Formatting Output in Python

Leigh J. Little SUNY Brockport November 18, 2020

Contents

1 Introduction

This chapter discusses customizing the appearance of your output in Python. This is optional, but can make the results that you display to the screen or write to a file easier to read.

2 Format Specifiers

There have been numerous changes to the process of formatting data in the Python language since its inception. We will focus on what is currently the recommended way to code formatting statements however, you should be aware that there are other methods than we will examine here and you may see other approaches used in Python codes.

2.1 Integer Formatting

The best way to see how to format data is through examples. Integers are the easiest type of data to format. Suppose you have the code:

```
x = [1,4,10,50,105]
for i in range(len(x)):
     print(x[i])
```

This gives the output

```
1
4
10
50
105
```

Note that the digits do not line up.

We can write this list in a way that allows the digits to be properly aligned using the `format` method. This method has the syntax

```
print(string.format(a,b,c,d,.....))
```

In the above command, `string` is a string that contains information about how the quantities `a`, `b`, *etc.* should be displayed. The string can also contain static text labels. The quantities `a`, `b`, *etc.* can be anything (integers, floats or other strings).

Consider the example below

```
x = [1,4,10,50,105]
for i in x:
     print('{0:3d}'.format(i))
```

This gives the output

```
  1
  4
 10
 50
105
```

The `{0:3d}` portion of the print command is called a *format specifier*. It indicates how the integer value `i` should be printed. In this case, we have specified that `i` should be printed using 3 place decimal formatting.

The `0` in the format specifier `{0:3d}` refers to the argument number in the `format` method. For example, if we want to write two arrays as a column, we can do

```
x = [1,4,10,50,105]
y = [90,140,675,9834,1558]
for i in range(len(x))
     print('{0:3d}, {1:4d}'.format(x[i],y[i]))
```

This gives the output

```
  1,   90
  4,  140
 10,  675
 50, 9834
105, 1558
```

In this example, we have printed the elements of `x` with 3 place decimal formatting and the elements of `y` with 4 place formatting.

```
print('{0:3d}, {1:4d}'.format(x[i],y[i]))
        ^       ^
        ^       ^  --- second argument to format printed with 4 place decimal formatting
        ^ ---- first argument to format printed with 3 digit formatting.
```

The key to obtaining quality formatted output relies on knowing something about the magnitude of the numbers being output. For example, if we used 3 place formatting for the elements of `y` we would obtain the output

```
  1,  90
  4, 140
 10, 675
 50, 9834
105, 1558
```

The values of `x` line up, but the `y` values do not. This is because we used 3 digit formatting for numbers with 4 digits. You can always format for more digits than you have, but you should never format for fewer digits, for example

```
x = [1,4,10,50,105]
y = [90,140,675,9834,1558]
for i in range(len(x))
     print('{0:6d}, {1:6d}'.format(x[i],y[i]))
```

gives the output

```
     1,     90
     4,    140
    10,    675
    50,   9834
   105,   1558
```

Note the greater separation between the values.

Finally, the comma between the values is printed because it is included between the quotes in the format specifier. If you don't want the comma to be printed, you can use

```
print('{0:6d} {1:6d}'.format(x[i],y[i]))
```

2.2 Floating-point Formatting

Floating-point formatting is the trickiest formatting. This is because it is easy to obtain nonzero values that display as 0. Consider the example

```
x = [1.2, 3.44, 5.66, 7.897]
for i in range(len(x)):
     print('{0:.4f}'.format(x[i]))
```

This gives the output

```
1.2000
3.4400
5.6600
7.8970
```

In this case, we have used the floating-point specifier `{0:.4f}`. The `0.4f` part indicates that the values should be printed to 4 places after the decimal point. Values that have fewer places than this are padded with zeros. In the event that there are more decimal places than are indicated in the format specifier, the values are rounded to fit the specifier. For example,

```
x = [1.2, 3.44, 5.66, 7.897]
for i in range(len(x)):
     print('{0:.2f}'.format(x[i]))
```

gives the output

```
1.20
3.44
5.66
7.90
```

The last value has been rounded to 2 decimal places.

Suppose the previous code were run with the list

```
x = [1.2, 0.0005, 0.34, 0.45, 0.00323]
```

instead. This would give the output

```
1.20
0.00
0.34
0.45
0.00
```

Note that two of the values round to 0 for this formatting. This is why it is important to know something about the range of values you expect to be writing. If you had a program that ran for several hours and accidentally chose the wrong formatting, you would have to run the program again with proper formatting to get meaningful output.

Suppose instead that you defined

```
x = [-1.3, 2.3, 4.5, -5.6, 6.3]
```

and ran the 2 digit formatting example with this `x`. You would get the output

```
-1.30
2.30
4.50
-5.60
6.30
```

Notice how the negative signs cause the numbers to lose their alignment. This can be handled by putting a padding space in the format specifier.

```
print('{0: .2f}'.format(x[i]))
          ^ ---- note the space here
```

If you make this change, the output becomes

```
-1.30
 2.30
 4.50
-5.60
 6.30
```

2.3 Exponential Formatting

Exponential formatting has an advantage over floating-point formatting in that it can't generate display values of 0 for nonzero values. Consider the example

```
x = [1.2, 3.44, 5.66, 7.897]
for i in range(len(x)):
     print('{0:.2e}'.format(x[i]))
```

Here, we have used the `e` formatting instead of the `f` formatting. This gives the output

```
1.20e+00
3.44e+00
5.66e+00
7.90e+00
```

The digits are rounded to two digits after the decimal place. This formatting will work equally well for the case when

```
x = [1.2, 0.0005, 0.34, 0.45, 0.00323]
```

If this value of `x` is used, the output will be

```
1.20e+00
5.00e-04
3.40e-01
4.50e-01
3.23e-03
```

Note that all numbers are displayed and rounded to 2 digits. As with floating-point formatting, you can add padding to the format specifier (`{0: .2e}`) in the case that you have both positive and negative numbers to display.

In the event you want to print out all the digits of a double-precision floating-point value, you can use the specifier

```
print('{0:.16e}'.format(x[i]))
```

Note that we have only touched on the main types of formatting that likely to be used in scientific applications. There are many other format specifiers (for example, formatting of dates and times) that exist. In addition, any of the `print` statements in the above examples can be replaced with a `f.write(....)` if you are writing to a file.