COMPUTING WITH STRINGS

One critical part of computing is the ability to store and retrieve data stored in files. There are two basic types of data files: ***binary*** and ***plain text***. ***Binary*** files contain data that have been encoded in binary for storage and processing purposes; i.e. the data are stored as a sequence of bytes. ***Plain text*** files contain data that have been stored as text. In this case, all data are converted to text and then encoded using a mapping (e.g., ASCII or UNICODE) from each character to a unique numerical code. This is why, for example, if you were to open a binary file in a text editor, what you usually see is an unintelligible display of characters produced when each sequence of eight bits in the binary file is treated as a character code and translated accordingly.

Many binary files contain headers, or ***metadata***, that provide a set of instructions for how a program should interpret the stored data (e.g., the format and layout of the text in this document). This metadata is usually proprietary and is what allows for the creation of and distinction among special file formats (e.g., Microsoft Office Documents, JPEGs, and MP3s). In many cases, only the program that created the original binary file is capable of interpreting the metadata and decoding the stored data. This can be problematic if we have any information that needs to be manipulated by multiple programs. While this difficulty is in part overcome by recent movement to make file formats readable by multiple programs, such as the interchange between MS Office or Adobe programs, most of the time we have to fall back on the old standard and store data in *plain text* files to make the data accessible to other software! This issue is of special concern to individuals interesting in manipulating and analyzing sets of data, a process that may involve the use of multiple programs: e.g., specialized or personally written computer programs for processing, recording, or producing data, spreadsheet for entering and organizing the data, statistical software for analysis, and graphical display software for visualization. Thus, it has become a standard in data analysis to save data in text files to facilitate transfer of information from one program to the next.

In this chapter, we will take a first look at the problem of plain text file input and output, which at its most basic, is just a form of ***string processing***.

Do this: Look up ASCII and UNICODE on Wikipedia. What are the differences between the two encodings?

STRING PROCESSING

String processing, or the translation and manipulation of information stored as text, is a very important programming task. To understand what is involved in this task, we first need to review a bit about strings.

Strings

We have already spent quite a bit of time using and manipulating strings. In programming, strings are a special data type that describes any information stored as a sequence of characters. A ***character*** is a unit of information (a piece of data) that corresponds to a symbol in the written form of a natural language. In Python, a ***string*** is a literal formed by enclosing two or more characters in either single or double quotation marks, e.g.:

```
>>> str1 = 'spam'
>>> type(str1)
<type 'string'>
```

We have already done a bit of string manipulation. We know how to print strings and we've also (in the last chapter) had a bit of experience getting string-valued input from the user (in the Rock, Paper, Scissors game). We've also learned that strings are Python sequences and so can be manipulated in the same way we manipulate any other sequence. For example, we can perform the following operations:

```
>>> 'spam' + 'eggs'
'spameggs'

>>> 3 * 'spam'
'spamspamspam '

>>> (3 * 'spam') + (5 * 'eggs')
'spamspamspameggseggseggseggseggs'

>>> len('spam')
4

>>> for ch in 'Spam!'
               print ch
S p a m !
```

Basic sequence operations that can be performed on strings are all summarized in the table below:

Operator	Meaning
`+`	Concatenation
`*`	Repetition
`<string>[]`	Indexing
`<string>[:]`	Slice
`len(<string>)`	Length
`for <var> in <string>`	Iteration through characters

Strings as Objects

However, strings are more than just your typical Python sequence. Because a lot of basic problems in computing involve the manipulation of sequences of characters, there are a lot string-specific operations that are frequently performed. Because of this, Python strings are in fact objects and Python provides a set of special operations that can only be performed on strings, for example:

```
>>> s = 'Spam and Eggs'

>>> s.capwords() # capitalize the words
'Spam And Eggs'

>>> s.upper() # make all upper case
'SPAM AND EGGS'

>>> s.replace('and','or') # replace one part with another
'Spam or Eggs'

>>> s.count('a') # count no times a is in the string
2

>>> s.split() # split string into words
['Spam', 'and', 'Eggs']
```

