52
>>> f.write (str(x))
```

An alternative is to use the **format operator** `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, and the second operand is a tuple of expressions. The result is a string that contains the values of the expressions, formatted according to the format string.

As a simple example, the **format sequence** `"%d"` means that the first expression in the tuple should be formatted as an integer. Here the letter *d* stands for "decimal":

```
>>> cars = 52
>>> "%d" % cars
'52'
```

The result is the string `'52'`, which is not to be confused with the integer value `52`.

A format sequence can appear anywhere in the format string, so we can embed a value in a sentence:

```
>>> cars = 52
>>> "In July we sold %d cars." % cars
'In July we sold 52 cars.'
```

The format sequence `"%f"` formats the next item in the tuple as a floating-point number, and `"%s"` formats the next item as a string:

```
>>> "In %d days we made %f million %s." % (34,6.1,'dollars')
'In 34 days we made 6.100000 million dollars.'
```

By default, the floating-point format prints six decimal places.

The number of expressions in the tuple has to match the number of format sequences in the string. Also, the types of the expressions have to match the format sequences:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dollars'
TypeError: illegal argument type for built-in operation
```

In the first example, there aren't enough expressions; in the second, the expression is the wrong type.

For more control over the format of numbers, we can specify the number of digits as part of the format sequence:

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6.100000'
```

The number after the percent sign is the minimum number of spaces the number will take up. If the value provided takes fewer digits, leading spaces are added. If the number of spaces is negative, trailing spaces are added:

```
>>> "%-6d" % 62
'62    '
```

For floating-point numbers, we can also specify the number of digits after the decimal point:

```
>>> "%12.2f" % 6.1
'        6.10'
```

In this example, the result takes up twelve spaces and includes two digits after the decimal. This format is useful for printing dollar amounts with the decimal points aligned.

For example, imagine a dictionary that contains student names as keys and hourly wages as values. Here is a function that prints the contents of the dictionary as a formatted report:

```
def report (wages) :
  students = wages.keys()
  students.sort()
  for student in students :
    print "%-20s %12.2f" % (student, wages[student])
```

To test this function, we'll create a small dictionary and print the contents:

```
>>> wages = {'mary': 6.23, 'joe': 5.45, 'joshua': 4.25}
>>> report (wages)
joe                          5.45
joshua                       4.25
mary                         6.23
```

By controlling the width of each value, we guarantee that the columns will line up, as long as the names contain fewer than twenty-one characters and the wages are less than one billion dollars an hour.

11.3 Directories

When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Python looks for it in the current directory.

If you want to open a file somewhere else, you have to specify the **path** to the file, which is the name of the directory (or folder) where the file is located:

```
>>>   f = open("/usr/share/dict/words","r")
>>>   print f.readline()
Aarhus
```

This example opens a file named `words` that resides in a directory named `dict`, which resides in `share`, which resides in `usr`, which resides in the top-level directory of the system, called `/`.

You cannot use `/` as part of a filename; it is reserved as a delimiter between directory and filenames.

The file `/usr/share/dict/words` contains a list of words in alphabetical order, of which the first is the name of a Danish university.

11.4 Pickling

In order to put values into a file, you have to convert them to strings. You have already seen how to do that with `str`:

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

The problem is that when you read the value back, you get a string. The original type information has been lost. In fact, you can't even tell where one value ends and the next begins:

```
>>>   f.readline()
'12.3[1, 2, 3]'
```

The solution is **pickling**, so called because it "preserves" data structures. The `pickle` module contains the necessary commands. To use it, import `pickle` and then open the file in the usual way:

```
>>> import pickle
>>> f = open("test.pck","w")
```

To store a data structure, use the `dump` method and then close the file in the usual way:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Then we can open the file for reading and load the data structures we dumped:

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x
12.3
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'>
```

Each time we invoke `load`, we get a single value from the file, complete with its original type.

11.5 Exceptions

Whenever a runtime error occurs, it creates an **exception**. Usually, the program stops and Python prints an error message.

For example, dividing by zero creates an exception:

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

So does accessing a nonexistent list item:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

Or accessing a key that isn't in the dictionary:

```
>>> b = {}
>>> print b['what']
KeyError: what
```

Or trying to open a nonexistent file:

```
>>> f = open("Idontexist", "r")
IOError: [Errno 2] No such file or directory: 'Idontexist'
```

In each case, the error message has two parts: the type of error before the colon, and specifics about the error after the colon. Normally Python also prints a traceback of where the program was, but we have omitted that from the examples.

Sometimes we want to execute an operation that could cause an exception, but we don't want the program to stop. We can **handle** the exception using the `try` and `except` statements.

For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

```
filename = raw_input('Enter a file name: ')
try:
  f = open (filename, "r")
except IOError:
  print 'There is no file named', filename
```

The `try` statement executes the statements in the first block. If no exceptions occur, it ignores the `except` statement. If an exception of type `IOError` occurs, it executes the statements in the `except` branch and then continues.

We can encapsulate this capability in a function: `exists` takes a filename and returns true if the file exists, false if it doesn't:

```
def exists(filename):
  try:
    f = open(filename)
    f.close()
    return True
  except IOError:
    return False
```

You can use multiple `except` blocks to handle different kinds of exceptions. The *Python Reference Manual* has the details.

If your program detects an error condition, you can make it **raise** an exception. Here is an example that gets input from the user and checks for the value 17. Assuming that 17 is not valid input for some reason, we raise an exception.

```
def inputNumber () :
  x = input ('Pick a number: ')
```

```
  if x == 17 :
    raise ValueError, '17 is a bad number'
  return x
```

The `raise` statement takes two arguments: the exception type and specific information about the error. `ValueError` is one of the exception types Python provides for a variety of occasions. Other examples include `TypeError`, `KeyError`, and my favorite, `NotImplementedError`.

If the function that called `inputNumber` handles the error, then the program can continue; otherwise, Python prints the error message and exits:

```
>>> inputNumber ()
Pick a number: 17
ValueError: 17 is a bad number
```

The error message includes the exception type and the additional information you provided.

> *As an exercise, write a function that uses* `inputNumber` *to input a number from the keyboard and that handles the* `ValueError` *exception.*

11.6 Glossary

file: A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

directory: A named collection of files, also called a folder.

path: A sequence of directory names that specifies the exact location of a file.

text file: A file that contains printable characters organized into lines separated by newline characters.

break statement: A statement that causes the flow of execution to exit a loop.

continue statement: A statement that causes the current iteration of a loop to end. The flow of execution goes to the top of the loop, evaluates the condition, and proceeds accordingly.

format operator: The `%` operator takes a format string and a tuple of expressions and yields a string that includes the expressions, formatted according to the format string.

format string: A string that contains printable characters and format sequences that indicate how to format values.

format sequence: A sequence of characters beginning with `%` that indicates how to format a value.

pickle: To write a data value in a file along with its type information so that it can be reconstituted later.

exception: An error that occurs at runtime.

handle: To prevent an exception from terminating a program using the `try` and `except` statements.

raise: To signal an exception using the `raise` statement.

Chapter 12

Classes and objects

12.1 User-defined compound types

Having used some of Python's built-in types, we are ready to create a user-defined type: the `Point`.

Consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, $(0, 0)$ represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

A natural way to represent a point in Python is with two floating-point values. The question, then, is how to group these two values into a compound object. The quick and dirty solution is to use a list or tuple, and for some applications that might be the best choice.

An alternative is to define a new user-defined compound type, also called a **class**. This approach involves a bit more effort, but it has advantages that will be apparent soon.

A class definition looks like this:

```
class Point:
  pass
```

Class definitions can appear anywhere in a program, but they are usually near the beginning (after the `import` statements). The syntax rules for a class definition are the same as for other compound statements (see Section 4.4).

This definition creates a new class called `Point`. The **pass** statement has no effect; it is only necessary because a compound statement must have something in its body.

By creating the `Point` class, we created a new type, also called `Point`. The members of this type are called **instances** of the type or **objects**. Creating a new instance is called **instantiation**. To instantiate a `Point` object, we call a function named (you guessed it) `Point`:

```
blank = Point()
```

The variable `blank` is assigned a reference to a new `Point` object. A function like `Point` that creates new objects is called a **constructor**.

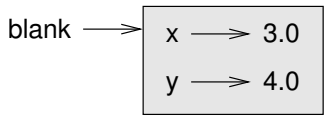
12.2 Attributes

We can add new data to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.uppercase`. In this case, though, we are selecting a data item from an instance. These named items are called **attributes**.

The following state diagram shows the result of these assignments:



The variable `blank` refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

We can read the value of an attribute using the same syntax:

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

The expression `blank.x` means, "Go to the object `blank` refers to and get the value of `x`." In this case, we assign that value to a variable named `x`. There is no

conflict between the variable `x` and the attribute `x`. The purpose of dot notation is to identify which variable you are referring to unambiguously.

You can use dot notation as part of any expression, so the following statements are legal:

```
print '(' + str(blank.x) + ', ' + str(blank.y) + ')'
distanceSquared = blank.x * blank.x + blank.y * blank.y
```

The first line outputs `(3.0, 4.0)`; the second line calculates the value 25.0.

You might be tempted to print the value of `blank` itself:

```
>>> print blank
<__main__.Point instance at 80f8e70>
```

The result indicates that `blank` is an instance of the `Point` class and it was defined in `__main__`. `80f8e70` is the unique identifier for this object, written in hexadecimal (base 16). This is probably not the most informative way to display a `Point` object. You will see how to change it shortly.

> *As an exercise, create and print a* `Point` *object, and then use* `id` *to print the object's unique identifier. Translate the hexadecimal form into decimal and confirm that they match.*

12.3 Instances as arguments

You can pass an instance as an argument in the usual way. For example:

```
def printPoint(p):
  print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

`printPoint` takes a point as an argument and displays it in the standard format. If you call `printPoint(blank)`, the output is `(3.0, 4.0)`.

> *As an exercise, rewrite the* `distance` *function from Section 5.2 so that it takes two* `Point`*s as arguments instead of four numbers.*

12.4 Sameness

The meaning of the word "same" seems perfectly clear until you give it some thought, and then you realize there is more to it than you expected.

For example, if you say, "Chris and I have the same car," you mean that his car and yours are the same make and model, but that they are two different cars.

If you say, "Chris and I have the same mother," you mean that his mother and yours are the same person.[1] So the idea of "sameness" is different depending on the context.

When you talk about objects, there is a similar ambiguity. For example, if two `Point`s are the same, does that mean they contain the same data (coordinates) or that they are actually the same object?

To find out if two references refer to the same object, use the `is` operator. For example:

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 is p2
False
```

Even though `p1` and `p2` contain the same coordinates, they are not the same object. If we assign `p1` to `p2`, then the two variables are aliases of the same object:

```
>>> p2 = p1
>>> p1 is p2
True
```

This type of equality is called **shallow equality** because it compares only the references, not the contents of the objects.

To compare the contents of the objects—**deep equality**—we can write a function called `samePoint`:

```
def samePoint(p1, p2) :
  return (p1.x == p2.x) and (p1.y == p2.y)
```

Now if we create two different objects that contain the same data, we can use `samePoint` to find out if they represent the same point.

[1] Not all languages have the same problem. For example, German has different words for different kinds of sameness. "Same car" in this context would be "gleiche Auto," and "same mother" would be "selbe Mutter."

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> samePoint(p1, p2)
True
```

Of course, if the two variables refer to the same object, they have both shallow and deep equality.

12.5 Rectangles

Let's say that we want a class to represent a rectangle. The question is, what information do we have to provide in order to specify a rectangle? To keep things simple, assume that the rectangle is oriented either vertically or horizontally, never at an angle.

There are a few possibilities: we could specify the center of the rectangle (two coordinates) and its size (width and height); or we could specify one of the corners and the size; or we could specify two opposing corners. A conventional choice is to specify the upper-left corner of the rectangle and the size.

Again, we'll define a new class:

```
class Rectangle:
  pass
```

And instantiate it:

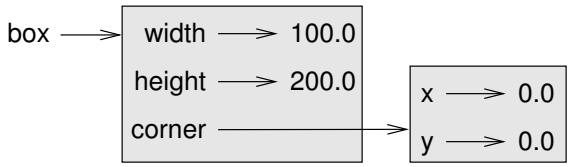
```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```

This code creates a new `Rectangle` object with two floating-point attributes. To specify the upper-left corner, we can embed an object within an object!

```
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The dot operator composes. The expression `box.corner.x` means, "Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`."

The figure shows the state of this object:



12.6 Instances as return values

Functions can return instances. For example, `findCenter` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
def findCenter(box):
  p = Point()
  p.x = box.corner.x + box.width/2.0
  p.y = box.corner.y - box.height/2.0
  return p
```

To call this function, pass `box` as an argument and assign the result to a variable:

```
>>> center = findCenter(box)
>>> printPoint(center)
(50.0, -100.0)
```

12.7 Objects are mutable

We can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, we could modify the values of `width` and `height`:

```
box.width = box.width + 50
box.height = box.height + 100
```

We could encapsulate this code in a method and generalize it to grow the rectangle by any amount:

```
def growRect(box, dwidth, dheight) :
  box.width = box.width + dwidth
  box.height = box.height + dheight
```

The variables dwidth and dheight indicate how much the rectangle should grow in each direction. Invoking this method has the effect of modifying the Rectangle that is passed as an argument.

For example, we could create a new Rectangle named bob and pass it to growRect:

```
>>> bob = Rectangle()
>>> bob.width = 100.0
>>> bob.height = 200.0
>>> bob.corner = Point()
>>> bob.corner.x = 0.0
>>> bob.corner.y = 0.0
>>> growRect(bob, 50, 100)
```

While growRect is running, the parameter box is an alias for bob. Any changes made to box also affect bob.

> *As an exercise, write a function named* moveRect *that takes a* Rectangle *and two parameters named* dx *and* dy*. It should change the location of the rectangle by adding* dx *to the* x *coordinate of* corner *and adding* dy *to the* y *coordinate of* corner*.*

12.8 Copying

Aliasing can make a program difficult to read because changes made in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

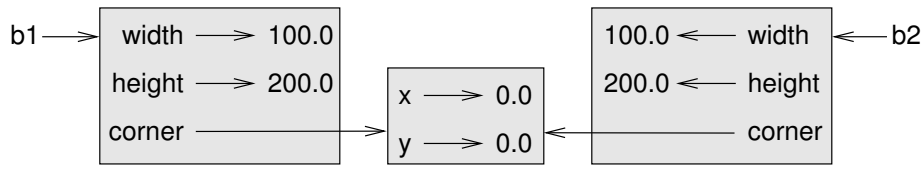
```
>>> import copy
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
False
>>> samePoint(p1, p2)
True
```

Once we import the copy module, we can use the copy method to make a new Point. p1 and p2 are not the same point, but they contain the same data.

To copy a simple object like a `Point`, which doesn't contain any embedded objects, `copy` is sufficient. This is called **shallow copying**.

For something like a `Rectangle`, which contains a reference to a `Point`, `copy` doesn't do quite the right thing. It copies the reference to the `Point` object, so both the old `Rectangle` and the new one refer to a single `Point`.

