

# FUTURE VISION BIE

One Stop for All Study Materials  
& Lab Programs



*Future Vision*

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : <https://hemanthrajhemu.github.io>

Gain Access to All Study Materials according to VTU,  
CSE – Computer Science Engineering,  
ISE – Information Science Engineering,  
ECE - Electronics and Communication Engineering  
& MORE...

Join Telegram to get Instant Updates: [https://bit.ly/VTU\\_TELEGRAM](https://bit.ly/VTU_TELEGRAM)

Contact: MAIL: [futurevisionbie@gmail.com](mailto:futurevisionbie@gmail.com)

INSTAGRAM: [www.instagram.com/hemanthraj\\_hemu/](http://www.instagram.com/hemanthraj_hemu/)

INSTAGRAM: [www.instagram.com/futurevisionbie/](http://www.instagram.com/futurevisionbie/)

WHATSAPP SHARE: <https://bit.ly/FVBIESHARE>

7.6	Letting the user choose the file name . . . . .	85
7.7	Using <code>try</code> , <code>except</code> , and <code>open</code> . . . . .	86
7.8	Writing files . . . . .	87
7.9	Debugging . . . . .	88
7.10	Glossary . . . . .	89
7.11	Exercises . . . . .	89
<b>8</b>	<b>Lists</b>	<b>91</b>
8.1	A list is a sequence . . . . .	91
8.2	Lists are mutable . . . . .	92
8.3	Traversing a list . . . . .	92
8.4	List operations . . . . .	93
8.5	List slices . . . . .	94
8.6	List methods . . . . .	94
8.7	Deleting elements . . . . .	95
8.8	Lists and functions . . . . .	96
8.9	Lists and strings . . . . .	97
8.10	Parsing lines . . . . .	98
8.11	Objects and values . . . . .	99
8.12	Aliasing . . . . .	100
8.13	List arguments . . . . .	100
8.14	Debugging . . . . .	102
8.15	Glossary . . . . .	105
8.16	Exercises . . . . .	105
<b>9</b>	<b>Dictionaries</b>	<b>107</b>
9.1	Dictionary as a set of counters . . . . .	109
9.2	Dictionaries and files . . . . .	110
9.3	Looping and dictionaries . . . . .	111
9.4	Advanced text parsing . . . . .	113
9.5	Debugging . . . . .	114
9.6	Glossary . . . . .	115
9.7	Exercises . . . . .	115

<b>10 Tuples</b>	<b>117</b>
10.1 Tuples are immutable . . . . .	117
10.2 Comparing tuples . . . . .	118
10.3 Tuple assignment . . . . .	120
10.4 Dictionaries and tuples . . . . .	121
10.5 Multiple assignment with dictionaries . . . . .	122
10.6 The most common words . . . . .	123
10.7 Using tuples as keys in dictionaries . . . . .	124
10.8 Sequences: strings, lists, and tuples - Oh My! . . . . .	124
10.9 Debugging . . . . .	125
10.10 Glossary . . . . .	125
10.11 Exercises . . . . .	126
<b>11 Regular expressions</b>	<b>127</b>
11.1 Character matching in regular expressions . . . . .	128
11.2 Extracting data using regular expressions . . . . .	129
11.3 Combining searching and extracting . . . . .	132
11.4 Escape character . . . . .	136
11.5 Summary . . . . .	136
11.6 Bonus section for Unix / Linux users . . . . .	137
11.7 Debugging . . . . .	138
11.8 Glossary . . . . .	138
11.9 Exercises . . . . .	139
<b>12 Networked programs</b>	<b>141</b>
12.1 Hypertext Transfer Protocol - HTTP . . . . .	141
12.2 The world's simplest web browser . . . . .	142
12.3 Retrieving an image over HTTP . . . . .	144
12.4 Retrieving web pages with <code>urllib</code> . . . . .	146
12.5 Reading binary files using <code>urllib</code> . . . . .	147
12.6 Parsing HTML and scraping the web . . . . .	148
12.7 Parsing HTML using regular expressions . . . . .	148
12.8 Parsing HTML using BeautifulSoup . . . . .	150
12.9 Bonus section for Unix / Linux users . . . . .	153
12.10 Glossary . . . . .	153
12.11 Exercises . . . . .	154

# Chapter 8

## Lists

### 8.1 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] []
```

## 8.2 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-th 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 relationship 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
```

## 8.3 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]]
```

## 8.4 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.

## 8.5 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 operations 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']
```

## 8.6 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 accidentally write `t = t.sort()`, you will be disappointed with the result.

