IIT Madras
BSc Degree

Home Lesson-3.5

Lesson-3.4

Formatted printing

Consider the following program:

```
1 name = input()
2 print('Hi,', name, '!')
```

When this code is executed with `Sachin` as the input, we get the following output:

```
1 Hi, Sachin !
```

This looks messy as there is an unwanted space after the name. This is a formatting issue. Python provides some useful tools to format text the way we want.

f-strings

The first method that we will look at is called formatted string literals or f-strings for short. Let us jump into the syntax:

```
1 name = input()
2 print(f'Hi, {name}!')
```

When this code is executed with `Sachin` as the input, we get the following output:

```
1 Hi, Sachin!
```

The messy formatting has been corrected. Let us take a closer look at the string inside the `print` command:

```
f'Hi, {name}'
```

This is called a formatted string literal or f-string. The `f` in front of the string differentiates f-strings from normal strings. f-string is an object which when evaluated results in a string. The value of the variable `name` is inserted in place of `{name}` in the f-string. Two things are important for f-strings to do our bidding:

- The `f` in front of the string.
- The curly braces enclosing the variable.

Let us see what happens if we miss one of these two:

```
name = 'Sachin'
print('Hi, {name}!')
print(f'Hi, name!')
```

This will give the output:

```
Hi, {name}!
Hi, name!
```

Let us now look at few other examples:

```
l, b = int(input()), int(input())
print(f'The length of the rectangle is {l} units')
print(f'The breadth of the rectangle is {b} units')
print(f'The area of the rectangle is {l * b} square units')
```

For `l = 4, b = 5`, the output is:

```
The length of the rectangle is 4 units
The breadth of the rectangle is 5 units
The area of the rectangle is 20 square units
```

Going back to the code, lines 2 and 3 are quite clear. Notice that line-4 has an expression — `l * b` — inside the curly braces and not just a variable. f-strings allow any valid Python expression inside the curly braces. If the f-string has some `{expression}` in it, the interpreter will substitute the value of `expression` in the place of `{expression}`. Another example:

```
x = int(input())
print(f'Multiplication table for {x}')
for i in range(1, 11):
    print(f'{x} X {i} \t=\t {x * i}')
```

For an input of 3, this will give the following result:

```
Multiplication table for 3
3 X 1   =    3
3 X 2   =    6
3 X 3   =    9
3 X 4   =    12
3 X 5   =    15
3 X 6   =    18
3 X 7   =    21
3 X 8   =    24
3 X 9   =    27
3 X 10  =    30
```

The `\t` is a tab character. It has been added before and after the `=`. Remove both the tabs and run the code. Do you see any change in the output?

Till now we have used f-strings within the `print` statement. Nothing stops us from using it to define other string variables:

```
name = input()
qual = input()
gender = input()
if qual == 'phd':
    name_respect = f'Dr. {name}'
elif gender == 'male':
    name_respect = f'Mr. {name}'
elif gender == 'female':
    name_respect = f'Ms. {name}'
print(f'Hello, {name_respect}')
```

Try to guess what this code is doing.

`format()`

Another way to format strings is using a string method called `format()`.

```
name = input()
print('Hi, {}!'.format(name))
```

In the above string, the curly braces will be replaced by the value of the variable `name`. Another example:

```
l, b = int(input()), int(input())
print('The length of the rectangle is {} units'.format(l))
print('The breadth of the rectangle is {} units'.format(b))
print('The area of the rectangle is {} square units'.format(l * b))
```

Let us now print the multiplication table using `format`:

```
x = int(input())
for i in range(1, 11):
    print('{} X {} \t=\t {}'.format(x, i, x * i))
```

The output will be identical to the one we saw when we used f-strings. Some points to note in line-3 of this code-block. There are three pairs of curly braces. The values that go into these three positions are given as three arguments in the `format` function. Starting from the left, the first pair of curly braces in the string is replaced by the first argument in `format`, the second pair by the second argument and so on. Few more examples:

First, consider the following code:

```
fruit1 = 'apple'
fruit2 = 'banana'
print('{} and {} are fruits'.format(fruit1, fruit2))
```

