3 Basic Python Programming: *for* loops and reading files

In the last tutorial we covered a range of topics including lists and how to define your own functions. In this tutorial we will continue this whirlwind introduction to Python and cover what are called *for* loops and also learn how to read information from files. The next tutorial, Tutorial 4, will be our 'easy day'—a breather. In that tutorial we will go through a set of exercises that will allow you to practice and apply what you've learned.

3.1 *For* Loops

Frequently we would like to perform the same actions on each element in a list—that is we would like to iterate through a list performing a sequence of commands. To do this we use the 'for' statement, which has the following format:

```
for name in List:
        command1
        command2
        …
```

Here's a concrete example:

```
>>> for noun in ['dog', 'cat', 'poodle', 'Maine Coon cat']:
...     print noun
...
dog
cat
poodle
Maine Coon cat
```

The word 'noun' in the 'for noun in …' is nothing special—it's just the name I made up. I could have used 'n', 's1234', or whatever. The following all give identical results

```
>>> for n in ['dog', 'cat', 'poodle', 'Maine Coon cat']:
```

```
...     print n


>>> for s1234 in ['dog', 'cat', 'poodle', 'Maine Coon cat']:
...     print s1234


>>> for element_from_pet_list in ['dog', 'cat', 'poodle', 'Maine Coon cat']:
...     print element_from_pet_list
```

What is important in this example is that the name used in the *for* line:

```
>>> for n in ['dog', 'cat', 'poodle', 'Maine Coon cat']:
```

matches the name used in the print line:

```
...     print n
```

If they don't ....

```
>>> for n in ['dog', 'cat', 'poodle', 'Maine Coon cat']:
...     print noun
...
Traceback (most recent call last):
  File "<interactive input>", line 2, in ?
NameError: name 'noun' is not defined
>>>
```

Also, you'll notice that indenting is used to delimit the block of lines associated with the *for* loop. (The indenting is handled automatically in the interactive window. To end the indented block enter a blank line)

Let's look at another example:

```
>>> pets = ['dog', 'cat', 'poodle', 'Maine Coon cat']        1
>>> total = 0                                                2
>>> for pet in pets:                                         3
...     length = len(pet)                                    4
...     total = total + length                               5
...     print length                                         6
...
3
3
6
14
>>> total                                                    12
26
```

Here I put lines numbers on the right so I can talk about what I typed—the numbers are not part of what I typed in.

line 1: We create a list named 'pets'
line 2: We create a name 'total' with the value 0.
line 3: The start of the *for* loop. We are iterating through the list *pets*, each element of the list is in turn given the name *pet*. That is, the first time through the loop pet equals 'dog', the second time through

the loop pet equals ‘cat’, and so on. We loop through the indented block of code for each item in the pets list. After we process the last item, ‘Maine Coon cat’ we exit the loop.

lines 4 & 5: We set the name *length* to the length of whatever string is named ‘pet’. For example,

the first time through the loop
The value of *pet* is ‘dog’ and the length of ‘dog’ is 3. Line 5 is

```
 ...   total = total + length
```

The first time through the loop the value of *total* is 0 and the value of *length* is 3 so the following substitution takes place:

```
...   total = total + length
                |       |
...   total =   0   +   3
```

Now total equals 3

the second time through the loop
The value of *pet* is ‘cat’ and the length of ‘cat’ is 3. Line 5 is

```
 ...   total = total + length
```

The value of *total* is currently 3 (from the length of ‘dog’) and the value of *length* is 3 so the following substitution takes place:

```
...   total = total + length
                |       |
...   total =   3   +   3
```

Now total equals 6

the third time through the loop
The value of *pet* is ‘poodle’. The new value of total equals the old value (6) plus the length of ‘poodle’ (6). Now total equals 12.

the fourth time through the loop
The value of *pet* is ‘Maine Coon cat’. The new value of total equals the old value (12) plus the length of ‘Maine Coon cat” (14). Now total equals 26.

line 6: We print the value of length. The first time through the loop we print the length of ‘dog’, the next time ‘cat’, etc.).

As you can see in line 12, the value of total after the *for* loop is 26.