## 8.7 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.



## 8.8 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.

## 8.9 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.

## 8.10 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>*

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.

## 8.11 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` operator.

```
>>> 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.

## 8.12 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.

## 8.13 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']
```

**Exercise 1:** Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`. Then write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

## 8.14 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

```
word = word.strip()
```

It is tempting to write list code like this:

```
t = t.sort()           # WRONG!
```

Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at:

[docs.python.org/library/stdtypes.html#common-sequence-operations](https://docs.python.org/library/stdtypes.html#common-sequence-operations)

The methods and operators that only apply to mutable sequences are documented at:

[docs.python.org/library/stdtypes.html#mutable-sequence-types](https://docs.python.org/library/stdtypes.html#mutable-sequence-types)

2. Pick an idiom and stick with it.

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

To add an element, you can use the `append` method or the `+` operator. But don't forget that these are right:

```
t.append(x)
t = t + [x]
```

And these are wrong:

```
t.append([x])         # WRONG!
t = t.append(x)       # WRONG!
t + [x]               # WRONG!
t = t + x             # WRONG!
```

Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

```
orig = t[:]
t.sort()
```

In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

#### 4. Lists, `split`, and files

When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the *guardian* pattern when it comes writing programs that read through a file and look for a “needle in the haystack”.

Let’s revisit our program that is looking for the day of the week on the from lines of our file:

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

Since we are breaking this line into words, we could dispense with the use of `startswith` and simply look at the first word of the line to determine if we are interested in the line at all. We can use `continue` to skip lines that don’t have “From” as the first word as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print(words[2])
```

This looks much simpler and we don’t even need to do the `rstrip` to remove the newline at the end of the file. But is it better?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

It kind of works and we see the day from the first line (Sat), but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever, and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a `print` statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output, but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable `words` right before line five. We even add a prefix “Debug:” to the line so we can keep our regular output separate from our debug output.



```

for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From' : continue
    print(words[2])

```

When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

```

Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range

```

Each debug line is printing the list of words which we get when we `split` the line into words. When the program fails, the list of words is empty `[]`. If we open the file in a text editor and look at the file, at that point it looks as follows:

```

X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan 5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

```

Details: <http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772>

The error occurs when our program encounters a blank line! Of course there are “zero words” on a blank line. Why didn’t we think of that when we were writing the code? When the code looks for the first word (`word[0]`) to check to see if it matches “From”, we get an “index out of range” error.

This of course is the perfect place to add some *guardian* code to avoid checking the first word if the first word is not there. There are many ways to protect this code; we will choose to check the number of words we have before we look at the first word:

```

fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])

```

First we commented out the debug print statement instead of removing it, in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use `continue` to skip to the next line in the file.

We can think of the two `continue` statements as helping us refine the set of lines which are “interesting” to us and which we want to process some more. A line which has no words is “uninteresting” to us so we skip to the next line. A line which does not have “From” as its first word is uninteresting to us so we skip it.

The program as modified runs successfully, so perhaps it is correct. Our guardian statement does make sure that the `words[0]` will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, “What might go wrong?”

**Exercise 2:** Figure out which line of the above program is still not properly guarded. See if you can construct a text file which causes the program to fail and then modify the program so that the line is properly guarded and test it to make sure it handles your new text file.

**Exercise 3:** Rewrite the guardian code in the above example without two `if` statements. Instead, use a compound logical expression using the `or` logical operator with a single `if` statement.

## 8.15 Glossary

**aliasing** A circumstance where two or more variables refer to the same object.  
**delimiter** A character or string used to indicate where a string should be split.  
**element** One of the values in a list (or other sequence); also called items.  
**equivalent** Having the same value.  
**index** An integer value that indicates an element in a list.  
**identical** Being the same object (which implies equivalence).  
**list** A sequence of values.  
**list traversal** The sequential accessing of each element in a list.  
**nested list** A list that is an element of another list.  
**object** Something a variable can refer to. An object has a type and a value.  
**reference** The association between a variable and its value.

## 8.16 Exercises

**Exercise 4:** Download a copy of the file [www.py4e.com/code3/romeo.txt](http://www.py4e.com/code3/romeo.txt). Write a program to open the file *romeo.txt* and read it line by line. For each line, split the line into a list of words using the `split` function. For each word, check to see if the word is already in a list. If the word is not in the list, add it to the list. When the program completes, sort and print the resulting words in alphabetical order.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

**Exercise 5:** Write a program to read through the mail box data and when you find line that starts with “From”, you will split the line into words using the split function. We are interested in who sent the message, which is the second word on the From line.

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

You will parse the From line and print out the second word for each From line, then you will also count the number of From (not From:) lines and print out a count at the end. This is a good sample output with a few lines removed:

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

**Exercise 6:** Rewrite the program that prompts the user for a list of numbers and prints out the maximum and minimum of the numbers at the end when the user enters “done”. Write the program to store the numbers the user enters in a list and use the max() and min() functions to compute the maximum and minimum numbers after the loop completes.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

## 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 dictionary, 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 type that can be converted to 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. I won't explain why hash functions are so magical, but you can read more about it at [wikipedia.org/wiki/Hash\\_table](http://wikipedia.org/wiki/Hash_table).

**Exercise 1:** Download a copy of the file [www.py4e.com/code3/words.txt](http://www.py4e.com/code3/words.txt)

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.

## 9.1 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:

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 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 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:

```
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.

## 9.2 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

```

In our `else` statement, we use the more compact alternative for incrementing a variable. `counts[word] += 1` is equivalent to `counts[word] = counts[word] + 1`. Either method can be used to change the value of a variable by any desired amount. Similar alternatives exist for `-=`, `*=`, and `/=`.

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](http://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.

## 9.3 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 described 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.

## 9.4 Advanced text parsing

In the above example using the file *romeo.txt*, we made the file as simple as possible 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 separated 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 `tostr` but we will use the `deletestr` 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.

## 9.5 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data 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 “completely illogical”.

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

**Pretty print the output** Formatting debugging output can make it easier to spot an error.

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

## 9.6 Glossary

**dictionary** A mapping from a set of keys to their corresponding values.

**hashtable** The algorithm used to implement Python dictionaries.

**hash function** A function used by a hashtable to compute the location for a key.

**histogram** A set of counters.

**implementation** A way of performing a computation.

**item** Another name for a key-value pair.

**key** An object that appears in a dictionary as the first part of a key-value pair.

**key-value pair** The representation of the mapping from a key to a value.

**lookup** A dictionary operation that takes a key and finds the corresponding value.

**nested loops** When there are one or more loops “inside” of another loop. The inner loop runs to completion each time the outer loop runs once.

**value** An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word “value”.

## 9.7 Exercises

**Exercise 2:** Write a program that categorizes each mail message by which day of the week the commit was done. To do this look for lines that start with “From”, then look for the third word and keep a running count of each of the days of the week. At the end of the program print out the contents of your dictionary (order does not matter).

**Sample Line:**

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

**Sample Execution:**

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

**Exercise 3:** Write a program to read through a mail log, build a histogram using a dictionary to count how many messages have come from each email address, and print the dictionary.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

**Exercise 4:** Add code to the above program to figure out who has the most messages in the file. After all the data has been read and the dictionary has been created, look through the dictionary using a maximum loop (see Chapter 5: Maximum and minimum loops) to find who has the most messages and print how many messages the person has.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
```

```
Enter a file name: mbox.txt
zqian@umich.edu 195
```

