Chapter 8

Lists

A list is a sequence

Like a string, a ***list*** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called ***elements*** or sometimes ***items***.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]

['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is ***nested***.

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']

numbers = [17, 123]

empty = []

print(cheeses, numbers, empty)

['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> print(cheeses[0]) Cheddar
```

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
numbers = [17, 123]

numbers[1] = 5
```

```
print(numbers)

[17,  5]
```

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This rela-tionship is called a ***mapping***; each index "maps to" one of the elements.

List indices work the same way as string indices:
Any integer expression can be used as an index.

If you try to read or write an element that does not exist, you get an

IndexError.

If an index has a negative value, it counts backward from the end of the list. The in operator also works on lists.

```
cheeses  =  ['Cheddar',  'Edam',  'Gouda']

'Edam'  in  cheeses

True

>>> 'Brie' in cheeses False
```

Traversing a list

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
for     cheese in cheeses: print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len:

```
for     i in range(len(numbers)): numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

A for loop over an empty list never executes the body:

```
for x in empty:
print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

List operations

The + operator concatenates lists:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
[0] * 4 [0, 0, 0, 0]

[1, 2, 3] * 3

[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats four times. The second example repeats the list three times.

List slices

The slice operator also works on lists:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']

t[1:3]

['b', 'c']

>>> t[:4]

['a', 'b', 'c', 'd']

>>> t[3:]

['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]

['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing opera-tions that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']

t[1:3] = ['x', 'y']

print(t)
```

```
['a', 'x', 'y', 'd', 'e', 'f']
```

List methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

```
t = ['a', 'b', 'c']

t.append('d')

print(t)
```

```
['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements:

```
t1 = ['a', 'b', 'c']

t2 = ['d', 'e']

t1.extend(t2)

print(t1)
```

```
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

```
t = ['d', 'c', 'e', 'b', 'a']

t.sort()

print(t)
```

```
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None. If you acciden-tally write t = t.sort(), you will be disappointed with the result.

Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

```
t = ['a', 'b', 'c']
```

```
x = t.pop(1)

print(t)

['a', 'c']

>>> print(x) b
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the del operator:

```
t = ['a', 'b', 'c']

del t[1]

print(t)

['a', 'c']
```

If you know the element you want to remove (but not the index), you can use remove:

```
t = ['a', 'b', 'c']

t.remove('b')

print(t)

['a', 'c']
```

The return value from remove is None.

To remove more than one element, you can use del with a slice index:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']

del t[1:5]

print(t)

['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

Lists and functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
nums = [3, 41, 12, 9, 74, 15]
```

```
print(len(nums))
```

```
6
```

```
print(max(nums))
```

```
74
```

```
print(min(nums))
```

```
3
```

```
print(sum(nums))
```

```
154
```

```
print(sum(nums)/len(nums))
```

```
25
```

The sum() function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

We could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list.

First, the program to compute an average without a list:

```
total = 0 count = 0 while (True):

inp = input('Enter a number: ') if inp == 'done': break

value = float(inp) total = total + value count = count + 1

average = total / count print('Average:', average)

# Code: http://www.py4e.com/code3/avenum.py
```

In this program, we have count and total variables to keep the number and running total of the user's numbers as we repeatedly prompt the user for a number.

We could simply remember each number as the user entered it and use built-in functions

to compute the sum and count at the end.

```
numlist = list() while (True):

inp = input('Enter a number: ') if inp == 'done': break

value  =  float(inp)
numlist.append(value)

average = sum(numlist) / len(numlist) print('Average:', average)
```

 # Code: http://www.py4e.com/code3/avelist.py

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
s  =  'spam'

t  =  list(s)

print(t)

['s',  'p',  'a',  'm']
```

Because list is the name of a built-in function, you should avoid using it as a variable name. I also avoid the letter l because it looks too much like the number 1. So that's why I use t.

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

