CptS 111 — Lab #10

Modules, String Formatting, and String Methods

Learning Objectives:

- Use three different string formatting methods
- Use a number of different string methods
- Access several modules using different `import` commands

Prerequisites:

- Exposure to different string formatting methods
- Exposure to string methods
- Basics of modules

Task 1: Different approaches to string formatting

String formatting in Python has evolved over time. In Python 2.x, `print` was a statement rather than a function. A `print` command was issued, without parentheses, followed by the string to be printed using `%`-formatting (called modulo-formatting because of the use of the `%` character). Both `%`-formatting and the `.format()` method were supported starting with Python 2.6 when the `.format()` method was introduced. In Python 3.0, `print()` became a function which could be used with both `%`-formatting and the `.format()` method. Many Python programmers preferred the old `%`-formatting approach and continued to use it. However, the introduction of f-strings in Python 3.6 seems to have started a change; f-strings are growing in popularity because of their simplicity and ease of use. Because Python 3.6, and above, currently supports all three approaches, it's a good idea to be familiar with all of them.

```
>>> # Let's start with a simple example.  Given a string, integer, and
>>> # float, how are they printed using the three string formatting
>>> # methods when their simplest forms are used?
>>> name = 'The Rock'
>>> wt_kg = 118
>>> wt_lb = 259.6
>>>
>>> # f-string method (henceforth, SF1)
>>> print(f'{name} weighs {wt_kg} kg or {wt_lb} lbs.')
The Rock weighs 118 kg or 259.6 lbs.
>>>
>>> # %-formatting method (henceforth, SF2)
>>> print('%s weighs %d kg or %f lbs.' % (name, wt_kg, wt_lb))
The Rock weighs 118 kg or 259.600000 lbs.
>>>
>>> # .format() method (henceforth, SF3)
>>> print('{} weighs {} kg or {} lbs.'.format(name, wt_kg, wt_lb))
The Rock weighs 118 kg or 259.6 lbs.
>>>
```

```
>>> # We note that SF1 and SF3 give the same result, but SF2 prints
>>> # extra zeros that aren't part of the variable.  To obtain the same
>>> # results as SF1 and SF3, we must do the following.
>>> print('%s weighs %d kg or %.1f lbs.' % (name, wt_kg, wt_lb))
The Rock weighs 118 kg or 259.6 lbs.
>>>
>>> # Next, try the following,
>>> print('%d weighs %d kg or %.1f lbs.' % (name, wt_kg, wt_lb))
>>>
>>> # What happened with the example above?
>>>
>>> # SF1 and SF2 have some special powers.  Let's first consider SF1.
>>> # Many programmers really like this new method because it's more
>>> # intuitive than the other two methods, and it's also simpler to use.
>>> # Consider the following,
>>> x = 5
>>> print(f'{x} squared is {x ** 2}.')
5 squared is 25.
>>>
>>> # We can obtain the same results using SF2 and SF3 but we can't use
>>> # x and x**2 inside the formatted string, i.e., between quotes
>>> # Instead, they're outside the formatted string, e.g.,
>>> print('%d squared is %d.' % (x, x ** 2))
5 squared is 25.
>>> print('{} squared is {}.'.format(x, x ** 2))
5 squared is 25.
>>>
>>> # For SF2, %s, %d, and %f are called conversion specifiers, and
>>> # as the name implies, they can be used to convert variables.  For
>>> # example:
>>> print('%s weighs %.1f kg or %d lbs.' % (name, wt_kg, wt_lb))
The Rock weighs 118.0 kg or 259 lbs.
>>>
>>> # or
>>> print('%s weighs %s kg or %s lbs.' % (name, wt_kg, wt_lb))
The Rock weighs 118 kg or 259.6 lbs.
>>>
>>> # Next, try the following,
>>> print('%d weighs %d kg or %.1f lbs.' % (name, wt_kg, wt_lb))
>>>
>>> # What happened with the example above?
>>>
>>> # Using an f-string is simpler and more human-friendly, especially
>>> # if we have a lot of replacement fields.  Let's consider a few
>>> # other examples using f-strings, but note that the same results
>>> # can be obtained using the other string formatting methods.
>>> print(f'Two to the power of 32 is {2 ** 32}.')
>>> name = {'first': 'Joseph', 'last': 'Gordon-Levitt'}
>>> print(f''{name['first']} {name['last']} is an American actor.'')
```

```
>>> print(f''{name['first'].upper()} {name['last'].upper()} is an\
 American actor.'')  # Use a \ to continue a string to the next line.
>>>
>>> # In the first example above, we've used numbers inside the f-string.
>>> # In the next two, we see how to use dictionary entries with
>>> # f-strings and also that we can use string methods inside
>>> # f-strings.
>>>
>>> # Note that we can use any of the three string formatting methods
>>> # in for-loops.  For example,
>>> names = ['Sam', 'Mohamed', 'Yang', 'Maria', 'Mani']
>>> for name in names:
        print('Hello, {}!'.format(name))
>>>
>>> # There's one last thing I want to mention about SF3, i.e., the
>>> # .format() method.  As with f-string string formatting (SF1),
>>> # we can use format specifiers with them.  Try the following.
>>> pi = 3.141592653
>>> print('Pi to two-decimal places is {:.2f}'.format(pi))
>>>
>>> # Finally, we can use uppercase F in f-strings:
>>> print(F'Pi to two-decimal places of accuracy is {pi:.2f}')
```