**Exercise 5:** This program records the domain name (instead of the address) where the message was sent from instead of who the mail came from (i.e., the whole email address). At the end of the program, print out the contents of your dictionary.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

# Chapter 10

## Tuples

### 10.1 Tuples are immutable

A tuple<sup>1</sup> 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 parentheses 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:

---

<sup>1</sup>Fun fact: The word “tuple” comes from the names given to sequences of numbers of varying lengths: single, double, triple, quadruple, quintuple, sextuple, septuple, etc.

```
>>> 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')
```

## 10.2 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.

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 element 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.



## 10.3 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:<sup>2</sup>

```
>>> 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
```

---

<sup>2</sup>Python does not translate the syntax literally. For example, if you try this with a dictionary, it will not work as might expect.

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
```

## 10.4 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.

## 10.5 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.

## 10.6 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.

## 10.7 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.

## 10.8 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 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 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:

1. 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.
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. 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.

## 10.9 Debugging

Lists, dictionaries and tuples are known generically as *data structures*; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain 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 composition, or perhaps you write some code and forget the shape of your data and introduce an error. 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.

## 10.10 Glossary

**comparable** A type where one value can be checked to see if it is greater than, less than, or equal to another value of the same type. Types which are comparable can be put in a list and sorted.

**data structure** A collection of related values, often organized in lists, dictionaries, tuples, etc.

**DSU** Abbreviation of “decorate-sort-undecorate”, a pattern that involves building a list of tuples, sorting, and extracting part of the result.

**gather** The operation of assembling a variable-length argument tuple.

**hashable** A type that has a hash function. Immutable types like integers, floats, and strings are hashable; mutable types like lists and dictionaries are not.

**scatter** The operation of treating a sequence as a list of arguments.

**shape (of a data structure)** A summary of the type, size, and composition of a data structure.

**singleton** A list (or other sequence) with a single element.

**tuple** An immutable sequence of elements.

**tuple assignment** An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

## 10.11 Exercises

**Exercise 1:** Revise a previous program as follows: Read and parse the “From” lines and pull out the addresses from the line. Count the number of messages from each person using a dictionary.

After all the data has been read, print the person with the most commits by creating a list of (count, email) tuples from the dictionary. Then sort the list in reverse order and print out the person who has the most commits.

Sample Line:

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

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
```

```
Enter a file name: mbox.txt
zqian@umich.edu 195
```

**Exercise 2:** This program counts the distribution of the hour of the day for each of the messages. You can pull the hour from the “From” line by finding the time string and then splitting that string into parts using the colon character. Once you have accumulated the counts for each hour, print out the counts, one per line, sorted by hour as shown below.

```
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

**Exercise 3:** Write a program that reads a file and prints the *letters* in decreasing order of frequency. Your program should convert all the input to lower case and only count the letters a-z. Your program should not count spaces, digits, punctuation, or anything other than the letters a-z. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at [https://wikipedia.org/wiki/Letter\\_frequencies](https://wikipedia.org/wiki/Letter_frequencies).

# 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. For more detail on regular expressions, see:

[https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)

<https://docs.python.org/library/re.html>

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.

## 11.1 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 between 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.

## 11.2 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 characters *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 beginning 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 “>” at the end of “`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.

### 11.3 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>*

When we run the program, we see the data nicely filtered to show only the lines we are looking for.

