UNIT-IV

Dictionaries

A Dictionary Is a Mapping

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called **keys**, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

In mathematical language, a dictionary represents a **mapping** from keys to values, so you can also say that each key "maps to" a value. As an example, we'll build a dictio- nary that maps from English to Spanish words, so the keys and the values are all strings.

The function dictcreates a new dictionary with no items. Because dictis the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

The squiggly brackets, {}, represent an empty dictionary. To add items to the dictio- nary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one'to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> eng2sp
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictio- nary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print eng2sp, you might be surprised:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs might not be the same. If you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> eng2sp['two']
'dos'
```

The key 'two'always maps to the value 'dos'so the order of the items doesn't mat- ter.

If the key isn't in the dictionary, you get an exception:

```
>>> eng2sp['four']
KeyError: 'four'
```

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp) 3
```

The in operator works on dictionaries, too; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order, as in "Searching" on page 89. As the list gets longer, the search time gets longer in direct proportion.

For dictionaries, Python uses an algorithm called a **hashtable** that has a remarkable property: the in operator takes about the same amount of time no matter how many items are in the dictionary. I explain how that's possible in "Hashtables" on page 251, but the explanation might not make sense until you've read a few more chapters.

Dictionary as a Collection of Counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corre- sponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
def histogram(s): d =
    dict() for c in s:
        if c not in d: d[c]
            = 1
        else:
            d[c] += 1
    return d
```

The name of the function is histogram, which is a statistical term for a collection of counters (or frequencies).

The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we cre- ate a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c].

Here's how it works:

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on.

Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

As an exercise, use get to write histogram more concisely. You should be able to eliminate the if statement.

Looping andDictionaries

If you use a dictionary in a forstatement, it traverses the keys of the dictionary. For example, print_histprints each key and the corresponding value:

```
def print_hist(h): for c
    in h:
        print(c, h[c])
```

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h) a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order. To traverse the keys in sorted order, you can use the built-in function sorted:

```
>>> for key in sorted(h):
...         print(key, h[key]) a 1
o 1
p 1
r 2
t 1
```

Reverse Lookup

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a **lookup**.

But what if you have vand you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverselookup**; you have to search.

Here is a function that takes a value and returns the first key that maps to that value:

```
def reverse_lookup(d, v): for k
    in d:
        if d[k] == v:
            return k raise
    LookupError()
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before: raise. The **raise statement** causes an exception; in this case it causes a LookupError, which is a built-in exception used to indicate that a lookup operation failed.

If we get to the end of the loop, that means vdoesn't appear in the dictionary as a value, so we raise an exception.

Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> k
'r'
```

And an unsuccessful one:

```
>>> k = reverse_lookup(h, 3) Traceback
(most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

The raise statement can take a detailed error message as an optional argument. For example:

```
>>> raise LookupError('value does not appear in the dictionary') Traceback
(most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

Dictionaries and Lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dic- tionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters.

Here is a function that inverts a dictionary:

```
def invert_dict(d):
    inverse = dict() for
    key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

Each time through the loop, key gets a key from d and val gets the corresponding value. If val is not in inverse, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

Figure 11-1 is a state diagram showing hist and inverse. A dictionary is represented as a box with the type dict above it and the key-value pairs inside. If the values are integers, floats or strings, I draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.

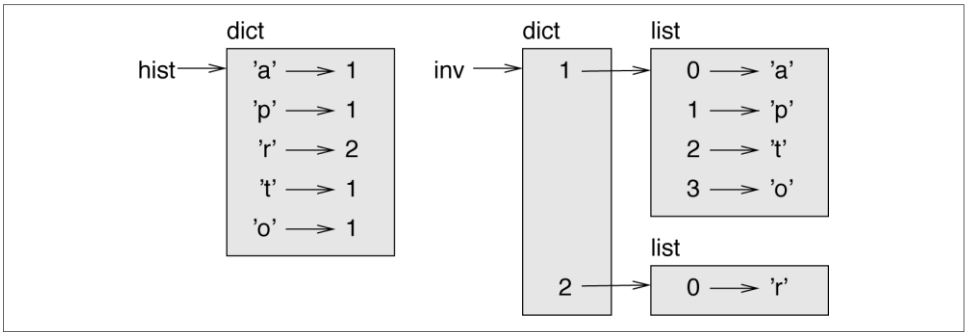

Figure 11-1. State diagram.

