M30299 – Programming

moodle.port.ac.uk

Week 4 – Worksheet P4: Strings and Files

Introduction

This worksheet is designed to acquaint you with Python's string type, and with reading and writing files. It assumes that you have a fair understanding of the material covered in the first few worksheets. Work though this worksheet at your own pace and, as always, feel free to experiment with the language using the shell until you fully understand each concept. You should aim to complete this worksheet by next week's practical session. Your solutions to the programming exercises should be written to a file pract4.py, and this file should be available for you to show one of the teaching staff in next week's practical.

The string data type

Using Pyzo's shell, some `print` statements, and the built-in function `type`, let's experiment with some string values and related numeric values:

```
"Hello World"
print("Hello World")
type("Hello World")
string1 = "Hello"
string2 = 'World'
print(string1, string2)
type(string1)
type(string2)
"42"
type("42")
42
type(42)
```

Notice here that: (i) a string value (or any string expression) entered directly at the shell prompt (as we have done with arithmetic expressions) causes Python to respond with its value (and so the quotes are included); (ii) `print` displays strings without quotes; (iii) we can use single or double quotes for strings; and (iv) values such as `"42"` and 42 have different types.

Basic string operations

Let's now experiment with the basic string operations: `+` (concatenation), `*` (repetition) and `len` (length). Type

```
string1 + string2
string1 + " " + string2
string1 * 5
string1 * 5 + string2
string1 + " " * 5 + string2
(string1 + " ") * 5 + string2
(string1 + "\n") * 5
print((string1 + "\n") * 5)
```

```
len("Hi")
len(string1)
len("")
```

Notice here the use of the special ***newline character*** `"\n"`, and that * has higher precedence than +.

String indexing and slicing

Try the following ***string indexing*** expressions:

```
myString = "Hello there"
myString[0]
myString[6]
print(myString[0], myString[6])
myString[-1]
myString[-7]
type(myString[6])
myString[11]
```

Notice in the last two lines that: (i) a single character is also a string; and (ii) if we attempt to access a character at a non-existent position, Python reports an error.

Now, let's try some ***string slicing*** expressions:

```
myString[2:5]
myString[0:5]
myString[2:]
myString[6:-1]
myString[:-4]
myString[:]
```

Notice that when one of the numbers is omitted in a slicing expression, Python will automatically assume you mean the start or end of the string.

Loops and strings

Type the following loop:

```
for ch in myString:
    print(ch)
```

Here we see that, since a string is a ***sequence***, we can loop through its elements (characters).

String methods

Let's now try some further string operations that take the form of ***methods***:

```
myString.upper()
"Don't SHOUT".lower()
myString.center(30)
myString.find("er")
myString.replace("e", "ai")
myString.count("e")
myString.count("z")
myString.split()
words = myString.split()
```

```
words[0]
words[1]
aString = "a:list:of:words"
aString.split(":")
```

Experiment with these operations until you are sure you understand what they do.

String formatting

String formatting is achieved using the `format` method and ***format specifiers*** at ***slots*** (slots are numbered starting from 0). For example, try:

```
x = 10.123456789
y = 20
print("The values are {0:0.4f} and {1:8}.".format(x, y))
```

Here, the first number `x` is formatted using 4 decimal places, and the second number `y` is displayed using 8 characters, right justified (note the leading spaces). After importing the math module, try:

```
print("The value {0:0.4f} is pi".format(math.pi))
print("The value {0:0.2f} is pi".format(math.pi))
print("The value {0:8.2f} is pi".format(math.pi))
print("The value {0:<8.2f} is pi".format(math.pi))
```

Try experimenting with formatting until you understand how it works.

Conversion to other types

To convert a value from a string to a numeric type, we can use `int` or `float`. We can convert ints and floats to strings using `str`. Try the following:

```
string1 = "85"
string2 = "42.5"
int(string1)
float(string2)
float(string1)
int("Hello")
int(string2)
x = 42
str(x)
```

Notice how Python reports an error if a conversion is not possible.

Numerical encoding of characters

Every character we use is represented using a binary sequence and therefore has an equivalent integer value (i.e. the integer represented by the same sequence of binary digits). The built-in function `ord` gives the numeric (ordinal) value of a character. Try:

```
ord("a")
ord("b")
ord("A")
ord("z")
ord("b") - ord("a") + 1
ord("z") - ord("a") + 1
```

The `chr` function does the opposite of `ord`; try;

```
chr(98)
chr(120)
chr(960)
chr(8364)
```

Navigating the filesystem

We'll now use Python's `os` module to move around the filesystem, listing the names of the files we see. Begin by importing the `os` module:

```
import os
```

Now, to find out the full ***pathname*** of the folder we are currently in, try:

```
os.getcwd()
```

To obtain a list of the names of the files and subfolders within this folder, do:

```
os.listdir()
```

Now, let's move to one of the subfolders. This will of course depend on what folders you have in your filespace. For example, to move to a folder called `programming` you should enter:

```
os.chdir("programming")
```

Now try `os.getcwd()` and `os.listdir()` to show the current folder path and its contents. Finally, to move back up to the parent folder, try:

```
os.chdir("..")
```

(followed by `os.getcwd()` and `os.listdir()`). Experiment with moving around the filesystem until you are comfortable with using these commands.