```

X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000

```

But now we have to solve the problem of extracting the numbers. While it would be simple enough to use `split`, we can use another feature of regular expressions to both search and parse the line at the same time.

Parentheses are another special character in regular expressions. When you add parentheses to a regular expression, they are ignored when matching the string. But when you are using `findall()`, parentheses indicate that while you want the whole expression to match, you only are interested in extracting a portion of the substring that matches the regular expression.

So we make the following change to our program:

```

# 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.
# Then print the number if it is greater than zero.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+'), line)
    if len(x) > 0:
        print(x)

```

*# Code: <http://www.py4e.com/code3/re11.py>*

Instead of calling `search()`, we add parentheses around the part of the regular expression that represents the floating-point number to indicate we only want `findall()` to give us back the floating-point number portion of the matching string.

The output from this program is as follows:

```

['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..

```

The numbers are still in a list and need to be converted from strings to floating point, but we have used the power of regular expressions to both search and extract the information we found interesting.

As another example of this technique, if you look at the file there are a number of lines of the form:

Details: <http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772>

If we wanted to extract all of the revision numbers (the integer number at the end of these lines) using the same technique as above, we could write the following program:

```
# Search for lines that start with 'Details: rev='
# followed by numbers and '.'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9.]*)', line)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re12.py
```

Translating our regular expression, we are looking for lines that start with `Details:`, followed by any number of characters (`.*`), followed by `rev=`, and then by one or more digits. We want to find lines that match the entire expression but we only want to extract the integer number at the end of the line, so we surround `[0-9]+` with parentheses.

When we run the program, we get the following output:

```
['39772']
['39771']
['39770']
['39769']
...
```

Remember that the `[0-9]+` is “greedy” and it tries to make as large a string of digits as possible before extracting those digits. This “greedy” behavior is why we get all five digits for each number. The regular expression library expands in both directions until it encounters a non-digit, or the beginning or the end of a line.

Now we can use regular expressions to redo an exercise from earlier in the book where we were interested in the time of day of each mail message. We looked for lines of the form:

From [stephen.marquard@uct.ac.za](mailto:stephen.marquard@uct.ac.za) Sat Jan 5 09:14:16 2008

and wanted to extract the hour of the day for each line. Previously we did this with two calls to `split`. First the line was split into words and then we pulled out the fifth word and split it again on the colon character to pull out the two characters we were interested in.

While this worked, it actually results in pretty brittle code that is assuming the lines are nicely formatted. If you were to add enough error checking (or a big try/except block) to insure that your program never failed when presented with incorrectly formatted lines, the code would balloon to 10-15 lines of code that was pretty hard to read.

We can do this in a far simpler way with the following regular expression:

```
^From .* [0-9][0-9]:
```

The translation of this regular expression is that we are looking for lines that start with `From` (note the space), followed by any number of characters (`.*`), followed by a space, followed by two digits `[0-9][0-9]`, followed by a colon character. This is the definition of the kinds of lines we are looking for.

In order to pull out only the hour using `findall()`, we add parentheses around the two digits as follows:

```
^From .* ([0-9][0-9]):
```

This results in the following program:

```
# Search for lines that start with From and a character
# followed by a two digit number between 00 and 99 followed by ':'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0: print(x)

# Code: http://www.py4e.com/code3/re13.py
```

When the program runs, it produces the following output:

```
['09']
['18']
['16']
['15']
...
```



## 11.4 Escape character

Since we use special characters in regular expressions to match the beginning or end of a line or specify wild cards, we need a way to indicate that these characters are “normal” and we want to match the actual character such as a dollar sign or caret.

We can indicate that we want to simply match a character by prefixing that character with a backslash. For example, we can find money amounts with the following regular expression.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+' ,x)
```

Since we prefix the dollar sign with a backslash, it actually matches the dollar sign in the input string instead of matching the “end of line”, and the rest of the regular expression matches one or more digits or the period character. *Note:* Inside square brackets, characters are not “special”. So when we say `[0-9.]`, it really means digits or a period. Outside of square brackets, a period is the “wild-card” character and matches any character. Inside square brackets, the period is a period.

## 11.5 Summary

While this only scratched the surface of regular expressions, we have learned a bit about the language of regular expressions. They are search strings with special characters in them that communicate your wishes to the regular expression system as to what defines “matching” and what is extracted from the matched strings. Here are some of those special characters and character sequences:

`^` Matches the beginning of the line.

`$` Matches the end of the line.

`.` Matches any character (a wildcard).

`\s` Matches a whitespace character.