A complete list of string object methods is provided below:

Method	Meaning
`.capitalize()`	capitalize only first letter
`.capwords()`	capitalize first letter of all words
`.lower()`	make whole string lower case
`.upper()`	make whole string upper case
`.count(substring)`	count # times substring occurs
`.find(substring)`	find first position where substring occurs
`.rfind(substring)`	like find but searches from the right
`.join(list)`	concatenates a list of strings
`.replace(oldsub, newsub)`	replaces old substring with new one
`.split(character)`	splits the list everywhere character appears, if no character is specified splits at whitespaces
`.ljust(width)`	left justify string in field of given width
`.center(width)`	center string in field of given width
`.rjust(width)`	right justify string in field of given width

Do This: Enter the following at the IDLE prompt:

```
>>> sentence = 'The quick brown fox jumped over the lazy dogs.'
```

Use the string methods and sequence operations introduced in the preceding two sections to manipulate `sentence` in various ways. Try each one out until you are sure you understand how each works.

INPUT/OUTPUT AS STRING MANIPULATION

Before we begin to manipulate strings, we first need to obtain textual data from some source. Text may be obtained through direct interaction with the user (e.g., through the keyboard) or may be loaded into a program through the process of reading a stored file. Let's start by looking at text obtained through interactive processes of input through the keyboard.

Raw Input

In the last chapter, we experienced getting textual data from the user first hand when we asked the user to provide a choice (`'Rock', 'Paper', 'Scissors'`) in the Rock, Paper, Scissors! game. This was done with the following command:

```
yourChoice = input('Please enter Rock, Paper, or Scissors: ')
```

Executing this command would result in the following:

```
>>> yourChoice = input('Please enter Rock, Paper, or Scissors: ')
Please enter Rock, Paper, or Scissors:|
```

where Python relinquishes control of the program to the user and waits to proceed until some input is entered at the prompt (after the text string). But what do we enter? Well, many of you may have been frustrated to realize that the only way to avoid the following:

```
>>> yourChoice = input('Please enter Rock, Paper, or Scissors: ')
Please enter Rock, Paper, or Scissors: Rock

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    yourChoice = input('Please enter Rock, Paper, or Scissors: ')
  File "<string>", line 1, in <module>
NameError: name 'Rock' is not defined
```

Was to type quotes around the string input so that it evaluates the string as a literal, as follows:

```
>>> yourChoice = input('Please enter Rock, Paper, or Scissors: ')
Please enter Rock, Paper, or Scissors: 'Rock'
```

Do this: Enter the following command in IDLE.

```
>>> answer = input('Enter a mathematical expression: ')
```

When prompted for input, enter the expression `2 + 9`. Then print the value stored in the variable `answer`.

What happened? The answer to that question explains why entering `Rock` as opposed to `'Rock'` in our Rock, Paper, Scissors example caused an error in the first place. The answer is simply that ***an input statement is a delayed expression***. Thus, when we enter the name, `Rock`, this is the same as executing the following assignment statement:

```
>>> yourChoice = Rock
```

The statement reads: lookup the value stored in the variable `Rock` and then store that value in the variable `yourChoice`. Since `Rock` was never defined, Python cannot find the name and gives you the result is a `NameError`. Putting quotes around the entered string is one way around this error, but it is kind of tedious to require that a user enter quotes each time s/he wants to input a string value.