```
s  =  'pining  for  the  fjords'

t  =  s.split()

print(t)

['pining',  'for',  'the',  'fjords']

>>> print(t[2]) the
```

Once you have used split to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list.

You can call split with an optional argument called a ***delimiter*** that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
s = 'spam-spam-spam'

delimiter = '-'

s.split(delimiter)
```

```
['spam', 'spam', 'spam']
```

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
t = ['pining', 'for', 'the', 'fjords']

delimiter = ' '

delimiter.join(t)
```

```
'pining for the fjords'
```

In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string, “”, as a delimiter.

Parsing lines

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the “interesting lines” and then ***parse*** the line to find some interesting ***part*** of the line. What if we wanted to print out the day of the week from those lines that start with “From”?

From stephen.marquard@uct.ac.za*Sat* Jan 5 09:14:16 2008

The split method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with “From”, split those lines, and then print out the third word in the line:

```
fhand = open('mbox-short.txt') for line in fhand:

line = line.rstrip()
```

```
if not line.startswith('From '): continue words = line.split()

print(words[2])

#  Code:  http://www.py4e.com/code3/search5.py
```

Here we also use the contracted form of the if statement where we put the continue on the same line as the if. This contracted form of the if functions the same as if the continue were on the next line and indented.
The program produces the following output:

```
Sat

Fri

Fri

Fri

...
```

Later, we will learn increasingly sophisticated techniques for picking the lines to work on and how we pull those lines apart to find the exact bit of information we are looking for.

Objects and values

If we execute these assignment statements:

a = 'banana' b = 'banana'

we know that a and b both refer to a string, but we don't know whether they refer to the ***same*** string. There are two possible states:

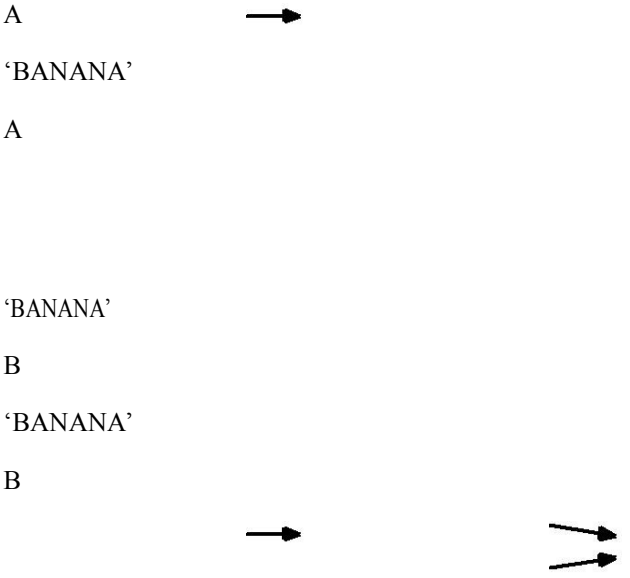

Figure 8.1: Variables and Objects

In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the is oper-ator.

a = 'banana'

```
b = 'banana'
```

```
a is b
```

```
True
```

In this example, Python only created one string object, and both a and b refer to it.

But when you create two lists, you get two objects:

```
a = [1, 2, 3]
```

```
b = [1, 2, 3]
```

```
a is b
```

```
False
```

In this case we would say that the two lists are ***equivalent***, because they have the same elements, but not ***identical***, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using “object” and “value” interchangeably, but it is more precise to say that an object has a value. If you execute a = [1,2,3], a refers to a list object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

```
a = [1, 2, 3]


b = a


b is a


True
```

The association of a variable with an object is called a ***reference***. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is ***aliased***.

If the aliased object is mutable, changes made with one alias affect the other:

```
b[0] = 17


print(a) [17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana' b = 'banana'
```

it almost never makes a difference whether a and b refer to the same string or not.

List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete_head removes the first element from a list:

```
def delete_head(t): del t[0]
```

Here's how it is used:

```
letters = ['a', 'b', 'c']

delete_head(letters)

print(letters)
```

```
['b', 'c']
```

The parameter t and the variable letters are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

```
t1 = [1, 2]

t2 = t1.append(3)
```

```
print(t1)
```

```
[1, 2, 3]
```

```
print(t2)
```

```
None
```

```
t3 = t1 + [3]
```

```
print(t3)
```

```
[1, 2, 3]
```

```
>>> t2 is t3 False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function ***does not*** delete the head of a list:

```
def bad_delete_head(t):
```

```
t = t[1:]                                    # WRONG!
```

The slice operator creates a new list and the assignment makes t refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, tail returns all but the first element of a list:

```
def                                    tail(t): return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
letters = ['a', 'b', 'c']


rest = tail(letters)


print(rest)


['b', 'c']
```

Chapter 9

Dictionaries

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

You can think of a dictionary as a mapping between a set of indices (which are called ***keys***) and a set of values. Each key maps to a value. The association of a key and a value is called a ***key-value pair*** or sometimes an ***item***.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

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

```
eng2sp = dict()


print(eng2sp)


{}
```

The curly 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:

```
>>> print(eng2sp) {'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print eng2sp, you might be surprised:

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

```
print(eng2sp)
```

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

The order of the key-value pairs is not the same. In fact, 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:

```
>>> print(eng2sp['two'])
```

'dos'

The key 'two' always maps to the value "dos" so the order of the items doesn't matter.

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

```
>>> print(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; 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 the values as a list, and then use the in operator:

```
vals  =  list(eng2sp.values())
```

```
'uno'  in  vals
```

```
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a ***hash table*** that has a remarkable property: the in operator takes about the same amount of time no matter how many items there are in a dictionary.

Exercise 1: [wordlist2]

Write a program that reads the words in words.txt and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.

Dictionary as a set 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:

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.

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.

You could create a dictionary with characters as keys and counters as the corresponding 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 implementa-tion 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:

```
word = 'brontosaurus' d = dict()

for c in word:

if c not in d: d[c] = 1

else:

d[c] = d[c] + 1
```

```
print(d)
```

We are effectively computing a ***histogram***, which is a statistical term for a set of counters (or frequencies).

The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create 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 the output of the program:

```
{'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:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}


print(counts.get('jan', 0))


100


>>> print(counts.get('tim', 0)) 0
```

We can use get to write our histogram loop more concisely. Because the get method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the if statement.

```
word = 'brontosaurus' d = dict()


for c in word:
```

```
d[c] = d.get(c,0) + 1 print(d)
```

The use of the get method to simplify this counting loop ends up being a very commonly used “idiom” in Python and we will use it many times in the rest of the book. So you should take a moment and compare the loop using the if statement and in operator with the loop using the get method. They do exactly the same thing, but one is more succinct.

Dictionaries and files

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let’s start with a very simple file of words taken from the text of ***Romeo and Juliet***.

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included.

But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief

We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary.

You will see that we have two for loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called ***nested loops*** because one of the loops is the ***outer*** loop and the other loop is the ***inner*** loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating “more quickly” and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

```
fname = input('Enter the file name: ') try:

fhand = open(fname) except:

print('File cannot be opened:', fname) exit()

counts = dict() for line in fhand:

words = line.split() for word in words:

if word not in counts: counts[word] = 1

else:

counts[word]  +=  1

print(counts)

#  Code:  http://www.py4e.com/code3/count1.py
```

When we run the program, we see a raw dump of all of the counts in unsorted hash order. (the romeo.txt file is available at www.py4e.com/code3/romeo.txt)

```
python  count1.py
```

```
Enter  the  file  name:  romeo.txt
```

```
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1, 'is': 3, 'through': 1, 'pale': 1, 'yonder': 1, 'what':
1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1, 'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
```

```
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1, 'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

It is a bit inconvenient to look through the dictionary to find the most common words and their counts, so we need to add some more Python code to get us the output that will be more helpful.