`\S` Matches a non-whitespace character (opposite of `\s`).

`*` Applies to the immediately preceding character(s) and indicates to match zero or more times.

`*?` Applies to the immediately preceding character(s) and indicates to match zero or more times in “non-greedy mode”.

`+` Applies to the immediately preceding character(s) and indicates to match one or more times.

`+?` Applies to the immediately preceding character(s) and indicates to match one or more times in “non-greedy mode”.

? Applies to the immediately preceding character(s) and indicates to match zero or one time.

?? Applies to the immediately preceding character(s) and indicates to match zero or one time in “non-greedy mode”.

[aeiou] Matches a single character as long as that character is in the specified set. In this example, it would match “a”, “e”, “i”, “o”, or “u”, but no other characters.

[a-zA-Z0-9] You can specify ranges of characters using the minus sign. This example is a single character that must be a lowercase letter or a digit.

[^A-Za-z] When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything *other than* an uppercase or lowercase letter.

( ) When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using `findall()`.

\b Matches the empty string, but only at the start or end of a word.

\B Matches the empty string, but not at the start or end of a word.

\d Matches any decimal digit; equivalent to the set [0-9].

\D Matches any non-digit character; equivalent to the set [^0-9].

## 11.6 Bonus section for Unix / Linux users

Support for searching files using regular expressions was built into the Unix operating system since the 1960s and it is available in nearly all programming languages in one form or another.

As a matter of fact, there is a command-line program built into Unix called *grep* (Generalized Regular Expression Parser) that does pretty much the same as the `search()` examples in this chapter. So if you have a Macintosh or Linux system, you can try the following commands in your command-line window.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

This tells `grep` to show you lines that start with the string “From:” in the file *mbox-short.txt*. If you experiment with the `grep` command a bit and read the documentation for `grep`, you will find some subtle differences between the regular expression support in Python and the regular expression support in `grep`. As an example, `grep` does not support the non-blank character `\S` so you will need to use the slightly more complex set notation `[^ ]`, which simply means match a character that is anything other than a space.

## 11.7 Debugging

Python has some simple and rudimentary built-in documentation that can be quite helpful if you need a quick refresher to trigger your memory about the exact name of a particular method. This documentation can be viewed in the Python interpreter in interactive mode.

You can bring up an interactive help system using `help()`.

```
>>> help()

help> modules
```

If you know what module you want to use, you can use the `dir()` command to find the methods in the module as follows:

```
>>> import re
>>> dir(re)
[. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

You can also get a small amount of documentation on a particular method using the `dir` command.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.

>>>
```

The built-in documentation is not very extensive, but it can be helpful when you are in a hurry or don't have access to a web browser or search engine.

## 11.8 Glossary

**brittle code** Code that works when the input data is in a particular format but is prone to breakage if there is some deviation from the correct format. We call this “brittle code” because it is easily broken.

**greedy matching** The notion that the `+` and `*` characters in a regular expression expand outward to match the largest possible string.

**grep** A command available in most Unix systems that searches through text files looking for lines that match regular expressions. The command name stands for “Generalized Regular Expression Parser”.

**regular expression** A language for expressing more complex search strings. A regular expression may contain special characters that indicate that a search only matches at the beginning or end of a line or many other similar capabilities.

**wild card** A special character that matches any character. In regular expressions the wild-card character is the period.

## 11.9 Exercises

**Exercise 1:** Write a simple program to simulate the operation of the `grep` command on Unix. Ask the user to enter a regular expression and count the number of lines that matched the regular expression:

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author
```

```
$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-
```

```
$ python grep.py
Enter a regular expression: java$
mbox.txt had 4175 lines that matched java$
```

**Exercise 2:** Write a program to look for lines of the form:

```
New Revision: 39772
```

Extract the number from each of the lines using a regular expression and the `findall()` method. Compute the average of the numbers and print out the average as an integer.

```
Enter file:mbox.txt
38549
```

```
Enter file:mbox-short.txt
39756
```