Let’s define a function, big_length, that does a similar thing:

```
def big_length(alist):
    """add the length of all the strings in the list"""
    total = 0
    for item in alist:
        total = total + len(item)
    return total
```

To perform the function big_length with the argument alist:	`def big_length(alist):`
First create the name 'total' with the value zero	`total = 0`
Then for every item in alist:	`for item in alist:`
Set the new value of total to be the old value of total plus the length of each item	`total = total + len(item)`
Finally, return the value of total	`return total`

Here's an example of big_length's behavior:

```
>>> big_length(pets)
26
>>> big_length(['apples', 'bananas'])
13
```

Let's define another function that uses a *for* loop:

```
def print_list(alist):
    """print each element of a list one per line"""
    for item in alist:
        print item
```

This time, just for practice, we'll define the function in a new program window. Save the program and then run it. The result of running it will be that the Python interpreter will now have a definition for 'print_list'. We can try the function using the PythonWin Interactive Window:

```
>>> print_list(pets)
dog
cat
poodle
Maine Coon cat
>>> grades = [92, 87, 91, 96, 100, 95]
>>> print_list(grades)
92
87
91
96
100
95
>>> print_list(['Ann', 'Ben', 'Clara', 'Sara'])
Ann
Ben
Clara
Sara
>
```

Exercise 3.1 – The Average

Can you write a function, average, that will give the average of a list of numbers?

```
>>> grades = [92, 87, 91, 96, 100, 95]
>>> average(grades)
93
>>> grades2 = [67, 75, 82, 78, 83]
>>> average(grades2)
```

```
    77
```

To see one solution, see the solution section at the end of this chapter.

3.2 Reading Files

Learning to program requires not only learning different statements but also learning little scripts (sequences of statements). In this way you can construct your program from large building blocks instead of laboring over it line by line. A script that is particularly useful is one for reading files. The basic template of this looks like this in English:

```
open a file for reading
read information from it
close the file
```

In Python it looks like this:

```
infile = open(infilename, 'r')
lines = list(infile)
infile.close()
```

Let's look at these lines one by one.

Opening the file

The first line calls the function, *open*. This function takes two arguments: the name of file to open, and the mode of the file. By 'mode' we mean that we can open a file for reading or we can open it for writing. In this case we want to read information from a file so the mode is 'r' (for '**r**ead'). The name of the file is a string and it represents the name of the file including possibly the path to that file.
For example, suppose I have a folder on my C drive called 'AI', in that folder is another folder called 'python', and in that folder is the file I want to read, 'sample.txt'. Windows[1] specifies the directory path for this as:

```
C:\AI\python\sample.txt
```

In Chapter 1 we noted that the backslash character '\' has special meaning in Python strings—namely that the next character should be interpreted in some special way. In order to get actual backslashes in a Python string we need to put a backslash before each of them. For example:

```
filename = 'C:\\AI\\python\\sample.txt'
```

Writing all these backslashes can become tedious, so in most cases, you can use a shorthand provided by Python. Put the single character 'r' (for '**r**aw string') at the front of your string (with no space):

```
filename = r'C:\AI\python\sample.txt'
```

Note that Python sees the same thing regardless of whether you type the extra backslashes or you put the 'r' in front (as long as you do one of them):

```
>>> print 'C:\\AI\\python\\sample.txt'
```

[1] Different operating systems specify paths differently. Windows specifies a path using backslashes (as in, \AI\Python\sample.txt); the Macintosh OS specifies a path using double colons (AI::Python::sample.txt) and Unix and Linux specify a path using forward slashes (/AI/Python/sample.txt),

```
C:\AI\python\sample.txt
>>> print r'C:\AI\python\sample.txt'
C:\AI\python\sample.txt
```

Now that we've identified our filename, we can open this file for reading:

```
infile = open(filename, 'r')
```

or

```
infile = open(r'C:\AI\python\sample.txt', 'r')
```

Reading information from the file
In the next line in the sample above,

```
lines = list(infile)
```

the list() function creates a list from all the lines of the file (in this case all the lines of sample.txt). This list is then assigned the name *lines*. For example, if sample.txt contained the lines:

```
Ann  541-1360
Ben  541-1298
Sara 524-9963
```

After the statement

```
lines = list(infile)
```

lines would equal

```
['Ann 541-1360\n', 'Ben 541-1298\n', 'Sara 524-9963\n']
```

(The '\n' represents the newline character.)