Reading files

Download the file `quotation.txt` from the unit web-site to a folder of your choice. Navigate to this folder using the commands learnt above. Type:

```
inFile = open("quotation.txt", "r")
```

to open the file, and then enter the following three lines to read the ***entire*** contents of the file as a string (using the `read` method), show this string, and then output the string.

```
contents = inFile.read()
contents
print(contents)
```

Close the file with:

```
inFile.close()
```

The `readlines` method reads the file contents into a ***list*** of strings (one for each line). Try:

```
inFile = open("quotation.txt", "r")
contents = inFile.readlines()
contents
inFile.close()
```

The following code will re-open the file, read the ***first line*** into a string using the `readline`

method, and show this string:

```
inFile = open("quotation.txt", "r")
line = inFile.readline()
line
```

Repeat the second and third statements above to read through the other lines of the file. When the `readline` method no longer gives you a new line (i.e. you have finished reading the file), close the file again:

```
inFile.close()
```

The final way to read a file is by treating it as a ***sequence*** of lines (strings), and using a `for` loop to loop though these lines:

```
inFile = open("quotation.txt", "r")
for line in inFile:
   print(line)
inFile.close()
```

(Remember that you'll need to press return twice after the `print` statement to tell the shell that you've finished the code within the loop.) Notice that the `print` statement above results in blank lines being output between each line of text. This is because each line in the file includes a newline character "\n", and `print` always adds a newline character to whatever it outputs. To prevent this, change the `print` statement above to:

```
print(line[:-1])
```

so that the strings' newline characters are not output.

Writing files

Creating and writing to textfiles is straightforward. To create a file `myfile.txt` for writing, we do:

```
outFile = open("myfile.txt", "w")
```

Now, to write to the file, we just use `print` statements specifying that we want our output to go to the file rather than to the screen. Try:

```
print("Hello World!", file=outFile)
print(42, file=outFile)
print("Goodbye World!", file=outFile)
```

Once you have finished writing lines of text to the file, close it in the normal way:

```
outFile.close()
```

and use an editor (e.g. Pyzo's) to look at its contents.

Programming exercises

Your attempted solutions to the following exercises should be written to a single file which you should call pract4.py. Begin by adding your details (name, student number, etc.) at the top of this file.

Strings

1. Write a `personalGreeting` function which, after asking for the user's name, outputs a personalised greeting. E.g., for user input Sam, the function should output the greeting `Hello Sam, nice to see you!` (note the details of the spaces and punctuation).

2. Write a `formalName` function which asks the user to input his/her given name and family name, and then outputs a more formal version of their name. E.g. on input Sam and Brown, the function should output `S. Brown` (again note the spacing and punctuation).

3. Copy the `kilos2pounds` conversion function from your pract1.py file. Modify this function so that its output takes the form of a message such as "12.34 kilos is equal to 27.15 pounds", where ***both*** the user's kilos value and the calculated pounds values are displayed to ***two decimal places***.

4. Suppose the University decides that students' email addresses should be made up of the first 4 letters of their surname, the first letter of their forename, and the final two digits of the year they entered the university, separated by dots. Write a function called `generateEmail` that outputs an email address given a student's details. (E.g., if the user enters the following information: Sam, Brown and 2022, the function should output:

   ```
   brow.s.22@myport.ac.uk
   ```

5. A teacher awards letter grades for test marks as follows: 8, 9 or 10 marks give an A, 6 or 7 marks give B, 4 or 5 marks give C, and 0, 1, 2 or 3 marks all give F. Using string indexing, write a function `gradeTest` which asks the user for a mark (between 0 and 10) and displays the corresponding grade.

6. Write a function `graphicLetters` which first asks the user to enter a word, opens a graphics window, and then allows the user to display the letters of the word at different locations by clicking the mouse once for each letter. (Use the `setSize` method of the `Text` class to make the letters appear big.)

7. Write a `singASong` function which outputs a "song" based on a single word. The user should be asked for the song's word, how many lines long the song should be, and how many times the word should be repeated on each line. For example, if the user enters the word "dum" and the numbers 2 and 4, the function should then output the following song (note that the spaces are important):

   ```
   dum dum dum dum
   dum dum dum dum
   ```

8. Write a function `exchangeTable` that gives a table of euros values and their equivalent values in pounds, using an exchange rate of 1.17 euros to the pound. The euros values should be 0, 1, 2, ..., 20, and should be right justified. The pounds values should be right justified and given to two decimal places (i.e. with decimal points lined up and with pence values after the points).

9. Write a `makeInitialism` function that allows the user to enter a phrase, and then displays the first letters of the words in capitals for that phrase. For example, if the

user enters "University of Portsmouth", the function should display UOP. (Hint: first use the `split` method to find the words in the inputted string.)

10. [harder] Write a `nameToNumber` function that asks the user for their name and converts it into a numerical value by adding up the "values" of its letters (where 'a' is 1, 'b' is 2, ... 'z' is 26). So, for example, "Sam" has the value $19 + 1 + 13 = 33$.

Strings and Files

11. Write a `fileInCaps` function which displays the contents of a file in capital letters. The name of the input file should be entered by the user.

12. [harder] The file `rainfall.txt` from the module's Moodle page contains rainfall data in mm for several UK cities for a particular day, in the form:

```
Portsmouth 9
London 5
Southampton 12
```

    Write a function `rainfallChart` that displays this data as a textual bar chart using one asterisk for each mm of rainfall; e.g., given the above data the output should be:

```
Portsmouth      *********
London          *****
Southampton     ************
```

    Now write a graphical version `graphicalRainfallChart` that displays a similar bar chart in a graphical window but uses filled rectangles instead of sequences of asterisks.

13. [harder] Write a `rainfallInInches` function that reads the rainfall data from rainfall.txt, and outputs the data to a file rainfallInches.txt where all the mm values are converted to inches (there are 25.4mm in an inch). The inch values should be given to two decimal places, so the Portsmouth line above will become:

```
Portsmouth 0.35
```

14. [harder] In Linux, there is a command called `wc` which reports the number of characters, words and lines in a file. Write a function `wc` which performs the same task. The name of the file should be entered by the user.