Looping and dictionaries

If you use a dictionary as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100} for key in counts:
```

```
print(key,  counts[key])
```

Here's what the output looks like:

```
jan 100 chuck 1 annie 42
```

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have de-scribed earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

```
counts
```

```
= {
'chuck'
:
1 , 'annie' : 42, 'jan': 100}
for key in
counts:



if
counts[key]
>
10 :


print(key,
counts[key])
```

The for loop iterates through the ***keys*** of the dictionary, so we must use the index operator to retrieve the corresponding ***value*** for each key. Here's what the output looks like:

```
jan 100 annie 42
```

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the keys method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100} lst = list(counts.keys())

print(lst) lst.sort()

for key in lst:

print(key, counts[key])
```

Here's what the output looks like:

['jan', 'chuck', 'annie'] annie 42

chuck 1 jan 100

First you see the list of keys in unsorted order that we get from the keys method. Then we see the key-value pairs in order from the for loop.

Advanced text parsing

In the above example using the file romeo.txt, we made the file as simple as possi-ble by removing all punctuation by hand. The actual text has lots of punctuation, as shown below.

But, soft! what light through yonder window breaks?

It is the east, and Juliet is the sun.

Arise, fair sun, and kill the envious moon,

Who is already sick and pale with grief,

Since the Python split function looks for spaces and treats words as tokens sep-arated by spaces, we would treat the words “soft!” and “soft” as ***different*** words and create a separate dictionary entry for each word.

Also since the file has capitalization, we would treat “who” and “Who” as different words with different counts.

We can solve both these problems by using the string methods lower, punctuation, and translate. The translate is the most subtle of the methods. Here is the documentation for translate:

line.translate(str.maketrans(fromstr, tostr, deletestr))

Replace the characters in** fromstr **with the character in the same position in** tostr **and delete all characters that are in** deletestr**. The** fromstr **and** tostr **can be empty strings and the** deletestr **parameter can be omitted.

We will not specify the table but we will use the deletechars parameter to delete all of the punctuation. We will even let Python tell us the list of characters that it considers “punctuation”:

```
import string

string.punctuation

'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

The parameters used by translate were different in Python 2.0.

We make the following modifications to our program:

```
import  string
fname = input('Enter the file name: ') try:

fhand = open(fname) except:

print('File cannot be opened:', fname) exit()

counts  =
dict()
for  line
in  fhand:
line
=  line.rstrip()
  line = line.translate(line.maketrans('', '', string.punctuation)) line = line.lower()

 words = line.split()
 for      word in words:

if word not in counts: counts[word] = 1

else:
```

```
counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count2.py
```

Part of learning the “Art of Python” or “Thinking Pythonically” is realizing that Python often has built-in capabilities for many common data analysis problems. Over time, you will see enough example code and read enough of the documentation to know where to look to see if someone has already written something that makes your job much easier.

The following is an abbreviated version of the output:

```
Enter the file name: romeo-full.txt

{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2, 'kinsmen': 2, 'what': 11, 'thinkst': 1,
'love': 24, 'cloak': 1, a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40, 'maiden': 1,
'whiteupturned': 1, 'juliet': 32, 'gentleman': 1, 'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Looking through this output is still unwieldy and we can use Python to give us exactly what we are looking for, but to do so, we need to learn about Python ***tuples***. We will pick up this example once we learn about tuples.

Chapter 10

Tuples

Tuples are immutable

A tuple[1] is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are ***immutable***. Tuples are also ***comparable*** and ***hashable*** so we can sort lists of them and use tuples as key values in Python dictionaries.

