Lecture 1.3 : Formatted string output

Introduction

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

```
x = 1/3
print(x)

from math import pi
print(pi)
```

```
0.3333333333333333
3.141592653589793
```

- Unfortunately `print()` alone affords us little control over the *format* of the printed string. 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 alone will not allow us to do that. We need a more sophisticated approach.

f-strings

- We can use *f-strings* to format a string prior to printing.
- An f-string is similar to a normal string except it starts with an f (surprise!) and contains *placeholders* for the data we wish to be specially formatted.
- Inside the placeholders we specify *which* data we want displayed and *how* we want it to be displayed.

```
print(f'{x}')
print(f'{pi}')
```

```
0.3333333333333333
3.141592653589793
```

- Each f-string above contains a single placeholder (a placeholder is delimited by curly brackets). Into each placeholder we insert the name of a variable whose value we wish to print. Above we are printing `x` and `pi`.
- So far our f-strings have not yielded results different to those produced using the `print()` function. The power of f-strings 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 are printing. 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.
- This all 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 `pi` to five decimal places we would write:

```
print(f'{x:.2f}')
print(f'{pi:.5f}')
```

```
0.33
3.14159
```

- 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(f'{x:8.2f}')
print(f'{pi:1.5f}')
```

```
    0.33
3.14159
```

- Sometimes we want to pad with zeros rather than spaces.

```
hour, min, sec = 3, 4, 5
print(f'The current time is {hour:02d}:{min:02d}:{sec:02d}')

hour, min, sec = 13, 14, 15
print(f'The current time is {hour:02d}:{min:02d}:{sec:02d}')
```

```
The current time is 03:04:05
The current time is 13:14:15
```

- We can include as many placeholders in the f-string as we wish.

```
print(f'x has the value {x:.2f} and pi has the value {pi:.5f}')
```

```
x has the value 0.33 and pi has the value 3.14159
```

- Suppose we want to print our times 12 multiplication table. We could do it like this but the output is not nicely aligned (which we find upsetting):

```
for i in range(1, 13):
   print(str(i) + ' * 12 = ' + str(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
```

- Specifying minimum widths is useful when we want to align output. 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(f'{i:2d} * 12 = {i*12:3d}')
```

```
 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 `pi` to some user-defined number of decimal places. Can we do that? Yes, with nested placeholders. Below we first print `pi` to 3 decimal places. Next we replace the 3

in the f-string with a placeholder. This allows us to insert at runtime an arbitrary value for the precision.

```
print(f'{pi:.3f}')
N = 5
print(f'{pi:.{N}f}')
```

```
3.142
3.14159
```

- Below we use this technique to display `pi` to various decimal places inside a `for` loop.

```
for i in range(10):
   print(f'{pi:.{i}f}')
```

```
3
3.1
3.14
3.142
3.1416
3.14159
3.141593
3.1415927
3.14159265
3.141592654
```