String Formatting!

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

```
player = 'Thomas'
points = 33

'Last night, '+player+' scored '+str(points)+' points.'      # concatenation
f'Last night, {player} scored {points} points.'              # string formatting
```

Formatting with the .format() method

A better way to format objects into your strings for print statements is with the string `.format()` method. The syntax is:

```
'String here {} then also {}'.format('something1','something2')
```

For example…

```
In [13]: print('This is a string with an {}'.format('insert'))
```

```
This is a string with an insert
```

Few codes showing how .format() is used…

```
In [14]: print('The {2} {1} {0}'.format('fox','brown','quick'))
```

```
The quick brown fox
```

2. Inserted objects can be assigned keywords:

```
In [15]: print('First Object: {a}, Second Object: {b}, Third Object: {c}'.format(a=1,b='Two',c=12.3))
```

```
First Object: 1, Second Object: Two, Third Object: 12.3
```

3. Inserted objects can be reused, avoiding duplication:

In [16]:
```
print('A %s saved is a %s earned.' %('penny','penny'))
# vs.
print('A {p} saved is a {p} earned.'.format(p='penny'))
```

```
A penny saved is a penny earned.
A penny saved is a penny earned.
```

Formatted String Literals (f-strings)

Introduced in Python 3.6, f-strings offer several benefits over the older `.format()` string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through `.format(var).`

In [21]:
```
name = 'Fred'

print(f"He said his name is {name}.")
```

```
He said his name is Fred.
```

Pass `!r` to get the string representation:

In [22]:
```
print(f"He said his name is {name!r}")
```

```
He said his name is 'Fred'
```