TAKE YOUR CODE TO THE **NEXT LEVEL**

SUPERCHARGED PYTHON

BRIAN OVERLAND | JOHN BENNETT

Supercharged Python

The following examples use a `{:,}` print field. This is a simple specification because it just involves a comma to the immediate right of the colon—all inside a print field.

```
fss1 = 'The USA owes {:,} dollars.'
print(fss1.format(21000000000))
fss2 = 'The sun is {:,} miles away.'
print(fss2.format(93000000))
```

These statements print

```
The USA owes 21,000,000,000,000 dollars.
The sun is 93,000,000 miles away.
```

The next example uses the comma in combination with ***`fill`*** and ***`align`*** characters **`*`** and **`>`**, respectively. The ***`width`*** specifier is 12. Notice that the comma (,) appears just after ***`width`***; it's the last item before the closing curly brace.

```
n = 4500000
print('The amount on the check was ${:*>12,}'.format(n))
```

This example prints

```
The amount on the check was $***4,500,000
```

The print width of 12 includes room for the number that was printed, including the commas (a total of nine characters); therefore, this example uses three fill characters. The fill character in this case is an asterisk (*). The dollar sign ($) is not part of this calculation because it is a literal character and is printed as is.

If there is a leading-zero character as described in Section 5.8.4 (as opposed to a 0 fill character), the zeros are also grouped with commas. Here's an example:

```
print('The amount is {:011,}'.format(13000))
```

This example prints

```
The amount is 000,013,000
```

In this case, the leading zeros are grouped with commas, because all the zeros are considered part of the number itself.

A print-field size of 12 (or any other multiple of 4), creates a conflict with the comma, because an initial comma cannot be part of a valid number. Therefore, Python adds an additional leading zero in that special case.

```
n = 13000
print('The amount is {:012,}'.format(n))
```

This prints

```
The amount is 0,000,013,000
```

But if 0 is specified as a fill character instead of as a leading zero, the zeros are not considered part of the number and are not grouped with commas. Note the placement of the 0 here relative to the right justify (>) sign. This time it's just to the *left* of this sign.

```
print('The amount is {:0>11,}'.format(n))
```

This prints

```
The amount is 0000013,000
```

5.8.6 Controlling Precision

The *`precision`* specifier is a number provided primarily for use with floating-point values, although it can also be used with strings. It causes rounding and truncation. The precision of a floating-point number is the maximum number of digits to be printed, both to the right and to the left of the decimal point.

Precision can also be used, in the case of fixed-point format (which has an **`f`** type specifier), to ensure that an exact number of digits are always printed to the *right* of the decimal point, helping floating-point values to line up in a table.

Placement: Precision is always a number to the immediate right of a decimal point (.). It's the last item in a *`spec`* field, with the exception of the one-letter *`type`* specifier described in the next section.

Key Syntax

```
.precision
```

Here are some simple examples in which precision is used to limit the total number of digits printed.

```
pi = 3.14159265
phi = 1.618

fss = '{:.2} + {:.2} = {:.2}'
print(fss.format(pi, phi, pi + phi))
```

These statements print the following results. Note that each number has exactly two total digits:

```
3.1 + 1.6 = 4.8
```

This statement looks inaccurate, due to rounding errors. For each number, only two digits total are printed. Printing three digits for each number yields better results.

```
pi = 3.14159265
phi = 1.618

fss = '{:.3} + {:.3} = {:.3}'
print(fss.format(pi, phi, pi + phi))
```

This prints

```
3.14 + 1.62 = 4.76
```

The last digit to appear, in all cases of limited precision, is rounded as appropriate.

If you want to use *precision* to print numbers in fixed-point format, combine *width* and *precision* with an **f** type specifier at the end of the print field. Here's an example:

```
fss = '  {:10.3f}\n  {:10.3f}'
print(fss.format(22.1, 1000.007))
```

This prints

```
      22.100
    1000.007
```

Notice how well things line up in this case. In this context (with the **f** type specifier) the precision specifies not the total number of digits but *the number of digits just to the right of the decimal point*—which are padded with trailing zeros if needed.