If we create a box, `b1`, in the usual way and then make a copy, `b2`, using `copy`, the resulting state diagram looks like this:



This is almost certainly not what we want. In this case, invoking `growRect` on one of the `Rectangles` would not affect the other, but invoking `moveRect` on either would affect both! This behavior is confusing and error-prone.

Fortunately, the `copy` module contains a method named `deepcopy` that copies not only the object but also any embedded objects. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> b2 = copy.deepcopy(b1)
```

Now `b1` and `b2` are completely separate objects.

We can use `deepcopy` to rewrite `growRect` so that instead of modifying an existing `Rectangle`, it creates a new `Rectangle` that has the same location as the old one but new dimensions:

```
def growRect(box, dwidth, dheight) :
  import copy
  newBox = copy.deepcopy(box)
  newBox.width = newBox.width + dwidth
  newBox.height = newBox.height + dheight
  return newBox
```

An an exercise, rewrite `moveRect` *so that it creates and returns a new* `Rectangle` *instead of modifying the old one.*

12.9 Glossary

class: A user-defined compound type. A class can also be thought of as a template for the objects that are instances of it.

instantiate: To create an instance of a class.

instance: An object that belongs to a class.

object: A compound data type that is often used to model a thing or concept in the real world.

constructor: A method used to create new objects.

attribute: One of the named data items that makes up an instance.

shallow equality: Equality of references, or two references that point to the same object.

deep equality: Equality of values, or two references that point to objects that have the same value.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

Chapter 13

Classes and functions

13.1 Time

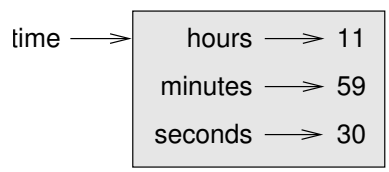
As another example of a user-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time:
  pass
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hours = 11
time.minutes = 59
time.seconds = 30
```

The state diagram for the `Time` object looks like this:



As an exercise, write a function `printTime` *that takes a* `Time` *object as an argument and prints it in the form* `hours:minutes:seconds`.

As a second exercise, write a boolean function `after` *that takes two* `Time` *objects,* `t1` *and* `t2`*, as arguments, and returns* `True` *if* `t1` *follows* `t2` *chronologically and* `False` *otherwise.*

13.2 Pure functions

In the next few sections, we'll write two versions of a function called `addTime`, which calculates the sum of two `Time`s. They will demonstrate two kinds of functions: pure functions and modifiers.

The following is a rough version of `addTime`:

```
def addTime(t1, t2):
  sum = Time()
  sum.hours = t1.hours + t2.hours
  sum.minutes = t1.minutes + t2.minutes
  sum.seconds = t1.seconds + t2.seconds
  return sum
```

The function creates a new `Time` object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no side effects, such as displaying a value or getting user input.

Here is an example of how to use this function. We'll create two `Time` objects: `currentTime`, which contains the current time; and `breadTime`, which contains the amount of time it takes for a breadmaker to make bread. Then we'll use `addTime` to figure out when the bread will be done. If you haven't finished writing `printTime` yet, take a look ahead to Section 14.2 before you try this:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds =  30

>>> breadTime = Time()
>>> breadTime.hours =  3
>>> breadTime.minutes =  35
>>> breadTime.seconds =  0

>>> doneTime = addTime(currentTime, breadTime)
>>> printTime(doneTime)
```

The output of this program is `12:49:30`, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minutes column or the extra minutes into the hours column.

Here's a second corrected version of the function:

```
def addTime(t1, t2):
  sum = Time()
  sum.hours = t1.hours + t2.hours
  sum.minutes = t1.minutes + t2.minutes
  sum.seconds = t1.seconds + t2.seconds

  if sum.seconds >= 60:
    sum.seconds = sum.seconds - 60
    sum.minutes = sum.minutes + 1

  if sum.minutes >= 60:
    sum.minutes = sum.minutes - 60
    sum.hours = sum.hours + 1

  return sum
```

Although this function is correct, it is starting to get big. Later we will suggest an alternative approach that yields shorter code.

13.3 Modifiers

There are times when it is useful for a function to modify one or more of the objects it gets as arguments. Usually, the caller keeps a reference to the objects it passes, so any changes the function makes are visible to the caller. Functions that work this way are called **modifiers**.

`increment`, which adds a given number of seconds to a `Time` object, would be written most naturally as a modifier. A rough draft of the function looks like this:

```
def increment(time, seconds):
  time.seconds = time.seconds + seconds

  if time.seconds >= 60:
    time.seconds = time.seconds - 60
    time.minutes = time.minutes + 1

  if time.minutes >= 60:
    time.minutes = time.minutes - 60
    time.hours = time.hours + 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if the parameter `seconds` is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until `seconds` is less than sixty. One solution is to replace the `if` statements with `while` statements:

```
def increment(time, seconds):
  time.seconds = time.seconds + seconds

  while time.seconds >= 60:
    time.seconds = time.seconds - 60
    time.minutes = time.minutes + 1

  while time.minutes >= 60:
    time.minutes = time.minutes - 60
    time.hours = time.hours + 1
```

This function is now correct, but it is not the most efficient solution.

> *As an exercise, rewrite this function so that it doesn't contain any loops.*
>
> *As a second exercise, rewrite* `increment` *as a pure function, and write function calls to both versions.*

13.4 Which is better?

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient.

In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

13.5 Prototype development versus planning

In this chapter, we demonstrated an approach to program development that we call **prototype development**. In each case, we wrote a rough draft (or prototype) that performed the basic calculation and then tested it on a few cases, correcting flaws as we found them.

Although this approach can be effective, it can lead to code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **planned development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a `Time` object is really a three-digit number in base 60! The `second` component is the "ones column," the `minute` component is the "sixties column," and the `hour` component is the "thirty-six hundreds column."

When we wrote `addTime` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert a `Time` object into a single number and take advantage of the fact that the computer knows how to do arithmetic with numbers. The following function converts a `Time` object into an integer:

```
def convertToSeconds(t):
  minutes = t.hours * 60 + t.minutes
  seconds = minutes * 60 + t.seconds
  return seconds
```

Now, all we need is a way to convert from an integer to a `Time` object:

```
def makeTime(seconds):
  time = Time()
  time.hours = seconds // 3600
  time.minutes = (seconds%3600) // 60
  time.seconds = seconds%60
  return time
```

You might have to think a bit to convince yourself that this function is correct. Assuming you are convinced, you can use it and `convertToSeconds` to rewrite `addTime`:

```
def addTime(t1, t2):
  seconds = convertToSeconds(t1) + convertToSeconds(t2)
  return makeTime(seconds)
```

This version is much shorter than the original, and it is much easier to demonstrate that it is correct.

As an exercise, rewrite `increment` *the same way.*

13.6 Generalization

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`convertToSeconds` and `makeTime`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two `Time`s to find the duration between them. The naïve approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

13.7 Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. We mentioned this word before but did not define it carefully. It is not easy to define, so we will try a couple of approaches.

First, consider something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In our opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

13.8 Glossary

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are fruitless.

functional programming style: A style of program design in which the majority of functions are pure.

prototype development: A way of developing programs starting with a prototype and gradually testing and improving it.

planned development: A way of developing programs that involves high-level insight into the problem and more planning than incremental development or prototype development.

algorithm: A set of instructions for solving a class of problems by a mechanical, unintelligent process.

Chapter 14

Classes and methods

14.1 Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support **object-oriented programming**.

It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

- Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.
- Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the `Time` class defined in Chapter 13 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the `Point` and `Rectangle` classes correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. Strictly speaking, these features are not necessary. For the most part, they provide an alternative syntax for things we have already done, but in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in the `Time` program, there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as an argument.

This observation is the motivation for **methods**. We have already seen some methods, such as `keys` and `values`, which were invoked on dictionaries. Each method is associated with a class and is intended to be invoked on instances of that class.

Methods are just like functions, with two differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it simply by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

14.2 printTime

In Chapter 13, we defined a class named `Time` and you wrote a function named `printTime`, which should have looked something like this:

```
class Time:
  pass

def printTime(time):
  print str(time.hours) + ":" + \
        str(time.minutes) + ":" + \
        str(time.seconds)
```

To call this function, we passed a `Time` object as an argument:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> printTime(currentTime)
```

To make `printTime` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
  def printTime(time):
    print str(time.hours) + ":" +  \
          str(time.minutes) + ":" +  \
          str(time.seconds)
```

Now we can invoke `printTime` using dot notation.

```
>>> currentTime.printTime()
```

As usual, the object on which the method is invoked appears before the dot and the name of the method appears after the dot.

The object on which the method is invoked is assigned to the first parameter, so in this case `currentTime` is assigned to the parameter `time`.

By convention, the first parameter of a method is called `self`. The reason for this is a little convoluted, but it is based on a useful metaphor.

The syntax for a function call, `printTime(currentTime)`, suggests that the function is the active agent. It says something like, "Hey `printTime`! Here's an object for you to print."

In object-oriented programming, the objects are the active agents. An invocation like `currentTime.printTime()` says "Hey `currentTime`! Please print yourself!"

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

14.3 Another example

Let's convert `increment` (from Section 13.3) to a method. To save space, we will leave out previously defined methods, but you should keep them in your version:

```
class Time:
  #previous method definitions here...

  def increment(self, seconds):
    self.seconds = seconds + self.seconds

    while self.seconds >= 60:
      self.seconds = self.seconds - 60
```

```
    self.minutes = self.minutes + 1

  while self.minutes >= 60:
    self.minutes = self.minutes - 60
    self.hours = self.hours + 1
```

The transformation is purely mechanical—we move the method definition into the class definition and change the name of the first parameter.

Now we can invoke `increment` as a method.

```
currentTime.increment(500)
```

Again, the object on which the method is invoked gets assigned to the first parameter, `self`. The second parameter, `seconds` gets the value `500`.

> *As an exercise, convert* `convertToSeconds` *(from Section 13.5) to a method in the* `Time` *class.*

14.4 A more complicated example

The `after` function is slightly more complicated because it operates on two `Time` objects, not just one. We can only convert one of the parameters to `self`; the other stays the same:

```
class Time:
  #previous method definitions here...

  def after(self, time2):
    if self.hour > time2.hour:
      return 1
    if self.hour < time2.hour:
      return 0

    if self.minute > time2.minute:
      return 1
    if self.minute < time2.minute:
      return 0

    if self.second > time2.second:
      return 1
    return 0
```

We invoke this method on one object and pass the other as an argument:

```
if doneTime.after(currentTime):
  print "The bread is not done yet."
```

You can almost read the invocation like English: "If the done-time is after the current-time, then..."

14.5 Optional arguments

We have seen built-in functions that take a variable number of arguments. For example, `string.find` can take two, three, or four arguments.

It is possible to write user-defined functions with optional argument lists. For example, we can upgrade our own version of `find` to do the same thing as `string.find`.

This is the original version from Section 7.7:

```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

This is the new and improved version:

```
def find(str, ch, start=0):
  index = start
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

The third parameter, `start`, is optional because a default value, `0`, is provided. If we invoke `find` with only two arguments, it uses the default value and starts from the beginning of the string:

```
>>> find("apple", "p")
1
```

If we provide a third argument, it **overrides** the default:

```
>>> find("apple", "p", 2)
2
>>> find("apple", "p", 3)
-1
```

As an exercise, add a fourth parameter, `end`*, that specifies where to stop looking.*

Warning: This exercise is a bit tricky. The default value of `end` *should be* `len(str)`*, but that doesn't work. The default values are evaluated when the function is defined, not when it is called. When* `find` *is defined,* `str` *doesn't exist yet, so you can't find its length.*

14.6 The initialization method

The **initialization method** is a special method that is invoked when an object is created. The name of this method is `__init__` (two underscore characters, followed by `init`, and then two more underscores). An initialization method for the `Time` class looks like this:

```
class Time:
  def __init__(self, hours=0, minutes=0, seconds=0):
    self.hours = hours
    self.minutes = minutes
    self.seconds = seconds
```

There is no conflict between the attribute `self.hours` and the parameter `hours`. Dot notation specifies which variable we are referring to.

When we invoke the `Time` constructor, the arguments we provide are passed along to `init`:

```
>>> currentTime = Time(9, 14, 30)
>>> currentTime.printTime()
9:14:30
```

Because the arguments are optional, we can omit them:

```
>>> currentTime = Time()
>>> currentTime.printTime()
0:0:0
```

Or provide only the first:

```
>>> currentTime = Time (9)
>>> currentTime.printTime()
9:0:0
```

Or the first two:

```
>>> currentTime = Time (9, 14)
>>> currentTime.printTime()
9:14:0
```

Finally, we can make assignments to a subset of the parameters by naming them explicitly:

```
>>> currentTime = Time(seconds = 30, hours = 9)
>>> currentTime.printTime()
9:0:30
```

14.7 Points revisited

Let's rewrite the `Point` class from Section 12.1 in a more object-oriented style:

```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

The initialization method takes x and y values as optional parameters; the default for either parameter is 0.

The next method, `__str__`, returns a string representation of a `Point` object. If a class provides a method named `__str__`, it overrides the default behavior of the Python built-in `str` function.

```
>>> p = Point(3, 4)
>>> str(p)
'(3, 4)'
```

Printing a `Point` object implicitly invokes `__str__` on the object, so defining `__str__` also changes the behavior of `print`:

```
>>> p = Point(3, 4)
>>> print p
(3, 4)
```

When we write a new class, we almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is almost always useful for debugging.

14.8 Operator overloading

Some languages make it possible to change the definition of the built-in operators when they are applied to user-defined types. This feature is called **operator overloading**. It is especially useful when defining new mathematical types.

For example, to override the addition operator +, we provide a method named __add__:

```
class Point:
  # previously defined methods here...

  def __add__(self, other):
    return Point(self.x + other.x, self.y + other.y)
```

As usual, the first parameter is the object on which the method is invoked. The second parameter is conveniently named other to distinguish it from self. To add two Points, we create and return a new Point that contains the sum of the x coordinates and the sum of the y coordinates.

Now, when we apply the + operator to Point objects, Python invokes __add__:

```
>>>   p1 = Point(3, 4)
>>>   p2 = Point(5, 7)
>>>   p3 = p1 + p2
>>>   print p3
(8, 11)
```

The expression p1 + p2 is equivalent to p1.__add__(p2), but obviously more elegant.

> *As an exercise, add a method* __sub__(self, other) *that overloads the subtraction operator, and try it out.*

There are several ways to override the behavior of the multiplication operator: by defining a method named __mul__, or __rmul__, or both.

If the left operand of * is a Point, Python invokes __mul__, which assumes that the other operand is also a Point. It computes the **dot product** of the two points, defined according to the rules of linear algebra:

```
def __mul__(self, other):
  return self.x * other.x + self.y * other.y
```

If the left operand of * is a primitive type and the right operand is a Point, Python invokes __rmul__, which performs **scalar multiplication**:

```
def __rmul__(self, other):
  return Point(other * self.x,  other * self.y)
```

The result is a new Point whose coordinates are a multiple of the original coordinates. If other is a type that cannot be multiplied by a floating-point number, then __rmul__ will yield an error.

This example demonstrates both kinds of multiplication:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

What happens if we try to evaluate p2 * 2? Since the first operand is a Point, Python invokes __mul__ with 2 as the second argument. Inside __mul__, the program tries to access the x coordinate of other, which fails because an integer has no attributes:

```
>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
```

Unfortunately, the error message is a bit opaque. This example demonstrates some of the difficulties of object-oriented programming. Sometimes it is hard enough just to figure out what code is running.

For a more complete example of operator overloading, see Appendix B.

14.9 Polymorphism

Most of the methods we have written only work for a specific type. When you create a new object, you write methods that operate on that type.

But there are certain operations that you will want to apply to many types, such as the arithmetic operations in the previous sections. If many types support the same set of operations, you can write functions that work on any of those types.

For example, the `multadd` operation (which is common in linear algebra) takes three arguments; it multiplies the first two and then adds the third. We can write it in Python like this:

```
def multadd (x, y, z):
  return x * y + z
```

This method will work for any values of `x` and `y` that can be multiplied and for any value of `z` that can be added to the product.

We can invoke it with numeric values:

```
>>> multadd (3, 2, 1)
7
```

Or with `Point`s:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print multadd (2, p1, p2)
(11, 15)
>>> print multadd (p1, p2, 1)
44
```

In the first case, the `Point` is multiplied by a scalar and then added to another `Point`. In the second case, the dot product yields a numeric value, so the third argument also has to be a numeric value.