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ? TypeError: list
objects are unhashable
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**.

A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.

That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in the next chapter.

Since dictionaries are mutable, they can't be used as keys, but they *can* be used as values.

Memos

If you played with the fibonacci function from "One More Example" on page 68, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the runtime increases quickly.

To understand why, consider Figure 11-2, which shows the **call graph** for fibonacci with n=4.

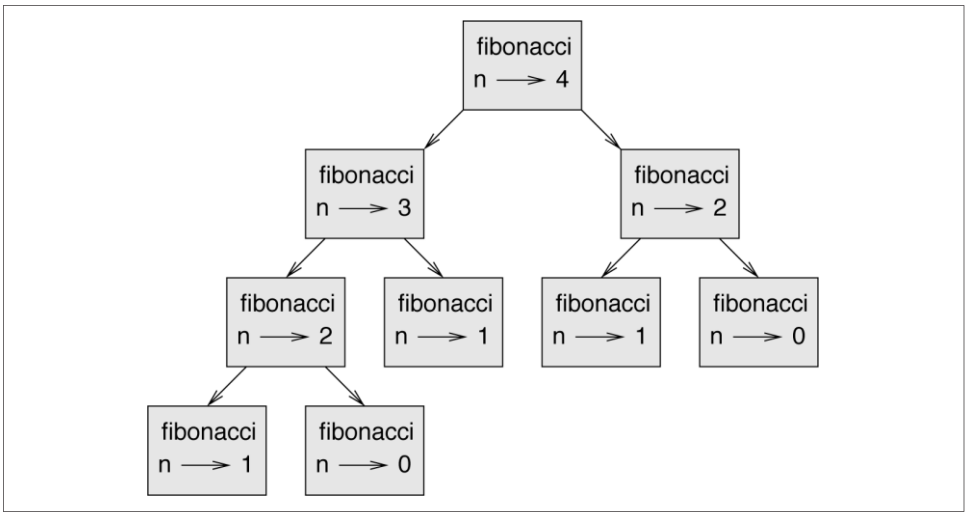

Figure 11-2. Call graph.

A call graph shows a set of 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 fibo nacci 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 ineffi- cient solution to the problem, and it gets worse as the argument gets bigger.

One 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 **memo**. Here is a "memoized" version of fibonacci:

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

def fibonacci(n): if n
      in known:
            return known[n]

      res = fibonacci(n-1) + fibonacci(n-2)
      known[n] = res
      return res
```

known is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictio- nary, and return it.

If you run this version of fibonacci and compare it with the original, you will find that it is much faster.

Global Variables

In the previous example, known is created outside the function, so it belongs to the special frame called __main__. Variables in __main__ are sometimes called **global** because they can be accessed from any function. Unlike local variables, which disap- pear when their function ends, global variables persist from one function call to the next.

It is common to use global variables for **flags**; that is, boolean variables that indicate ("flag") whether a condition is true. For example, some programs use a flag named verbose to control the level of detail in the output:

```
verbose = True

def example1(): if
    verbose:
        print('Running example1')
```

If you try to reassign a global variable, you might be surprised. The following exam- ple is supposed to keep track of whether the function has been called:

```
been_called = False

def example2():
    been_called = True              # WRONG
```

But if you run it you will see that the value of been_called doesn't change. The prob- lem is that example2 creates a new local variable named been_called. The local vari- able goes away when the function ends, and has no effect on the global variable.

To reassign a global variable inside a function you have to **declare** the global variable before you use it:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

The **global statement** tells the interpreter something like, "In this function, when I say been_called, I mean the global variable; don't create a local one."

Here's an example that tries to update a global variable:

```
count = 0

def example3():
    count = count + 1               # WRONG
```

If you run it you get:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python assumes that count is local, and under that assumption you are reading it before writing it. The solution, again, is to declare count global:

```
def example3():
    global count
```

```
        count += 1
```

If a global variable refers to a mutable value, you can modify the value without declaring the variable:

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

def example4():
    known[2] = 1
```

So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:

```
def example5(): global
    known known =
    dict()
```

Global variables can be useful, but if you have a lot of them, and you modify them frequently, they can make programs hard to debug.

Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking the output by hand. Here are some suggestions for debugging large datasets:

Scale down the input:

If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first n lines.