A more effective way is just to tell Python not to evaluate the input. We can do this using an alternative function, **`raw_input`**, that is exactly like **`input`**, except it does not evaluate the expression that the user types. The expression is instead handed to the program as a string of text. For example, the following:

```
>>> yourChoice = raw_input('Please enter Rock, Paper, or
Scissors: ')
Please enter Rock, Paper, or Scissors: Rock
```

no longer results in an error. Moreover, if we try to access the value saved in yourChoice, we see that it is a string:

```
>>> type(yourChoice)
<type 'str'>
>>> yourChoice
'Rock'
```

Do This: We've seen that computers manipulate numbers by storing them in binary notation (sequences of zeros and ones) and that the computer contains circuitry designed to do arithmetic on those sequences. How then, does a computer manipulate strings? Pretty much in the same way; i.e., manipulating textual information is no different from manipulating numerical information. Computers manipulate strings by transforming character data into numerical data using an encoding such as ASCII or UNICODE. That numerical representation can then be translated into binary sequences and processed by the CPU. Interestingly, it is this same process that is at the foundation of ***encryption*** or the process of encoding information for the purpose of keeping it secret or transmitting it privately.

The study of encryption methods is known as ***cryptography*** and is a very important computer science subdiscipline. The translation of text from ASCII or UNICODE and back again is a very simple type of encryption called a ***substitution cipher***. Each character is replaced by a corresponding symbol. We can write a simple program to automate this process of turning the

messages into a sequence of numbers and back again. The algorithm for encoding a message is easy:

```
get the message to encode from the user
for each character in the message:
     convert each character to a number and print
```

Getting the message from the user is simple: we can just use the `raw_input` function:

```
message = raw_input('Please enter the message: ')
```

And we know that we can just use a for loop to iterate through every character in the message:

```
for ch in message:
```

Now all we need to do is convert each character to a number. The simplest approach is to use the ASCII standard, which we can do by using a built-in Python function **`ord`**, which provides the ASCII code for any character passed as an argument. Now we can assemble the encoding function as follows:

```
def encode:
    # get message
    message = raw_input('Please enter the message: ')

    # iterate through the message
    for ch in message:
        # print out encoded message
        print ord(ch), # adding a comma to print on one line

    print # extra print to move to next line
```

Decoding is a little more difficult. The overall decoding algorithm is pretty similar to the encoding problem:

```
get the sequence of numbers to decode from the user
for each number in the sequence:
     convert the number to the appropriate character
     add the character to the end of the decoded message
print out the message
```

Again, Python provides a function, `chr`, for converting an ASCII code back into a character. Getting individual ASCII codes, however, requires a bit of string processing.

For example, consider the following encoded message, obtained using the `raw_input` command:

```
encoding = '72 101 108 108 111 32 87 111 114 108 100 33'
```

To translate this code back into the original text, we need to translate it from a single string into a sequence of numbers that can be evaluated by the **chr** function. We can do this by first using the `split` method to split the string into a list of strings. We can then convert each element of the list into a number using the **`eval`** function. `eval` takes any string and evaluates it as if it were a Python expression. Here are some examples:

```
>>> eval('500')
500
>>> eval('3 + 7')
10
>>> x, y = 3.5, 4.7
>>> eval('x * y')
16.45
```

Conversion

You can also use standard Python conversion functions to convert between strings and other data types:

```
int(<expression>)
long(<expression>)
float(<expression>)
str(<expression>)
```

Using `split` and `eval`, we can write our decoder program as follows:

```
def decode():
    # get the coded message
    code = raw_input('Enter code: ')

    # loop through each substring and build ASCII message
    message = ''
    for numStr in code.split(): #
        asciiNum = eval(numStr) # convert digits to num
        message = message + chr(asciiNum) # append chr to message

    # output result
    print 'Decoded message is : ', message
```

String Formatting

We've been playing for a while with the problem of input, now lets take a minute to talk about output. One of the more important tasks in outputting information is ensuring the readability and ease of processing saved data. In other words, we want to make it as easy to input any stored information back into a program as possible. One way to accomplish this task is to pay careful attention to the format of data.

