420-SN1-RE Programming in Science - Lab Exercise 15

October 20, 2024

Introduction

Goals for this lab:

- Formatting strings and printing
- Reinforcing ideas learned in previous labs

Formatting in Python

There are a few ways to print statements in python. The current *recommended* approach is to use `f-strings`. Here is an example.

```
school = 'Marianopolis College'
year_founded = 1908
city = 'Westmount'
province = 'Quebec'
country = 'Canada'
print(f'{school} was founded in {year_founded} in {city}, {province}, {country}.')
```

In this lab, we outline some important formatting options. For more complete documentation, see https://docs.python.org/3/tutorial/inputoutput.html.

`f-strings`

We often want to print data from a program in a form that is easily readable by humans. Using *formatted string literals* (`f-strings`) is a way that you can control the format of the output. To create an `f-string`:

- place the letter `f` or `F` immediately before a string literal
- use variable names or expressions inside of curly braces `{}` to place their values into the string

Remark: Python will compute certain expressions inside of `f-strings`. We can either assign names to these strings, or print them directly.

```
x = 5
y = f'{x} squared is {x**2} and cubed is {x**3}.'
print(y)
print(f'{x} squared is {x**2} and cubed is {x**3}.')
```

Formatting

When we place variables or expressions into strings, we can use optional "format specifications" to define how individual values are presented. The general structure is as follows (Note that the order matters):

`{expression[:[width][.precision][type]]}`

- type the expression (variable name)
- use an optional colon `:` (formatting follows)
- an optional width (specifies the minimum number of characters to print)
- an optional period `.` (separates width and precision)
- an optional precision (specifies the number of digits to print after the decimal)
- the type
  - `d` - for decimal integer (`int`)
  - `e` - for scientific float
  - `f` - for decimal float
  - `s` - for string (`str`)
- This is a simplification, see the complete specification for details. https://docs.python.org/3/library/string.html#formatspec.

Exercise 0: Practicing with f-strings

For this exercise, you will use the IDLE shell to experiment with f-strings. Nothing you do here will need to be submitted.

1. Start the IDLE shell and create a few values by typing the following assignment statements:

   ```
   x = 1000
   y = 21
   z = "Marianopolis"
   ls = [30, 2, -14]
   ```

2. You can obviously print these using normal calls to the `print()` function:

   ```
   print(x / y)
   for v in ls:
       print(v)
   ```

   However, the resulting text may not be well aligned.

3. Python's f-strings provide a way to get more control over your program's output. Try the following and compare the results to the prior example:

   ```
   print(f'{x / y:8.3f}')
   for v in ls:
       print(f'{v:4d}')
   ```

   The two things to notice are both how the alignment changes and how the precision restricts the number of digits printed after the decimal point.

4. You don't have to use format specifications if you don't need them. Try the following examples:

```
text = f"There were only {y} chairs for {ls[0]} students."
print(text)
print(f'{ls[1]} minus {ls[2]} equals {ls[1] - ls[2]}')
print(f'{z} College was founded in {x+908}')
```

5. If you do want more control over the formatting of the information, you can add the format specification after the optional colon character:

   ```
   n = 0.01
   while n < 10000:
       print(f'{n:8.2f}')
       n *= 10
   ```

   Here the format specification indicates that the value of `n` should be printed as a decimal using *at least* 8 characters, with exactly 2 characters after the decimal point.

6. Re-run the previous example using the 'e' type instead of the 'f' type. Notice how the output changes.

7. Remember that the width specifies a *minimum* width. Run the following example and see what happens:

   ```
   n = 1
   while n <= 100000:
       print(f'{n:4d}')
       n *= 10
   ```

   What happens when the number is too big to fit into only 4 characters?

Exercise 1: Letter frequencies in English

Similar to the previous lab, write a program, `letfreq.py`, to compute the letter frequencies of English text. Your program should prompt the user to enter a file name, the create a dictionary that represents the number of times each letter appears in the text.

Print the frequencies as decimal fractions of the overall number of letters in the text, ignoring all non-letter characters.