If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types:

Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks:

Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "insane".

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check".

Format the output:

Formatting debugging output can make it easier to spot an error. We saw an example in "Debugging" on page 70. The pprint module provides a pprint func- tion that displays

built-in types in a more human-readable format (pprint stands for “pretty print”).

Again, time you spend building scaffolding can reduce the time you spend debugging.

Tuples

Tuples AreImmutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the ele- ments of the sequence:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a built-in function, you should avoid using it as a vari- able name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

And the slice operator selects a range of elements:

```
>>> t[1:3]
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

This statement makes a new tuple and then makes trefer to it.

The relational operators work with tuples and other sequences; Python starts by com- paring the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Tuple Assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap aand b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evalu- ated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from splitis a list with two elements; the first element is assigned to uname, the second to domain:

```
>>> uname
'monty'
>>> domain
'python.org'
```

Tuples as Return Values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y. It is better to compute them both at the same time.

The built-in function divmod takes two arguments and returns a tuple of two values: the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

max and min are built-in functions that find the largest and smallest elements of a sequence. min_max computes both and returns a tuple of two values.

Variable-Length Argument Tuples

Functions can take a variable number of arguments. A parameter name that begins with * **gathers** arguments into a tuple. For example, printall takes any number of arguments and prints them:

```
def printall(*args):
    print(args)
```

The gather parameter can have any name you like, but args is conventional. Here's how the function works:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

The complement of gather is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator. For exam- ple, divmod takes exactly two arguments; it doesn't work with a tuple:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
>>> divmod(*t)
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, max and min can take any number of arguments:

```
>>> max(1, 2, 3)
3
```

But sum does not:

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

As an exercise, write a function called sumall that takes any number of arguments and returns their sum.

Lists and Tuples

zip is a built-in function that takes two or more sequences and returns a list of tuples where each tuple contains one element from each sequence. The name of the func- tion refers to a zipper, which joins and interleaves two rows of teeth.

This example zips a string and a list:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

The result is a **zip object** that knows how to iterate through the pairs. The most com- mon use of zip is in a for loop:

```
>>> for pair in zip(s, t):
...         print(pair)
... ('a', 0)
('b', 1)
('c', 2)
```

A zip object is a kind of **iterator**, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

If you want to use list operators and methods, you can use a zip object to make a list:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one:

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a forloop to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letterand number. The output of this loop is:

```
0 a
1 b
2 c
```

If you combine zip, forand tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, has_matchtakes two sequen- ces, t1and t2, and returns Trueif there is an index isuch that t1[i]==t2[i]:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2): if x ==
        y:
            return True
    return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate:

```
for index, element in enumerate('abc'):
    print(index, element)
```

The result from enumerate is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence. In this example, the output is

```
0 a
1 b
2 c
```

Again.

Dictionaries andTuples

Dictionaries have a method called itemsthat returns a sequence of tuples, where each tuple is a key-value pair:

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

The result is a dict_itemsobject, which is an iterator that iterates the key-value pairs. You can use it in a forloop like this:

```
>>> for key, value in d.items():
...         print(key, value)
...
c 2
a 0
```

```
b 1
```

As you should expect from a dictionary, the items are in no particular order.

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Combining dictwith zipyields a concise way to create a dictionary:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method updatealso takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, firstand number, we could write:

```
directory[last, first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary:

```
for last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to lastand first, then prints the name and corresponding telephone number.

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple ('Cleese', 'John') would appear as in Figure 12-1.

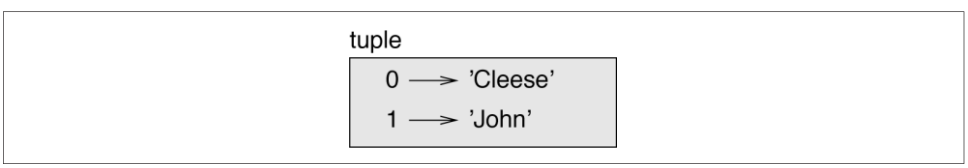

Figure 12-1. State diagram.

But in a larger diagram you might want to leave out the details. For example, a dia- gram of the telephone directory might appear as in Figure 12-2.

Figure 12-2. State diagram.

Here the tuples are shown using Python syntax as a graphical shorthand. The tele- phone number in the diagram is the complaints line for the BBC, so please don't call it.

Sequences of Sequences

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how should you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like sort and reverse, which modify existing lists. But Python provides the built-in function sorted, which takes any sequence and returns a new list with the same elements in sorted order, and

reversed, which takes a sequence and returns an iterator that traverses the list in reverse order.

Debugging

Lists, dictionaries and tuples are examples of **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, or dictionaries that con- tain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call **shape errors**; that is, errors caused when a data structure has the wrong type, size, or structure. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

To help debug these kinds of errors, I have written a module called structshape that provides a function, also called structshape, that takes any kind of data structure as an argument and returns a string that summarizes its shape. You can download it from *http://thinkpython2.com/code/structshape.py*.

Here's the result for a simple list:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t) 'list
of 3 int'
```