Consider the following program, which illustrates the mathematical concept known as chaos:

```
# program: chaos.py

def main():
    print 'This program illustrates a chaotic function'
    x = input('Enter a number between 0 and 1: ')

    for i in range(10):
        x = 3.9 * x * (1 - x)
        print x

main()
```

Upon executing, this program will produce the following:

```
This program illustrates a chaotic function
Enter a number between 0 and 1: .5
0.975
0.0950625
0.335499922266
0.869464925259
0.442633109113
0.962165255337
0.141972779362
0.4750843862
0.972578927537
0.104009713267
```

Which is nice, but wouldn't it be a lot more readable like this:

```
0.97500
0.09506
0.33550
0.86946
```

```
0.44263
0.96217
0.14197
0.47508
0.97258
0.10401
```

In other words, to increase the readability (and prettiness, after all we are studying the visualization design!) of our output, wouldn't it be nice to be able to format what is printed?

We can do this by changing the last line of the program as follows:

```
print '%7.5f' % x
```

When used with strings, the % sign is a symbol that indicates a string formatting operation. In general, the ***string formatting operator*** is used as follows:

```
<template-string> % (<values>)
```

Percent signs inside the template-string mark ***slots*** into which the values are inserted. There most be exactly one slot per value. Each slot is described by ***format specifier*** that tells Python how the value for that slot should be output. In our example, the template contains a single specifier: `%7.5f`. The value of `x` will be inserted into the template in place of the specifier. The specifier tells Python that the value of `x` is a floating point number (`f`) that should be rounded to 5 decimal places (`.5`). It also tells us that the value should be printed in a total of 7 spaces (`7`). We can also specify formats for strings (`s`) and integers (`d`). The easiest way to get the hang of formatting is to play around. Refer back to the lecture for a list of formatting options.

Do this: What if we wanted our `chaos.py` program to be a better example of how chaos works? Well the definition of chaos is that extremely small changes in starting conditions can result in vastly different outcomes. Thus, to really illustrate chaos, we need to be able to compare multiple outcomes. But that would be difficult to do if all the results were printed in one long run on list. Ideally we would want to output a table, such as the one below:

```
    Start Value               1               2               3
--------------- --------------- --------------- ---------------
           0.50         0.97500         0.09506         0.33550
           0.20         0.62400         0.91503         0.30321
           0.10         0.35100         0.88842         0.38662
```

Certainly, this result is very readable and ... very useable. Below is another version of the `chaos.py` program that produces the example above. It is expands the original version to use `raw_input` to get a sequence of inputs (separated by spaces), converts those inputs to numerical values much the same way the decoding function did and then output the results in table format, where each row corresponds to the change over time in x for a given starting condition and each column is an interval of time (index). This version also has an additional input statement that allows the user to specify the number of iterations. Read through the program carefully and make sure you understand how it works.

```
# program: chaos.py

def main():
    print 'This program illustrates a chaotic function'

    # get input from user
    sequence = raw_input('Enter a sequence of nums between 0 and 1: ')
    sequence = sequence.split() # split the sequence by spaces

    # let user specify number of iterations
    numIterations = input('Enter number of iterations (integer): ')

    # print out table titles
    print '%15s' % 'Start Value', # comma insures printing on one line
    for i in range(numIterations):
        print '%15d' % (i + 1),
    print # move to next line

    # print out a dashed line under each heading
    line = '---------------'
    for y in range(numIterations):
        print line,
    print line # print the line the last time and move to a new line

    # iterate over the sequence of nums
    for num in sequence:
        x = eval(num)
        print '%15.2f' % x, # comma insures printing on one line

        for index in range(numIterations):
            x = 3.9 * x * (1 - x)
            print '%15.5f' % x,
        print  # move to next line

main() # invoke main
```

So, now we can print to the screen very nicely formatted data. Which is good, but...what happens when we exit IDLE? Well, all of our calculations and nicely formatted output will be lost! Ideally we would like to take our beautiful table and save it to a file so that we can use it later!

FILE PROCESSING

A file is a sequence of data stored in some secondary memory (usually some sort of disk drive). As discussed, files can have many formats, but today we are only concerned with text files. In Python, text files are very flexible and powerful file formats, especially considering how easy it is to convert back and forth between strings and other types.

