Lecture 1.3 : Formatted string output

Introduction

- So far we have been using `print()` to send output to `stdout`.

```
>>> x = 1/3
>>> print(x)
0.3333333333333333
>>> import math
>>> print(math.pi)
3.141592653589793
```

- Unfortunately `print()` affords us no control over the *format* of the output it produces. For instance, what if we want to display a floating point number to some specific number of decimal places? If we calculate an average price for example it would not make sense to go beyond two decimal places when displaying it. The `print()` function will not allow us to do that. We need a more sophisticated approach.

The `format()` method

- We invoke Python's `format()` method to *prepare* a string for printing. The string upon which we invoke the `format()` method is called the *format string*. It looks like a normal string except it contains *placeholders* (also known as *replacement fields*) for the data we wish to be specially formatted. Inside the placeholders we specify *how* we want the data to be displayed. The arguments to the `format()` method are the items of data which will be inserted into each placeholder and formatted accordingly.

- The above sounds more complicated than it is. Let's look at some examples of the `format()` method in action.

```
>>> print('{}'.format(x))
0.3333333333333333
>>> print('{}'.format(math.pi))
3.141592653589793
```

- When we write `'{}'.format(x)` we are invoking the `format()` method on the string `'{}'` where `{}` is a placeholder. Into this placeholder is inserted the value of argument `x` to produce a finished string. That string then serves as an argument to the enclosing `print()` function.

- So far `format()` has not yielded results any different to those produced using the `print()` function. The power of `format()` however comes from the *format commands* that can be placed inside the `{}` placeholders. They control *how* the data inserted into those placeholders is displayed.

- The general structure of a *format command* is `{[:[align] [minimum_width] [.precision] [type]]}`. Square brackets indicate optional arguments.

- The `align` field is used to control whether the printed value is centred, left justified or right justified. Values include `^` for centred, `<` for left justified and `>` for right justified.
- The `type` field specifies the type of value we want to insert where commonly used types include `s` (for string), `d` (for integer) and `f` (for floating point).
- The `minimum_width` field specifies the desired *overall* minimum width for this value once printed.
- The `.precision` field specifies the number of digits to follow the decimal point when printing a floating point type.
- Again, this sounds more complicated than it is. Let's look at some examples in order to make things clearer. To print `x` to two decimal places and `math.pi` to five decimal places we would write:

```
>>> print('{:.2f}'.format(x))
0.33
>>> print('{:.5f}'.format(math.pi))
3.14159
```

- `{:.2f}` means insert here a floating point number (`type=f`) and display it to two decimal places (`precision=2`).
- `{:.5f}` means insert here a floating point number (`type=f`) and display it to five decimal places (`precision=5`).
- By specifying a minimum width we can cause leading spaces to be inserted in order to pad the number out to an overall width that equals the minimum width (there will be no padding if the width of the number already equals or exceeds the minimum width):

```
>>> print('{:8.2f}'.format(x))
    0.33
>>> print('{:8.5f}'.format(math.pi))
 3.14159
```

- We can include as many placeholders in the format command as we wish. The first argument to the format method is matched with the first placeholder, the second argument with the second placeholder, etc. For example:

```
>>> print('The value of x is {:.2f} while pi is {:.3f}'.format(x, math.pi))
The value of x is 0.33 while pi is 3.142
```

- Specifying minimum widths is useful when we want to align output. Suppose we want to print our times 12 multiplication table. We could do it like this but the output is not nicely aligned:

```
>>> for i in range(1,13):
...     print(i, '* 12 = ', i*12)
...
1 * 12 =  12
2 * 12 =  24
3 * 12 =  36
4 * 12 =  48
5 * 12 =  60
6 * 12 =  72
```

```
7 * 12 =  84
8 * 12 =  96
9 * 12 =  108
10 * 12 =  120
11 * 12 =  132
12 * 12 =  144
```

- If we write it like this then the output is neatly aligned (because all numbers are right-justified and padded out to the specified minimum width):

```
>>> for i in range(1,13):
...     print('{:2d} * {:2d} = {:3d}'.format(i, 12, i*12))
...
 1 * 12 =  12
 2 * 12 =  24
 3 * 12 =  36
 4 * 12 =  48
 5 * 12 =  60
 6 * 12 =  72
 7 * 12 =  84
 8 * 12 =  96
 9 * 12 = 108
10 * 12 = 120
11 * 12 = 132
12 * 12 = 144
```

Nested placeholders

- Suppose we want to print `math.pi` to some user-defined number of decimal places. Can we do that? Yes, with nested placeholders.

```
>>> print('{:.3f}'.format(math.pi))
3.142
>>> N = 3
>>> print('{:.{}f}'.format(math.pi, N))
3.142
```

- In the first example we simply print `math.pi` to 3 decimal places. In the second example we replace the 3 in the format command with a placeholder. This allows us to insert at runtime an arbitrary value. Arguments are matched with placeholders from left-to-right so `math.pi` matches the outer placeholder while `N` matched the inner one. Below we use this technique to display `math.pi` to various decimal places inside a `for` loop.

```
>>> for i in range(0,10):
...     print('{:.{}f}'.format(math.pi, i))
...
3
3.1
3.14
3.142
3.1416
3.14159
3.141593
3.1415927
3.14159265
3.141592654
```