M30299 – Programming

Worksheet 4: Strings and Files

This worksheet teaches you about Python's string type. You will also learn about reading from and writing to files. We assume that you have a fair understanding of the material covered in the earlier worksheets.

As always, through this worksheet at your own pace. Once you have finished the instructions in this document, start working on the programming exercises and show us your solutions in the next practical session. For further support, book a one-to-one session with the Academic Tutors (Simon Jones or Eleni Noussi) using their Moodle page. Additionally, join the Discord channel using these instructions to get help with your questions.

The string data type

Using the shell, let's experiment with some string values:

Shell

```
"Hello World"
print("Hello World")
string_1 = "Hello"
string_2 = 'World'
string_1
string_2
```

Notice that we can use single or double quotes for strings.

A string value (or any string expression) entered at the shell (as we did in worksheet 2 with arithmetic expressions) causes Python to respond with its value. So, the quotes are included when Python responds.

Shell

```
print("Hello World")
print(string_1, string_2)
```

Note that the `print` function displays strings without quotes.

Let's use the built-in function `type` on our strings:

Shell

```
type("Hello World")
type(string_1)
type(string_2)
```

Check the type of the following two values (guess the outcome before you enter them in the shell):

Shell

```
"42"
type("42")
42
type(42)
```

Observe that values such as `"42"` and `42` have different types.

Basic string operations

Let's create some more string variables so we can experiment with the basic string operations: **+** (concatenation) and ***** (repetition):

Shell

```
course = 'Computer Science'
student_id = "up1234567"
print(course, student_id)
"Welcome to: " + course + " " + student_id
```

Next, try the following:

Shell

```
2 * student_id
"hey " + 4 * student_id
("Welcome to:" + " ") * 4 + course
"Welcome to:" + " " * 4 + course
"Your ID number is:" + (" " * 4) + student_id
"Your ID number is:" + " " * 4 + student_id
```

Notice that * has higher precedence than +.

Next, we see that `\n` (**the newline character**) is interpreted as a new line when printed:

Shell

```
print("Hello:" + " " + student_id)
print("Hello:" + "\n" + student_id)
print("Hello:\n" + student_id)
print((student_id + "\n") * 5)
```

Similarly, `\t` is the **tab character**:

Shell

```
print("Welcome to:\t" + course)
print("\n" * 2 + "Your ID number is:\t" + student_id)
```

Indexing and slicing strings

Length of a string

Strings have a length, which we can determine using the `len` function:

Shell

```
len("Hi")
len(string_1)
len("")
my_string = "Hello World"
len(my_string)
```

Indexing a string

Try the following string indexing expressions:

Shell

```
my_string[0]
my_string[10]
print(my_string[0], my_string[10])
my_string[6]
type(my_string[6])
```

Notice in the last two lines that a single character is also a string.

If we attempt to access a character at a non-existent index, Python reports an error:

Shell

```
my_string[11]
```

We can access the character **from the end** of the string with negative indices:

Shell

```
my_string[-1]
my_string[-6]
```

Slicing a string

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

Shell

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

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 at the shell:

Shell

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

(You may have to enter blank lines at the end.) Here we see that, since a string is a sequence, we can loop through its elements (characters).

String methods

Now try some further string operations that take the form of **methods**:

Shell

```
"Don't SHOUT".lower()
my_string = "Hello!"
my_string.lower()
my_string.upper()
```

Check what is inside `my_string` after you run the expressions above.

Next, try the following:

Shell

```
my_string.find("lo")
my_string.find("mp")
my_string.count("l")
my_string.count("z")
words = my_string.split()
words[0]
words[1]
a_string = "a:list:of:words"
a_string.split(":")
my_string.split("ll")
```

Once again, check the content of `my_string` after you have done the above.

Chaining methods

You can apply (string) methods one after another to an expression. Enter the following lines one by one in the shell:

Shell

```
"hello".upper()
"hello".upper().replace('E', '3')
"hello".upper().replace('E', '3').replace('O', '0')
```

Character and Number Conversion

Every character we use in a string is represented by a binary sequence and therefore has an equivalent integer value. The `chr` function takes the integer and returns the corresponding character:

Shell

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

On the other hand, the `ord` function takes a character and returns the corresponding integer:

Shell

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

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

String formatting

Python includes the concept of `f`-strings, which provide a way to embed expressions inside strings, using curly braces `{}`. They're called `f`-strings because you must prefix the string with the letter `f`. Here is a basic example:

Shell

```
name = "Alice"
age = 22
height = 165.65
print(f"My name is {name}, I am {age} years old and I am {height} cm tall.")
```

You can add a **format specifier** after a colon to control how the values of the embedded expressions are formatted:

Shell

```
x = 10.123456789
y = 20
print(f"The values are {x} and {y}.")
print(f"The values displayed in a field of 8 characters: {x:8.4f},{y:8}")
```

Here we format `x` using 8 characters in total and to two decimal places, and we format `y` using 8 characters in total. Spaces are added for padding. Count these padding spaces and the digits (and decimal point) in the outputted `x` and `y` values to check that they both add up to 8.

Now try:

Shell

```
from math import pi
print(f"pi is approximately {pi}")
print(f"pi rounded to 2 decimal places is {pi:.2f}")
print(f"pi rounded to 4 decimal places is {pi:.4f}")
print(f"pi displayed in 8 characters is {pi:8.2f}")
print(f"pi displayed in 8 characters is {pi:<8.2f} (left justified)")
```

In the last statement, we use the `<` symbol to give a left justified format (so that padding spaces are added after the value.