A fancier program might write "list of 3 ints", but it was easier not to deal with plu- rals. Here's a list of lists:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

If the elements of the list are not the same type, structshape groups them, in order, by type:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Here's a list of tuples:

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

And here's a dictionary with three items that map integers to strings:

```
>>> d = dict(lt)
>>> structshape(d) 'dict
of 3 int->str'
```

If you are having trouble keeping track of your data structures, structshape can help.

Random Numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be difficult, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that gen- erate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The random module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series. To see a sample, run this loop:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

The function randint takes parameters low and high and returns an integer between low and high (including both):

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use choice:

```
>>> t = [1, 2, 3]
>>> random.choice(t) 2
>>> random.choice(t) 3
```

The random module also provides functions to generate random values from continu- ous distributions including Gaussian, exponential, gamma, and a few more.

Exercise 13-5.

Write a function named choose_from_hist that takes a histogram as defined in "Dic- tionary as a Collection of Counters" on page 127 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

your function should return 'a' with probability *2/3* and 'b' with probability *1/3*.

Word Histogram

You should attempt the previous exercises before you go on. You can download my solution from *http://thinkpython2.com/code/analyze_book1.py*. You will also need *http://thinkpython2.com/code/emma.txt*.

Here is a program that reads a file and builds a histogram of the words in the file:

```
import string

def process_file(filename): hist =
    dict()
    fp = open(filename) for
    line in fp:
        process_line(line, hist) return
    hist

def process_line(line, hist): line =
    line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace) word =
        word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

This program reads emma.txt, which contains the text of *Emma* by Jane Austen.

process_file loops through the lines of the file, passing them one at a time to pro cess_line. The histogram hist is being used as an accumulator.

process_line uses the string method replace to replace hyphens with spaces before using split to break the line into a list of strings. It traverses the list of words and uses strip and lower to remove punctuation and convert to lowercase. (It is short- hand to say that strings are "converted"; remember that strings are immutable, so methods like strip and lower return new strings.)

Finally, process_line updates the histogram by creating a new item or incrementing an existing one.

To count the total number of words in the file, we can add up the frequencies in the histogram:

```
def total_words(hist):
    return sum(hist.values())
```

The number of different words is just the number of items in the dictionary:

```
def different_words(hist): return
    len(hist)
```

Here is some code to print the results:

```
print('Total number of words:', total_words(hist)) print('Number of
different words:', different_words(hist))
```

And the results:

```
Total number of words: 161080
Number of different words: 7214
```

Most Common Words

To find the most common words, we can make a list of tuples, where each tuple con- tains a word and its frequency, and sort it.

The following function takes a histogram and returns a list of word-frequency tuples:

```
def most_common(hist): t
    = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

In each tuple, the frequency appears first, so the resulting list is sorted by frequency. Here is a loop that prints the 10 most common words:

```
t = most_common(hist)
print('The most common words are:') for
freq, word in t[:10]:
    print(word, freq, sep='\t')
```

I use the keyword argument sep to tell print to use a tab character as a "separator", rather than a space, so the second column is lined up. Here are the results from *Emma*:

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

This code can be simplified using the key parameter of the sort function. If you are curious, you can read about it at *https://wiki.python.org/moin/HowTo/Sorting*.

Optional Parameters

We have seen built-in functions and methods that take optional arguments. It is pos- sible to write programmer-defined functions with optional arguments, too. For exam- ple, here is a function that prints the most common words in a histogram:

```
def print_most_common(hist, num=10): t =
    most_common(hist)
    print('The most common words are:') for
    freq, word in t[:num]:
        print(word, freq, sep='\t')
```

The first parameter is required; the second is optional. The **default value** of num is 10. If you only provide one argument:

```
print_most_common(hist)
```

numgets the default value. If you provide two arguments:

```
print_most_common(hist, 20)
```

numgets the value of the argument instead. In other words, the optional argument **overrides** the default value.