A function like this that can take arguments with different types is called **polymorphic**.

As another example, consider the method `frontAndBack`, which prints a list twice, forward and backward:

```
def frontAndBack(front):
  import copy
  back = copy.copy(front)
  back.reverse()
  print str(front) + str(back)
```

Because the `reverse` method is a modifier, we make a copy of the list before reversing it. That way, this method doesn't modify the list it gets as an argument.

Here's an example that applies `frontAndBack` to a list:

```
>>>   myList = [1, 2, 3, 4]
>>>   frontAndBack(myList)
[1, 2, 3, 4][4, 3, 2, 1]
```

Of course, we intended to apply this function to lists, so it is not surprising that it works. What would be surprising is if we could apply it to a `Point`.

To determine whether a function can be applied to a new type, we apply the fundamental rule of polymorphism:

> **If all of the operations inside the function can be applied to the type, the function can be applied to the type.**

The operations in the method include `copy`, `reverse`, and `print`.

`copy` works on any object, and we have already written a `__str__` method for `Point`s, so all we need is a `reverse` method in the `Point` class:

```
def reverse(self):
  self.x , self.y = self.y, self.x
```

Then we can pass `Point`s to `frontAndBack`:

```
>>>   p = Point(3, 4)
>>>   frontAndBack(p)
(3, 4)(4, 3)
```

The best kind of polymorphism is the unintentional kind, where you discover that a function you have already written can be applied to a type for which you never planned.

14.10 Glossary

object-oriented language: A language that provides features, such as user-defined classes and inheritance, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

override: To replace a default. Examples include replacing a default value with a particular argument and replacing a default method by providing a new method with the same name.

initialization method: A special method that is invoked automatically when a new object is created and that initializes the object's attributes.

operator overloading: Extending built-in operators (`+`, `-`, `*`, `>`, `<`, etc.) so that they work with user-defined types.

dot product: An operation defined in linear algebra that multiplies two `Points` and yields a numeric value.

scalar multiplication: An operation defined in linear algebra that multiplies each of the coordinates of a `Point` by a numeric value.

polymorphic: A function that can operate on more than one type. If all the operations in a function can be applied to a type, then the function can be applied to a type.

Chapter 15

Sets of objects

15.1 Composition

By now, you have seen several examples of composition. One of the first examples was using a method invocation as part of an expression. Another example is the nested structure of statements; you can put an `if` statement within a `while` loop, within another `if` statement, and so on.

Having seen this pattern, and having learned about lists and objects, you should not be surprised to learn that you can create lists of objects. You can also create objects that contain lists (as attributes); you can create lists that contain lists; you can create objects that contain objects; and so on.

In this chapter and the next, we will look at some examples of these combinations, using `Card` objects as an example.

15.2 `Card` objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, the rank of Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is not as obvious what type the attributes

should be. One possibility is to use strings containing words like `"Spade"` for suits and `"Queen"` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By "encode," we do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by "encode" is "to define a mapping between a sequence of numbers and the items I want to represent." For example:

Spades	$\mapsto$	3
Hearts	$\mapsto$	2
Diamonds	$\mapsto$	1
Clubs	$\mapsto$	0

An obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

The reason we are using mathematical notation for these mappings is that they are not part of the Python program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```
class Card:
  def __init__(self, suit=0, rank=2):
    self.suit = suit
    self.rank = rank
```

As usual, we provide an initialization method that takes an optional parameter for each attribute. The default value of `suit` is 0, which represents Clubs.

To create a Card, we invoke the Card constructor with the suit and rank of the card we want.

```
threeOfClubs = Card(3, 1)
```

In the next section we'll figure out which card we just made.

15.3 Class attributes and the __str__ method

In order to print `Card` objects in a way that people can easily read, we want to map the integer codes onto words. A natural way to do that is with lists of strings. We assign these lists to **class attributes** at the top of the class definition:

```
class Card:
  suitList = ["Clubs", "Diamonds", "Hearts", "Spades"]
  rankList = ["narf", "Ace", "2", "3", "4", "5", "6", "7",
              "8", "9", "10", "Jack", "Queen", "King"]

  #init method omitted

  def __str__(self):
    return (self.rankList[self.rank] + " of " +
            self.suitList[self.suit])
```

A class attribute is defined outside of any method, and it can be accessed from any of the methods in the class.

Inside `__str__`, we can use `suitList` and `rankList` to map the numerical values of `suit` and `rank` to strings. For example, the expression `self.suitList[self.suit]` means "use the attribute `suit` from the object `self` as an index into the class attribute named `suitList`, and select the appropriate string."

The reason for the `"narf"` in the first element in `rankList` is to act as a place keeper for the zero-eth element of the list, which should never be used. The only valid ranks are 1 to 13. This wasted item is not entirely necessary. We could have started at 0, as usual, but it is less confusing to encode 2 as 2, 3 as 3, and so on.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(1, 11)
>>> print card1
Jack of Diamonds
```

Class attributes like `suitList` are shared by all `Card` objects. The advantage of this is that we can use any `Card` object to access the class attributes:

```
>>> card2 = Card(1, 3)
>>> print card2
3 of Diamonds
>>> print card2.suitList[1]
Diamonds
```

The disadvantage is that if we modify a class attribute, it affects every instance of the class. For example, if we decide that "Jack of Diamonds" should really be called "Jack of Swirly Whales," we could do this:

```
>>> card1.suitList[1] = "Swirly Whales"
>>> print card1
Jack of Swirly Whales
```

The problem is that *all* of the Diamonds just became Swirly Whales:

```
>>> print card2
3 of Swirly Whales
```

It is usually not a good idea to modify class attributes.

15.4 Comparing cards

For primitive types, there are conditional operators (`<`, `>`, `==`, etc.) that compare values and determine when one is greater than, less than, or equal to another. For user-defined types, we can override the behavior of the built-in operators by providing a method named `__cmp__`. By convention, `__cmp__` has two parameters, `self` and `other`, and returns 1 if the first object is greater, -1 if the second object is greater, and 0 if they are equal to each other.

Some types are completely ordered, which means that you can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are completely ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why you cannot compare apples and oranges.

The set of playing cards is partially ordered, which means that sometimes you can compare cards and sometimes not. For example, you know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to make cards comparable, you have to decide which is more important, rank or suit. To be honest, the choice is arbitrary. For the sake of choosing, we will say that suit is more important, because a new deck of cards comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `__cmp__`:

```
def __cmp__(self, other):
  # check the suits
  if self.suit > other.suit: return 1
  if self.suit < other.suit: return -1
  # suits are the same... check ranks
  if self.rank > other.rank: return 1
  if self.rank < other.rank: return -1
  # ranks are the same... it's a tie
  return 0
```

In this ordering, Aces appear lower than Deuces (2s).

As an exercise, modify `__cmp__` *so that Aces are ranked higher than Kings.*

15.5 Decks

Now that we have objects to represent `Card`s, the next logical step is to define a class to represent a `Deck`. Of course, a deck is made up of cards, so each `Deck` object will contain a list of cards as an attribute.

The following is a class definition for the `Deck` class. The initialization method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck:
  def __init__(self):
    self.cards = []
    for suit in range(4):
      for rank in range(1, 14):
        self.cards.append(Card(suit, rank))
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Since the outer loop iterates four times, and the inner loop iterates thirteen times, the total number of times the body is executed is fifty-two (thirteen times four). Each iteration creates a new instance of `Card` with the current suit and rank, and appends that card to the `cards` list.

The `append` method works on lists but not, of course, tuples.

15.6 Printing the deck

As usual, when we define a new type of object we want a method that prints the contents of an object. To print a `Deck`, we traverse the list and print each `Card`:

```
class Deck:
  ...
  def printDeck(self):
    for card in self.cards:
      print card
```

Here, and from now on, the ellipsis (...) indicates that we have omitted the other methods in the class.

As an alternative to `printDeck`, we could write a `__str__` method for the `Deck` class. The advantage of `__str__` is that it is more flexible. Rather than just printing the contents of the object, it generates a string representation that other parts of the program can manipulate before printing, or store for later use.

Here is a version of `__str__` that returns a string representation of a `Deck`. To add a bit of pizzazz, it arranges the cards in a cascade where each card is indented one space more than the previous card:

```
class Deck:
  ...
  def __str__(self):
    s = ""
    for i in range(len(self.cards)):
      s = s + " "*i + str(self.cards[i]) + "\n"
    return s
```

This example demonstrates several features. First, instead of traversing `self.cards` and assigning each card to a variable, we are using `i` as a loop variable and an index into the list of cards.

Second, we are using the string multiplication operator to indent each card by one more space than the last. The expression `" "*i` yields a number of spaces equal to the current value of `i`.

Third, instead of using the `print` command to print the cards, we use the `str` function. Passing an object as an argument to `str` is equivalent to invoking the `__str__` method on the object.

Finally, we are using the variable `s` as an **accumulator**. Initially, `s` is the empty string. Each time through the loop, a new string is generated and concatenated with the old value of `s` to get the new value. When the loop ends, `s` contains the complete string representation of the `Deck`, which looks like this:

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
 2 of Clubs
  3 of Clubs
   4 of Clubs
    5 of Clubs
     6 of Clubs
      7 of Clubs
       8 of Clubs
        9 of Clubs
         10 of Clubs
          Jack of Clubs
           Queen of Clubs
            King of Clubs
             Ace of Diamonds
```

And so on. Even though the result appears on 52 lines, it is one long string that contains newlines.

15.7 Shuffling the deck

If a deck is perfectly shuffled, then any card is equally likely to appear anywhere in the deck, and any location in the deck is equally likely to contain any card.

To shuffle the deck, we will use the `randrange` function from the `random` module. With two integer arguments, `a` and `b`, `randrange` chooses a random integer in the range `a <= x < b`. Since the upper bound is strictly less than `b`, we can use the length of a list as the second argument, and we are guaranteed to get a legal index. For example, this expression chooses the index of a random card in a deck:

```
random.randrange(0, len(self.cards))
```

An easy way to shuffle the deck is by traversing the cards and swapping each card with a randomly chosen one. It is possible that the card will be swapped with itself, but that is fine. In fact, if we precluded that possibility, the order of the cards would be less than entirely random:

```
class Deck:
  ...
  def shuffle(self):
    import random
    nCards = len(self.cards)
    for i in range(nCards):
      j = random.randrange(i, nCards)
      self.cards[i], self.cards[j] = self.cards[j], self.cards[i]
```

Rather than assume that there are fifty-two cards in the deck, we get the actual length of the list and store it in `nCards`.

For each card in the deck, we choose a random card from among the cards that haven't been shuffled yet. Then we swap the current card (`i`) with the selected card (`j`). To swap the cards we use a tuple assignment, as in Section 9.2:

```
self.cards[i], self.cards[j] = self.cards[j], self.cards[i]
```

> *As an exercise, rewrite this line of code without using a sequence assignment.*

15.8 Removing and dealing cards

Another method that would be useful for the `Deck` class is `removeCard`, which takes a card as an argument, removes it, and returns `True` if the card was in the deck and `False` otherwise:

```
class Deck:
  ...
  def removeCard(self, card):
    if card in self.cards:
      self.cards.remove(card)
      return True
    else:
      return False
```

The `in` operator returns true if the first operand is in the second, which must be a list or a tuple. If the first operand is an object, Python uses the object's `__cmp__` method to determine equality with items in the list. Since the `__cmp__` in the `Card` class checks for deep equality, the `removeCard` method checks for deep equality.

To deal cards, we want to remove and return the top card. The list method `pop` provides a convenient way to do that:

```
class Deck:
  ...
  def popCard(self):
    return self.cards.pop()
```

Actually, `pop` removes the *last* card in the list, so we are in effect dealing from the bottom of the deck.

One more operation that we are likely to want is the boolean function `isEmpty`, which returns true if the deck contains no cards:

```
class Deck:
  ...
  def isEmpty(self):
    return (len(self.cards) == 0)
```

15.9 Glossary

encode: To represent one set of values using another set of values by constructing a mapping between them.

class attribute: A variable that is defined inside a class definition but outside any method. Class attributes are accessible from any method in the class and are shared by all instances of the class.

accumulator: A variable used in a loop to accumulate a series of values, such as by concatenating them onto a string or adding them to a running sum.

Chapter 16

Inheritance

16.1 Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class.

The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. It is called "inheritance" because the new class inherits all of the methods of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class. The new class may be called the **child** class or sometimes "subclass."

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be scattered among several modules. Also, many of the things that can be done using inheritance can be done as elegantly (or more so) without it. If the natural structure of the problem does not lend itself to inheritance, this style of programming can do more harm than good.

In this chapter we will demonstrate the use of inheritance as part of a program that plays the card game Old Maid. One of our goals is to write code that could be reused to implement other card games.

16.2 A hand of cards

For almost any card game, we need to represent a hand of cards. A hand is similar to a deck, of course. Both are made up of a set of cards, and both require operations like adding and removing cards. Also, we might like the ability to shuffle both decks and hands.

A hand is also different from a deck. Depending on the game being played, we might want to perform some operations on hands that don't make sense for a deck. For example, in poker we might classify a hand (straight, flush, etc.) or compare it with another hand. In bridge, we might want to compute a score for a hand in order to make a bid.

This situation suggests the use of inheritance. If `Hand` is a subclass of `Deck`, it will have all the methods of `Deck`, and new methods can be added.

In the class definition, the name of the parent class appears in parentheses:

```
class Hand(Deck):
  pass
```

This statement indicates that the new `Hand` class inherits from the existing `Deck` class.

The `Hand` constructor initializes the attributes for the hand, which are `name` and `cards`. The string `name` identifies this hand, probably by the name of the player that holds it. The name is an optional parameter with the empty string as a default value. `cards` is the list of cards in the hand, initialized to the empty list:

```
class Hand(Deck):
  def __init__(self, name=""):
    self.cards = []
    self.name = name
```

For just about any card game, it is necessary to add and remove cards from the deck. Removing cards is already taken care of, since `Hand` inherits `removeCard` from `Deck`. But we have to write `addCard`:

```
class Hand(Deck):
  ...
  def addCard(self,card) :
    self.cards.append(card)
```

Again, the ellipsis indicates that we have omitted other methods. The list `append` method adds the new card to the end of the list of cards.

16.3 Dealing cards

Now that we have a `Hand` class, we want to deal cards from the `Deck` into hands. It is not immediately obvious whether this method should go in the `Hand` class or in the `Deck` class, but since it operates on a single deck and (possibly) several hands, it is more natural to put it in `Deck`.

`deal` should be fairly general, since different games will have different requirements. We may want to deal out the entire deck at once or add one card to each hand.

`deal` takes three parameters: the deck, a list (or tuple) of hands, and the total number of cards to deal. If there are not enough cards in the deck, the method deals out all of the cards and stops:

```
class Deck :
  ...
  def deal(self, hands, nCards=999):
    nHands = len(hands)
    for i in range(nCards):
      if self.isEmpty(): break    # break if out of cards
      card = self.popCard()       # take the top card
      hand = hands[i % nHands]    # whose turn is next?
      hand.addCard(card)          # add the card to the hand
```

The last parameter, `nCards`, is optional; the default is a large number, which effectively means that all of the cards in the deck will get dealt.

The loop variable `i` goes from 0 to `nCards-1`. Each time through the loop, a card is removed from the deck using the list method `pop`, which removes and returns the last item in the list.

The modulus operator (`%`) allows us to deal cards in a round robin (one card at a time to each hand). When `i` is equal to the number of hands in the list, the expression `i % nHands` wraps around to the beginning of the list (index 0).

16.4 Printing a Hand

To print the contents of a hand, we can take advantage of the `printDeck` and `__str__` methods inherited from `Deck`. For example:

```
>>> deck = Deck()
>>> deck.shuffle()
>>> hand = Hand("frank")
>>> deck.deal([hand], 5)
>>> print hand
Hand frank contains
2 of Spades
 3 of Spades
  4 of Spades
   Ace of Hearts
    9 of Clubs
```

It's not a great hand, but it has the makings of a straight flush.

Although it is convenient to inherit the existing methods, there is additional information in a `Hand` object we might want to include when we print one. To do that, we can provide a `__str__` method in the `Hand` class that overrides the one in the `Deck` class:

```
class Hand(Deck)
  ...
  def __str__(self):
    s = "Hand " + self.name
    if self.isEmpty():
      return s + " is empty\n"
    else:
      return s + " contains\n" + Deck.__str__(self)
```

Initially, `s` is a string that identifies the hand. If the hand is empty, the program appends the words `is empty` and returns the result.

Otherwise, the program appends the word `contains` and the string representation of the `Deck`, computed by invoking the `__str__` method in the `Deck` class on `self`.

It may seem odd to send `self`, which refers to the current `Hand`, to a `Deck` method, until you remember that a `Hand` is a kind of `Deck`. `Hand` objects can do everything `Deck` objects can, so it is legal to send a `Hand` to a `Deck` method.

In general, it is always legal to use an instance of a subclass in place of an instance of a parent class.

16.5 The `CardGame` class

The `CardGame` class takes care of some basic chores common to all games, such as creating the deck and shuffling it:

```
class CardGame:
  def __init__(self):
    self.deck = Deck()
    self.deck.shuffle()
```

This is the first case we have seen where the initialization method performs a significant computation, beyond initializing attributes.

To implement specific games, we can inherit from `CardGame` and add features for the new game. As an example, we'll write a simulation of Old Maid.

The object of Old Maid is to get rid of cards in your hand. You do this by matching cards by rank and color. For example, the 4 of Clubs matches the 4 of Spades since both suits are black. The Jack of Hearts matches the Jack of Diamonds since both are red.

To begin the game, the Queen of Clubs is removed from the deck so that the Queen of Spades has no match. The fifty-one remaining cards are dealt to the players in a round robin. After the deal, all players match and discard as many cards as possible.

When no more matches can be made, play begins. In turn, each player picks a card (without looking) from the closest neighbor to the left who still has cards. If the chosen card matches a card in the player's hand, the pair is removed. Otherwise, the card is added to the player's hand. Eventually all possible matches are made, leaving only the Queen of Spades in the loser's hand.

In our computer simulation of the game, the computer plays all hands. Unfortunately, some nuances of the real game are lost. In a real game, the player with the Old Maid goes to some effort to get their neighbor to pick that card, by displaying it a little more prominently, or perhaps failing to display it more prominently, or even failing to fail to display that card more prominently. The computer simply picks a neighbor's card at random.

16.6 `OldMaidHand` class

A hand for playing Old Maid requires some abilities beyond the general abilities of a `Hand`. We will define a new class, `OldMaidHand`, that inherits from `Hand` and provides an additional method called `removeMatches`:

```
class OldMaidHand(Hand):
  def removeMatches(self):
    count = 0
    originalCards = self.cards[:]
```

```
    for card in originalCards:
      match = Card(3 - card.suit, card.rank)
      if match in self.cards:
        self.cards.remove(card)
        self.cards.remove(match)
        print "Hand %s: %s matches %s" % (self.name,card,match)
        count = count + 1
    return count
```

We start by making a copy of the list of cards, so that we can traverse the copy while removing cards from the original. Since `self.cards` is modified in the loop, we don't want to use it to control the traversal. Python can get quite confused if it is traversing a list that is changing!

For each card in the hand, we figure out what the matching card is and go looking for it. The match card has the same rank and the other suit of the same color. The expression `3 - card.suit` turns a Club (suit 0) into a Spade (suit 3) and a Diamond (suit 1) into a Heart (suit 2). You should satisfy yourself that the opposite operations also work. If the match card is also in the hand, both cards are removed.

The following example demonstrates how to use `removeMatches`:

```
>>> game = CardGame()
>>> hand = OldMaidHand("frank")
>>> game.deck.deal([hand], 13)
>>> print hand
Hand frank contains
Ace of Spades
 2 of Diamonds
  7 of Spades
   8 of Clubs
    6 of Hearts
     8 of Spades
      7 of Clubs
       Queen of Clubs
        7 of Diamonds
         5 of Clubs
          Jack of Diamonds
           10 of Diamonds
            10 of Hearts

>>> hand.removeMatches()
Hand frank: 7 of Spades matches 7 of Clubs
Hand frank: 8 of Spades matches 8 of Clubs
```

```
Hand frank: 10 of Diamonds matches 10 of Hearts
>>> print hand
Hand frank contains
Ace of Spades
 2 of Diamonds
  6 of Hearts
   Queen of Clubs
    7 of Diamonds
     5 of Clubs
      Jack of Diamonds
```

Notice that there is no __init__ method for the OldMaidHand class. We inherit it from Hand.

16.7 OldMaidGame class

Now we can turn our attention to the game itself. OldMaidGame is a subclass of CardGame with a new method called play that takes a list of players as an argument.

Since __init__ is inherited from CardGame, a new OldMaidGame object contains a new shuffled deck:

```
class OldMaidGame(CardGame):
  def play(self, names):
    # remove Queen of Clubs
    self.deck.removeCard(Card(0,12))

    # make a hand for each player
    self.hands = []
    for name in names :
      self.hands.append(OldMaidHand(name))

    # deal the cards
    self.deck.deal(self.hands)
    print "---------- Cards have been dealt"
    self.printHands()

    # remove initial matches
    matches = self.removeAllMatches()
    print "---------- Matches discarded, play begins"
    self.printHands()
```

```
    # play until all 50 cards are matched
    turn = 0
    numHands = len(self.hands)
    while matches < 25:
      matches = matches + self.playOneTurn(turn)
      turn = (turn + 1) % numHands

    print "---------- Game is Over"
    self.printHands()
```

Some of the steps of the game have been separated into methods. `removeAllMatches` traverses the list of hands and invokes `removeMatches` on each:

```
class OldMaidGame(CardGame):
  ...
  def removeAllMatches(self):
    count = 0
    for hand in self.hands:
      count = count + hand.removeMatches()
    return count
```

As an exercise, write `printHands` *which traverses* `self.hands` *and prints each hand.*

`count` is an accumulator that adds up the number of matches in each hand and returns the total.

When the total number of matches reaches twenty-five, fifty cards have been removed from the hands, which means that only one card is left and the game is over.

The variable `turn` keeps track of which player's turn it is. It starts at 0 and increases by one each time; when it reaches `numHands`, the modulus operator wraps it back around to 0.

The method `playOneTurn` takes an argument that indicates whose turn it is. The return value is the number of matches made during this turn:

```
class OldMaidGame(CardGame):
  ...
  def playOneTurn(self, i):
    if self.hands[i].isEmpty():
      return 0
    neighbor = self.findNeighbor(i)
    pickedCard = self.hands[neighbor].popCard()
    self.hands[i].addCard(pickedCard)
    print "Hand", self.hands[i].name, "picked", pickedCard
    count = self.hands[i].removeMatches()
    self.hands[i].shuffle()
    return count
```

If a player's hand is empty, that player is out of the game, so he or she does nothing and returns 0.

Otherwise, a turn consists of finding the first player on the left that has cards, taking one card from the neighbor, and checking for matches. Before returning, the cards in the hand are shuffled so that the next player's choice is random.

The method `findNeighbor` starts with the player to the immediate left and continues around the circle until it finds a player that still has cards:

```
class OldMaidGame(CardGame):
  ...
  def findNeighbor(self, i):
    numHands = len(self.hands)
    for next in range(1,numHands):
      neighbor = (i + next) % numHands
      if not self.hands[neighbor].isEmpty():
        return neighbor
```

If `findNeighbor` ever went all the way around the circle without finding cards, it would return `None` and cause an error elsewhere in the program. Fortunately, we can prove that that will never happen (as long as the end of the game is detected correctly).

We have omitted the `printHands` method. You can write that one yourself.

The following output is from a truncated form of the game where only the top fifteen cards (tens and higher) were dealt to three players. With this small deck, play stops after seven matches instead of twenty-five.

```
>>> import cards
>>> game = cards.OldMaidGame()
>>> game.play(["Allen","Jeff","Chris"])
```

```
---------- Cards have been dealt
Hand Allen contains
King of Hearts
 Jack of Clubs
  Queen of Spades
   King of Spades
    10 of Diamonds

Hand Jeff contains
Queen of Hearts
 Jack of Spades
  Jack of Hearts
   King of Diamonds
    Queen of Diamonds

Hand Chris contains
Jack of Diamonds
 King of Clubs
  10 of Spades
   10 of Hearts
    10 of Clubs

Hand Jeff: Queen of Hearts matches Queen of Diamonds
Hand Chris: 10 of Spades matches 10 of Clubs
---------- Matches discarded, play begins
Hand Allen contains
King of Hearts
 Jack of Clubs
  Queen of Spades
   King of Spades
    10 of Diamonds

Hand Jeff contains
Jack of Spades
 Jack of Hearts
  King of Diamonds

Hand Chris contains
Jack of Diamonds
 King of Clubs
  10 of Hearts
```

```
Hand Allen picked King of Diamonds
Hand Allen: King of Hearts matches King of Diamonds
Hand Jeff picked 10 of Hearts
Hand Chris picked Jack of Clubs
Hand Allen picked Jack of Hearts
Hand Jeff picked Jack of Diamonds
Hand Chris picked Queen of Spades
Hand Allen picked Jack of Diamonds
Hand Allen: Jack of Hearts matches Jack of Diamonds
Hand Jeff picked King of Clubs
Hand Chris picked King of Spades
Hand Allen picked 10 of Hearts
Hand Allen: 10 of Diamonds matches 10 of Hearts
Hand Jeff picked Queen of Spades
Hand Chris picked Jack of Spades
Hand Chris: Jack of Clubs matches Jack of Spades
Hand Jeff picked King of Spades
Hand Jeff: King of Clubs matches King of Spades
---------- Game is Over
Hand Allen is empty

Hand Jeff contains
Queen of Spades

Hand Chris is empty
```

So Jeff loses.

16.8 Glossary

inheritance: The ability to define a new class that is a modified version of a previously defined class.

parent class: The class from which a child class inherits.

child class: A new class created by inheriting from an existing class; also called a "subclass."

Chapter 17

Linked lists

17.1 Embedded references

We have seen examples of attributes that refer to other objects, which we called **embedded references** (see Section 12.8). A common data structure, the **linked list**, takes advantage of this feature.

Linked lists are made up of **nodes**, where each node contains a reference to the next node in the list. In addition, each node contains a unit of data called the **cargo**.

A linked list is considered a **recursive data structure** because it has a recursive definition.

> A linked list is either:
>
> - the empty list, represented by `None`, or
> - a node that contains a cargo object and a reference to a linked list.

Recursive data structures lend themselves to recursive methods.

17.2 The `Node` class

As usual when writing a new class, we'll start with the initialization and `__str__` methods so that we can test the basic mechanism of creating and displaying the new type:

```
class Node:
  def __init__(self, cargo=None, next=None):
    self.cargo = cargo
    self.next  = next

  def __str__(self):
    return str(self.cargo)
```

As usual, the parameters for the initialization method are optional. By default, both the cargo and the link, `next`, are set to `None`.

The string representation of a node is just the string representation of the cargo. Since any value can be passed to the `str` function, we can store any value in a list.

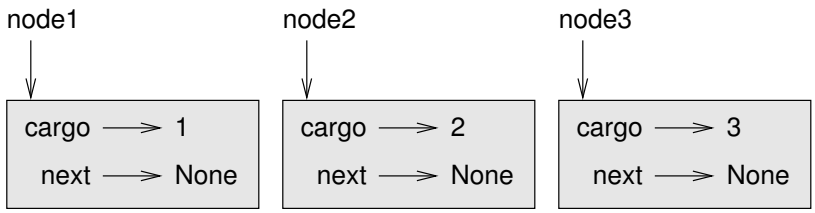
To test the implementation so far, we can create a `Node` and print it:

```
>>> node = Node("test")
>>> print node
test
```

To make it interesting, we need a list with more than one node:

```
>>> node1 = Node(1)
>>> node2 = Node(2)
>>> node3 = Node(3)
```

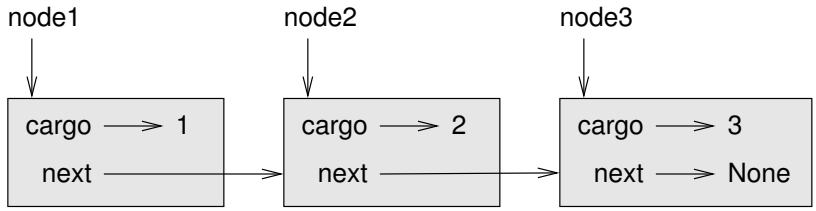
This code creates three nodes, but we don't have a list yet because the nodes are not **linked**. The state diagram looks like this:



To link the nodes, we have to make the first node refer to the second and the second node refer to the third:

```
>>> node1.next = node2
>>> node2.next = node3
```

The reference of the third node is `None`, which indicates that it is the end of the list. Now the state diagram looks like this:



Now you know how to create nodes and link them into lists. What might be less clear at this point is why.

17.3 Lists as collections

Lists are useful because they provide a way to assemble multiple objects into a single entity, sometimes called a **collection**. In the example, the first node of the list serves as a reference to the entire list.

To pass the list as an argument, we only have to pass a reference to the first node. For example, the function `printList` takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end:

```
def printList(node):
  while node:
    print node,
    node = node.next
  print
```

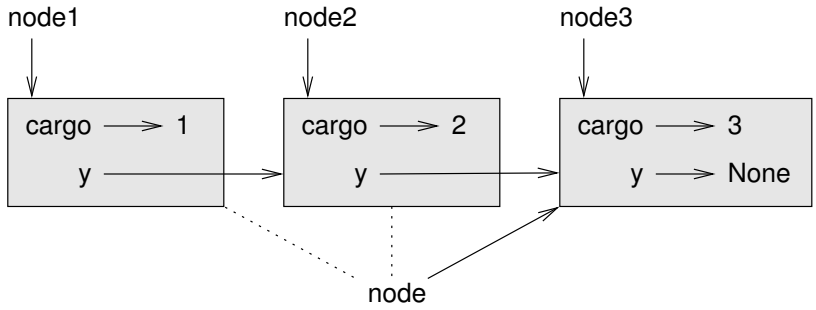
To invoke this function, we pass a reference to the first node:

```
>>> printList(node1)
1 2 3
```

Inside `printList` we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the `next` value from each node to get to the next node.

To traverse a linked list, it is common to use a loop variable like `node` to refer to each of the nodes in succession.

This diagram shows the nodes in the list and the values that `node` takes on:



By convention, lists are often printed in brackets with commas between the elements, as in `[1, 2, 3]`*. As an exercise, modify* `printList` *so that it generates output in this format.*

17.4 Lists and recursion

It is natural to express many list operations using recursive methods. For example, the following is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head); and the rest (called the tail).
2. Print the tail backward.
3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backward. But if we assume that the recursive call works—the leap of faith—then we can convince ourselves that this algorithm works.

All we need are a base case and a way of proving that for any list, we will eventually get to the base case. Given the recursive definition of a list, a natural base case is the empty list, represented by `None`:

```
def printBackward(list):
  if list == None: return
  head = list
  tail = list.next
  printBackward(tail)
  print head,
```

The first line handles the base case by doing nothing. The next two lines split the list into `head` and `tail`. The last two lines print the list. The comma at the end of the last line keeps Python from printing a newline after each node.

We invoke this function as we invoked `printList`:

```
>>> printBackward(node1)
3 2 1
```

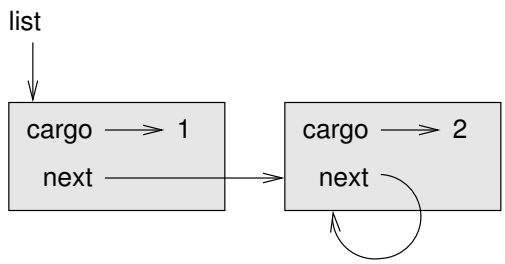
The result is a backward list.

You might wonder why `printList` and `printBackward` are functions and not methods in the `Node` class. The reason is that we want to use `None` to represent the empty list and it is not legal to invoke a method on `None`. This limitation makes it awkward to write list-manipulating code in a clean object-oriented style.

Can we prove that `printBackward` will always terminate? In other words, will it always reach the base case? In fact, the answer is no. Some lists will make this function crash.

17.5 Infinite lists

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself:



If we invoke `printList` on this list, it will loop forever. If we invoke `printBackward`, it will recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasionally useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction.

Regardless, it is problematic that we cannot prove that `printList` and `printBackward` terminate. The best we can do is the hypothetical statement, "If the list contains no loops, then these functions will terminate." This sort of claim is called a **precondition**. It imposes a constraint on one of the arguments and describes the behavior of the function if the constraint is satisfied. You will see more examples soon.

17.6 The fundamental ambiguity theorem

One part of `printBackward` might have raised an eyebrow:

```
head = list
tail = list.next
```

After the first assignment, `head` and `list` have the same type and the same value. So why did we create a new variable?

The reason is that the two variables play different roles. We think of `head` as a reference to a single node, and we think of `list` as a reference to the first node of a list. These "roles" are not part of the program; they are in the mind of the programmer.

In general we can't tell by looking at a program what role a variable plays. This ambiguity can be useful, but it can also make programs difficult to read. We often use variable names like `node` and `list` to document how we intend to use a variable and sometimes create additional variables to disambiguate.

We could have written `printBackward` without `head` and `tail`, which makes it more concise but possibly less clear:

```
def printBackward(list) :
  if list == None : return
  printBackward(list.next)
  print list,
```

Looking at the two function calls, we have to remember that `printBackward` treats its argument as a collection and `print` treats its argument as a single object.

The **fundamental ambiguity theorem** describes the ambiguity that is inherent in a reference to a node:

> **A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.**

17.7 Modifying lists

There are two ways to modify a linked list. Obviously, we can change the cargo of one of the nodes, but the more interesting operations are the ones that add, remove, or reorder the nodes.

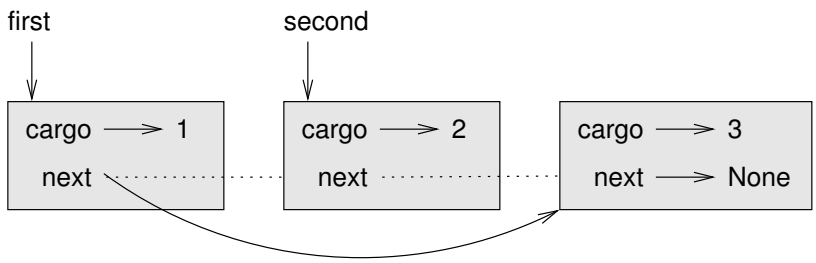
As an example, let's write a function that removes the second node in the list and returns a reference to the removed node:

```
def removeSecond(list):
  if list == None: return
  first = list
  second = list.next
  # make the first node refer to the third
  first.next = second.next
  # separate the second node from the rest of the list
  second.next = None
  return second
```

Again, we are using temporary variables to make the code more readable. Here is how to use this function:

```
>>> printList(node1)
1 2 3
>>> removed = removeSecond(node1)
>>> printList(removed)
2
>>> printList(node1)
1 3
```

This state diagram shows the effect of the operation:



What happens if you invoke this function and pass a list with only one element (a **singleton**)? What happens if you pass the empty list as an argument? Is there a precondition for this function? If so, fix the function to handle a violation of the precondition in a reasonable way.

17.8 Wrappers and helpers

It is often useful to divide a list operation into two functions. For example, to print a list backward in the format `[3 2 1]` we can use the `printBackward` function to print `3 2 1` but we need a separate function to print the brackets. Let's call it `printBackwardNicely`:

```
def printBackwardNicely(list) :
  print "[",
  printBackward(list)
  print "]",
```

Again, it is a good idea to check functions like this to see if they work with special cases like an empty list or a singleton.

When we use this function elsewhere in the program, we invoke `printBackwardNicely` directly, and it invokes `printBackward` on our behalf. In that sense, `printBackwardNicely` acts as a **wrapper**, and it uses `printBackward` as a **helper**.

17.9 The `LinkedList` class

There are some subtle problems with the way we have been implementing lists. In a reversal of cause and effect, we'll propose an alternative implementation first and then explain what problems it solves.

First, we'll create a new class called `LinkedList`. Its attributes are an integer that contains the length of the list and a reference to the first node. `LinkedList` objects serve as handles for manipulating lists of `Node` objects:

```
class LinkedList :
  def __init__(self) :
    self.length = 0
    self.head   = None
```

One nice thing about the `LinkedList` class is that it provides a natural place to put wrapper functions like `printBackwardNicely`, which we can make a method of the `LinkedList` class:

```
class LinkedList:
  ...
  def printBackward(self):
    print "[",
    if self.head != None:
      self.head.printBackward()
    print "]",

class Node:
  ...
  def printBackward(self):
    if self.next != None:
```

```
      tail = self.next
      tail.printBackward()
    print self.cargo,
```

Just to make things confusing, we renamed `printBackwardNicely`. Now there are two methods named `printBackward`: one in the `Node` class (the helper); and one in the `LinkedList` class (the wrapper). When the wrapper invokes `self.head.printBackward`, it is invoking the helper, because `self.head` is a `Node` object.

Another benefit of the `LinkedList` class is that it makes it easier to add or remove the first element of a list. For example, `addFirst` is a method for `LinkedLists`; it takes an item of cargo as an argument and puts it at the beginning of the list:

```
class LinkedList:
  ...
  def addFirst(self, cargo):
    node = Node(cargo)
    node.next = self.head
    self.head = node
    self.length = self.length + 1
```

As usual, you should check code like this to see if it handles the special cases. For example, what happens if the list is initially empty?

17.10 Invariants

Some lists are "well formed"; others are not. For example, if a list contains a loop, it will cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the `length` value in the `LinkedList` object should be equal to the actual number of nodes in the list.

Requirements like these are called **invariants** because, ideally, they should be true of every object all the time. Specifying invariants for objects is a useful programming practice because it makes it easier to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are times when they are violated. For example, in the middle of `addFirst`, after we have added the node but before we have incremented `length`, the invariant is violated. This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while. Normally, we require that every method that violates an invariant must restore the invariant.

If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.

17.11 Glossary

embedded reference: A reference stored in an attribute of an object.

linked list: A data structure that implements a collection using a sequence of linked nodes.

node: An element of a list, usually implemented as an object that contains a reference to another object of the same type.

cargo: An item of data contained in a node.

link: An embedded reference used to link one object to another.

precondition: An assertion that must be true in order for a method to work correctly.

fundamental ambiguity theorem: A reference to a list node can be treated as a single object or as the first in a list of nodes.

singleton: A linked list with a single node.

wrapper: A method that acts as a middleman between a caller and a helper method, often making the method easier or less error-prone to invoke.

helper: A method that is not invoked directly by a caller but is used by another method to perform part of an operation.

invariant: An assertion that should be true of an object at all times (except perhaps while the object is being modified).

Chapter 18

Stacks

18.1 Abstract data types

The data types you have seen so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the `Card` class represents a card using two integers. As we discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An **abstract data type**, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

- It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.
- Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.
- Well-known ADTs, such as the Stack ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.
- The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the **client** code, from the code that implements the ADT, called the **provider** code.

18.2 The Stack ADT

In this chapter, we will look at one common ADT, the **stack**. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include dictionaries and lists.

An ADT is defined by the operations that can be performed on it, which is called an **interface**. The interface for a stack consists of these operations:

`__init__`**:** Initialize a new empty stack.

`push`**:** Add a new item to the stack.

`pop`**:** Remove and return an item from the stack. The item that is returned is always the last one that was added.

`isEmpty`**:** Check whether the stack is empty.

A stack is sometimes called a "last in, first out" or LIFO data structure, because the last item added is the first to be removed.

18.3 Implementing stacks with Python lists

The list operations that Python provides are similar to the operations that define a stack. The interface isn't exactly what it is supposed to be, but we can write code to translate from the Stack ADT to the built-in operations.

This code is called an **implementation** of the Stack ADT. In general, an implementation is a set of methods that satisfy the syntactic and semantic requirements of an interface.

Here is an implementation of the Stack ADT that uses a Python list:

```
class Stack :
  def __init__(self) :
    self.items = []

  def push(self, item) :
    self.items.append(item)
```

```
  def pop(self) :
    return self.items.pop()

  def isEmpty(self) :
    return (self.items == [])
```

A `Stack` object contains an attribute named `items` that is a list of items in the stack. The initialization method sets `items` to the empty list.

To push a new item onto the stack, `push` appends it onto `items`. To pop an item off the stack, `pop` uses the homonymous[1] list method to remove and return the last item on the list.

Finally, to check if the stack is empty, `isEmpty` compares `items` to the empty list.

An implementation like this, in which the methods consist of simple invocations of existing methods, is called a **veneer**. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

18.4 Pushing and popping

A stack is a **generic data structure**, which means that we can add any type of item to it. The following example pushes two integers and a string onto the stack:

```
>>> s = Stack()
>>> s.push(54)
>>> s.push(45)
>>> s.push("+")
```

We can use `isEmpty` and `pop` to remove and print all of the items on the stack:

```
while not s.isEmpty() :
  print s.pop(),
```

The output is `+ 45 54`. In other words, we just used a stack to print the items backward! Granted, it's not the standard format for printing a list, but by using a stack, it was remarkably easy to do.

You should compare this bit of code to the implementation of `printBackward` in Section 17.4. There is a natural parallel between the recursive version

[1] same-named

of `printBackward` and the stack algorithm here. The difference is that `printBackward` uses the runtime stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, except that it uses a `Stack` object instead of the runtime stack.

18.5 Using a stack to evaluate postfix

In most programming languages, mathematical expressions are written with the operator between the two operands, as in `1+2`. This format is called **infix**. An alternative used by some calculators is called **postfix**. In postfix, the operator follows the operands, as in `1 2 +`.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack:

- Starting at the beginning of the expression, get one term (operator or operand) at a time.
  - If the term is an operand, push it on the stack.
  - If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.
- When you get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

As an exercise, apply this algorithm to the expression `1 2 + 3 *`.

This example demonstrates one of the advantages of postfix—there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write `(1 + 2) * 3`.

As an exercise, write a postfix expression that is equivalent to `1 + 2 * 3`.

18.6 Parsing

To implement the previous algorithm, we need to be able to traverse a string and break it into operands and operators. This process is an example of **parsing**, and the results—the individual chunks of the string—are called **tokens**. You might remember these words from Chapter 1.

Python provides a `split` method in both the `string` and `re` (regular expression) modules. The function `string.split` splits a string into a list using a single character as a **delimiter**. For example:

```
>>> import string
>>> string.split("Now is the time"," ")
['Now', 'is', 'the', 'time']
```

In this case, the delimiter is the space character, so the string is split at each space.

The function `re.split` is more powerful, allowing us to provide a regular expression instead of a delimiter. A regular expression is a way of specifying a set of strings. For example, `[A-z]` is the set of all letters and `[0-9]` is the set of all digits. The `^` operator negates a set, so `[^0-9]` is the set of every character that is not a digit, which is exactly the set we want to use to split up postfix expressions:

```
>>> import re
>>> re.split("([^0-9])", "123+456*/")
['123', '+', '456', '*', '', '/', '']
```

Notice that the order of the arguments is different from `string.split`; the delimiter comes before the string.

The resulting list includes the operands `123` and `456` and the operators `*` and `/`. It also includes two empty strings that are inserted as "phantom operands," whenever an operator appears without a number before or after it.

18.7 Evaluating postfix

To evaluate a postfix expression, we will use the parser from the previous section and the algorithm from the section before that. To keep things simple, we'll start with an evaluator that only implements the operators `+` and `*`:

```
def evalPostfix(expr):
  import re
  tokenList = re.split("([^0-9])", expr)
  stack = Stack()
  for token in tokenList:
    if  token == '' or token == ' ':
      continue
    if  token == '+':
      sum = stack.pop() + stack.pop()
      stack.push(sum)
    elif token == '*':
      product = stack.pop() * stack.pop()
      stack.push(product)
    else:
      stack.push(int(token))
  return stack.pop()
```

The first condition takes care of spaces and empty strings. The next two conditions handle operators. We assume, for now, that anything else must be an operand. Of course, it would be better to check for erroneous input and report an error message, but we'll get to that later.

Let's test it by evaluating the postfix form of `(56+47)*2`:

```
>>> print evalPostfix ("56 47 + 2 *")
206
```

That's close enough.

18.8 Clients and providers

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct—in accord with the specification of the ADT—and not how it will be used.

Conversely, the client *assumes* that the implementation of the ADT is correct and doesn't worry about the details. When you are using one of Python's built-in types, you have the luxury of thinking exclusively as a client.

Of course, when you implement an ADT, you also have to write client code to test it. In that case, you play both roles, which can be confusing. You should make some effort to keep track of which role you are playing at any moment.

18.9 Glossary

abstract data type (ADT): A data type (usually a collection of objects) that is defined by a set of operations but that can be implemented in a variety of ways.

interface: The set of operations that define an ADT.

implementation: Code that satisfies the syntactic and semantic requirements of an interface.

client: A program (or the person who wrote it) that uses an ADT.

provider: The code (or the person who wrote it) that implements an ADT.

veneer: A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it improves or standardizes the interface seen by the client.

generic data structure: A kind of data structure that can contain data of any type.

infix: A way of writing mathematical expressions with the operators between the operands.

postfix: A way of writing mathematical expressions with the operators after the operands.

parse: To read a string of characters or tokens and analyze its grammatical structure.

token: A set of characters that are treated as a unit for purposes of parsing, such as the words in a natural language.

delimiter: A character that is used to separate tokens, such as punctuation in a natural language.

Chapter 19

Queues

This chapter presents two ADTs: the Queue and the Priority Queue. In real life, a **queue** is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served. There are exceptions, though. At airports, customers whose flights are leaving soon are sometimes taken from the middle of the queue. At supermarkets, a polite customer might let someone with only a few items go first.

The rule that determines who goes next is called the **queueing policy**. The simplest queueing policy is called **FIFO**, for "first-in-first-out." The most general queueing policy is **priority queueing**, in which each customer is assigned a priority and the customer with the highest priority goes first, regardless of the order of arrival. We say this is the most general policy because the priority can be based on anything: what time a flight leaves; how many groceries the customer has; or how important the customer is. Of course, not all queueing policies are "fair," but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations. The difference is in the semantics of the operations: a queue uses the FIFO policy; and a priority queue (as the name suggests) uses the priority queueing policy.

19.1 The Queue ADT

The Queue ADT is defined by the following operations:

`__init__`: Initialize a new empty queue.

`insert`: Add a new item to the queue.

`remove`: Remove and return an item from the queue. The item that is returned is the first one that was added.

`isEmpty`: Check whether the queue is empty.

19.2 Linked Queue

The first implementation of the Queue ADT we will look at is called a **linked queue** because it is made up of linked `Node` objects. Here is the class definition:

```
class Queue:
  def __init__(self):
    self.length = 0
    self.head = None

  def isEmpty(self):
    return (self.length == 0)

  def insert(self, cargo):
    node = Node(cargo)
    node.next = None
    if self.head == None:
      # if list is empty the new node goes first
      self.head = node
    else:
      # find the last node in the list
      last = self.head
      while last.next: last = last.next
      # append the new node
      last.next = node
    self.length = self.length + 1

  def remove(self):
    cargo = self.head.cargo
    self.head = self.head.next
    self.length = self.length - 1
    return cargo
```

The methods `isEmpty` and `remove` are identical to the `LinkedList` methods `isEmpty` and `removeFirst`. The `insert` method is new and a bit more complicated.

We want to insert new items at the end of the list. If the queue is empty, we just set `head` to refer to the new node.

Otherwise, we traverse the list to the last node and tack the new node on the end. We can identify the last node because its `next` attribute is `None`.

There are two invariants for a properly formed `Queue` object. The value of `length` should be the number of nodes in the queue, and the last node should have `next` equal to `None`. Convince yourself that this method preserves both invariants.

19.3 Performance characteristics

Normally when we invoke a method, we are not concerned with the details of its implementation. But there is one "detail" we might want to know—the performance characteristics of the method. How long does it take, and how does the run time change as the number of items in the collection increases?

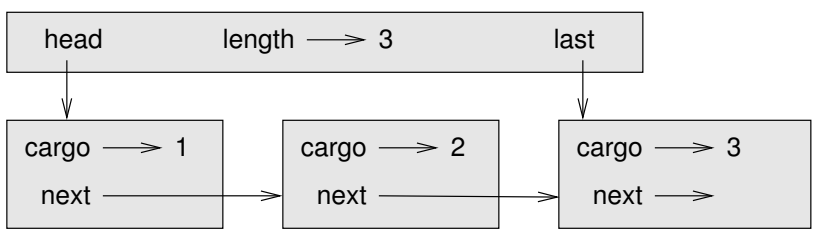
First look at `remove`. There are no loops or function calls here, suggesting that the runtime of this method is the same every time. Such a method is called a **constant time** operation. In reality, the method might be slightly faster when the list is empty since it skips the body of the conditional, but that difference is not significant.

The performance of `insert` is very different. In the general case, we have to traverse the list to find the last element.

This traversal takes time proportional to the length of the list. Since the runtime is a linear function of the length, this method is called **linear time**. Compared to constant time, that's very bad.

19.4 Improved Linked Queue

We would like an implementation of the Queue ADT that can perform all operations in constant time. One way to do that is to modify the Queue class so that it maintains a reference to both the first and the last node, as shown in the figure:



The `ImprovedQueue` implementation looks like this:

```
class ImprovedQueue:
  def __init__(self):
    self.length = 0
    self.head   = None
    self.last   = None

  def isEmpty(self):
    return (self.length == 0)
```

So far, the only change is the attribute `last`. It is used in `insert` and `remove` methods:

```
class ImprovedQueue:
  ...
  def insert(self, cargo):
    node = Node(cargo)
    node.next = None
    if self.length == 0:
      # if list is empty, the new node is head and last
      self.head = self.last = node
    else:
      # find the last node
      last = self.last
      # append the new node
      last.next = node
      self.last = node
    self.length = self.length + 1
```

Since `last` keeps track of the last node, we don't have to search for it. As a result, this method is constant time.

There is a price to pay for that speed. We have to add a special case to `remove` to set `last` to `None` when the last node is removed:

```
class ImprovedQueue:
  ...
  def remove(self):
    cargo     = self.head.cargo
    self.head = self.head.next
    self.length = self.length - 1
    if self.length == 0:
      self.last = None
    return cargo
```

This implementation is more complicated than the Linked Queue implementation,

and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal—both `insert` and `remove` are constant time operations.

> *As an exercise, write an implementation of the Queue ADT using a Python list. Compare the performance of this implementation to the* `ImprovedQueue` *for a range of queue lengths.*

19.5 Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. Again, the interface is:

`__init__`**:** Initialize a new empty queue.

`insert`**:** Add a new item to the queue.

`remove`**:** Remove and return an item from the queue. The item that is returned is the one with the highest priority.

`isEmpty`**:** Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is the item in the queue that has the highest priority. What the priorities are and how they compare to each other are not specified by the Priority Queue implementation. It depends on which items are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might go from highest to lowest, but if they are golf scores, we would go from lowest to highest. As long as we can compare the items in the queue, we can find and remove the one with the highest priority.

This implementation of Priority Queue has as an attribute a Python list that contains the items in the queue.

```
class PriorityQueue:
  def __init__(self):
    self.items = []

  def isEmpty(self):
    return self.items == []

  def insert(self, item):
    self.items.append(item)
```

The initialization method, `isEmpty`, and `insert` are all veneers on list operations. The only interesting method is `remove`:

```
class PriorityQueue:
  ...
  def remove(self):
    maxi = 0
    for i in range(1,len(self.items)):
      if self.items[i] > self.items[maxi]:
        maxi = i
    item = self.items[maxi]
    self.items[maxi:maxi+1] = []
    return item
```

At the beginning of each iteration, `maxi` holds the index of the biggest item (highest priority) we have seen *so far*. Each time through the loop, the program compares the `i`-eth item to the champion. If the new item is bigger, the value of `maxi` is set to `i`.

When the `for` statement completes, `maxi` is the index of the biggest item. This item is removed from the list and returned.

Let's test the implementation:

```
>>> q = PriorityQueue()
>>> q.insert(11)
>>> q.insert(12)
>>> q.insert(14)
>>> q.insert(13)
>>> while not q.isEmpty(): print q.remove()
14
13
12
11
```

If the queue contains simple numbers or strings, they are removed in numerical or alphabetical order, from highest to lowest. Python can find the biggest integer or string because it can compare them using the built-in comparison operators.

If the queue contains an object type, it has to provide a `__cmp__` method. When `remove` uses the `>` operator to compare items, it invokes the `__cmp__` for one of the items and passes the other as an argument. As long as the `__cmp__` method works correctly, the Priority Queue will work.

19.6 The Golfer class

As an example of an object with an unusual definition of priority, let's implement a class called Golfer that keeps track of the names and scores of golfers. As usual, we start by defining __init__ and __str__:

```
class Golfer:
  def __init__(self, name, score):
    self.name = name
    self.score= score

  def __str__(self):
    return "%-16s: %d" % (self.name, self.score)
```

__str__ uses the format operator to put the names and scores in neat columns.

Next we define a version of __cmp__ where the lowest score gets highest priority. As always, __cmp__ returns 1 if self is "greater than" other, -1 if self is "less than" other, and 0 if they are equal.

```
class Golfer:
  ...
  def __cmp__(self, other):
    if self.score < other.score: return  1   # less is more
    if self.score > other.score: return -1
    return 0
```

Now we are ready to test the priority queue with the Golfer class:

```
>>> tiger = Golfer("Tiger Woods",    61)
>>> phil  = Golfer("Phil Mickelson", 72)
>>> hal   = Golfer("Hal Sutton",     69)
>>>
>>> pq = PriorityQueue()
>>> pq.insert(tiger)
>>> pq.insert(phil)
>>> pq.insert(hal)
>>> while not pq.isEmpty(): print pq.remove()
Tiger Woods    : 61
Hal Sutton     : 69
Phil Mickelson : 72
```

As an exercise, write an implementation of the Priority Queue ADT using a linked list. You should keep the list sorted so that removal is a constant time operation. Compare the performance of this implementation with the Python list implementation.

19.7 Glossary

queue: An ordered set of objects waiting for a service of some kind.

Queue: An ADT that performs the operations one might perform on a queue.

queueing policy: The rules that determine which member of a queue is removed next.

FIFO: "First In, First Out," a queueing policy in which the first member to arrive is the first to be removed.

priority queue: A queueing policy in which each member has a priority determined by external factors. The member with the highest priority is the first to be removed.

Priority Queue: An ADT that defines the operations one might perform on a priority queue.

linked queue: An implementation of a queue using a linked list.

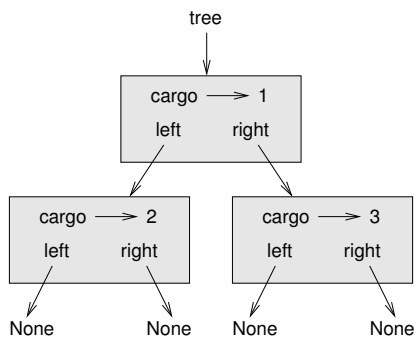
constant time: An operation whose runtime does not depend on the size of the data structure.

linear time: An operation whose runtime is a linear function of the size of the data structure.

Chapter 20

Trees

Like linked lists, trees are made up of nodes. A common kind of tree is a **binary tree**, in which each node contains a reference to two other nodes (possibly null). These references are referred to as the left and right subtrees. Like list nodes, tree nodes also contain cargo. A state diagram for a tree looks like this:



To avoid cluttering up the picture, we often omit the `Nones`.

The top of the tree (the node `tree` refers to) is called the **root**. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called **leaves**. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in another metaphor—the family tree. The top node is sometimes called a **parent** and the nodes it refers to are its **children**. Nodes with the same parent are called **siblings**.

Finally, there is a geometric vocabulary for talking about trees. We already mentioned left and right, but there is also "up" (toward the parent/root) and "down" (toward the children/leaves). Also, all of the nodes that are the same distance from the root comprise a **level** of the tree.

We probably don't need three metaphors for talking about trees, but there they are.

Like linked lists, trees are recursive data structures because they are defined recursively.

A tree is either:

- the empty tree, represented by `None`, or
- a node that contains an object reference and two tree references.

20.1 Building trees

The process of assembling a tree is similar to the process of assembling a linked list. Each constructor invocation builds a single node.

```
class Tree:
  def __init__(self, cargo, left=None, right=None):
    self.cargo = cargo
    self.left  = left
    self.right = right

  def __str__(self):
    return str(self.cargo)
```

The `cargo` can be any type, but the arguments for `left` and `right` should be tree nodes. `left` and `right` are optional; the default value is `None`.

To print a node, we just print the cargo.

One way to build a tree is from the bottom up. Allocate the child nodes first:

```
left = Tree(2)
right = Tree(3)
```

Then create the parent node and link it to the children:

```
tree = Tree(1, left, right);
```

We can write this code more concisely by nesting constructor invocations:

```
>>> tree = Tree(1, Tree(2), Tree(3))
```

Either way, the result is the tree at the beginning of the chapter.

20.2 Traversing trees

Any time you see a new data structure, your first question should be, "How do I traverse it?" The most natural way to traverse a tree is recursively. For example, if the tree contains integers as cargo, this function returns their sum:

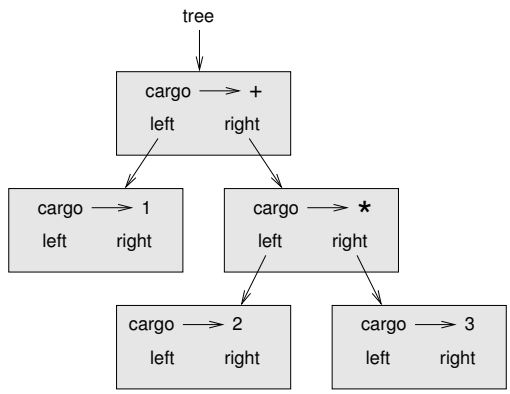
```
def total(tree):
  if tree == None: return 0
  return total(tree.left) + total(tree.right) + tree.cargo
```

The base case is the empty tree, which contains no cargo, so the sum is 0. The recursive step makes two recursive calls to find the sum of the child trees. When the recursive calls complete, we add the cargo of the parent and return the total.

20.3 Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the computation unambiguously. For example, the infix expression `1 + 2 * 3` is ambiguous unless we know that the multiplication happens before the addition.

This expression tree represents the same computation:



The nodes of an expression tree can be operands like `1` and `2` or operators like `+` and `*`. Operands are leaf nodes; operator nodes contain references to their operands. (All of these operators are **binary**, meaning they have exactly two operands.)

We can build this tree like this:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
```

Looking at the figure, there is no question what the order of operations is; the multiplication happens first in order to compute the second operand of the addition.

