CSC108: String Formatting

1 String Concatenation

To combine multiple strings for output (or for storage as a longer string), string concatenation can be used. Here is an example:

```
>>> first = 'Foresta'
>>> second = 'Village'
>>> print(first + ' ' + second)
Foresta Village
>>> amazing_song = 'A truly amazing song is ' + first + ' ' + second
>>> print(amazing_song)
A truly amazing song is Foresta Village
```

For small examples involving only a few strings, string concatenation is fine. But watch what happens when we try to combine more information, where some of that information is stored as integers. What we want to do is print a message like:
`The largest of the numbers 4, 3, and 8 is 8.`
and where the three integers are referred to by variables. Here is a first attempt:

```
>>> a = 4
>>> b = 3
>>> c = 8
>>> print('The largest of the numbers ' + a + ', ' + b + ', and ' + c + ' is ' +
          max(a, b, c))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The problem here is that `a`, `b`, and `c` are integers, and string concatenation works only on strings. So we have to first convert the integers to strings, and then concatenate:

```
>>> print('The largest of the numbers ' + str(a) + ', ' + str(b) + ', and ' +
          str(c) + ' is ' + max(a, b, c))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Oops, there's still a problem. The problem is that `max` returns an integer and, again, we can't concatenate that integer to a string. Let's fix that:

```
>>> print('The largest of the numbers ' + str(a) + ', ' + str(b) + ', and ' +
          str(c) + ' is ' + str(max(a, b, c)))
The largest of the numbers 4, 3, and 8 is 8
```

Rather than explicitly gluing everything together with `+` and converting every non-string into a string, we can instead use **string formatting**. String formatting simplifies our code and gives us more powerful techniques for combining multiple pieces of data.

2 Positional String Formatting

A **format string** is like a template that determines how your string will be produced. It looks like a regular string, except that it contains replacement fields in curly braces, and those fields will be replaced by other data before the string is displayed.
Let's go back to our initial example, where we used string concatenation to produce the (highly-accurate) message:

```
>>> first = 'Foresta'
>>> second = 'Village'
>>> amazing_song = 'A truly amazing song is ' + first + ' ' + second
>>> print(amazing_song)
A truly amazing song is Foresta Village
```

To produce this message, we combined a string literal (`A truly amazing song is `) with two string variables. A template for this string can therefore be defined to have two placeholders where the `first` and `second` strings can later be inserted. Here is how this format string would look:

```
>>> fmt = 'A truly amazing song is {0} {1}'
>>> print(fmt)
A truly amazing song is {0} {1}
```

Notice that if we were to take this string and put `Foresta` in place of `{0}` and `Village` in place of `{1}` then we'd arrive at the same string as when we used concatenation. The `format` method of strings allows us to make these replacements.
The syntax of `format` is as follows, for some format string `s`:
`s.format(data_0, data_1, ..., data_n)`
Then, `format` will replace the `{0}` in your format string with `data_0`, `{1}` with `data_1`, and so on. The number of arguments you pass to `format` should be the same as the number of different values to which you refer to using the `{n}` syntax.
Let's use this to reproduce the message about the amazing song. Compare this to how we produced this message using string concatenation:

```
>>> fmt = 'A truly amazing song is {0} {1}'
>>> first = 'Foresta'
>>> second = 'Village'
>>> print(fmt.format(first, second))
A truly amazing song is Foresta Village
>>> amazing_song = fmt.format(first, second)
>>> amazing_song
'A truly amazing song is Foresta Village'
```

At this point, since we have our format string `fmt`, we can reuse it to tell people about other great songs:

```
>>> print(fmt.format("Terra's", "Theme"))
A truly amazing song is Terra's Theme
>>> fmt.format('Meridian', 'Dance')
'A truly amazing song is Meridian Dance'
```

Of course, it only works for songs that have two words (unless you put multiple words in one of the two placeholders). Sometimes, you just want to use a format string once to combine some data and print it out. In these cases, you do not first have to assign the format string to a variable:

```
>>> print('My two letters are {0} and {1}'.format(a, b))
My two letters are h and i
>>> print('My two letters are {0} and {0}'.format(a))
My two letters are h and h
```

(In the second example, note how I used the same replacement field twice in the format string.)
The great thing about string formatting is that you are not restricted to inserting strings into the replacement fields. Here is an example showing that integers and floats can go in there too:

```
>>> kg = 50.5
>>> age = 29
>>> print('Mass: {0}, age: {1}.'.format(kg, age))
Mass: 50.5, age: 29.
```

2.1 Maximum of Three Numbers

Remember that "maximum of three numbers" example that we solved using string concatenation in the first section of this handout? Now we have everything necessary to rewrite that using string formatting. Notice that `str` conversions are no longer necessary, and that we avoid all of the `+` operators.

```
>>> print('The largest of the numbers {0}, {1}, and {2} is {3}'
          .format(a, b, c, max(a, b, c)))
The largest of the numbers 4, 3, and 8 is 8
```

2.2 Field Width and Precision

The `{n}` syntax we have been using is only a small piece of what can actually be put in those curly braces for string formatting. To give you a better idea of the power of string formatting, we'll consider two additional features here: minimum field width, and precision.

The **minimum field width** determines the minimum number of characters that will be occupied by a field. If a field would otherwise take up fewer than this number of characters, it will be padded so that it's width is the minimum. The minimum width is specified by appending a colon and the minimum width inside the curly braces, like this:

```
>>> '{0:5} pendants'.format(3)
'    3 pendants'
>>> '{0:10} pendants'.format(3)
'         3 pendants'
>>> '{0:2} pendants'.format(3)
' 3 pendants'
>>> '{0:1} pendants'.format(3)
'3 pendants'
```

This can be useful when you want to display text in columns:

```
>>> songfmt = '{0:6}{1:20}{2:20}'
>>> print(songfmt.format('Year', 'Composer', 'Song'))
Year  Composer            Song
>>> print(songfmt.format('1995', 'Tamura', 'Be Absent-Minded'))
1995  Tamura              Be Absent-Minded
```

When formatting floating-point numbers, they might have tons of decimal digits. To get control over the number of digits printed, you can include `.p`, where `p` indicates the precision:

```
>>> '{0}'.format(31 / 3)
'10.333333333333334'
>>> '{0:.5}'.format(31 / 3)
'10.333'
>>> '{0:.4}'.format(31 / 3)
'10.33'
>>> '{0:.3}'.format(31 / 3)
'10.3'
```

Of course, you can combine the minimum field width and precision with a format string like `'{0:8.3}'` — try it!

3 Keyword String Formatting

Rather than using positional fields, such as `{0}` or `{1}`, we can instead use keywords that can be more meaningful than numbers. Here is an example: instead of using the field numbers `{0}`, `{1}`, and `{2}`, we use more descriptive names.

```
>>> fmt = 'Today is {month} {day}, {year}'
>>> fmt.format(month='Sep', day=18, year=2012)
'Today is Sep 18, 2012'
>>> mo = 'Sep'
>>> day = 18
>>> year = 2012
>>> fmt.format(month=mo, day=day, year=year)
'Today is Sep 18, 2012'
```

Notice that when calling `format`, we use the syntax `month='Sep'` rather than just `'Sep'`. This is because we have to provide the names (such as `month`) that match the names in the curly braces of the format string.