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 to help us quickly identify tuples when we look at Python code:

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

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

```
t1 = ('a',)
```

```
type(t1)
```

```
<type 'tuple'>
```

Without the comma Python treats (’a’) as an expression with a string in paren-theses that evaluates to a string:

```
t2 = ('a')
```

```
type(t2) <type 'str'>
```

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

```
t = tuple()

print(t)

()
```

If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
t = tuple('lupins')

print(t)

('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a constructor, you should avoid using it as a variable name.

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

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

print(t[0])

'a'
```

And the slice operator selects a range of elements.

```
>>> print(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
```

You can’t modify the elements of a tuple, but you can replace one tuple with another:

```
t = ('A',) + t[1:]
```

```
print(t)
```

```
('A', 'b', 'c', 'd', 'e')
```

Comparing tuples

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, 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
```

The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called ***DSU*** for

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort the list of tuples using the Python built-in sort, and

Undecorate by extracting the sorted elements of the sequence.

[DSU]

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
txt = 'but soft what light in yonder window breaks' words = txt.split()

t = list()

for word in words: t.append((len(word), word))

t.sort(reverse=True)

res = list()

for length, word in t: res.append(word)

print(res)

# Code: http://www.py4e.com/code3/soft.py
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

sort compares the first element, length, first, and only considers the second el-ement to break ties. The keyword argument reverse=True tells sort to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in ***reverse*** alphabetical order, so “what” appears before “soft” in the following list.

The output of the program is as follows:

['yonder', 'window', 'breaks', 'light', 'what', 'soft', 'but', 'in']

Of course the line loses much of its poetic impact when turned into a Python list and sorted in descending word length order.

Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

```
m = [ 'have', 'fun' ]

x, y = m

x

'have'

>>> y

'fun'

>>>
```

It is not magic, Python ***roughly*** translates the tuple assignment syntax to be the following:[2]

```
m = [ 'have', 'fun' ]

x = m[0]

y = m[1]

x

'have'
```

```
>>> y

'fun'

>>>
```

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
m = [ 'have', 'fun' ]

(x, y) = m

x

'have'

>>> y

'fun'

>>>
```

A particularly clever application of tuple assignment allows us to ***swap*** the values of two variables in a single statement:

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

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must 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 split is a list with two elements; the first element is assigned to uname, the second to domain.

```
print(uname) monty
```

```
print(domain) python.org
```

Dictionaries and tuples

Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair:

```
d = {'a':10, 'b':1, 'c':22}
```

```
t = list(d.items())
```

```
print(t)
```

```
[('b', 1), ('a', 10), ('c', 22)]
```

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

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

```
d = {'a':10, 'b':1, 'c':22}
```

```
t = list(d.items())
```

```
t
```

```
[('b', 1), ('a', 10), ('c', 22)]
```

```
t.sort()
```

```
t
```

```
[('a', 10), ('b', 1), ('c', 22)]
```

The new list is sorted in ascending alphabetical order by the key value.

Multiple assignment with dictionaries

Combining items, tuple assignment, and for, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
for key, val in list(d.items()): print(val, key)
```

This loop has two ***iteration variables*** because items returns a list of tuples and key, val is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.

For each iteration through the loop, both key and value are advanced to the next key-value pair in the dictionary (still in hash order).

The output of this loop is:

```
10 a
```

```
22 c
```

```
1 b
```

Again, it is in hash key order (i.e., no particular order).

If we combine these two techniques, we can print out the contents of a dictionary sorted by the ***value*** stored in each key-value pair.