Expression trees have many uses. The example in this chapter uses trees to translate expressions to postfix, prefix, and infix. Similar trees are used inside compilers to parse, optimize, and translate programs.

20.4 Tree traversal

We can traverse an expression tree and print the contents like this:

```
def printTree(tree):
  if tree == None: return
  print tree.cargo,
  printTree(tree.left)
  printTree(tree.right)
```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, and then print the entire right subtree. This way of traversing a tree is called a **preorder**, because the contents of the root appear *before* the contents of the children. For the previous example, the output is:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
>>> printTree(tree)
+ 1 * 2 3
```

This format is different from both postfix and infix; it is another notation called **prefix**, in which the operators appear before their operands.

You might suspect that if you traverse the tree in a different order, you will get the expression in a different notation. For example, if you print the subtrees first and then the root node, you get:

```
def printTreePostorder(tree):
  if tree == None: return
  printTreePostorder(tree.left)
  printTreePostorder(tree.right)
  print tree.cargo,
```

The result, `1 2 3 * +`, is in postfix! This order of traversal is called **postorder**.

Finally, to traverse a tree **inorder**, you print the left tree, then the root, and then the right tree:

```
def printTreeInorder(tree):
  if tree == None: return
  printTreeInorder(tree.left)
  print tree.cargo,
  printTreeInorder(tree.right)
```

The result is `1 + 2 * 3`, which is the expression in infix.

To be fair, we should point out that we have omitted an important complication. Sometimes when we write an expression in infix, we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

> *As an exercise, modify* `printTreeInorder` *so that it puts parentheses around every operator and pair of operands. Is the output correct and unambiguous? Are the parentheses always necessary?*

If we do an inorder traversal and keep track of what level in the tree we are on, we can generate a graphical representation of a tree:

```
def printTreeIndented(tree, level=0):
  if tree == None: return
  printTreeIndented(tree.right, level+1)
  print '  '*level + str(tree.cargo)
  printTreeIndented(tree.left, level+1)
```

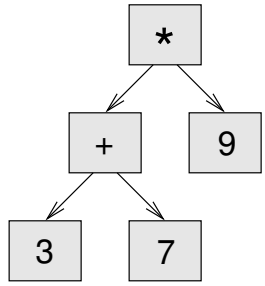
The parameter `level` keeps track of where we are in the tree. By default, it is initially 0. Each time we make a recursive call, we pass `level+1` because the child's level is always one greater than the parent's. Each item is indented by two spaces per level. The result for the example tree is:

```
>>> printTreeIndented(tree)
    3
  *
    2
+
  1
```

If you look at the output sideways, you see a simplified version of the original figure.

20.5 Building an expression tree

In this section, we parse infix expressions and build the corresponding expression trees. For example, the expression `(3+7)*9` yields the following tree:



Notice that we have simplified the diagram by leaving out the names of the attributes.

The parser we will write handles expressions that include numbers, parentheses, and the operators `+` and `*`. We assume that the input string has already been tokenized into a Python list. The token list for `(3+7)*9` is:

```
['(', 3, '+', 7, ')', '*', 9, 'end']
```

The `end` token is useful for preventing the parser from reading past the end of the list.

> *As an exercise, write a function that takes an expression string and returns a token list.*

The first function we'll write is `getToken`, which takes a token list and an expected token as arguments. It compares the expected token to the first token on the list: if they match, it removes the token from the list and returns true; otherwise, it returns false:

```
def getToken(tokenList, expected):
  if tokenList[0] == expected:
    del tokenList[0]
    return True
  else:
    return False
```

Since `tokenList` refers to a mutable object, the changes made here are visible to any other variable that refers to the same object.

The next function, `getNumber`, handles operands. If the next token in `tokenList` is a number, `getNumber` removes it and returns a leaf node containing the number; otherwise, it returns `None`.

```
def getNumber(tokenList):
  x = tokenList[0]
  if not isinstance(x, int): return None
  del tokenList[0]
  return Tree (x, None, None)
```

Before continuing, we should test `getNumber` in isolation. We assign a list of numbers to `tokenList`, extract the first, print the result, and print what remains of the token list:

```
>>> tokenList = [9, 11, 'end']
>>> x = getNumber(tokenList)
>>> printTreePostorder(x)
9
>>> print tokenList
[11, 'end']
```

The next method we need is `getProduct`, which builds an expression tree for products. A simple product has two numbers as operands, like `3 * 7`.

Here is a version of `getProduct` that handles simple products.

```
def getProduct(tokenList):
  a = getNumber(tokenList)
  if getToken(tokenList, '*'):
    b = getNumber(tokenList)
    return Tree ('*', a, b)
  else:
    return a
```

Assuming that `getNumber` succeeds and returns a singleton tree, we assign the first operand to `a`. If the next character is `*`, we get the second number and build an expression tree with `a`, `b`, and the operator.

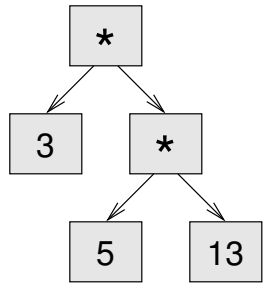
If the next character is anything else, then we just return the leaf node with `a`. Here are two examples:

```
>>> tokenList = [9, '*', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9 11 *
```

```
>>> tokenList = [9, '+', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9
```

The second example implies that we consider a single operand to be a kind of product. This definition of "product" is counterintuitive, but it turns out to be useful.

Now we have to deal with compound products, like like `3 * 5 * 13`. We treat this expression as a product of products, namely `3 * (5 * 13)`. The resulting tree is:



With a small change in `getProduct`, we can handle an arbitrarily long product:

```
def getProduct(tokenList):
  a = getNumber(tokenList)
  if getToken(tokenList, '*'):
    b = getProduct(tokenList)       # this line changed
    return Tree ('*', a, b)
  else:
    return a
```

In other words, a product can be either a singleton or a tree with `*` at the root, a number on the left, and a product on the right. This kind of recursive definition should be starting to feel familiar.

Let's test the new version with a compound product:

```
>>> tokenList = [2, '*', 3, '*', 5 , '*', 7, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
2 3 5 7 * * *
```

Next we will add the ability to parse sums. Again, we use a slightly counterintuitive definition of "sum." For us, a sum can be a tree with `+` at the root, a product on the left, and a sum on the right. Or, a sum can be just a product.

If you are willing to play along with this definition, it has a nice property: we can represent any expression (without parentheses) as a sum of products. This property is the basis of our parsing algorithm.

`getSum` tries to build a tree with a product on the left and a sum on the right. But if it doesn't find a `+`, it just builds a product.

```
def getSum(tokenList):
  a = getProduct(tokenList)
  if getToken(tokenList, '+'):
    b = getSum(tokenList)
    return Tree ('+', a, b)
  else:
    return a
```

Let's test it with `9 * 11 + 5 * 7`:

```
>>> tokenList = [9, '*', 11, '+', 5, '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 * 5 7 * +
```

We are almost done, but we still have to handle parentheses. Anywhere in an expression where there can be a number, there can also be an entire sum enclosed in parentheses. We just need to modify `getNumber` to handle **subexpressions**:

```
def getNumber(tokenList):
  if getToken(tokenList, '('):
    x = getSum(tokenList)         # get the subexpression
    getToken(tokenList, ')')      # remove the closing parenthesis
    return x
  else:
    x = tokenList[0]
    if not isinstance(x, int): return None
    tokenList[0:1] = []
    return Tree (x, None, None)
```

Let's test this code with `9 * (11 + 5) * 7`:

```
>>> tokenList = [9, '*', '(', 11, '+', 5, ')', '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 5 + 7 * *
```

The parser handled the parentheses correctly; the addition happens before the multiplication.

In the final version of the program, it would be a good idea to give `getNumber` a name more descriptive of its new role.

20.6 Handling errors

Throughout the parser, we've been assuming that expressions are well-formed. For example, when we reach the end of a subexpression, we assume that the next character is a close parenthesis. If there is an error and the next character is something else, we should deal with it.

```
def getNumber(tokenList):
  if getToken(tokenList, '('):
    x = getSum(tokenList)
    if not getToken(tokenList, ')'):
      raise ValueError, 'missing parenthesis'
    return x
  else:
    # the rest of the function omitted
```

The `raise` statement creates an exception; in this case a `ValueError`. If the function that called `getNumber`, or one of the other functions in the traceback, handles the exception, then the program can continue. Otherwise, Python will print an error message and quit.

> *As an exercise, find other places in these functions where errors can occur and add appropriate* `raise` *statements. Test your code with improperly formed expressions.*

20.7 The animal tree

In this section, we develop a small program that uses a tree to represent a knowledge base.

The program interacts with the user to create a tree of questions and animal names. Here is a sample run:

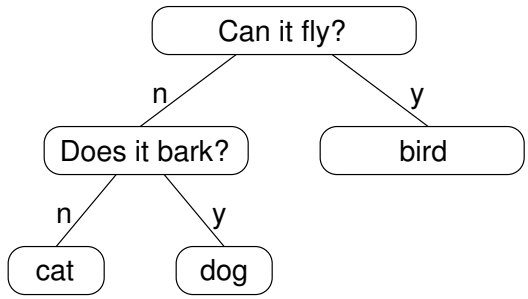
```
Are you thinking of an animal? y
Is it a bird? n
What is the animals name? dog
What question would distinguish a dog from a bird? Can it fly
If the animal were dog the answer would be? n

Are you thinking of an animal? y
Can it fly? n
Is it a dog? n
What is the animals name? cat
What question would distinguish a cat from a dog? Does it bark
If the animal were cat the answer would be? n

Are you thinking of an animal? y
Can it fly? n
Does it bark? y
Is it a dog? y
I rule!

Are you thinking of an animal? n
```

Here is the tree this dialog builds:



At the beginning of each round, the program starts at the top of the tree and asks the first question. Depending on the answer, it moves to the left or right child and continues until it gets to a leaf node. At that point, it makes a guess. If the guess is not correct, it asks the user for the name of the new animal and a question that distinguishes the (bad) guess from the new animal. Then it adds a node to the tree with the new question and the new animal.

Here is the code:

```
def animal():
  # start with a singleton
  root = Tree("bird")

  # loop until the user quits
  while True:
    print
    if not yes("Are you thinking of an animal? "): break

    # walk the tree
    tree = root
    while tree.getLeft() != None:
      prompt = tree.getCargo() + "? "
      if yes(prompt):
        tree = tree.getRight()
      else:
        tree = tree.getLeft()

    # make a guess
    guess = tree.getCargo()
    prompt = "Is it a " + guess + "? "
    if yes(prompt):
      print "I rule!"
      continue

    # get new information
    prompt  = "What is the animal's name? "
    animal  = raw_input(prompt)
    prompt  = "What question would distinguish a %s from a %s? "
    question = raw_input(prompt % (animal,guess))

    # add new information to the tree
    tree.setCargo(question)
    prompt = "If the animal were %s the answer would be? "
    if yes(prompt % animal):
      tree.setLeft(Tree(guess))
      tree.setRight(Tree(animal))
    else:
      tree.setLeft(Tree(animal))
      tree.setRight(Tree(guess))
```

The function `yes` is a helper; it prints a prompt and then takes input from the user. If the response begins with *y* or *Y*, the function returns true:

```
def yes(ques):
  from string import lower
  ans = lower(raw_input(ques))
  return (ans[0] == 'y')
```

The condition of the outer loop is `True`, which means it will continue until the `break` statement executes, if the user is not thinking of an animal.

The inner `while` loop walks the tree from top to bottom, guided by the user's responses.

When a new node is added to the tree, the new question replaces the cargo, and the two children are the new animal and the original cargo.

One shortcoming of the program is that when it exits, it forgets everything you carefully taught it!

> *As an exercise, think of various ways you might save the knowledge tree in a file. Implement the one you think is easiest.*

20.8 Glossary

binary tree: A tree in which each node refers to zero, one, or two dependent nodes.

root: The topmost node in a tree, with no parent.

leaf: A bottom-most node in a tree, with no children.

parent: The node that refers to a given node.

child: One of the nodes referred to by a node.

siblings: Nodes that share a common parent.

level: The set of nodes equidistant from the root.

binary operator: An operator that takes two operands.

subexpression: An expression in parentheses that acts as a single operand in a larger expression.

preorder: A way to traverse a tree, visiting each node before its children.

prefix notation: A way of writing a mathematical expression with each operator appearing before its operands.

postorder: A way to traverse a tree, visiting the children of each node before the node itself.

inorder: A way to traverse a tree, visiting the left subtree, then the root, and then the right subtree.

Appendix A

Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a `def` statement yields the somewhat redundant message `SyntaxError: invalid syntax`.
- Runtime errors are produced by the runtime system if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes a runtime error of "maximum recursion depth exceeded."
- Semantic errors are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

A.1 Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages

are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.

2. Check that you have a colon at the end of the header of every compound statement, including `for`, `while`, `if`, and `def` statements.

3. Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.

4. Make sure that any strings in the code have matching quotation marks.

5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an `invalid token` error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!

6. An unclosed bracket—`(`, `{`, or `[`—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.

7. Check for the classic `=` instead of `==` inside a conditional.

If nothing works, move on to the next section...

A.1.1 I can't get my program to run no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run. If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run (or import) it again. If the compiler doesn't find the new error, there is probably something wrong with the way your environment is set up.

If this happens, one approach is to start again with a new program like "Hello, World!," and make sure you can get a known program to run. Then gradually add the pieces of the new program to the working one.

A.2 Runtime errors

Once your program is syntactically correct, Python can import it and at least start running it. What could possibly go wrong?

A.2.1 My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the "Flow of Execution" section below.

A.2.2 My program hangs.

If a program stops and seems to be doing nothing, we say it is "hanging." Often that means that it is caught in an infinite loop or an infinite recursion.

- If there is a particular loop that you suspect is the problem, add a `print` statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop."

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the "Infinite Loop" section below.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. If that happens, go to the "Infinite Recursion" section below.

  If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the "Infinite Recursion" section.

- If neither of those steps works, start testing other loops and other recursive functions and methods.

- If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the "Flow of Execution" section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while x > 0 and y < 0 :
  # do something to x
  # do something to y

  print  "x: ", x
  print  "y: ", y
  print  "condition: ", (x > 0 and y < 0)
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `false`. If the loop keeps going, you will be able to see the values of `x` and `y`, and you might figure out why they are not being updated correctly.

Infinite Recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a `print` statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of Execution

If you are not sure how the flow of execution is moving through your program, add `print` statements to the beginning of each function with a message like "entering function `foo`," where `foo` is the name of the function.

Now when you run the program, it will print a trace of each function as it is invoked.

A.2.3 When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked *that*, and so on. In other words, it traces the path of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError: You are trying to use a variable that doesn't exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.

TypeError: There are several possible causes:

- You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
- There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.

- You are passing the wrong number of arguments to a function or method. For methods, look at the method definition and check that the first parameter is `self`. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

KeyError: You are trying to access an element of a dictionary using a key value that the dictionary does not contain.

AttributeError: You are trying to access an attribute or method that does not exist.

IndexError: The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a `print` statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

A.2.4 I added so many `print` statements I get inundated with output.

One of the problems with using `print` statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out `print` statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting an array, sort a *small* array. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

A.3 Semantic errors

In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed `print` statements is often short compared to setting up the debugger, inserting and removing breakpoints, and "walking" the program to where the error is occurring.

A.3.1 My program doesn't work.

You should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other Python modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

A.3.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].addCard (self.hands[self.findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = self.findNeighbor (i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard (pickedCard)
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Python, you might write:

```
y = x / 2 * math.pi
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
y = x / (2 * math.pi)
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the rules of precedence.