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

Dictionary Subtraction

Finding the words from the book that are not in the word list from words.txtis a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in the other (the words in the list).

subtract takes dictionaries d1 and d2 and returns a new dictionary that contains all the keys from d1 that are not in d2. Since we don't really care about the values, we set them all to None:

```
def subtract(d1, d2): res =
        dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in words.txt, we can use process_file to build a histogram for words.txt, and then subtract:

```
words = process_file('words.txt') diff =
subtract(hist, words)

print("Words in the book that aren't in the word list:") for word in
diff:
    print(word, end=' ')
```

Here are some of the results from *Emma*:

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness friend's
venice apartment ...
```

Some of these words are names and possessives. Others, like "rencontre", are no longer in common use. But a few are common words that should really be in the list!

Exercise 13-6.

Python provides a data structure called set that provides many common set opera- tions. You can read about them in "Sets" on page 226, or read the documentation at *http://docs.python.org/3/library/stdtypes.html#types-set*.

Write a program that uses set subtraction to find words in the book that are not in the word list.

Solution: *http://thinkpython2.com/code/analyze_book2.py*.

Random Words

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

```
def random_word(h): t
    = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

The expression `[word]*freq` creates a list with `freq` copies of the string `word`. The `extend` method is similar to `append` except that the argument is a sequence.

This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improve- ment is to build the list once and then make multiple selections, but the list is still big.

An alternative is:

1. Use `keys` to get a list of the words in the book.
2. Build a list that contains the cumulative sum of the word frequencies (see Exer- cise 10-2). The last item in this list is the total number of words in the book, *n*.

3. Choose a random number from 1 to *n*. Use a bisection search (See Exercise 10-10) to find the index where the random number would be inserted in the cumulative sum.
4. Use the index to find the corresponding word in the word list.

Exercise 13-7.

Write a program that uses this algorithm to choose a random word from the book. Solution:

http://thinkpython2.com/code/analyze_book3.py.

Markov Analysis

If you choose words from the book at random, you can get a sense of the vocabulary, but you probably won't get a sentence:

this the small regard harriet which knightley's it most things

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb.

One way to measure these kinds of relationships is Markov analysis, which character- izes, for a given sequence of words, the probability of the words that might come next. For example, the song "Eric, the Half a Bee" begins:

> Half a bee, philosophically,
> Must, ipso facto, half not be. But
> half the bee has got to be Vis a
> vis, its entity. D'you see?
>
> But can a bee be said to be Or
> not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

In this text, the phrase "half the" is always followed by the word "bee", but the phrase "the bee" might be followed by either "has" or "is".

The result of Markov analysis is a mapping from each prefix (like "half the" and "the bee") to all possible suffixes (like "has" and "is").

Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

For example, if you start with the prefix "Half a", then the next word has to be "bee", because the prefix only appears once in the text. The next prefix is "a bee", so the next suffix might be "philosophically", "be" or "due".

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length.

Exercise 13-8.

Markov analysis:

1. Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length 2, but you should write the program in a way that makes it easy to try other lengths.
2. Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from *Emma* with prefix length 2:

   > He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.

   For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

   What happens if you increase the prefix length? Does the random text make more sense?
3. Once your program is working, you might want to try a mash-up: if you combine text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.

Credit: This case study is based on an example from Kernighan and Pike, *The Practice of Programming*, Addison-Wesley, 1999.

You should attempt this exercise before you go on; then you can can download my solution from *http://thinkpython2.com/code/markov.py*. You will also need *http://thinkpython2.com/code/emma.txt*.

Data Structures

Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:

- How to represent the prefixes.
- How to represent the collection of possible suffixes.
- How to represent the mapping from each prefix to the collection of possible suf- fixes.

The last one is easy: a dictionary is the obvious choice for a mapping from keys to corresponding values.

For the prefixes, the most obvious options are string, list of strings, or tuple of strings. For the suffixes, one option is a list; another is a histogram (dictionary).

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is "Half a", and the next word is "bee", you need to be able to form the next prefix, "a bee".

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can't append or remove, but you can use the addition operator to form a new tuple:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

shift takes a tuple of words, prefix, and a string, word, and forms a new tuple that has all the words in prefix except the first, and word added to the end.

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently (see Exercise 13-7).

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is runtime. Sometimes there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that the in operator is faster for dictionaries than for lists, at least when the number of elements is large.

But often you don't know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This approach is called **benchmarking**. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is

no need to go on. If not, there are tools, like the profile module, that can identify the places in a program that take the most time.

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after runtime.

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This would be a net win if the time saved during generation exceeded the time spent in conversion.

Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are five things to try:

Reading:

Examine your code, read it back to yourself, and check that it says what you meant to say.

Running:

Experiment by making changes and running different versions. Often if you dis- play the right thing at the right place in the program, the problem becomes obvi- ous, but sometimes you have to build scaffolding.

Ruminating:

Take some time to think! What kind of error is it: syntax, runtime, or semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

Rubberducking:

If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called **rubber duck debugging**. I am not making this up; see *https://en.wikipedia.org/wiki/Rubber_duck_debugging*.

Retreating:

> At some point, the best thing to do is back off and undo recent changes until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more pos- sibilities, try to think of a test that would eliminate one of them.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

Files

Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writ- ing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, pickle, that makes it easy to store program data.

Reading and Writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in "Reading Word Lists" on page 99.

To write a file, you have to open it with mode 'w'as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

openreturns a file object that provides methods for working with the file. The write method puts data into the file:

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1) 24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call writeagain, it adds the new data to the end of the file:

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2) 24
```

