Sequences: Strings, Lists, and Files

Read: C*hapter 5, Sections 11.1-11.3 from Chapter 11 in the textbook*

- **Strings:** So far we have examined in depth two numerical types of data: integers (int) and floating point numbers (float). Another type we have seen and we will study here is the **string** type. As we already know, a string is just a sequence of characters:

```
>>> str1 = “Hello”
>>> str2 = 'Help!'
>>> type(str1)
>>> print(str1, str2)
```

- You can think of a string as just a list of characters. For example

  'Help!' = ['H', 'e', 'l', 'p', '!']

  (Note that we write 'H' to denote the character H, just like a string of just one character; in this way, we will never confuse the character, say, 'x', with a variable named x.) Since a string is nothing more than a list of characters, we can use it anywhere we use lists. For example, try:

```
>>> for c in “Hello!”:
        print(c)
```

  This should print the string (list) “Hello!” one character at a time. In a list (say, a string) the elements are indexed starting from 0; so, in list(string) “Hello!” element no. 0 is 'H', element no. 1 is 'e', and element no. 5 is '!'. Using this indexing scheme, we can actually access portions of a string (or a list in general):

```
>>> greet = “Hello Bin”
>>> greet[0]
>>> greet[8]
>>> greet[0:8]
>>> greet[4:]
>>> greet[6:9]
>>> greet[-2]
>>> greet[9]
```

  A few things to notice here: The string greet has length 9 characters, as you can see, and as the function len(<str>) can tell you:

```
>>> len(greet)
```

  The characters are indexed starting from 0 to 8. If we want the portion of the string between indices a and b, ***without including element of index b,*** we write a:b. So in line 4 above, greet[0:8] is the portion between the 0'th and 7'th element in the string. In line 6 you tried to

give a negative index (greet[-2]); in this case Python starts counting from the end of the string towards the beginning. But in the last line you tried to index element no. 9 starting from the left (since 9 is positive), but there are only indices 0...8! So you fell out of the indexing range (as the error message tells you).

- **Characters:** A string is a list of characters. Each character is internally encoded according to a standard coding diagram (usually Unicode (or ASCII)). To see the code for a character, use function ord(<char>):

```
>>> ord('A')
>>> ord(']')
```

To see what character is encoded by a specific code-number, use function chr(<code>):

```
>>> chr(65)
>>> chr(245)
>>> print(chr(68))
>>> print(chr(7))
```

- **Lists:** As we saw, strings are a special case of list. In general, a list is simply a sequence of items. The items don't need to be of the same type:

```
>>> mylist = [1, 2, 'F', -1.0, “three”]
>>> mylist
>>> type(mylist)
>>> len(mylist)
>>> type(mylist[4])
>>> type(mylist[3])
>>> yourlist = mylist[3:5]
>>> yourlist
```

- **List/String operations:** There are operations that work on lists (or strings, since they are lists, too), like + and *:

```
>>> 3*”Hello”
>>> “Hello”+” “+”Bob”
>>> 4*[1, 2, 3]
>>> [1,2,3] + [4,5,6]
```

- **Lists/Strings as objects:** Lists/strings so far have been treated as simple sequences of elements. In fact they are much more than that: each list (or string) is actually an **object**. For now, it is enough for us to know that an object is a collection of data and operations that can act on these data. For example, for a string object, a piece of data inside the object is the list of characters itself, while an operation (or **method** as these internal operations are known) is invoked using the 'dot' notation. In the following example, mystring is a string object; we set its internal data

in the first line, and then we apply its split() method (which splits a string sequence at the spaces) on these data:

```
>>> mystring = “Hello Bob, my friend”
>>> mystring.split()
```

A list of string methods can be found in Table 5.2, and more about such methods can be found in Section 5.5. In fact, you don't have to define an object variable like mystring in order to assign a string object (“Hello Bob, my friend”) to it. Try

```
>>> “Hello Bob, my friend”.split()
```

The actual object then is “Hello Bob, my friend”, and mystring is just a variable whose value is a whole object. This is too simplistic, but for know this is enough **Obect-Oriented Programming (OOP)** for us. Lists are also objects; see Section 5.6 in the text.

- Recall that int(<expr>) and float(<expr>) convert the numerical value returned by <expr> to an integer or a float respectively. Recall also that eval(<str>) takes a string <str> of the form “<expr>”, strips it of the quotes, and then calculates the value of <expr>:

```
>>> eval(“4+5”)
>>> eval(input(“Give an expression: “))
Give an expression: 4+5
9
```

The function str(<expr>) does the opposite: it takes expression <expr>, it calculates its value, and then returns this value as a string:

```
>>> str(3+4*7)
'31'
```

- **String formatting:** Most of the times, we want our output to be a predetermined string with a few placeholders that we fill with our own choices of values. This predetermined string acts as a template and the formatting is done by a *method* (see above) of this string called format():

  <template-string>.format(<values>)

  The placeholders are denoted by {0}, {1}, {2}... In fact, we can even specify for a particular placeholder how exactly the value will be used. For example, try

```
>>> “Hello {0} {1}, you have ${2:10}.”.format(“Mrs.”, “Smith”, 100000)
```

The third placeholder {2:10} reserves 10 positions for the third value (here 100000).

