When creating strings to present as messages to the user of a program, you will want to *format* them in such a way that they are clear and easy to understand. While it is possible to format strings using concatenation (+), this can quickly become tedious. Fortunately, Python provides other ways to format strings. One powerful way of doing so is to use the string **`format`** method.

The following code demonstrates a simple use of the **`format`** method:

```
1  country = "France"
2  capital = "Paris"
3  sentence = "The capital of {} is {}.".format(country, capital)
4  print(sentence)
```

The string format method is used in line 3 of the above code. **`format`** returns a new, formatted string based upon the initial *format string*, which is the string you call the **`format`** method on, and the arguments. In the above example, the format string is **`"The capital of {} is {}."`**. In general, the format string can be quite complex, but the key idea is that curly braces indicate a *replacement field*. Text not in curly braces appears unchanged in the output string returned by **`format`**. In this example, there are two replacement fields and two arguments. The first replacement field is replaced in the output string with the first argument and the second replacement field is replaced in the output with the second argument. So, this program will print **`The capital of France is Paris.`**.

In this simple example, the two arguments were both strings, but this does not have to be the case. They can be any Python objects, which will be converted to strings and inserted into the output. Also, note that since curly braces indicate replacement fields, they cannot appear directly elsewhere in the format string. If you would like to have a curly brace in the output, you need to use {{ or }}. Double braces will appear as a single brace in the output string.

In order to provide more control of the formatted output, you can also provide a number that indicates which argument should be used. This allows you to repeat arguments or use them out of order:

```
1  mood1 = "happy"
2  mood2 = "sad"
3  sentence1 = "I feel {0}, do you feel {0}?  Or are you {1}? I'm not sure if we
     should be {0}.".format(mood1, mood2)
4  print(sentence1)
```

As this example shows, you can use the arguments in any order and as many times as you would like. Note, however, that you can either number the replacement fields automatically (using **`{}`**) or manually (explicitly giving the argument number for each replacement field), but you cannot switch back and forth between these two methods. Either all replacement fields need to be automatically numbered, or you need to manually indicate which argument to use for every replacement field.

You can also give a *format specification* for a replacement field. The format specification is preceeded by a colon and must come after the argument index (if there is one) in the replacement field. The format specification indicates how to format the individual field. It can be used to align strings (or numbers):

```
1  name1 = "Pierre"
2  age1 = 7
3  name2 = "May"
4  age2 = 13
5
6  line1 = "{0:^7} {1:>3}".format(name1, age1)
7  line2 = "{0:^7} {1:>3}".format(name2, age2)
8  print(line1)
9  print(line2)
```

In the format strings on lines 6 and 7, you can see that after the argument index in each replacement field there is a colon. The format specifiers after the colon include a number. This number is the width of the field in the output. The output will contain exactly that many characters for that field (padded with spaces), regardless of how wide the argument is. In this example the width is preceded by a symbol which indicates how the field should be aligned: < for left aligned, > for right aligned, and ^ for centered.

The format specification can also include a numerical precision and a type option. The precision is a number preceded by a period and must appear after the alignment and width (if they are included). Finally, the type option is a single letter that always appears at the end of the format specification which modifies the way things are formatted. The precision specifier can be useful when formatting floating point numbers:

```
1  num = 3.283663293
2  output = "{0:>10.3f} {0:.2f}".format(num)
3  print(output)
```

When formatting floating point numbers, the precision has different meanings depending on the specified type. With the `f` ("fixed point") type, which is used in the above example, the floating point number will be formatted such that there are n digits after the decimal point, where n is the specified precision. As the example shows, you can combine this with a field width and alignment (although you do not have to). The default floating point type, `g` ("general format"), instead uses the precision to indicate the number of significant digits to include in the formatted string.

There are many other options you can specify in the format specification. If you are interested, you can read the full documentation. Be warned, however, that the complete format specification syntax is quite complex!

Mark as completed