When you are done writing, you should close the file:

```
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

Format Operator

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. The easiest way to do that is with str:

```
>>> x = 52
>>> fout.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**, which contains one or more **format sequen- ces**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence '%d' means that the second operand should be for- matted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

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

```
>>> 'I have spotted %d camels.' % camels 'I have
spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels') 'In 3 years I
have spotted 0.1 camels.'
```

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

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

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

For more information on the format operator, see *https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting*. A more powerful alternative is the string format method, which you can read about at *https://docs.python.org/3/library/stdtypes.html#str.format*.

Filenames and Paths

Files are organized into **directories** (also called "folders"). Every running program has a "current directory", which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current direc- tory.

The os module provides functions for working with files and directories ("os" stands for "operating system"). os.getcwd returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

cwd stands for "current working directory". The result in this example is /home/dins dale, which is the home directory of a user named dinsdale.

A string like '/home/dinsdale' that identifies a file or directory is called a **path**.

A simple filename, like memo.txt, is also considered a path, but it is a **relative path** because it relates to the current directory. If the current directory is /home/dinsdale, the filename memo.txt would refer to /home/dinsdale/memo.txt.

A path that begins with / does not depend on the current directory; it is called an **absolute path**. To find the absolute path to a file, you can use os.path.abspath:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path provides other functions for working with filenames and paths. For example, os.path.exists checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt') True
```

If it exists, os.path.isdir checks whether it's a directory:

```
>>> os.path.isdir('memo.txt') False
>>> os.path.isdir('/home/dinsdale') True
```

Similarly, os.path.isfile checks whether it's a file.

os.listdir returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories:

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

os.path.join takes a directory and a filename and joins them into a complete path.

Catching Exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an IOError:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like os.path.exists and os.path.isfile, but it would take a lot of time and code to check all the possibilities (if "Errno 21" is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the try statement does. The syntax is similar to an if...else statement:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python starts by executing the try clause. If all goes well, it skips the except clause and proceeds. If an exception occurs, it jumps out of the try clause and runs the except clause.

Handling an exception with a try statement is called **catching** an exception. In this example, the except clause prints an error message that is not very helpful. In gen- eral, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

Databases

A **database** is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest differ- ence between a database and a dictionary is that the database is on disk (or other per- manent storage), so it persists after the program ends.

The module dbm provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

The mode 'c' means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, dbm updates the database file:

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, dbm reads the file:

```
>>> db['cleese.png'] b'Photo
of John Cleese.'
```

The result is a **bytes object**, which is why it begins with b. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, dbm replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like keys and items, don't work with database objects. But iteration with a for loop works:

```
for key in db: print(key,
      db[key])
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

Pickling

A limitation of dbm is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The pickle module can help. It translates almost any type of object into a string suit- able for storage in a database, and then translates strings back into objects.

pickle.dumps takes an object as a parameter and returns a string representation (dumps is short for "dump string"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for pickle to inter- pret. pickle.loads("load string") reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2 [1,
2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use pickle to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called shelve.