- **Files:** A file is a sequence of data stored in the hard disk. Here we will deal with *text* files, i.e., the sequence of data is a string. This string can be seen to be a sequence of strings(lines) separated by a newline character ('\n'). For example the following string

  'Hello\nWorld\n\nGoodbye 32\n'

can be broken in 4 lines:

Hello (1st line)
World (2nd line)
(3rd line)
Goodbye 32 (4th line)

Actually this is what you are going to see if you print this string:

```
>>> print('Hello\nWorld\n\nGoodbye 32\n', end="")
```

where we set the ending of the print statement to `end=""` (empty string), because the default is `end="\n"` and `print()` outputs this `end` right after the string. Try

```
>>> print('Hello\nWorld\n\nGoodbye 32\n')
```

to see the difference.

- As you probably can guess, files are also objects with their own methods. The standard operations for a file are the following:
  - **Open a file**
    You can open a file and assign it to a variable as follows

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

    where <filename> is the name of the file we open, and mode is either `"r"` or `"w"` depending on whether we want to **read** or **write** in the file.
  - **Read from a file**
    After we open a file `<var>` for reading, we can read from it either the whole file as a *string* with

    ```
    <var>.read()
    ```

    or as a *list* of lines

    ```
    <var>.readlines()
    ```

    or just the next line as a *string*

    ```
    <var>.readline()
    ```

    By "next" we mean the next line after the last one we've read. In other words, there is a pointer that shows the current position in the file; initially it points to the beginning of the file, and every time you read another line it moves to the beginning of the next line in the file.
  - **Write to a file**

After we open a file <var> for writing, the file is empty (even if a file with <filename> existed, **it will be erased** – be very careful of this, or you'll lose your data!). Then we can write to the file by printing strings to it:

```
print(..., file=<var>)
```

- Close a file
  After we are done with a file we need to close it (for technical reasons):

```
<var>.close()
```

- The general sequence of actions on a file should be:
  - Open a file (for reading or writing)
  - Read or write
  - Close the file

  Here is an example:

```
>>> myfile = open("example.txt","w")
>>> print("First line",file=myfile)
>>> print("Second line",file=myfile)
>>> print("Third line",file=myfile)
>>> myfile.close()
>>> myfile = open("example.txt","r")
>>> file_contents = myfile.read()
>>> file_contents
>>> print(file_contents)
>>> test = myfile.readline()
>>> test
>>> myfile.close()
```

Note that after you read the whole file in `file_contents`, when you attempt to read a line and put it in `test`, `test` is the empty string. This happened because when you read the whole file, the file pointer points to the end of the file; hence, when you tried to read again from it, you read nothing. One way to "rewind" the pointer is to close and reopen the file:

```
>>> myfile = open("example.txt","r")
>>> for li in myfile.readlines():
        print(li,end="")
>>> myfile.close()
>>> myfile = open("example.txt","r")
>>> for li in myfile:
        print(li,end="")
>>> myfile.close()
```

Note that in the two for-loops above infile.readlines() and myfile itself are treated in the same way, namely as a sequence of lines.

Practice problems

1. Do all problems in Chapter 5.
2. Change username.py in the text to concatenate the first 2 letters of the user's first name and the last name. Change it to concatenate the first and last name in an interleaved fashion, e.g., “Johhny Doel” should produce “JDoohehlny”.
3. Write a program that asks the user for a sentence (it must end in a full-stop '.'). Then it prints the number of words in this sentence and the sentence itself but without the final '.'
4. Repeat the previous problem, but this time count the number of characters in the sentence *without counting the spaces between words.*
5. Create a text file (possibly using the IDLE text editor or your favourite text editor) and write a few lines in it. Then write a program that opens the file, reads it, prints out the contents, and then closes it.
6. Repeat the previous problem, but now write back in the file only the first two lines. After your code finishes running, look into the text file to make sure that only the first two lines are there.
7. Repeat the previous problem, but now what you write back in the file is every line repeated 14 times. (Hints: There are a few things to notice here: (a) You can multiply strings. (b) See the difference when you type

   >>> “String\n”*4

   and

   >>> print(“String\n”*4)

   in the command line.
8. Write a command that prints out the number 234.546346536 so that the precision is 2 decimal places and using a total width of 6 characters.
9. Rewrite the chaos.py program to get 2 numbers as inputs from the user, and then print two columns (one for each input) side-by-side of 20 lines of calculation (one column for each input). Each column must have a width of 6 and every number a precision of 4 decimal places.
10. Make a text file of student names (first and last name, one student per line). Then write a program that reads such a file and replaces it with a file with the same names, but in front of each name there must be now a serial number (starting from 1). The new file must look “nice”, i.e., there must be three distinct columns, with the number columns having a width of 5 characters, and each of the name columns having a width of 20 characters.