Project Explanation

When we are introduced to Python, the first statement we write is generally a call to the print() function. As we progress as programmers, we use the **`print()`** function to interact with the user through the Python shell and report the results of our computations. At some point in our development, we get to the point where we want to generate output as tables of numbers. Further, those numbers often have meaning in the form of physical units or currency that may require them to be formatted in a particular way.

You may have noticed in some of your programs that when you print a float, you get a very long string of digits after the decimal point that are not helpful in communicating the result. It's often desirable to print rows of numbers in columns that line up neatly. There are several methods that you can use to format numbers consistently in Python. Here, I'm going to describe the method of "formatted string literals" sometimes called f-strings.

The first thing you need to know is f-strings begin with an f followed by an apostrophe. An empty f-string looks like this: **`f''`**. This generates a null (empty) string … not very exciting. You can assign an f-string to a variable and put text between the quotes and it will just create a string **`s = f'this is an f-string'`** which is still not very exciting because you really don't need the **`f`** to do that. Things get exciting, though, when you start to insert f-string expressions.

f-string expressions are enclosed in curly braces from within an f-string. The simplest f-string expression just contains a variable name or Python expression. For example:
`s = f'result={result}'` just creates a string containing the text "result=" followed by the value of the variable result. This would have the same result as this statement:
`s='result='+str(result)`.

The advantage of the f-string is that you can specify the way you want result to look. For example, you can specify that result is in a 10-character wide column by adding **`:10`** after the variable … **`s = f'result={result:10}'`**. The column width is not required to be a constant … it could actually be a program variable. Suppose you want to vary the column width with a variable called width … you could replace **`:10`** with: **`{width}`** like this:
`s = f'result={result:{width}}'`. You can change the justification by adding a prefix to the column width specifier …

left-justified (**<**): **`s = f'result={result:<{width}}'`**
center-justified (**^**): **`s = f'result={result:^{width}}'`**
right-justified (**>**): **`s = f'result={result:>{width}}'`**

The case of float variables. When you simply print the value of a float variable, you don't always know how many digits are going to be printed to the right of the decimal point. You may get only one digit (e.g. 6.0) or you may get as many as 15 (e.g. 3.141592653589793). It all depends on the number represented by the float variable. In the case of expressing amounts of money (i.e. dollars) you really only care about two digits to the right of the decimal point (often called "precision").

f-string expressions give you a way to control the number of digits to the right of the decimal place and the precision of the strings you print. To specify exactly two digits after the decimal point, follow the

width parameter with **`.2f`**. For example,
`s = f'result={result:6.2f}'` will try to create a string representation of the value represented by **`result`** using a total of 6 characters. There will be a decimal point and the two digits of precision to the right (3 characters). This leaves 3 digits to the left of the decimal point. If **`result`** is a number larger than 999.99, digits will be added to the left and the width you specified will not be respected but there will still be two digits to the right of the decimal point. Here we say the precision is "fixed" at 2 decimal places (i.e. fixed precision).

If you are looking to print columns of fixed precision numbers, it's important to have an estimate of how big you expect your numbers to be and how many characters may be required to print the numbers. To print floating point numbers in consistent columns, it may be advisable to not specify any width at all. For example, **`s=f'result={result:0.2f}'`** will format the **`result`** variable as a float with two digits of precision using exactly the number of digits necessary and no more. Note that you can either specify zero 0 digits or leave the field blank … either way will give the same result.

The special case of currency. If you are trying to format monetary values, there are certain additional considerations. One important consideration is where you might want to place the currency symbol (e.g. dollar sign). Several different ways to format monetary values for printing in columns are shown below. Several involve nesting f-strings inside of f-strings. The best formatting for monetary values in columns are: the decimal points are always in the same column and the currency symbols are always in the same column. Try these out yourself. All of these examples print out the values in this list:
`amounts = [0.27,1.5,12.99,4.675,1204.1,12345]`

EXAMPLES

Symbols line up, decimal points don't always …

```
>>> for item in amounts:
        print(f'${item:6.2f}')


$  0.27
$  1.50
$ 12.99
$  4.67
$1204.10
$12345.00
```

Removing width criteria makes things worse …

```
>>> for item in amounts:
        print(f'${item:.2f}')


$0.27
$1.50
$12.99
$4.67
$1204.10
$12345.00
```

Adding width criteria makes things better …

```
>>> for item in amounts:
        print(f'${item:9.2f}')


$     0.27
$     1.50
$    12.99
$     4.67
$  1204.10
$ 12345.00
```

Nesting no-width format in a 15 character column …

```
>>> for item in amounts:
        print(f"{f'${item:.2f}':15}")


$0.27
$1.50
$12.99
$4.67
$1204.10
$12345.00
```

Right-justify the nested no-width format …

```
>>> for item in amounts:
        print(f"{f'${item:.2f}':>15}")


          $0.27
          $1.50
         $12.99
          $4.67
       $1204.10
      $12345.00
```

Formatted String Literals

Add a width of 10 for the nested format …

```
>>> for item in amounts:
        print(f"{f'${item:10.2f}':>15}")


$      0.27
$      1.50
$     12.99
$      4.67
$   1204.10
$  12345.00
```

Demonstrate two columns this way (method 1) …

```
>>> for item in amounts:
        print(f"{f'${item:10.2f}':>15}{f'${item*3:10.2f}':>15}")


$      0.27    $      0.81
$      1.50    $      4.50
$     12.99    $     38.97
$      4.67    $     14.02
$   1204.10    $   3612.30
$  12345.00    $  37035.00
```

Demonstrate two columns another way (method 2) …

```
>>> for item in amounts:
        print(f'${item:9.2f} ${item/3:9.2f}')
```

Method 1 allows for greater flexibility. All format parameters could be provided as variables. Method 2 is quick but not as flexible. Space between columns is hard-coded.

```
$     0.27 $     0.09
$     1.50 $     0.50
$    12.99 $     4.33
$     4.67 $     1.56
$  1204.10 $   401.37
$ 12345.00 $  4115.00
```

THE EXERCISES

Follow the instructions in *f-string_exercise-template.py* to complete the exercises. When you have successfully completed the exercises, this should be printed at the Python console …

To get these results, be sure to download and use the template file I provide.

```
_                    _
#1 ...
27 is a whole number
#2 ...
27 is <class 'int'>
#3 ...
pi to 6 digits is 3.14159
#4 ...
pi to 3 digits is 3.14
#5 ...
pi to 4 digits is 3.142
#6 ...
    27     3.14159      pi
#7 ...
    x         y         z
    27     3.14159      pi
#8 ...
$27.00    $3.14
#9 ...
  $27.00    $3.14
#10 ...
 $  27.00  $   3.14
```