After you've completed this task, show your work to your TA to get credit. Keep your IDLE session open for the next task.

Task 2: Practicing some string methods

It's probably best to use string methods in a program to see their power, but we want to practice quite a few of them so we'll do this in an IDLE Shell window.

```
>>> # We'll start by assigning some strings that we'll use in the
>>> # examples of the string methods that follow.
>>> title = 'The Adventures of Buckaroo Banzai Across the 8th Dimension'
>>> poem = '\tGod in his wisdom made the fly\n\tand then\
 forgot to tell us why.\n\n\n'
>>> print(title)
>>> print(poem)
>>>
>>> # Try the following string methods, and note how the results differ
>>> # from the originals.
>>> title.lower()
>>> title.upper()
>>> title.capitalize()
>>> title.title()
>>> title.replace('s', '$')
>>> print(title)
>>>
>>> # From the last statement, we see that the original string hasn't
```

```
>>> # changed!  This is because strings are immutable, i.e., they can't
>>> # be changed.
>>> title.count('a')
>>> title.count('A')
>>> title.lower().count('a')
>>>
>>> # We used chaining in the last statement.  What is the result?  Let's
>>> # move on to .find() and .rfind() which give the index of the first
>>> # occurrence of a character, searching in the forward and reverse
>>> # directions, respectively.
>>> title.find('8')
>>> title.rfind('8')
>>> title.find('A')
>>> title.rfind('A')
>>>
>>> # Explain the results in the last 4 examples.
>>> title = title.replace('s', '$')
>>> print(title)
>>>
>>> # Explain the two statements above.
>>>
>>> # Let's now look at .rstrip(), .lstrip(), and .strip().
>>> print(poem)
>>> print(poem.rstrip())
>>> print(poem.lstrip())
>>> print(poem.strip())
>>>
>>> # Compare and explain the results for the last three examples.
```

After you've completed this task, **EXPLAIN** your work to your TA to get credit.

Task 3: Modules—`math` again

In the next three tasks, you're going to write short programs using three different modules. In this task, you're going to use the `math` module to create a pretty good calculator! Use the IDLE Editor window, and save your program as `lab10_t3.py`. The program consists of a single import statement, a `main()` function, and a call to `main()`. Don't forget a docstring with your `main()` function. Before starting this task, let me reintroduce the `eval()` function which you used in PA #5.

```
>>> # The eval() function can be used to evaluate strings representing
>>> # operations.  The examples below should give you a good idea of
>>> # how to use eval()
>>> eval('2 ** 32')
4294967296
>>> x = 25
>>> eval('(x ** 0.5) * x')
125.0
>>> eval('sum([1, 2, 3, 4, 5])')
```

```
15
>>> eval('{1: 'dog', 2: 'cat', 3: 'anteater'}')
>>>
>>> # The last example shows you why we needed to use eval() on the
>>> # dictionary in PA #5, i.e., the .read() method produced a single
>>> # string.  The eval() function evaluated the string to give just
>>> # the dictionary.
>>>
>>> # Also, as shown in class, the eval() function can be used if you
>>> # want to have more than one value entered when prompted for
>>> # input.  Enter your height and weight after entering the statement
>>> # below.
>>> ht, wt = eval(input('Enter your height and weight separated
>>>                           by a comma: '))
>>> ht
>>> wt
>>>
>>> # You can see why eval() is so useful!  Note that eval() only works
>>> # for numbers.  We'll learn soon how to enter two strings.
```

Now let's begin!

