2020

A Guide to f-string Formatting in Python

Jacqueline Masloff, PhD

Introduction

The release of Python version 3.6 introduced formatted string literals, simply called “f-strings.”

They are called **`f-strings`** because you need to prefix a string with the letter 'f' in order to get an f-string. The letter 'f' also indicates that these strings are used for formatting. Although there are other ways for formatting strings, the Zen of Python states that simple is better than complex and practicality beats purity--and **`f-strings`** are really the most simple and practical way for formatting strings.

To use formatted string literals, begin a string with f or F before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between **{ }** characters that can refer to variables or literal values. **`f-strings`** support extensive modifiers that control the final appearance of the output string. Expressions in **`f-strings`** can be modified by a format specification.

Format specifications are used within replacement fields contained within a format string to define how individual values are presented. Each formattable type may define how the format specification is to be interpreted.

Alignment

There are several ways to align variables in **`f-strings`**. The various alignment options are as follows:

Option	Meaning
<	Forces the field to be left-aligned within the available space (this is the default for most objects).
>	Forces the field to be right aligned within the available space (this is the default for numbers).
=	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+000000120’. This alignment option is only valid for numeric types. It becomes the default when ‘0’ immediately precedes the field width.
^	Forces the field to be centered within the available space.

Data Types

There are many ways to represent strings and numbers when using f-strings. The most common ones that you will need for this course are shown on the next page:

Type	Meaning
s	String format—this is the default type for strings
d	Decimal Integer. Outputs the number in base 10
n	Number. This is the same as **d** except that it uses the current locale setting to insert the appropriate number separator characters.
e	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6.
f	Fixed-point notation. Displays the number as a fixed-point number. The default precision is 6.
n	Number. This is the same as **g**, except that it uses the current locale setting to insert the appropriate number separator characters.
%	Percentage. Multiplies the number by 100 and displays in fixed (**'f'**) format, followed by a percent sign.

The following is a basic example of the use of **f-strings**:

```
x = 4.5
print(f'This will print out the variable x: {x}')
```

The output of this code snippet the following:

```
This will print out the variable x: 4.5
```

The variable, **x**, is enclosed in curly braces (**{ }**) and the **f-string** understands that **x** is a **float** and displays it as assigned.

If you want to change the number of decimals displayed you would write this:

```
x = 4.5
print(f'This will print out the variable x: {x:.3f}')
```

The output of this code snippet is the following:

```
This will print out the variable x: 4.500
```

A colon is now added after the variable, which now controls the formatting of the float, specifying that three decimal places are to be displayed.

Field Width

Passing an integer after the "**:**"will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```
table = ['Sjoerd','Jack','Dcab']
for name in table:
```

```
    print(f'{name:10}')

print()
table2 = [4127, 4098,  7678]
for num in table2:
    print(f'{num:10}')
```

The output of this code snippet is this:

```
Sjoerd
Jack
Dcab

      4127
      4098
      7678
```

For the list, **table**, the strings are left-aligned in a field width of 10, while the numbers in the list, **table2**, are right-aligned in a field width of 10.

The following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
print(f'Number    Square       Cube')
for x in range(1, 11):
    print(f'{x:2d}       {x*x:3d}          {x*x*x:4d}')
```

The output of this code snippet is this:

```
Number    Square       Cube
 1          1             1
 2          4             8
 3          9            27
 4         16            64
 5         25           125
 6         36           216
 7         49           343
 8         64           512
 9         81           729
10        100          1000
```

The tab character (**\t**) can also be used in an **f-string** to line up columns, particularly when column headings are used:

```
print(f'Number\tSquare\tCube')
for x in range(1, 11):
    print(f'{x:2d}\t\t{x*x:3d}\t\t{x*x*x:4d}')
```

This produces the same output as the previous example. Either way is acceptable in Python.

Formatting Floating Point Numbers

The following takes the previous program and converts **x** to a **float()** so that formatting of floating-point numbers can be demonstrated:

```
print(f'Number\t\tSquare\t\t\tCube')
for x in range(1, 11):
    x = float(x)
    print(f'{x:5.2f}\t\t{x*x:6.2f}\t\t{x*x*x:8.2f}')
```

The output of this code snippet is this:

```
Number      Square          Cube
 1.00        1.00          1.00
 2.00        4.00          8.00
 3.00        9.00         27.00
 4.00       16.00         64.00
 5.00       25.00        125.00
 6.00       36.00        216.00
 7.00       49.00        343.00
 8.00       64.00        512.00
 9.00       81.00        729.00
10.00      100.00       1000.00
```

This also demonstrates how the use of a value for width will enable the columns to line up.

The following program demonstrates the use of strings, decimals, and floats, as well as tabs for a type of report that is often produced in a typical Python program. Notice the use of the dollar sign (**$**) just before the variables that are to be displayed as prices.

```
APPLES = .50
BREAD = 1.50
CHEESE = 2.25

numApples = 3
numBread = 4
numCheese = 2

prcApples = 3 * APPLES
prcBread = 4 * BREAD
prcCheese = 2 * CHEESE

strApples = 'Apples'
strBread = 'Bread'
strCheese = 'Cheese'
```

```
total = prcBread + prcBread + prcApples

print(f'{"My Grocery List":^30s}')
print(f'{"="*30}')
print(f'{strApples}\t{numApples:10d}\t\t${prcApples:>5.2f}')
print(f'{strBread}\t{numBread:10d}\t\t${prcBread:>5.2f}')
print(f'{strCheese}\t{numCheese:10d}\t\t${prcCheese:>5.2f}')
print(f'{"Total:":>19s}\t\t${total:>4.2f}')
```

The output of this code snippet is:

```
        My Grocery List
==============================
Apples              3       $ 1.50
Bread               4       $ 6.00
Cheese              2       $ 4.50
                Total:      $13.50
```

Formatting with Commas

Finally, commas are often need when formatting large numbers. The following shows the use of commas when numbers are aligned and when numbers do not required alignment:

```
number = 1000000
print(f'The number, 1000000, formatted with a comma
{number:,.2f}')
print(f'The number, 1000000, formatted with a comma and right-
aligned in a width of 15 {number:>15,.2f}')
```

The output of this code snippet the following:

```
The number, 1000000, formatted with a comma 1,000,000.00
The number, 1000000, formatted with a comma and right-aligned in a width of 15    1,000,000.00
```

Conclusion

Printing output in Python is facilitated with **`f-strings`** as it can be considered *What You See is What You Get* (WYSIWYG). The procedure is as follows:

- Placing between the quotation marks after the **`'f'`** the text that you want displayed
- Enclosing the variables to be displayed within the text in curly braces
- Within those curly braces, placing a colon (**`:`**) after the variable
- Formatting the variable using a format specification (width, alignment, data type) after the colon

Happy formatting!