To do this, we first make a list of tuples where each tuple is (value, key). The items method would give us a list of (key, value) tuples, but this time we want to sort by value, not key. Once we have constructed the list with the value-key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

```
d = {'a':10, 'b':1, 'c':22}

l = list()

for key, val in d.items() :

... l.append( (val, key) )

...

l

[(10, 'a'), (22, 'c'), (1, 'b')]

l.sort(reverse=True)

l

[(22, 'c'), (10, 'a'), (1, 'b')]

>>>
```

By carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

The most common words

Coming back to our running example of the text from ***Romeo and Juliet*** Act 2, Scene 2, we can augment our program to use this technique to print the ten most common words in the text as follows:

```
import string

fhand = open('romeo-full.txt') counts = dict()

for line in fhand:

line = line.translate(str.maketrans('', '', string.punctuation)) line = line.lower()

words = line.split() for word in words:

if word not in counts: counts[word] = 1

else:

counts[word] += 1

# Sort the dictionary by value lst = list()

for key, val in list(counts.items()): lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]: print(key, val)

# Code: http://www.py4e.com/code3/count3.py
```

The first part of the program which reads the file and computes the dictionary that maps each word to the count of words in the document is unchanged. But instead of simply printing out counts and ending the program, we construct a list of (val, key) tuples and then sort the list in reverse order.

Since the value is first, it will be used for the comparisons. If there is more than one tuple with the same value, it will look at the second element (the key), so tuples where the value is the same will be further sorted by the alphabetical order of the key.

At the end we write a nice for loop which does a multiple assignment iteration and prints out the ten most common words by iterating through a slice of the list (lst[:10]).

So now the output finally looks like what we want for our word frequency analysis.

61 i

42 and

40 romeo

34 to

34 the

32 thou

32 juliet

30 that

29 my

24 thee

The fact that this complex data parsing and analysis can be done with an easy-to-understand 19-line Python program is one reason why Python is a good choice as a language for exploring information.

Using tuples as keys in dictionaries

Because tuples are ***hashable*** and lists are not, if we want to create a ***composite*** key to use in a dictionary we must use a tuple as the key.

We would encounter a composite key if we wanted to create a telephone directory that maps from last-name, first-name pairs to telephone numbers. Assuming that we have defined the variables last, first, and number, we could write a dictionary assignment statement as follows:

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

The expression in brackets is a tuple. We could use tuple assignment in a for loop 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 last and first, then prints the name and corresponding telephone number.

Sequences: strings, lists, and tuples - Oh My!

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 enumer-ating 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 and why do 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:

In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.

If you want to use a sequence as a dictionary key, you have to use an im-mutable type like a tuple or string.

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. However Python provides the built-in functions sorted and reversed, which take any sequence as a parameter and return a new sequence with the same elements in a different order.

Chapter 11

Regular expressions

So far we have been reading through files, looking for patterns and extracting various bits of lines that we find interesting. We have been

using string methods like split and find and using lists and string slicing to extract portions of the lines.

This task of searching and extracting is so common that Python has a very powerful library called ***regular expressions*** that handles many of these tasks quite elegantly. The reason we have not introduced regular expressions earlier in the book is because while they are very powerful, they are a little complicated and their syntax takes some getting used to.

Regular expressions are almost their own little programming language for searching and parsing strings. As a matter of fact, entire books have been written on the topic of regular expressions. In this chapter, we will only cover the basics of regular expressions.

The regular expression library re must be imported into your program before you can use it. The simplest use of the regular expression library is the search() function. The following program demonstrates a trivial use of the search function.

```
Search for lines that contain 'From' import re


hand = open('mbox-short.txt') for line in hand:


line  =  line.rstrip()


if re.search('From:', line): print(line)
```

Code: http://www.py4e.com/code3/re01.py

We open the file, loop through each line, and use the regular expression search() to only print out lines that contain the string “From:”. This program does not

use the real power of regular expressions, since we could have just as easily used line.find() to accomplish the same result.

The power of the regular expressions comes when we add special characters to the search string that allow us to more precisely control which lines match the string. Adding these special characters to our regular expression allow us to do sophisticated matching and extraction while writing very little code.

