# String Formatting

In this lesson, we will learn about formatting strings in Python

WE'LL COVER THE FOLLOWING

- Wrapping Up

String formatting (AKA substitution) is the topic of substituting values into a base string. Most of the time, you will be inserting strings within strings; however you will also find yourself inserting integers and floats into strings quite often as well. There are two different ways to accomplish this task. We'll start with the old way of doing things and then move on to the new.

### Ye Olde Way of Substituting Strings

The easiest way to learn how to do this is to see a few examples. So here we go:

```
my_string = "I like %s" % "Python"
print(my_string) # 'I like Python'

var = "cookies"
newString = "I like %s" % var
print(newString) # 'I like cookies'

another_string = "I like %s and %s" % ("Python", var)
print(another_string)  # 'I like Python and cookies'
```

As you've probably guessed, the **%s** is the important piece in the code above. It tells Python that you may be inserting text soon. If you follow the string with a percent sign and another string or variable, then Python will attempt to insert it into the string. You can insert multiple strings by putting multiple instances of **%s** inside your string. You'll see that in the last example. Just note

that when you insert more than one string, you have to enclose the strings that you're going to insert with parentheses.

Now let's see what happens if we don't insert enough strings:

```
another_string = "I like %s and %s" % "Python"

print(another_string)
# TypeError: not enough arguments for format string
```

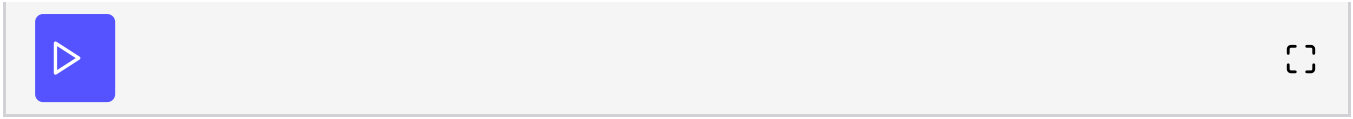
Oops! We didn't pass enough arguments to format the string! If you look carefully at the example above, you'll notice it has two instances of %s, so to insert strings into it, you have to pass it the same number of strings! Now we're ready to learn about inserting integers and floats. Let's take a look!

```
my_string = "%i + %i = %i" % (1,2,3)
print(my_string) # '1 + 2 = 3'

float_string = "%f" % (1.23)
print(float_string) # '1.230000'

float_string2 = "%.2f" % (1.23)
print(float_string2) # '1.23'

float_string3 = "%.2f" % (1.237)
print(float_string3) # '1.24'
```

The first example above is pretty obvious. We create a string that accept three arguments and we pass them in. Just in case you hadn't figured it out yet, no, Python isn't actually doing any addition in that first example. For the second example, we pass in a float. Note that the output includes a lot of extra zeroes. We don't want that, so we tell Python to limit it to two decimal places in the 3rd example ("%.2f"). The last example shows you that Python will do some rounding for you if you pass it a float that's greater than two decimal places.

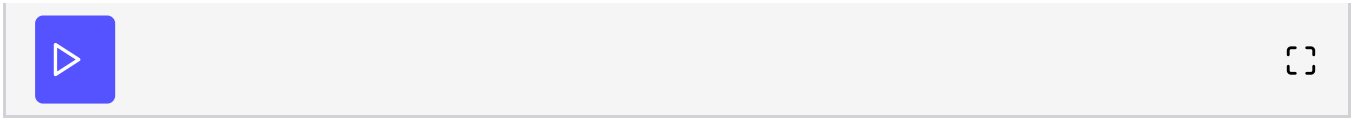
Now let's see what happens if we pass it bad data:

```
int_float_err = "%i + %f" % ("1", "2.00")
print(int_float_err)
# TypeError: %i format: a number is required, not str
```