- After you've written your header, use the correct `import` statement to import the `math` module so that dot notation isn't required.
- **`main()`**: This void function prompts the user for `f(x)` which is a mathematical expression that uses the variable `x` and is stored as a string variable, then prompts for the value of `x` which is a float, next evaluates `f(x)` using the `eval()` function, and then prints both the expression and the result as shown in the examples below. Note that the input and output should be separated by a blank line.

```
Enter f(x): (x + 27) / x ** 2
Enter the value of x: -9.234

f(x) = (x + 27) / x ** 2
f(-9.234) = 0.20835787396438535

Enter f(x): cos(pi * x / 2) + sin(x) - exp(-x)
Enter the value of x: 0.5

f(x) = cos(pi * x / 2) + sin(x) - exp(-x)
f(0.5) = 0.580001660078117
```

When your program is working properly, demonstrate it to your TA to get credit.

Task 4: Modules—`calendar`

For this task, you're going to use the `calendar` module. Use the IDLE Editor window, and save your program as `lab10_t4.py`. The program will again consist of a single import statement, a `main()` function, and a call to `main()`. Don't forget a docstring with your `main()` function. Most of the commands you need to use are given.

- After you've written your header, use any `import` statement you prefer to import the `calendar` module.
- **`main()`**: This void function prompts the user for an integer month `month` and an integer year `year` (see example below). It then uses the following statements to print a calendar of the month for the year entered. Note that you'll have to alter this code depending on the **`import`** statement you used. My code uses dot notation with an alias.

  ```
  my_cal = cal.TextCalendar(cal.SUNDAY)
  my_month = my_cal.formatmonth(year, month)
  print(my_month)
  ```

  Next (still in `main()`) open an output file called `calendar.txt`. The command below creates a calendar of the year. The integer arguments are for the column width, the lines per week, the number of spaces between columns, and the number of columns of months. Use a print statement to print a calendar of the year you chose to the output file.

  ```
  my_year = my_cal.formatyear(year, 2, 1, 3, 3)
  ```

  You can use the `with` command to open the output file, or you can use the `open()` function together with the `.close()` method. The results shown below are for `my_month` and the contexts of the output file `calendar.txt`.

```
Enter month [1-12]: 4
Enter year [4 digits]: 2023

     April 2023
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

Part of output file `calendar.txt`:

```
                                2023

      January               February              March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7              1  2  3  4              1  2  3  4
 8  9 10 11 12 13 14   5  6  7  8  9 10 11   5  6  7  8  9 10 11
15 16 17 18 19 20 21  12 13 14 15 16 17 18  12 13 14 15 16 17 18
22 23 24 25 26 27 28  19 20 21 22 23 24 25  19 20 21 22 23 24 25
29 30 31              26 27 28              26 27 28 29 30 31
```

When your program is working properly, demonstrate it to your TA to get credit. Be sure to show your TA the `calendar.txt` file stored in your current working directory.

Task 5: Modules—`random`

For the last task, you'll use the `randint()` function from the `random` module to create a guessing game. Use the IDLE Editor window, and save your program as `lab10_t5.py`. The program consists of a single import statement, a `main()` function, and a call to `main()`. Don't forget a docstring with your `main()` function.

- After you've written your header, import just the `randint()` function from the `random` module. This function generates a random integer between its two arguments, e.g., `randint(1, 10)` will give an integer between 1 and 10 inclusive of 1 and 10.
- **`main()`**: This void function prompts the user for the number of players, generates a random integer between 1 and 100 (this is the target value, i.e., the number players are trying to guess correctly), prints the message shown in the example, initializes the first player to 0, and then uses a **counting** `for`-loop which iterates up to 100 times. The first statement in the `for`-loop prompts the appropriate player for their guess. After the guess has been obtained, an `if-elif-else` construct should be used to determine whether the guess is higher than the target value, lower than the target value, or equal to the target value. If the guess is equal to the target value, that player wins, and a message is printed as shown in the example below. When a player wins, use the `break` command to exit the `for`-loop. If a guess is incorrect, the player number is set to the next player using a command such as the following.

  ```
  player = (player + 1) % num_players
  ```

  The modulo function has to be used to cycle through all the players as shown in the example.

```
Enter number of players: 3
I'm thinking of a number between 1 and 100.  Guess what it is.
Player 1? 75
Too high.
Player 2? 35
Too high.
Player 3? 17
Too high.
Player 1? 8
Too low.
Player 2? 12
Too high.
Player 3? 10
Too high.
Player 1? 9

***PLAYER 1 WINS!***
```

When your program is working properly, demonstrate it to your TA to get credit.

This week's lab was relatively light (compared to the previous 2!), but I hope you learned a few things. You've come a long way!