For example, the caret character is used in regular expressions to match "the beginning" of a line. We could change our program to only match lines where "From:" was at the beginning of the line as follows:

```
Search for lines that start with 'From' import re

hand = open('mbox-short.txt') for line in hand:

line = line.rstrip()

if re.search('^From:', line): print(line)
```

Code: http://www.py4e.com/code3/re02.py

Now we will only match lines that ***start with*** the string "From:". This is still a very simple example that we could have done equivalently with the startswith() method from the string library. But it serves to introduce the notion that regular expressions contain special action characters that give us more control as to what will match the regular expression.

Character matching in regular expressions

There are a number of other special characters that let us build even more powerful regular expressions. The most commonly used special character is the period or full stop, which matches any character.

In the following example, the regular expression “F..m:” would match any of the strings “From:”, “Fxxm:”, “F12m:”, or “F!@m:” since the period characters in the regular expression match any character.

```
Search for lines that start with 'F', followed by

2 characters, followed by 'm:'

import re

hand = open('mbox-short.txt') for line in hand:

line = line.rstrip()

if re.search('^F..m:', line): print(line)
```

```
# Code: http://www.py4e.com/code3/re03.py
```

This is particularly powerful when combined with the ability to indicate that a character can be repeated any number of times using the "*" or "+" characters in your regular expression. These special characters mean that instead of matching a single character in the search string, they match zero-or-more characters (in the case of the asterisk) or one-or-more of the characters (in the case of the plus sign).

We can further narrow down the lines that we match using a repeated ***wild card*** character in the following example:

```
Search for lines that start with From and have an at sign import re

hand = open('mbox-short.txt') for line in hand:


line = line.rstrip()


if re.search('^From:.+@', line): print(line)

Code: http://www.py4e.com/code3/re04.py
```

The search string "^From:.+@" will successfully match lines that start with "From:", followed by one or more characters (".+"), followed by an at-sign. So this will match the following line:

From: *stephen.marquard@* uct.ac.za

You can think of the ".+" wildcard as expanding to match all the characters be-tween the colon character and the at-sign.

From:.+@

It is good to think of the plus and asterisk characters as "pushy". For example, the following string would match the last at-sign in the string as the ".+" pushes outwards, as shown below:

From: *stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @* iupui.edu

It is possible to tell an asterisk or plus sign not to be so “greedy” by adding another character. See the detailed documentation for information on turning off the greedy behavior.

Extracting data using regular expressions

If we want to extract data from a string in Python we can use the findall() method to extract all of the substrings which match a regular expression. Let’s use the example of wanting to extract anything that looks like an email address from any line regardless of format. For example, we want to pull the email addresses from each of the following lines:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008 Return-Path: <postmaster@collab.sakaiproject.org>

for <source@collab.sakaiproject.org>; Received: (from apache@localhost)

Author: stephen.marquard@uct.ac.za

We don’t want to write code for each of the types of lines, splitting and slicing differently for each line. This following program uses findall() to find the lines with email addresses in them and extract one or more addresses from each of those lines.

```
import re

s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM' lst = re.findall('\S+@\S+', s)

print(lst)

# Code: http://www.py4e.com/code3/re05.py
```

The findall() method searches the string in the second argument and returns a list of all of the strings that look like email addresses. We are using a two-character sequence that matches a non-whitespace character (\S).

The output of the program would be:

['csev@umich.edu', 'cwen@iupui.edu']

Translating the regular expression, we are looking for substrings that have at least one non-whitespace character, followed by an at-sign, followed by at least one more non-whitespace character. The “\S+” matches as many non-whitespace characters as possible.

The regular expression would match twice (csev@umich.edu and cwen@iupui.edu), but it would not match the string "@2PM" because there are no non-blank char-acters ***before*** the at-sign. We can use this regular expression in a program to read all the lines in a file and print out anything that looks like an email address as follows:

```
# Search for lines that have an at sign between characters import re

hand = open('mbox-short.txt') for line in hand:

line  =  line.rstrip()

x = re.findall('\S+@\S+', line) if len(x) > 0:

print(x)

#  Code:  http://www.py4e.com/code3/re06.py
```

We read each line and then extract all the substrings that match our regular expression. Since findall() returns a list, we simply check if the number of elements in our returned list is more than zero to print only lines where we found at least one substring that looks like an email address.