Pipes

Most operating systems provide a command-line interface, also known as a **shell**. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix you can change directories with cd, display the contents of a directory with ls, and launch a web browser by typing (for example) firefox.

Any program that you can launch from the shell can also be launched from Python using a **pipe object**, which represents a running program.

For example, the Unix command ls -l normally displays the contents of the current directory in long format. You can launch ls with os.popen[1]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the ls process one line at a time with readline or get the whole thing at once with read:

```
>>> res = fp.read()
```

1 popen is deprecated now, which means we are supposed to stop using it and start using the subprocess mod- ule. But for simple cases, I find subprocess more complicated than necessary. So I am going to keep using popen until they take it away.

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print(stat)
None
```

The return value is the final status of the lsprocess; Nonemeans that it ended nor- mally (with no errors).

For example, most Unix systems provide a command called md5sumthat reads the contents of a file and computes a "checksum". You can read about MD5 at *http://en.wikipedia.org/wiki/Md5*. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe col- lapses).

You can use a pipe to run md5sumfrom Python and get the result:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res) 1e0033f0ed0656636de0d75144ba32e0
book.tex
>>> print(stat)
None
```

Writing Modules

Any file that contains Python code can be imported as a module. For example, sup- pose you have a file named wc.pywith the following code:

```
def linecount(filename):
    count = 0
    for line in open(filename): count
        += 1
    return count

print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object wc:

```
>>> wc
<module 'wc' from 'wc.py'>
```

The module object provides linecount:

```
>>> wc.linecount('wc.py') 7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new func- tions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if_name_== '_main_':
    print(linecount('wc.py'))
```

_name_is a built-in variable that is set when the program starts. If the program is running as a script,_name_has the value '_main_'; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

As an exercise, type this example into a file named wc.py and run it as a script. Then run the Python interpreter and import wc. What is the value of_name_when the module is being imported?

Warning: If you import a module that has already been imported, Python does noth- ing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function reload, but it can be tricky, so the safest thing to do is restart the interpreter and then import the mod- ule again.

Debugging

When you are reading and writing files, you might run into problems with white- space. These errors can be hard to debug because spaces, tabs and newlines are nor- mally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s) 1
2          3
  4
```

The built-in function repr can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s)) '1
2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different charac- ters to indicate the end of a line. Some systems use a newline, represented \n. Others use a return character, represented \r. Some use both. If you move files between dif- ferent systems, these inconsistencies can cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at *http://en.wikipedia.org/wiki/ Newline*. Or, of course, you could write one yourself.

Classes and Objects

At this point you know how to use functions to organize code and built-in types to organize data. The next step is to learn "object-oriented programming", which uses programmer-defined types to organize both code and data. Object-oriented program- ming is a big topic; it will take a few chapters to get there.

Programmer-Defined Types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called Point that represents a point in two- dimensional space.

In mathematical notation, points are often written in parentheses with a comma sepa- rating 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.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, x and y.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

A programmer-defined type is also called a **class**. A class definition looks like this:

```
class Point:
    """Represents a point in 2-D space."""
```

The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class defi- nition, but we will get back to that later.

Defining a class named Point creates a **class object**:

```
>>> Point
<class '__main__.Point'>
```

Because Point is defined at the top level, its "full name" is __main__.Point.

The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function:

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to blank.

Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

Every object is an instance of some class, so "object" and "instance" are interchangea- ble. But in this chapter I use "instance" to indicate that I am talking about a programmer-defined type.

Attributes

You can assign values 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.whitespace. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute", which is a verb.

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**; see Figure 15-1.

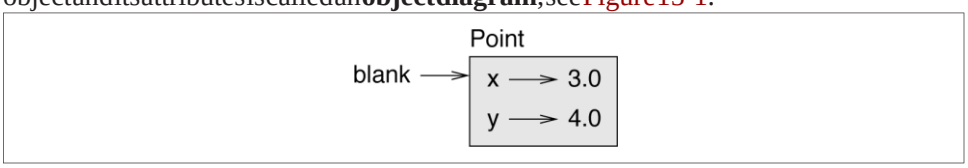

Figure 15-1. Object diagram.

