CIS 210 Fall 2019
Week 5 Lab

- **Demo - if __name__ == '__main__'**
- **Exercises - Python string formatting**

1. **Demo – if __name__ == '__main__'**

   - Code can be dual purpose – including executable code (scripting code) (e.g., call to main) that we want to execute, or being imported for the definitions only (as in p34).
   - We can import user-defined modules (.py files), the same as modules from the standard library.
   - We did this for p41 test functions when we imported countss code from p34.
   - We had to first document out the code in p34 that called main to start executing, else that code would run during the import.
   - It is often the case, that we want to import a .py file for its functions or other definitions. Typically in such cases we want Python to execute the function or other definitions, but not any function calls (“scripting” code).
   - We can document out all the scripting code, but there is a better way: we can use Python’s __name__ variable to check whether a function is being executed in the __main__ namespace, or has been imported.
   - __name__ is ‘__main__’ when code is executed via Run Module. __name__ is [name of module] when code is being imported.

lab_importer_name_main.py ... also notice examples of string formatting.

2. **Exercises – Python string formatting – review tutorial below before trying these – at seats:**

   a. Write a Python function to generate letterhead for a given name. The function should take in a single string and print it out surrounded by your letterhead. A simple example is here, but be creative.

```
********************
*             CIS 210             *
********************
```

(b) Write a Python function that returns a number that has been “stringified” into money format – meaning it has a dollar sign, uses thousands comma separator, and is rounded to 2 digits. moneyformat should take in a number and return a string.

For example:

```
>>> moneyformat(5)
$5.00
>>> moneyformat(1420.8823423)
$1,420.88
```

3. Given two lists of strings: one containing class titles and the second containing the corresponding class location, write a Python function that prints a 2-column table with neatly aligned columns. How wide should the columns be? For example,

```
Name       Location
CIS 210    WIL 100
CIS 211    LLCS 101
```
**Python string formatting**

The Python string format method is really a mini-language (similar to turtle functionality).

https://docs.python.org/3.4/library/string.html

Python 3.6 and higher also has f-strings (no format method required):

https://docs.python.org/3/whatsnew/3.6.html#whatsnew36-pep498

Also (another take and includes deprecated-but-C-language-like % string formatting syntax):

https://realpython.com/python-f-strings/

This is a short introduction to the string format method, with some examples of handy string formatting. Notice that the string format method returns a NEW string in which the specified formatting has been applied to the original, or “format”, string.

**String format uses {} inside a string to indicate where values should be inserted:**

```python
>>> print('Hello, { }.format('world'))
Hello, world.

>>> print(f'Hello, world')
```

**Notice that we can do this using variables instead of string literals:**

```python
>>> print(f'Hello, {name}.

>>> name = 'world

>>> print('Hello, { }.format(name))
Hello, world.
```

**Arguments to the format method supply the values that should be inserted in the {} placeholders.**

Note that these placeholders are PART of the string into which we are inserting. Note that the arguments are simply Python expressions, and can be string literals, integers, variables, etc.

```python
>>> print(f"Hello, {'w' + 'o' + 'r' + 'l' + 'd'}."

Hello, world.

>>> print('Hello, { }.format('w' + 'o' + 'r' + 'l' + 'd'))
Hello, world.
```

**Python docs for string formatting:**

https://docs.python.org/3/library/string.html#formatstrings

https://docs.python.org/3/library/string.html#formatspec

**Old style formatting** (see text ch. 5.2.3) – deprecated in Python but still in use elsewhere

```python
>>> print('Hello, %s. % (\'world\'))
Hello, world.
```
Use \{\text{commands}\} to provide additional information to the string format method:
- \{\text{fillchar}\}[\text{align}][\text{width}][\text{,}][\text{decimal places}][\text{type}]
- \text{Fillchar} – what char to put in open places – used with \text{align}
- \text{Align} (\text{right} >, \text{left} < \text{or center} ^*)
- \text{Width} – how many total spots to use
- \text{Comma} – use comma for thousands separator
- \text{Decimal places} – how many digits after the decimal (better than the round function)
- \text{Presentation types} – \text{d} for integers, \text{f} for floats, \text{b} for binary, \text{c} for characters, \text{%} for percentages
- More fields are available if you look
- All fields are optional if you don’t need them

For example:

```python
>>> print(f'a = {1.2:.3f}, b = {2:0>6}, c = {0.3:.1%}')
a = 1.200, b = 000002, c = 30.0%
```

```python
>>> my_string = "a = {:.3f}, b = {:0>6}, c = {:.1%}"  
>>> my_string = my_string.format(1.2, 2, 0.3)  
>>> print(my_string)  
a = 1.200, b = 000002, c = 30.0%
```

Another example:

```python
>>> print(f'a = {1.2:.1f}, b = {2:^10}, c = {3:>10}, d = {0.3:.2%}')
a = 1.2, b =     2     , c =     3     , d = 30.00%
```

```python
>>> my_string = 'a = {:.1f}, b = {:^10}, c = {:>10}, d = {:.2%}'  
>>> my_string = my_string.format(1.2, 2, 3, 0.3)  
>>> print(my_string)  
a = 1.2, b =     2     , c =     3     , d = 30.00%
```

Print with commas example:

```python
>>> print(f'{123456789:,}')
123,456,789
```

```python
>>> my_string = "{:,}".format(123456789)  
>>> print(my_string)
123,456,789
```

Format/print in binary:

```python
>>> f'{44:b}'  
'101100'
```

```python
>>> print(f'{44:b}')  
101100
```

```python
>>> print('{:b}'.format(44))
101100
```

Now a character:

```python
>>> print(f'{101:c}')
e
```

```python
>>> print('{:c}'.format(101))
e
```

```python
>>> print('{:b}'.format(44))
101100
```