If we run the program on mbox.txt we get the following output:

```
['wagnermr@iupui.edu']

['cwen@iupui.edu']

['<postmaster@collab.sakaiproject.org>']

['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']

['<source@collab.sakaiproject.org>;']
```

['<source@collab.sakaiproject.org>;']

['<source@collab.sakaiproject.org>;']

['apache@localhost)']

['source@collab.sakaiproject.org;']

Some of our email addresses have incorrect characters like "<" or ";" at the begin-ning or end. Let's declare that we are only interested in the portion of the string that starts and ends with a letter or a number.

To do this, we use another feature of regular expressions. Square brackets are used to indicate a set of multiple acceptable characters we are willing to consider matching. In a sense, the "\S" is asking to match the set of "non-whitespace characters". Now we will be a little more explicit in terms of the characters we will match.

Here is our new regular expression:

[a-zA-Z0-9]\S*@\S*[a-zA-Z]

This is getting a little complicated and you can begin to see why regular expressions are their own little language unto themselves. Translating this regular expression, we are looking for substrings that start with a ***single*** lowercase letter, uppercase letter, or number "[a-zA-Z0-9]", followed by zero or more non-blank characters ("\S*"), followed by an at-sign, followed by zero or more non-blank characters ("\S*"), followed by an uppercase or lowercase letter. Note that we switched from "+" to "*" to indicate zero or more non-blank characters since "[a-zA-Z0-9]" is already one non-blank character. Remember that the "*" or "+" applies to the single character immediately to the left of the plus or asterisk.

If we use this expression in our program, our data is much cleaner:

```
Search for lines that have an at sign between characters

The characters must be a letter or number

import re
```

```
hand = open('mbox-short.txt') for line in hand:


line  =  line.rstrip()


x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line) if len(x) > 0:


print(x)


#  Code:  http://www.py4e.com/code3/re07.py

...


['wagnermr@iupui.edu']


['cwen@iupui.edu']


['postmaster@collab.sakaiproject.org']


['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']


['source@collab.sakaiproject.org']

['source@collab.sakaiproject.org']


['source@collab.sakaiproject.org']


['apache@localhost']
```

Notice that on the “source@collab.sakaiproject.org” lines, our regular expression eliminated two letters at the end of the string (“>;”). This is because when we append “[a-zA-Z]” to the end of our regular expression, we are demanding that whatever string the regular expression parser finds must end with a letter. So when it sees the “>” after “sakaiproject.org>;” it simply stops at the last “matching” letter it found (i.e., the “g” was the last good match).

Also note that the output of the program is a Python list that has a string as the single element in the list.

Combining searching and extracting

If we want to find numbers on lines that start with the string "X-" such as:

X-DSPAM-Confidence: 0.8475

X-DSPAM-Probability: 0.0000

we don't just want any floating-point numbers from any lines. We only want to extract numbers from lines that have the above syntax.

We can construct the following regular expression to select the lines:

^X-.*: [0-9.]+

Translating this, we are saying, we want lines that start with "X-", followed by zero or more characters (".*"), followed by a colon (":") and then a space. After the space we are looking for one or more characters that are either a digit (0-9) or a period "[0-9.]+". Note that inside the square brackets, the period matches an actual period (i.e., it is not a wildcard between the square brackets).

This is a very tight expression that will pretty much match only the lines we are interested in as follows:

```
Search for lines that start with 'X' followed by any non

whitespace characters and ':'

followed by a space and any number.

The number can include a decimal.

import re
```

```
hand = open('mbox-short.txt') for line in hand:


line  =  line.rstrip()


if re.search('^X\S*: [0-9.]+', line): print(line)


# Code:  http://www.py4e.com/code3/re10.py
```