A.3.3 I've got a function or method that doesn't return what I expect.

If you have a `return` statement with a complex expression, you don't have a chance to print the `return` value before returning. Again, you can use a temporary variable. For example, instead of:

```
return self.hands[i].removeMatches()
```

you could write:

```
count = self.hands[i].removeMatches()
return count
```

Now you have the opportunity to display the value of `count` before returning.

A.3.4 I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these effects:

- Frustration and/or rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backward").
- Random-walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. We often find bugs when we are away from the computer and let our minds wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

A.3.5 No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you have exhausted the techniques described here. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have `print` statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

- If there is an error message, what is it and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

Appendix B

Creating a new data type

Object-oriented programming languages allow programmers to create new data types that behave much like built-in data types. We will explore this capability by building a `Fraction` class that works very much like the built-in numeric types: integers, longs and floats.

Fractions, also known as rational numbers, are values that can be expressed as a ratio of whole numbers, such as 5/6. The top number is called the numerator and the bottom number is called the denominator.

We start by defining a `Fraction` class with an initialization method that provides the numerator and denominator as integers:

```
class Fraction:
  def __init__(self, numerator, denominator=1):
    self.numerator = numerator
    self.denominator = denominator
```

The denominator is optional. A Fraction with just one parameter represents a whole number. If the numerator is n, we build the Fraction $n/1$.

The next step is to write a `__str__` method that displays fractions in a way that makes sense. The form "numerator/denominator" is natural here:

```
class Fraction:
  ...
  def __str__(self):
    return "%d/%d" % (self.numerator, self.denominator)
```

To test what we have so far, we put it in a file named `Fraction.py` and import it into the Python interpreter. Then we create a fraction object and print it.

```
>>> from Fraction import Fraction
>>> spam = Fraction(5,6)
>>> print "The fraction is", spam
The fraction is 5/6
```

As usual, the `print` command invokes the `__str__` method implicitly.

B.1 Fraction multiplication

We would like to be able to apply the normal addition, subtraction, multiplication, and division operations to fractions. To do this, we can overload the mathematical operators for `Fraction` objects.

We'll start with multiplication because it is the easiest to implement. To multiply fractions, we create a new fraction with a numerator that is the product of the original numerators and a denominator that is a product of the original denominators. `__mul__` is the name Python uses for a method that overloads the `*` operator:

```
class Fraction:
  ...
  def __mul__(self, other):
    return Fraction(self.numerator*other.numerator,
                    self.denominator*other.denominator)
```

We can test this method by computing the product of two fractions:

```
>>> print Fraction(5,6) * Fraction(3,4)
15/24
```

It works, but we can do better! We can extend the method to handle multiplication by an integer. We use the `isinstance` function to test if `other` is an integer and convert it to a fraction if it is.

```
class Fraction:
  ...
  def __mul__(self, other):
    if isinstance(other, int):
      other = Fraction(other)
    return Fraction(self.numerator   * other.numerator,
                    self.denominator * other.denominator)
```

Multiplying fractions and integers now works, but only if the fraction is the left operand:

```
>>> print Fraction(5,6) * 4
20/6
>>> print 4 * Fraction(5,6)
TypeError: __mul__ nor __rmul__ defined for these operands
```

To evaluate a binary operator like multiplication, Python checks the left operand first to see if it provides a `__mul__` that supports the type of the second operand. In this case, the built-in integer operator doesn't support fractions.

Next, Python checks the right operand to see if it provides an `__rmul__` method that supports the first type. In this case, we haven't provided `__rmul__`, so it fails.

On the other hand, there is a simple way to provide `__rmul__`:

```
class Fraction:
  ...
  __rmul__ = __mul__
```

This assignment says that the `__rmul__` is the same as `__mul__`. Now if we evaluate `4 * Fraction(5,6)`, Python invokes `__rmul__` on the `Fraction` object and passes 4 as a parameter:

```
>>> print 4 * Fraction(5,6)
20/6
```

Since `__rmul__` is the same as `__mul__`, and `__mul__` can handle an integer parameter, we're all set.

B.2 Fraction addition

Addition is more complicated than multiplication, but still not too bad. The sum of a/b and c/d is the fraction `(a*d+c*b)/(b*d)`.

Using the multiplication code as a model, we can write `__add__` and `__radd__`:

```
class Fraction:
  ...
  def __add__(self, other):
    if isinstance(other, int):
      other = Fraction(other)
    return Fraction(self.numerator   * other.denominator +
                    self.denominator * other.numerator,
                    self.denominator * other.denominator)

  __radd__ = __add__
```

We can test these methods with `Fraction`s and integers.

```
>>> print Fraction(5,6) + Fraction(5,6)
60/36
>>> print Fraction(5,6) + 3
23/6
>>> print 2 + Fraction(5,6)
17/6
```

The first two examples invoke `__add__`; the last invokes `__radd__`.

B.3 Euclid's algorithm

In the previous example, we computed the sum $5/6 + 5/6$ and got 60/36. That is correct, but it's not the best way to represent the answer. To **reduce** the fraction to its simplest terms, we have to divide the numerator and denominator by their **greatest common divisor (GCD)**, which is 12. The result is 5/3.

In general, whenever we create a new `Fraction` object, we should reduce it by dividing the numerator and denominator by their GCD. If the fraction is already reduced, the GCD is 1.

Euclid of Alexandria (approx. 325–265 BCE) presented an algorithm to find the GCD for two integers m and n:

> If n divides m evenly, then n is the GCD. Otherwise the GCD is the GCD of n and the remainder of m divided by n.

This recursive definition can be expressed concisely as a function:

```
def gcd (m, n):
  if m % n == 0:
    return n
  else:
    return gcd(n, m%n)
```

In the first line of the body, we use the modulus operator to check divisibility. On the last line, we use it to compute the remainder after division.

Since all the operations we've written create new `Fraction`s for the result, we can reduce all results by modifying the initialization method.

```
class Fraction:
  def __init__(self, numerator, denominator=1):
    g = gcd (numerator, denominator)
    self.numerator   =   numerator / g
    self.denominator = denominator / g
```

Now whenever we create a `Fraction`, it is reduced to its simplest form:

```
>>> Fraction(100,-36)
-25/9
```

A nice feature of `gcd` is that if the fraction is negative, the minus sign is always moved to the numerator.

B.4 Comparing fractions

Suppose we have two `Fraction` objects, `a` and `b`, and we evaluate `a == b`. The default implementation of `==` tests for shallow equality, so it only returns true if `a` and `b` are the same object.

More likely, we want to return true if a and b have the same value—that is, deep equality.

We have to teach fractions how to compare themselves. As we saw in Section 15.4, we can overload all the comparison operators at once by supplying a `__cmp__` method.

By convention, the `__cmp__` method returns a negative number if `self` is less than `other`, zero if they are the same, and a positive number if `self` is greater than `other`.

The simplest way to compare fractions is to cross-multiply. If $a/b > c/d$, then $ad > bc$. With that in mind, here is the code for `__cmp__`:

```
class Fraction:
  ...
  def __cmp__(self, other):
    diff = (self.numerator  * other.denominator -
            other.numerator * self.denominator)
    return diff
```

If `self` is greater than `other`, then `diff` will be positive. If `other` is greater, then `diff` will be negative. If they are the same, `diff` is zero.

B.5 Taking it further

Of course, we are not done. We still have to implement subtraction by overriding `__sub__` and division by overriding `__div__`.

One way to handle those operations is to implement negation by overriding `__neg__` and inversion by overriding `__invert__`. Then we can subtract by negating the second operand and adding, and we can divide by inverting the second operand and multiplying.

Next, we have to provide `__rsub__` and `__rdiv__`. Unfortunately, we can't use the same trick we used for addition and multiplication, because subtraction and division are not commutative. We can't just set `__rsub__` and `__rdiv__` equal to `__sub__` and `__div__`. In these operations, the order of the operands makes a difference.

To handle **unary negation**, which is the use of the minus sign with a single operand, we override `__neg__`.

We can compute powers by overriding `__pow__`, but the implementation is a little tricky. If the exponent isn't an integer, then it may not be possible to represent the result as a `Fraction`. For example, `Fraction(2) ** Fraction(1,2)` is the square root of 2, which is an irrational number (it can't be represented as a fraction). So it's not easy to write the most general version of `__pow__`.

There is one other extension to the `Fraction` class that you might want to think about. So far, we have assumed that the numerator and denominator are integers.

> *As an exercise, finish the implementation of the* `Fraction` *class so that it handles subtraction, division and exponentiation.*

B.6 Glossary

greatest common divisor (GCD): The largest positive integer that divides without a remainder into both the numerator and denominator of a fraction.

reduce: To change a fraction into an equivalent form with a GCD of 1.

unary negation: The operation that computes an additive inverse, usually denoted with a leading minus sign. Called "unary" by contrast with the binary minus operation, which is subtraction.

Appendix C

Recommendations for further reading

So where do you go from here? There are many directions to pursue, extending your knowledge of Python specifically and computer science in general.

The examples in this book have been deliberately simple, but they may not have shown off Python's most exciting capabilities. Here is a sampling of extensions to Python and suggestions for projects that use them.

- GUI (graphical user interface) programming lets your program use a windowing environment to interact with the user and display graphics.

  The oldest graphics package for Python is Tkinter, which is based on Jon Ousterhout's Tcl and Tk scripting languages. Tkinter comes bundled with the Python distribution.

  Another popular platform is wxPython, which is essentially a Python veneer over wxWindows, a C++ package which in turn implements windows using native interfaces on Windows and Unix (including Linux) platforms. The windows and controls under wxPython tend to have a more native look and feel than those of Tkinter and are somewhat simpler to program.

  Any type of GUI programming will lead you into event-driven programming, where the user and not the programmer determines the flow of execution. This style of programming takes some getting used to, sometimes forcing you to rethink the whole structure of a program.

- Web programming integrates Python with the Internet. For example, you can build web client programs that open and read a remote web page (almost) as easily as you can open a file on disk. There are also Python modules

that let you access remote files via ftp, and modules to let you send and receive email. Python is also widely used for web server programs to handle input forms.

- Databases are a bit like super files where data is stored in predefined schemas, and relationships between data items let you access the data in various ways. Python has several modules to enable users to connect to various database engines, both Open Source and commercial.

- Thread programming lets you run several threads of execution within a single program. If you have had the experience of using a web browser to scroll the beginning of a page while the browser continues to load the rest of it, then you have a feel for what threads can do.

- When speed is paramount Python extensions may be written in a compiled language like C or C++. Such extensions form the base of many of the modules in the Python library. The mechanics of linking functions and data is somewhat complex. SWIG (Simplified Wrapper and Interface Generator) is a tool to make the process much simpler.

C.1 Python-related web sites and books

Here are the authors' recommendations for Python resources on the web:

- The Python home page at `www.python.org` is the place to start your search for any Python related material. You will find help, documentation, links to other sites and SIG (Special Interest Group) mailing lists that you can join.

- The Open Book Project `www.ibiblio.com/obp` contains not only this book online but also similar books for Java and C++ by Allen Downey. In addition there are *Lessons in Electric Circuits* by Tony R. Kuphaldt, *Getting down with ...*, a set of tutorials on a range of computer science topics, written and edited by high school students, *Python for Fun*, a set of case studies in Python by Chris Meyers, and *The Linux Cookbook* by Michael Stultz, with 300 pages of tips and techniques.

- Finally if you go to Google and use the search string "python -snake -monty" you will get about 750,000 hits.

And here are some books that contain more material on the Python language:

- *Core Python Programming* by Wesley Chun is a large book at about 750 pages. The first part of the book covers the basic Python language features. The second part provides an easy-paced introduction to more advanced topics including many of those mentioned above.
- *Python Essential Reference* by David M. Beazley is a small book, but it is packed with information both on the language itself and the modules in the standard library. It is also very well indexed.
- *Python Pocket Reference* by Mark Lutz really does fit in your pocket. Although not as extensive as *Python Essential Reference* it is a handy reference for the most commonly used functions and modules. Mark Lutz is also the author of *Programming Python*, one of the earliest (and largest) books on Python and not aimed at the beginning programmer. His later book *Learning Python* is smaller and more accessible.
- *Python Programming on Win32* by Mark Hammond and Andy Robinson is a "must have" for anyone seriously using Python to develop Windows applications. Among other things it covers the integration of Python and COM, builds a small application with wxPython, and even uses Python to script windows applications such as Word and Excel.

C.2 Recommended general computer science books

The following suggestions for further reading include many of the authors' favorite books. They deal with good programming practices and computer science in general.

- *The Practice of Programming* by Kernighan and Pike covers not only the design and coding of algorithms and data structures, but also debugging, testing and improving the performance of programs. The examples are mostly C++ and Java, with none in Python.
- *The Elements of Java Style* edited by Al Vermeulen is another small book that discusses some of the finer points of good programming, such as good use of naming conventions, comments, and even whitespace and indentation (somewhat of a nonissue in Python). The book also covers programming by contract, using assertions to catch errors by testing preconditions and postconditions, and proper programming with threads and their synchronization.

- *Programming Pearls* by Jon Bentley is a classic book. It consists of case studies that originally appeared in the author's column in the *Communications of the ACM*. The studies deal with tradeoffs in programming and why it is often an especially bad idea to run with your first idea for a program. The book is a bit older than those above (1986), so the examples are in older languages. There are lots of problems to solve, some with solutions and others with hints. This book was very popular and was followed by a second volume.

- *The New Turing Omnibus* by A.K Dewdney provides a gentle introduction to 66 topics of computer science ranging from parallel computing to computer viruses, from cat scans to genetic algorithms. All of the topics are short and entertaining. An earlier book by Dewdney *The Armchair Universe* is a collection from his column *Computer Recreations* in *Scientific American*. Both books are a rich source of ideas for projects.

- *Turtles, Termites and Traffic Jams* by Mitchel Resnick is about the power of decentralization and how complex behavior can arise from coordinated simple activity of a multitude of agents. It introduces the language StarLogo, which allows the user to write programs for the agents. Running the program demonstrates complex aggregate behavior, which is often counterintuitive. Many of the programs in the book were developed by students in middle school and high school. Similar programs could be written in Python using simple graphics and threads.

- *Gödel, Escher and Bach* by Douglas Hofstadter. Put simply, if you found magic in recursion you will also find it in this bestselling book. One of Hofstadter's themes involves "strange loops" where patterns evolve and ascend until they meet themselves again. It is Hofstadter's contention that such "strange loops" are an essential part of what separates the animate from the inanimate. He demonstrates such patterns in the music of Bach, the pictures of Escher and Gödel's incompleteness theorem.

Index