Notes - Monday 12/21/20

Day 2 of working on the Udemy Python Bootcamp course! We're going to pickup with more info about strings.

String Properties and Methods

Strings are "immutable" or cannot be changed. So for example:

- If I set `name = "Bob"`
- And then I try to set 'B' to 'R', I'm not allowed to. I'll get an error.

Concatenate

If I do want to change that 0 position, I will have to use concatenation.

1. Slice out the parts of the string I want to keep by creating a new string based on the first string. So in this case, I wan't to create the string "last_letters" based on "name".
2. Concatenate using + so I would take the "last_letters" and then Concatenate "R" to make Rob.

- `last_letters = name[1:]`
- `'R' + last_letters`

Using Concatenate is a way to put strings together. I can also add a variable to a string and run the cell to generate a longer statement.

```
x = 'Hello World'
x = x + ' it is nice out`
x
```

So the first run would say "Hello World it is nice out". And running the cell again would add on more "it is nice out". Concatenate could be added or multiplied. Remember that it's easy to confuse integers and strings. This is a con of the flexibility of the dynamic typing in Python. For example:

- `2+3` will yield 5
- `"2"+"3"` will Concatenate the strings "2" and "3" and yield 23

Functions

Variables can be modified with Python built in functions. First, define the variable as a string. In my example, I want `x = 'Hello World'`.

Now if I type x. in VSCode, a list of functions will appear. Syntax for these functions are `variable.function()`. If you don't include (), then Python will think you're just asking what that function will do.

Example of functions include:

- upper for all upper case letters in the string
- lower for all lower case letters in the string
- split will take your string and turn it into a list. The default split will divide the string at each whitespace. I can tell split where to split the list up.

Important Note: Comments

To add comments to my code, create a line that begins with # then write the comment out

String Formatting for Printing - String Interpolation

There are situations when you want to put a variable or objects ("inject") into a string so you can print it. For example:

```
my_name = "Jose"
print("Hello" + my_name)
```

There's 2 methods to do this:

1. .format()
2. f-strings (formatted string literals)

.format()

The .format() syntax is: `print('urnormalstringhere {}'.format('urobject'))`

You can add in multiple objects and multiple {}. The syntax would be: `print('urstringhere {} {} {}'.format('object1','object2','object3'))`

And within the {} you can assign a position number and then set an order based on the positions in the format string. The syntax would be: `print('urstringhere {0} {1} {2}'.format('object1','object2','object3'))`

You can also assign the object keywords and then set an order based on the keywords. The syntax would be: `print('urstringhere {x} {y} {z}'.format(x='object1',y='object2',z='object3'))`

Floating point number formatting

Let's say we have a variable that's a division problem. And then we want to print the answer from the division problem in a cleaned up format.

So the example code would be:

```
result = 100/77
result
```

Then we want to print the result but just with 3 decimal places. The syntax would be: `{value:width.precisionf}`

So the print code would look like:

```
print("The result was {r:1.3f}".format(r=result))
```

Where value r is the variable result. And then it's passed inside the {} to adjust the float formatting.

- Value is the variable we set in the .format()
- Width is how many white spaces you want to show.
- Precision (f) is how many decimal places do you want to show
- If you don't want the precision decimal places and just want to set a number of positions, leave the f out.

f-string String Interpolation

f-string is a simplified version of the .format(). So Let's say my variable is `name = 'Bob'` and I want to print a statement that says "Hello his name is Bob".

The example code would be:

```
name = "Bob"
print(f'Hello, his name is {name}')
```

So the syntax is `print(f'urstringhere {urobjecthere}')`.

Multiple objects can be used in an f-string. And .format() and f-string can be combined.

More reading can be found here:

- pyformat
- f-strings

Lists

`list = [1,2,3,4,5]`

Lists are ordered sequences that can hold objects. Could be a list of integers or a mixed list of data types. They use [] brackets and commas to seperate objects in the list. Syntax is `[x,y,z]` but we probably want to name the list or assign it a variable. So the syntax would be `variablename = [x,y,z]`.

Various methods can be called off them. Some things you can do with lists:

- indexing (this should feel like manipulating a string)
- slicing (this should feel like manipulating a string)
- nesting lists
- check the length using `len()`
- concatenate lists together
- add functions (remember to include () to have the functions work and not have Python tell you what things are)
  - Append adds a new object to the end of the list which is called affecting the list in place - syntax would be `list.append()`
  - Pop removes objects off the list or by index location. Index location can be in normal index order or reverse index. - syntax would be `list.pop(x)` (leave x out if you want the last object taken away)
  - Sort will put your list in order. Note that this is done in place. The sorted output needs to be assigned a variable in the next line after the sorted output happens in order to save the sorting. - syntax is `list.sort()`
    - To save the sorted output do the following:

      ```
      list.sort()
      sorted_list = list
      ```

    - Don't do:
      `sorted_list = list.sort()`

Unlike a string, lists can be changed or are mutable. So If I have a list showing 1-5, I could change 1 to ONE by using:

```
list[0] = "ONE"
list
```

Revision #5
Created 22 December 2020 00:57:23 by cba88
Updated 22 December 2020 18:33:05 by cba88