We can also assign formatted strings to variables. Try the following in the shell:

Shell

```
message = f"pi rounded to 2 decimal places is {pi:.2f}"
message
print(message)
```

Navigating the filesystem

Computers organise data on different drives and folders. This structure is called the **filesystem**. We'll now use Python's `os` module to move around the filesystem and list the names of the files or folders that we see. Begin by importing the `os` module in the shell:

Shell

```
import os
```

Every file or folder on your computer has a **pathname**. This is a string that represents the address of that item. Let's find out the full pathname of the folder we are currently in:

Shell

```
os.getcwd()
```

To get a list of the names of the files and subfolders within this folder, enter:

Shell

```
os.listdir()
```

The shell always starts in a single location (folder) and you can move to other folders by executing the `chdir` (short for "change directory") function of the `os` module. Suppose for example we have a folder called "text_files". We could move to this folder by running the following:

Shell

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

Finally, to move back up to the parent folder, you would enter:

Shell

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

Use the `chdir`, `getcwd` and `listdir` functions to navigate around the filesystem on your computer until you are comfortable with them.

Reading files

Create a folder called "text_files" in your programming folder and download this `quotation.txt` file to the new folder. Next, import the `os` module, navigate to the new folder and check that `quotation.txt` exists:

Shell

```
import os
os.chdir("text_files")
os.getcwd()
os.listdir()
```

You might need to use the `chdir` function multiple times to navigate to the correct folder, depending on where your current working directory was at the start.

Once you can see the `quotation.txt` file, open it with the following line:

Shell

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

Now, read the **entire** contents of `input_file` using its `read` method and assign the text of it to the string variable `content` before printing it.

Shell

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

Once we are done with a file, we should always close it with the `close` method.

Shell

```
input_file.close()
```

There are other ways to read files:

Methods	Result	Example (run in shell)

`readlines()`	returns a list of strings where each element is a line in the file	`input_file = open("quotation.txt", "r")`
		
		`lines = input_file.readlines()`
		`for line in lines:`
		`  print(line)`
		
		`input_file.close()`

`readline()`	returns a single line of the file as a string (or an empty string if it reaches the end of the file)	`input_file = open("quotation.txt", "r")`
		
		`line = input_file.readline()`
		`while line:`
		`  print(line)`
		`  line = input_file.readline()`
		
		`input_file.close()`

Don't worry too much about the `while` loop, we will cover these later in the module.

You could alternatively use the `with` statement for file handling (this includes reading and writing to files). This way, you don't have to manually close the file (the file will automatically close once the body of the `with` is finished).

Try this for example. You may need to press enter multiple times to finish.

Shell

```
with open("text_files/quotation.txt", "r") as input_file:
    lines = input_file.readlines()
    for line in lines:
        print(line)
```

Writing files

Creating and writing text files is easier than reading them. To create a file `my_quotation.txt` for writing, we do:

Shell

```
output_file = open("my_quotation.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:

Shell

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

As before, remember to close the file with the `close` method:

Shell

```
output_file.close()
```

Your code should have created a file called `my_quotation.txt` inside the `text_files` folder. Open it in the editor and verify that your code has written our quote to it.

Programming exercises

Your solutions to the exercises should be written at the bottom of the `pract4.py` file. Solve as many problems as you can and be present at the next practical to get feedback.

Strings

1. Write a `personal_greeting` function which, after asking for the user's name, outputs a personalised greeting. For example, 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 `formal_name` function which asks the user to input their given name and family name, and then outputs a more formal version of their name. For example, on input Sam and Brown, the function should output S. Brown. Again, note the spacing and punctuation.
3. Copy the `kilos_to_ounces` conversion function from your `pract1.py` file into your `pract4.py` file. Modify this function so that its output takes the form of a message such as s "12.34 kilos is equal to 435.28 ounces" where both the user's kilos value and the calculated ounces 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 `generate_email` that outputs an email address given a student's details. For example, if the user enters the following information: Sam, Brown and 2024, the function should output: `brow.s.24@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 just string indexing (and not an if statement), write a function `grade_test` which asks the user for a mark (between 0 and 10) and displays the corresponding grade.
6. Write a function `graphic_letters` 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. You can use the `size` attribute of the `Text` class to make the letters appear big.

7. Write a `sing_a_song` 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 `exchange_table` that gives a table of euro values and their equivalent values in pounds, using an exchange rate of 1.17 euros to the pound. The euro values should be 0, 1, 2, ..., 20, and should be right justified. The pound 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 `make_initialism` 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.

Files

10. Write a `file_in_caps` function, which prints the contents of a file in capital letters. The name of the input file should be typed in by the user. Try your function with our `quotation.txt` file.

11. Jenny uses a text file to write down how much she spends on food every day of the week. Download the `spending.txt` file that she wrote last week, place it in your programming folder and open it to see its contents. Observe, for example, that Jenny spent £29.10 on Wednesday. Then write a function called `total_spending` that reads the `spending.txt` file and prints the total sum of money spent that week.

Harder exercises

12. [harder] Write a `name_to_number` 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.

13. [harder] The file `rainfall.txt` contains rainfall data in millimetres (mm) for several UK cities for a particular day. Open it to see the format of this file and download it.

Write a function `rainfall_chart` that displays this data as a textual bar chart using one asterisk for each mm of rainfall. For instance, the first three lines of the output should be:

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

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

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

    ```
    Portsmouth 0.35
    ```

15. [harder] On Unix operating systems, there is a command called `wc` which reports the number of characters, words, and lines in a file (you can read more about it here). Write a function `wc` which performs the same task. The name of the file should be entered by the user.