The example can be combined with other features, such as the thousands separator, which comes after the width but before precision. Therefore, in this example, each comma comes right after 10, the width specifier.

```
fss = '  {:10,.3f}\n  {:10,.3f}'
print(fss.format(22333.1, 1000.007))
```

This example prints

```
  22,333.100
   1,000.007
```

The fixed-point format **f**, in combination with *width* and *precision*, is useful for creating tables in which the numbers line up. Here's an example:

```
fss = ' {:10.2f}'
for x in [22.7, 3.1415, 555.5, 29, 1010.013]:
    print(fss.format(x))
```

This example prints

```
  22.70
   3.14
 555.50
  29.00
1010.01
```

5.8.7 "Precision" Used with Strings (Truncation)

When used with strings, the ***precision*** specifier potentially causes truncation. If the length of the string to be printed is greater than the ***precision***, the text is truncated. Here's an example:

```
print('{:.5}'.format('Superannuated.'))  # Prints 'Super'
print('{:.5}'.format('Excellent!'))      # Prints 'Excel'
print('{:.5}'.format('Sam'))             # Prints 'Sam'
```

In these examples, if the string to be printed is shorter than the ***precision***, there is no effect. But the next examples use a combination of ***fill*** character, ***alignment***, ***width***, and ***precision***.

```
fss = '{:*<6.6}'
```

Let's break down what these symbols mean.

- The ***fill*** and ***align*** characters are ***** and **<**, respectively. The **<** symbol specifies left justification, so asterisks are used for padding on the right, if needed.
- The ***width*** character is 6, so any string shorter than 6 characters in length is padded after being left justified.
- The ***precision*** (the character after the dot) is also 6, so any string longer than 6 characters is truncated.

Let's apply this format to several strings.

```
print(fss.format('Tom'))
print(fss.format('Mike'))
print(fss.format('Rodney'))
print(fss.format('Hannibal'))
print(fss.format('Mortimer'))
```

These statements could have easily been written by using the global **format** function. Notice the similarities as well as the differences; the previous examples involved the format string `'{:*<6.6}'`.

```
print(format('Tom', '*<6.6'))
print(format('Mike', '*<6.6'))
print(format('Rodney', '*<6.6'))
print(format('Hannibal', '*<6.6'))
print(format('Mortimer', '*<6.6'))
```

In either case—that is, for either block of code just shown—the output is

```
Tom***
Mike**
Rodney
Hannib
Mortim
```

The *width* and *precision* need not be the same. For example, the following format specifies a width of 5, so any string shorter than 5 is padded; but the precision is 10, so any string longer than 10 is truncated.

```
fss = '{:*<5.10}'
```

5.8.8 "Type" Specifiers

The last item in the *spec* syntax is the *type* specifier, which influences how the data to be printed is interpreted. It's limited to one character and has one of the values listed in Table 5.5.

Placement: When the *type* specifier is used, it's the very last item in the *spec* syntax.

Table 5.5. "Type" Specifiers Recognized by the Format Method

TYPE CHARACTER	DESCRIPTION
b	Display number in binary.
c	Translate a number into its ASCII or Unicode character.
d	Display number in decimal format (the default).
e	Display a floating-point value using exponential format, with lowercase `e`—for example, `12e+20.`
E	Same as **e**, but display with an uppercase `E`—for example, `12E+20`.
f or **F**	Display number in fixed-point format.
g	Use format **e** or **f**, whichever is shorter.
G	Same as **g**, but use uppercase `E`.

▼ *continued on next page*

Table 5.5. "Type" Specifiers Recognized by the Format Method (*continued*)

TYPE CHARACTER	DESCRIPTION
n	Use the local format for displaying numbers. For example, instead of printing `1,200.34`, the American format, use the European format: `1.200,34.`
o	Display integer in octal format (base 8).
x	Display integer in hexadecimal format, using lowercase letters to represent digits greater than 9.
X	Same as **x**, but uses uppercase letters for hex digits.
%	Displays a number as a percentage: Multiply by 100 and then add a percent sign (%).

The next five sections illustrate specific uses of the *`type`* specifier.

5.8.9 Displaying in Binary Radix

To print an integer in binary radix (base 2), use the **b** specifier. The result is a series of 1's and 0's. For example, the following statement displays 5, 6, and 16 in binary radix:

```
print('{:b}  {:b}  {:b}'.format(5, 6, 16))
```

This prints the following:

```
101  110  10000
```

You can optionally use the **#** specifier to automatically put in radix prefixes, such as **0b** for binary. This formatting character is placed after the *`fill`*, *`align`*, and *`sign`* characters if they appear but before the *`type`* specifier. (It also precedes *`width`* and *`precision`*.) Here's an example:

```
print('{:#b}'.format(7))
```

This prints

```
0b111
```

5.8.10 Displaying in Octal and Hex Radix

The octal (base 8) and hexadecimal (base 16) radixes are specified by the **o**, **x**, and **X** type specifiers. The last two specify lowercase and uppercase hexadecimal, respectively, for digits greater than 9.