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

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

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

The expression blank.x means, "Go to the object blank refers to and get the value of x." In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.

You can use dot notation as part of any expression. For example:

```
>>> '(%g, %g)' % (blank.x, blank.y) '(3.0,
4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

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

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

As an exercise, write a function called distance_between_points that takes two Points as arguments and returns the distance between them.

Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to repre-

---

sent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll imple- ment the first one, just as an example.

Here is the class definition:

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner. """
```

The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0 box.corner
= Point() box.corner.x = 0.0
box.corner.y = 0.0
```

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."

Figure 15-2 shows the state of this object. An object that is an attribute of another object is **embedded**.

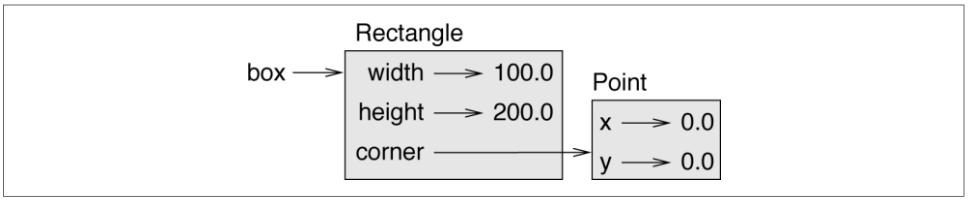

Figure 15-2. Object diagram.

Instances as Return Values

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

```
def find_center(rect): p =
    Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2 return p
```

Here is an example that passes box as an argument and assigns the resulting Point to center:

```
>>> center = find_center(box)
>>> print_point(center) (50,
100)
```

Objects Are Mutable

You 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 posi- tion, you can modify the values of width and height:

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

You can also write functions that modify objects. For example, grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

As an exercise, write a function named move_rectangle that takes a Rectangle and two numbers 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.

Copying

Aliasing can make a program difficult to read because changes 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 copymodule contains a function called copythat can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1and p2contain the same data, but they are not the same Point:

```
>>> print_point(p1) (3,
4)
>>> print_point(p2) (3,
4)
>>> p1 is p2
False
>>> p1 == p2
False
```

The isoperator indicates that p1and p2are not the same object, which is what we expected. But you might have expected ==to yield Truebecause these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be consideredequivalent. Atleast, notyet.

If you use copy.copyto duplicate a Rectangle, you will find that it copies the Rectan- gle object but not the embedded Point:

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner True
```

Figure 15-3 shows what the object diagram looks like. This operation is called a **shal- low copy** because it copies the object and any references it contains, but not the embedded objects.

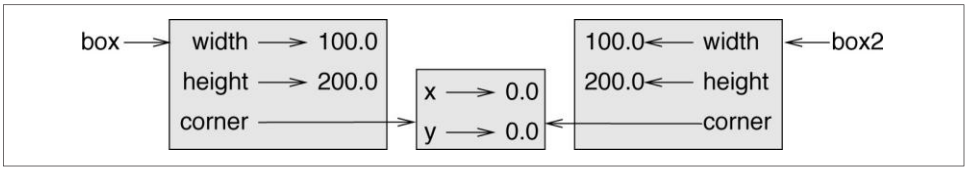

Figure 15-3. Object diagram.

For most applications, this is not what you want. In this example, invoking grow_rec tangleon one of the Rectangles would not affect the other, but invoking move_rec tangle on either wouldaffectboth!Thisbehaviorisconfusinganderror-prone.

Fortunately, the copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner False
```

box3and boxare completely separate objects.

As an exercise, write a version of move_rectanglethat creates and returns a new Rec- tangle instead of modifying the old one.

Debugging

When you start working with objects, you are likely to encounter some new excep- tions. If you try to access an attribute that doesn't exist, you get an AttributeError:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<class '_main_.Point'>
```

You can also use isinstance to check whether an object is an instance of a class:

```
>>> isinstance(p, Point) True
```

If you are not sure whether an object has a particular attribute, you can use the built- in function hasattr:

```
>>> hasattr(p, 'x') True
>>> hasattr(p, 'z') False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

You can also use a trystatement to see if the object has the attributes you need:

```
try:
    x = p.x
except AttributeError: x =
    0
```