Closing the file
The final line of the example is:

```
infile.close()
```

which closes the file.

Try this three line script yourself:

```
infile = open(infilename, 'r')
lines = list(infile)
infile.close()
```

First create a text file, called nouns.txt containing the following lines.

```
dog
cat
poodle
Maine Coon Cat
```

(Or be creative and type in whatever you want!)
Using the PythonWin Interactive Window open the file (substituting the name of your path and filename for mine):[2]

```
>>> infile = open(r'C:\AI\python\nouns.txt', 'r')
```

If Python cannot find your file it will give the following error:

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'sample.txt'
```

If you get this error check your path and filename (the pathname and filename are case sensitive).

Next, read the lines of your file into a list:

```
>>> nouns = list(infile)
```

Close your file:

```
>>> infile.close()
```

and finally take a look at the lines list:

```
>>> nouns
>> nouns
['dog\n', 'cat\n', 'poodle\n', 'Maine Coon Cat\n']
```

The file that you read needs to be a text document. You can create a text document using Notepad, WordPad (save the file as a 'Text Document'), or Microsoft Word (save the file as 'text only with line breaks').

3.3 Sentence generator – version 2

Suppose I combine the above with the functions I defined in the previous tutorial for the sentence generator (§2.6.2):

```
#
#  2.8.2 Sentence Generator
#

import random

infile = open(r'C:\AI\python\nouns.txt', 'r')
n = list(infile)
infile.close()

v   = ['likes', 'saw']
adj = ['sad', 'happy', 'silly']
```

[2] If you are using Macintosh or Linux machines make sure you use the correct directory separators as described in footnote 1.

```
det = ['a', 'the']


def select_lex(alist):
    """Returns a random element from the list"""
    return random.choice(alist)

def np():
    """Returns a random noun phrase"""
    return select_lex(det) + ' ' + select_lex(adj) + ' ' + select_lex(n)

def s():
    """Return a random sentence"""
    return np() + ' ' + select_lex(v) + ' ' + np()
```

The only difference from the previous version is the block that is in bold—reading the noun list from a file. Let's look at what happens when we use the *np* and *s* functions:

```
>>> np()
'the sad Maine Coon Cat\n'
>>> s()
'the happy poodle\n saw the sad cat\n'
>>> print s()
the happy cat
 saw a happy Maine Coon Cat
```

This looks pretty good. The only difference in output compared to the output of our previous version is that we have the '\n', or newline characters. (Notice that we type 'print s()' we see the newline characters realized as actual newlines.)

Where did these come from?

If you look at the nouns list:

```
>>> n
['dog\n', 'cat\n', 'poodle\n', 'Maine Coon Cat\n']
>>>
```

you'll see that every noun string ends with a newline. The original file looked like this:

```
dog
cat
poodle
Maine Coon Cat
```

That is, the file started with the word, *dog,* followed by a newline; then it had the word *cat* followed by a newline and so on. When we read the file into a list, the strings in that list contain newlines. We will look at one way of getting rid of these unwanted newlines right now. In the next tutorial we will cover alternative methods.

3.3.1 string.replace

The method replace, substitutes a new substring for an old substring. The template for this method is as follows:

```
        s.replace(old, new)
```

Here are some examples:

```
>>> s1 = 'Ann likes a dog'
>>> s1.replace('dog', 'poodle')      returns a string with occurrences of 'dog'
                                     changed to 'poodle'
'Ann likes a poodle'
>>> s1
'Ann likes a dog'                    but it doesn't change the value of s1
>>> s2 = 'Ann likes a dog and Mary likes a dog, too!'
>>> s2.replace('dog', 'poodle')
'Ann likes a poodle and Mary likes a poodle, too!'
>>>
```

This example shows that *replace* replaces all occurrences of the pattern.

As we have just seen, *replace* doesn't change the value—it just returns a new string. We can change the value for a name by doing the following:

```
>>> s2 = s2.replace('dog', 'poodle')      s2.replace() returns a string, and this returned
                                          string is assigned as the new value of s2

>>> s2
'Ann likes a poodle and Mary likes a poodle, too!'
```

You can get rid of one or more characters by replacing them with an empty string '':

```
>>> greeting = 'To Whom It May Concern:'
>>> greeting.replace(':', '')
'To Whom It May Concern'           The colon is removed  (replaced by the empty string)
```

Suppose I have the string:

```
        s3 = 'A dog saw a happy cat'
```

(You may want to cover the answers in the right column.)

How can you replace the 'A' with a 'The' and assign the resulting string the name s3?	`s3 = s3.replace('A', 'The')`

Test this by typing the name 's3' and see if it returns what you would expect.	`>>> s3`
	`'The dog saw a happy cat'`
	Yep! It looks good.

Replace 'a' with 'the' and see if the result is as you would expect.	`>>> s3 = s3.replace('a', 'the')`
	`>>> s3`
	`'The dog sthew the htheppy cthet'`

Did you get *'The dog sthew the htheppy cthet'*? If so, why?	The 'a' in *saw* was replaced with 'the' to yield *sthew* and so on.

How can you change the replace pattern to prevent this? (You'll need context.)	`>>> s3 = 'The dog saw a happy cat'`
	`>>> s3 = s3.replace(' a ', ' the ')`
	`>>> s3`

	`'The dog saw the happy cat'`

How can you get rid of the word 'happy'?	`>>> s3 = s3.replace(' happy', '')`
	`>>> s3`
	`'The dog saw the cat'`

Let's return to the list of nouns we read in from a file, and the function select_lex:

```
def select_lex(alist):
    "Returns a random element from the list"
    return random.choice(alist)
```

When we ran the function it would return a string terminated by a newline:

```
>>> select_lex(n)
'Maine Coon Cat\n'
```

Can you revise select_lex so it returns a string without the newline?[3]

3.4 A Chomskybot.

Of course, the fundamental error of regarding functional notions as categorial appears to correlate rather closely with the extended c-command discussed in connection with (34). On our assumptions, most of the methodological work in modern linguistics is not subject to a general convention regarding the forms of the grammar. From C1, it follows that the appearance of parasitic gaps in domains relatively inaccessible to ordinary extraction is to be regarded as the traditional practice of grammarians. Comparing these examples with their parasitic gap counterparts in (96) and (97), we see that the speaker-hearer's linguistic intuition is not to be considered in determining nondistinctness in the sense of distinctive feature theory. On the other hand, any associated supporting element is rather different from an abstract underlying order.

The above text was created by a program called *Comskybot* created by John Lawler.[4] Each sentence consists of four components: the initial phrase, the subject phrase, the verb phrase, and the terminating phrase.

initial phrase	**subject phrase**	**verb phrase**	**terminating phrase**
Of course,	*the fundamental error of regarding functional notions as categorial*	*appears to correlate rather closely with*	*the extended c-command discussed in connection with (34).*
On our assumptions,	*most of the methodological work in modern linguistics*	*is not subject to*	*a general convention regarding the forms of the grammar.*

[3] Here are several possibilities:

```
def select_lex(alist):
    "Returns a random element from the list"
    return random.choice(alist).replace('\n', '')

def select_lex2(alist):
    "Returns a random element from the list"
    a = random.choice(alist)
    return a.replace('\n', '')
```

[4] See http://www-personal.umich.edu/~jlawler/foggy.faq.html#how

From C1, it follows that	*the appearance of parasitic gaps in domains relatively inaccessible to ordinary extraction*	*is to be regarded as*	*the traditional practice of grammarians.*
Comparing these examples with their parasitic gap counterparts in (96) and (97), we see that	*the speaker-hearer's linguistic intuition*	*is not to be considered in determining*	*nondistinctness in the sense of distinctive feature theory.*
On the other hand,	*any associated supporting element*	*is rather different from*	*an abstract underlying order.*

John Lawler calls this method of generating text as the 'American Chinese Menu' method—you choose one phrase from Column A, one from Column B and so on. The actual programming mechanics is basically the same as our simple sentence generator. Your task is as follows:

1) From http://www.zacharski.org/python download the files initial.txt, subject.txt, verbal.txt, and terminating.txt. (To download a textfile using a Windows machine, right-click on the link and chose 'Save Target As …')
2) Write a program that generates a paragraph consisting of 4 sentences.
3) For hints see the solutions section at the end of this tutorial

Solutions

```
def average(alist):
    "Return the average of a list of numbers"
    total = 0
    num = len(alist)
    for item in alist:
        total = total + item
    return total / num
```