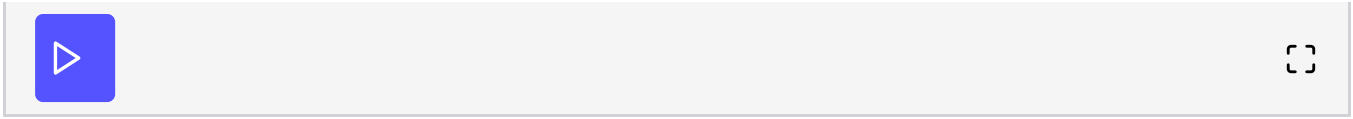
In this example, we pass it two strings instead of an integer and a float. This raises a TypeError and tells us that Python was expecting a number. This refers to not passing an integer, so let's fix that and see if that fixes the issue:

```
int_float_err = "%i + %f" % (1, "2.00")
print(int_float_err)
# TypeError: a float is required
```



Nope. We get the same error, but a different message that tells us we should have passed a float. As you can see, Python gives us pretty good information about what went wrong and how to fix it. If you fix the inputs appropriately, then you should be able to get this example to run. Let's give it a try:

```
int_float_str = "%i + %f" % (1, 2.00)
print(int_float_str)
```

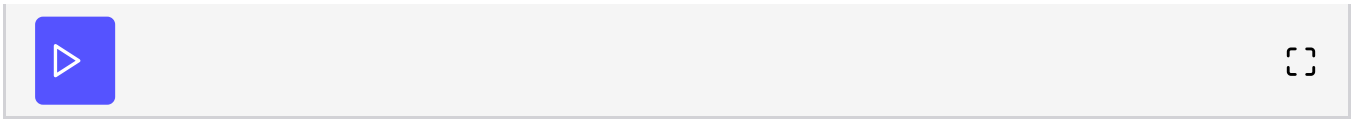


Let's move on to the new method of string formatting!

**Templates and the New String Formatting Methodology**

This new method was actually added back in Python 2.4 as string templates, but was added as a regular string method via the **format** method in Python 2.6. So it's not really a new method, just newer. Anyway, let's start with templates!

```
print("%(lang)s is fun!" % {"lang":"Python"})
# Python is fun!
```



This probably looks pretty weird, but basically we just changed our **%s** into **%**

**(lang)s**, which is basically the %s with a variable inside it. The second part is actually called a Python dictionary that we will be studying in the next section. Basically it's a key:value pair, so when Python sees the key "lang" in the string AND in the key of the dictionary that is passed in, it replaces that key with its value. Let's look at some more samples:

```
print("%(value)s %(value)s %(value)s !" % {"value":"SPAM"})
# SPAM SPAM SPAM !

print("%(x)i + %(y)i = %(z)i" % {"x":1, "y":2})
# KeyError: 'z'
```

In the first example, you'll notice that we only passed in one value, but it was inserted 3 times! This is one of the advantages of using templates. The second example has an issue in that we forgot to pass in a key, namely the "z" key. Here's how we will fix this error

```
print("%(x)i + %(y)i = %(z)i" % {"x":1, "y":2, "z":3})
# 1 + 2 = 3
```

Now let's look at how we can do something similar with the string's format method!

```
print("Python is as simple as {0}, {1}, {2}".format("a", "b", "c"))
# 'Python is as simple as a, b, c'

print("Python is as simple as {1}, {0}, {2}".format("a", "b", "c"))
# 'Python is as simple as b, a, c'

xy = {"x":0, "y":10}
print("Graph a point at where x={x} and y={y}".format(**xy))
# Graph a point at where x=0 and y=10
```

In the first two examples, you can see how we can pass items positionally. If we rearrange the order, we get a slightly different output. The last example

uses a dictionary like we were using in the templates above. However, we

have to extract the dictionary using the double asterisk to get it to work correctly here.

There are lots of other things you can do with strings, such as specifying a width, aligning the text, converting to different bases and much more. Be sure to take a look at some of the references below for more information.

> - Python's official documentation on the str type
> - String Formatting
> - More on String Formatting
> - Python 2.x documentation on unicode

## Wrapping Up #

We have covered a lot in this chapter. Let's review:

First we learned how to create strings themselves, then we moved on to the topic of string concatenation. After that we looked at some of the methods that the string object gives us. Next we looked at string slicing and we finished up by learning about string substitution.

In the next chapter, we will look at three more of Python's built-in data types: lists, tuples and dictionaries. Let's get to it!