ICS3U: Formatted Output

Constructing Strings Using `format`

To date, most of our output has involved displaying raw values, such as integers, or by using the `print` function to piece together simple strings. In some cases, we have used other functions such as `round` to change the appearance of floating-point values. For a greater variety of options, Python, the `format` string method to apply various formatting capabilities to a given string. Since it is a string method, it is applied either to a string literal or to a variable which is of type `str`. Much of the time, we will appy it to string literals as in the example below.

```
name = input("What is your name? ")
print("Hello, {}! How are you?".format(name))
```

The curly braces, `{}`, specify a **field**. This marks a location in the string where a value supplied to `format` will appear. They do not appear in the final string. If you wish to use a curly brace in a string, you must escape it as `\{` or `\}`.

Multiple values can be used with `format` by separating them with commas. When using more than one value, there are several ways to reference them. One method is to leave all fields blank. This will match each field with one value, in the order they appear.

```
>>> "The {} ate the {}.".format("cat", "mouse")
'The cat ate the mouse.'
```

Another option is to provide an index, *i*, in each field specifying the *i*th value to use. Remember that Python uses zero-based indexing, so the first value has an index of 0. This also allows us to repeat values without having to repeat them inside of `format`.

```
>>> "The {0} ate the {1}.".format("cat", "mouse")
'The cat ate the mouse.'
>>> "The {1} ate the {0}.".format("cat", "mouse")
'The mouse ate the cat.'
>>> "The {0} ate the {0}.".format("cat")
'The cat ate the cat.'
```

A third option is to use keyword arguments, although this often results in "bulkier" code, so it is generally not used.

```
>>> "The {eater} ate the {food}.".format(eater="cat", food="mouse")
'The cat ate the mouse.'
```

Advanced Formatting Options

All of the previous examples allow us to substitute values into various locations in a string, but they have not yet allowed us to change *how* those values are displayed. Inside of a field, it is possible to specify many different options that will alter a value's appearance. The general format of a field is `{index:formatting options}`. Most of the time we will not be specifying an index like in the last set of examples, but we can if we wish. As such, the examples below leave this portion blank.

All formatting options are, as their name suggests, optional. It is not necessary to include any of them, but doing so allows us to customize the ways in which the values are displayed to the screen. It is important to specify them in the correct order, according to the Python Format Mini-Language. More specifically, the full setup is as follows.

```
{index:[[fill]align][sign][width][grouping][.precision][type]}
```

Before we cover some of the different formatting options, it is important to know what **type** of data we are working with. Different data types have their own options. The table below shows some examples of how to specify a particular data type, and how the resulting string would appear. In many cases the results are the same. While it is often not a requirement to specify a data type, it is *always* the last option if included.

Type	Symbol	Example	Output
String	`s` or blank	`"I like {}.".format("apples")`	`I like apples.`
		`"I like {:s}.".format("apples")`	`I like apples.`
Integer	`d` or `n` or blank	`"The answer is {}!".format(42)`	`The answer is 42!`
		`"The answer is {:d}!".format(42)`	`The answer is 42!`
		`"The answer is {:n}!".format(42)`	`The answer is 42!`
Floating-point	`g`, `f`, `e`, `n` or blank (and more)	`"{}".format(1/7)`	`0.14285714285714285`
		`"{:g}".format(1/7)`	`0.142857`
		`"{:f}".format(1/7)`	`0.142857`
		`"{:n}".format(1/7)`	`0.142857`
		`"{:e}".format(1/7)`	`1.428571e-01`

Integers are typically displayed by leaving the field blank, or by using `d` when formatting is required. The `n` option uses the locale settings from the operating system. Floating-point values have many variations: `g` for general, `f` for fixed number of digits, `n` for a localized option, and `e` for exponential notation. When left blank, values are displayed using up to 17 digits. When `g`, `f`, `n` or `e` are used, the default precision is six decimals. This can be changed, as we will see later. There are capitalized options (`G`, `F` and `E`) as well, which you can read about in the documentation.

There are also ways to display values using other bases, including binary (`b`) and hexadecimal (`x` or `X`). It is even possible to display a Unicode character (using `c`) given its *decimal* representation.

```
>>> "{:b}".format(28)
'11100'
>>> "{:x}".format(28)
'1c'
>>> "{:X}".format(28)
'1C'
>>> "{:c}".format(2354)
'ल'
```

String **alignment** is done by using one of four characters (see the table) followed by a **width** (the number of characters to display the value in). It is not necessary to specify a type. Note that if a given value is larger than the specified width, Python will revert back to left-alignment. In the examples below, the '␣' character represents a space.

Alignment	Symbol	Example	Output
Left	<	`"{:<10}".format(-123)`	`-123␣␣␣␣␣␣`
Right	>	`"{:>10}".format(-123)`	`␣␣␣␣␣␣-123`
Centre	^	`"{:^10}".format(-123)`	`␣␣␣-123␣␣␣`
Pad after sign	=	`"{:=10}".format(-123)`	`-␣␣␣␣␣␣123`

The alignment character can be prefaced by a **fill** character, if desired. This will replace the spaces with that character.

```
>>> "{:*>10}".format(-123)
'******-123'
```

Following the alignment character, you can specify a **sign** option for numeric values. The default behaviour displays the sign for negative values but suppresses the sign for positive values. To show the sign for positive values as well, use `+`.

```
>>> "{:^10}".format(123)
'␣␣␣123␣␣␣␣'
>>> "{:^+10}".format(123)
'␣␣␣+123␣␣␣'
```

If you prefer using a comma to **group** thousands, you can add the `,` option before the type.

```
>>> "{:,d}".format(123456)
'123,456'
```

Note that on the examples above, a type (integer) was specified via the `d` option. It is also possible to use the operating system's native locale settings, but this requires using the `locale` module. Instead of using `d` for an integer type, or either `f` or `g` for a floating-point type, use `n` instead. On a system using the `en_CA` (Canadian English) locale where thousands are separated using spaces, it would look something like the code below.

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, '')
'en_CA.UTF-8'
>>> "{:n}".format(123456)
'123 456'
```

To set the **precision** and use a fixed number of decimals, use `.Nf` where *`N`* is the number of digits to display after the decimal point. Values may be rounded.

```
>>> "{:.3f}".format(3.14159)
'3.142'
```

This fixes the issue whereby Python removes trailing zeroes when displaying floating-point values.

```
>>> round(78.9012, 2)
78.9
>>> "{:.2f}".format(78.9012)
'78.90'
```

Finally, Python also provides the option to automatically display values as percentages using the `%` option. This multiplies a floating-point value by 100, and represents it using the `f` option with a percent sign appended to it.

```
>>> "{:%}".format(0.123456)
'12.345600%'
>>> "{:.2%}".format(0.123456)
'12.35%'
```

The last example uses the `.2` option to round the value to two decimals.