Multi-line Strings

When we think of a text file, the first thing that comes to mind is probably something a bit like what you are reading right now (except, without the formatting). In other words, a text file is pretty much one very long string. Except...well, just by looking at this page we can tell that most text files are actually composed of more than a single line of text. But we already know from using the `raw_input` command that pressing <RETURN> (i.e., go to a new line) denotes the end of a string! How then can a file be a single string, if there are multiple lines. The answer is that when you press the <RETURN> or <ENTER> key you are actually entering a very special sequence of characters, called ***newline***. Newline is a convention for marking the ***end-of-line***. In Python, and many other programming languages, the newline sequence is the special notation **`'\n'`**. We can enter other special characters (such as those reserved for in-string operations) using a similar notation (e.g., **`'\t'`** is a tab, **`'\\'`** allows us to output a backslash, **`'\"'`** quotes)

Do This: Try it out. Run the following command. What happens?

```
>>> print 'Hello\tWorld\n\nGoodbye\n'
```

Do This: Tabs and newline characters are really useful when formatting output. Try out the function below, which is another version of our `chaos.py` program that uses tabs instead of large amounts of spacing to format our output table:

```
# program: chaos.py

def main():
```

```
    print 'This program illustrates a chaotic function'

    # get input from user
    sequence = raw_input('Enter a sequence of nums between 0 and 1: ')
    sequence = sequence.split() # split the sequence by spaces

    # let user specify number of iterations
    numIterations = input('Enter number of iterations (integer): ')

    # print out table titles
    print 'Start Value', # comma insures printing on one line
    for i in range(numIterations):
        print '\t%7d' % (i + 1),
    print # move to next line

    # print out a dashed line under each heading
    line = '-----------\t'
    for y in range(numIterations):
        print line,
    print line # print the line the last time and move to a new line


    # iterate over the sequence of nums
    for num in sequence:
        x = eval(num)

        print '%7.2f' % x, # comma insures printing on one line

        for index in range(numIterations):
            x = 3.9 * x * (1 - x)
            print '\t%7.5f' % x,
        print  # move to next line

main() # invoke main
```

File Processing

The details of file processing vary significantly among different programming languages. Nonetheless, they all based on the same basic steps. First, we need to associate a file with a name (variable). This process is called ***opening*** a file. Once opened, a file can then be manipulated by manipulating the name with which it has been associated. Next, we need a set of operations that can manipulate files (which, incidentally, are another type of data!). Typically we have two operations: ***reading*** and ***writing***! Finally, when finished with a file we need to close it. ***Closing*** a file makes sure that any overhead involved in associating the file

with our program is finished. Also, it is very important to close a file as soon as your our done writing because most often information written to files is often not saved until the file is closed!

Working with text files in Python is easy. The first step is association (or assigning a file to a name). This can be done with the open command:

```
<filename> = open(<name>, <mode>)
```

Here, <name> is a string that provides the name of the file on the disk. The <mode> parameter is either `'r'` or `'w'` depending on whether we are reading or writing. For example:

```
infile = open('mydata.txt', 'r')
```

would open the file `'mydata.txt'` for ***reading***. `infile` is a standard name for files to which data is going to be written. When opening a file for reading, the file provided as the first argument to the open function (e.g., `'mydata.txt'`) must exist in your Python directory, otherwise Python will not be able to find it an error will occur.

Closing a file is equally simple. To close a file, just use the `close()` method:

```
<filename>.close()
```

So, with our example above, the command:

```
infile.close()
```

would close the file, `infile`.

Once we open a file for reading, there are several commands we can use to read information from the file. Each command reads data in as a string. The difference between them is how much information is read and stored. These functions are as follows:

- **`<filename>.read()`:** returns the entire remaining contents of a file as a single string
- **`<filename>.readline()`**: returns the next line of the file as a string
- **`<filename>.readlines()`**: returns all lines in the file as a list of strings

Typically, we only use the `readline()` method because of the overhead involved in loading an entire file into memory (what if your file was really large?).