Use the formatting methods you have learned to print these frequencies with an accuracy of 4 decimal places.

If you test your program with the included file `alice.txt`, this first part of your program output should be as follows:

```
Letter Frequency
    a: 0.0797
    b: 0.0142
    c: 0.0244
    d: 0.0445
    e: 0.1252
```

In `alice.txt`, the exact value for the letter 'a', for example, is 9805/123014=0.07970637, the number of times the letter 'a' appears in the text divided by the total number of letters in the text.

Using your program, determine the most frequent letter in the text. What is the second-most frequent letter? Remember these, as it will help you in your work for Exercise 3.

Exercise 2: The Caesar Cipher

The process of encryption hides a message in order to make it difficult to read for anyone who does not know some secret or password. Today there are many excellent encryption algorithms available, but some form of encryption has existed for centuries.

A *Caesar Cipher* is a very simple, and very easy to crack, way of encrypting a message. The concept, at least for the English language, is to pick an integer j between 1 and 25, and exchange every letter of a message with the letter j positions later in the alphabet. Each letter i of the alphabet is transformed into letter $i + j$. If $i + j$ is greater than 26, we have to go back to the beginning of the alphabet. For example a shift of 3 positions would give Table 1 (assuming lower-case letters only).

Table 1: Caesar cipher table for shift of 3

Normal	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
3-shift	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

Using Table 1, one can encrypt the message "hello" as "khoor", and "zebra" as "cheud" by reading the original letters from the top row and substituting the corresponding letter from the bottom row.

We have provided a partially-written program, `caesar.py`, to apply this simple cipher. Your job is to complete the code to ask for a file name, a shift amount, and then transform the text. We've provided one important function you will want to use:

- `build_translation_table(s)` - Given an integer shift `s`, creates a dictionary where each key is the "original" letter and each value is the "shifted" letter.

Your program should do the following:

1. Ask the user to enter a file name.
2. Ask the user to enter a shift amount.
3. Read the contents of the file into a `str`. This will be the original text.
4. Create the appropriate translation table.
5. Using the translation table, create a new `str` representing the shifted version of the original text (be sure you don't shift any of the characters that are not letters).
6. Print this shifted text.

See the provided `caesar.py` file for some additional hints.

Test your finished `caesar.py` on the provided file `sample.txt`. Here is an example of a correct run of the encryption program:

```
Enter the file name to encrypt or decrypt: sample.txt
Enter the shift amount (0-25): 1
Uif rvjdl cspxo gpy kvnqt pwfs uif mbaz eph.
```

The original text was "The quick brown fox jumps over the lazy dog." The result above is correct if each letter is shifted by one.

Exercise 2: Decrypting with the Caesar Cipher

One interesting property of the Caesar cipher is that the same algorithm can be used to both encrypt and decrypt a message. If a text is encrypted with a shift `s`, the resulting encrypted message can be decrypted by re-applying the cipher with a shift `26-s`.

The file `message.txt` that is included in this lab has been encrypted using the Caesar cipher. Your job is to figure out the correct shift to use to decode the encrypted file. You could do this the "brute force" way, which would involve trying all 25 possible shift values. Alternatively, you could use a statisical approach based on the observed *letter frequencies*. It turns out that, for the English language, one particular letter is almost always the most common, and another is almost always the second-most common.

Use your letter frequency program to determine the most common and second most common letters in `message.txt`. See if you can guess what the original letters must be. Once you know this, you can deduce the correct shift! Here's a hypothetical example: Suppose the most common letter in English was 'a', but you see that the most common letter in the encrypted text is 'd'. This implies that the text was shifted forward by 3. So to decrypt the text in this case, you could just shift the encrypted text by 26-3 or 23.

Decrypt the `message.txt` file and include the decrypted text in your submission for the lab.

Submission requirements

Submit your `letfreq.py`, your completed `caesar.py`, and your decrypted text for `message.txt`. Place them in a ZIP file and upload them to Omnivox.