In this code, the mapping is implicit. The first pair of curly braces is mapped to the first argument and so on. This can be made explicit by specifying which argument a particular curly braces will be mapped to:

```
fruit1 = 'apple'
fruit2 = 'banana'
print('{0} and {1} are fruits'.format(fruit1, fruit2))
```

The integer inside the curly braces gives the index of the argument in the `format` function. The arguments of the `format` function are indexed from 0 and start from the left. Changing the order of arguments will change the output. A third way of writing this as follows:

```
fruit1 = 'apple'
fruit2 = 'banana'
print('{string1} and {string2} are fruits'.format(string1 = fruit1, string2 =
fruit2))
```

This method uses the concept of keyword arguments which we will explore in the lessons on functions in the next chapter. Until then, let us put this last method on the back-burner.

Format specifiers

Consider the following code:

```
pi_approx = 22 / 7
print(f'The value of pi is approximately {pi_approx}')
```

This gives the following output:

```
The value of pi is approximately 3.142857142857143
```

There are too many numbers after the decimal point. In many real world applications, having two or at most three places after the decimal point is sufficient. In fact, having as many as fifteen numbers after the decimal point only confuses readers. Format specifiers are a way to solve this problem:

```
pi_approx = 22 / 7
print(f'The value of pi is approximately {pi_approx:.2f}')
```

This gives the following output:

```
The value of pi is approximately 3.14
```

Let us look at the content inside the curly braces: `{pi_approx:.2f}`. The first part before the `:` is the variable. Nothing new here. The part after `:` is called a format specifier. `.2f` means the following:

- `.` - this signifies the decimal point.
- `2` - since this comes after the decimal point, it stipulates that there should be exactly two numbers after the decimal point. In other words, the value (`pi_approx`) should be rounded off to two decimal places.
- `f` - this signifies that we are dealing with a `float` value.

Let us consider a variant of this code:

```
pi_approx = 22 / 7
print(f'The value of pi is approximately {pi_approx:.3f}')
```

This gives the following output:

```
The value of pi is approximately 3.143
```

Let us now take another example. Let us say we want to print the marks of three students in a class:

```
roll_1, marks_1 = 'BSC1001', 90.5
roll_2, marks_2 = 'BSC1002', 100
roll_3, marks_3 = 'BSC1003', 90.15
print(f'{roll_1}: {marks_1}')
print(f'{roll_2}: {marks_2}')
print(f'{roll_3}: {marks_3}')
```

This gives the following output:

```
BSC1001: 90.5
BSC1002: 100
BSC1003: 90.15
```

While this is not bad, we would like the marks to be right aligned and have a uniform representation for the marks. This is what we wish to see:

```
BSC1001:      90.50
BSC1002:     100.00
BSC1003:      90.15
```

This is much more neater. The following code helps us achieve this:

```
roll_1, marks_1 = 'BSC1001', 90.5
roll_2, marks_2 = 'BSC1002', 100
roll_3, marks_3 = 'BSC1003', 90.15
print(f'{roll_1}: {marks_1:10.2f}')
print(f'{roll_2}: {marks_2:10.2f}')
print(f'{roll_3}: {marks_3:10.2f}')
```

The part that might be confusing is the second curly braces in each of the print statements. Let us take a closer look: `{marks_1:10.2f}`. The part before the `:` is the variable. The part after the `:` is `10.2f`. Here again, `.2f` signifies that the float value should be rounded off to two decimal places. The `10` before the decimal point is the minimum width of the column used for printing this value. If the number has fewer than 10 characters (including the decimal point), this will be compensated by adding spaces before the number.

For a better understanding of this concept, let us turn to printing integers with a specific formatting. This time, we will use the `format` function:

```
print('{0:5d}'.format(1))
print('{0:5d}'.format(11))
print('{0:5d}'.format(111))
print('{:5d}'.format(1111))
print('{:5d}'.format(11111))
print('{:5d}'.format(111111))
```

This gives the following output:

```
    1
   11
  111
 1111
11111
111111
```

Points to note in the code:

- The `d` stands for integer.
- First three print statements have the index of the argument — `0` in this case — before the `:`. Last three statements do not have the index of the argument. In fact there is nothing before the `:`. Both representations are valid.
- The `5d` after the `:` means that the width of the column used for printing must be at least 5.
- Lines 1 to 4 have spaces before them as the integer being printed has fewer than five characters.