Do This: Below is a program that reads in and outputs data from a file using the `read` command. Implement and run the program. Modify the program to use first the `readline` and then the `readlines` command. How does the outcome change? Use the `myfile.txt` file as your input file (it is available for download from the course website).

```
# Program: printfile.py
# Description: prints a file to the screen

def main():
    fname = raw_input('Enter file name: ')
    infile = open(fname, 'r') # open for reading

    data = infile.read() # read in data
    print data # print data

    infile.close() # close file

main()
```

Do This: The following function processes only the first 5 lines of a file. Implement the program and run, using the `myfile.txt` file as your input file.

```
# Program: printpartialfile.py
# Description: prints a file to the screen

def main():
    fname = raw_input('Enter file name: ')
    infile = open(fname, 'r') # open for reading

    for x in range(5):
        line = infile.readline()
        print line[:-1]

main()
```

Why do we print only a slice of the line that was read in (i.e., `line[:-1]`). If you are having trouble answer this, try replacing the statement `print line[:-1]` with just `print line`. How is the outcome changed?

So, now that we know how to open, close, and read data from a file, all that is left is learning to save data to a file. Writing data to a file is pretty much the same as printing to the screen. As with reading, the first step is to open the file, using the open command. For example,

```
outfile = open('mydata.txt', 'w')
```

opens a file, `outfile`, for writing. `outfile` is a standard name for a variable that stores and output file. When opening a file for writing, the file provided as the first argument to the open function (e.g., `'mydata.txt'`) does not need exist, but will be created in your Python directory. However, if the file does exist, the act of opening for writing will overwrite its contents! So be careful!!!

Writing to a file is very simple. We just use the write method:

```
<filename>.write(<string>)
```

where the <string> passed as argument to the write command may be formatted in any of the ways discussed throughout this chapter. For example,

```
count = 5
outfile.write('The current count is %d.\n' %  count)
outfile.write('The next count is %d.\n'  % (count + 1))
```

writes the following to the file, `outfile`:

```
The current count is 5.
The next count is 6
```

And that is all there is to it!! But don't forget to close the file when you are done writing! Otherwise all your hard work might not get saved!

EXERCISES

1. Convert the `chaos.py` program (on page 13) so that it writes the data to the a file instead of printing to the screen. (10 pts)

2. For your histogram program write the following functions. As with last week you should just update your personal `histogram.py` file and submit that (plus the answer to #1) . An updated (answer) version is also available for download to use as a template/reference (`Histogram_Chapter6.py`) for any parts of last week's lab you were unable to complete.

   Two data files are available for your use: `total_losses.txt`, `paid_losses.txt`. These two files contain two sets of observations from the same larger dataset. `total_losses.txt` contains a list of the amount of reported flood damage (a loss) per county in the state of Texas over the past 22 years. `paid_losses.txt` reports similar data, except only includes counts of then number of losses that have been paid (i.e., those for whom folks suffering flood damage actually received compensation). The original source of the data is http://www.fema.gov/nfip/pcstat.htm.

   a. `importData`: function that imports observational data from a file. As stated, two files are available for download from the website. Open and view them in a text editor so that you can think about how to process the file (12 pts)

   b. `xTicks`: creates a list of tick labels, where each label is the midpoint of a bin (i.e., (binMax - binMin) / 2.0 (8 pts)

   c. `tickLabels`: converts a list of numeric tick labels to a list strings that can be later output to the screen or written to file(8 pts)

   d. `exportResults`: writes the data to a file in the format illustrated below. Here, Bin 1 label is the same as what you generated for the Bin 1 tick label. (20 pts)

Bin	Frequency	Relative Frequency
Bin 1 label	3	0.3
Bin 2 label	7	0.7

3. Make sure all of your functions work. First, make sure the `data = [<large list of numbers>]` statement is commented out or removed. Then, update your `main` program so that it calls `importData`. Load the either the data file. Calculate the

frequencies and relative frequencies of the data, given 10 bins. Determine the `xTicks` and the corresponding `tickLabels`. Use your `exportResults` function to write your data to a file. Submit the output file along with your final code. (5 pts)