Master the Format string

Basic Formatting

In [1]:
```
print('{} {}'.format('one', 'two'))
```

```
one two
```

In [2]:
```
print('{} {}'.format(1, 2))
```

```
1 2
```

Using Positional index

In [3]:
```
print("{1} {0}".format('one', 'two'))
```

```
two one
```

In [4]:
```
class Data(object):

    def __str__(self):
        return 'str'

    def __repr__(self):
        return 'repr'

print('{0!s} {0!r}'.format(Data()))
```

```
str repr
```

In Python 3 there exists an additional conversion flag that uses the output of repr(...) but uses ascii(...) instead.

Padding and aligning strings

1. Right allign

In [7]:
```
ls1 = [x**2 for x in range(3)]
ls2 = [x/2 for x in range(3)]
for i in range(3):
    print(f'{ls1[i]:>8}', end="")
print()
for i in range(3):
    print(f'{ls2[i]:>8}', end="")
```

```
       0       1       4
     0.0     0.5     1.0
```

1. Left allign

In [8]:
```
print('{:10}'.format('test'))
```

```
test
```

OR

In [9]:
```
print('{:<10}'.format('test'))
```

```
test
```

In [11]:
```
ls1 = [x**2 for x in range(3)]
ls2 = [x/2 for x in range(3)]
for i in range(3):
    print(f'{ls1[i]:<8}', end="")
print()
for i in range(3):
    print(f'{ls2[i]:<8}', end="")
```

```
0       1       4
0.0     0.5     1.0
```

In [12]:
```
print('{:_<10}'.format('test'))
```

```
test______
```

In [13]:
```
print('{:_>10}'.format('test'))
```

```
______test
```

1. Centre allign

In [15]:
```
print('{:^12}'.format('zip'))
```

```
    zip
```

In [16]:
```
print('{:_^12}'.format('zip'))
```

```
____zip_____
```

In [17]:
```
print('{:-^12}'.format('zip'))
```

```
----zip-----
```

In [18]:
```
print('{:$^12}'.format('zip'))
```

```
$$$$zip$$$$$
```

In [19]:
```
print('{:`^12}'.format('zip'))
```

```
````zip`````
```

I think you get the idea now.

Truncating long strings

In [20]:
```
print('{:.5}'.format('xylophone'))
```

```
xylop
```

Combining truncating and padding

In [28]:
```
print('{:-<10.5}'.format('xylophone'))
```

```
xylop-----
```

Numbers

In [33]:
```
print('{:d}'.format(42))
```

```
42
```

Floats

In [32]:
```
print('{:f}'.format(3.141592653589793))
```

```
3.141593
```

In [31]:
```
print('{:f}'.format(3))
```

```
3.000000
```

Padding numbers
```

In [35]:
```
print('{:40d}'.format(42))
```

```
                                      42
```

Note: *Numbers are aligning to right*

In [38]:
```
print('{:06.2f}'.format(3.141592653589793))
```

```
003.14
```

In [39]:
```
print('{:04d}'.format(42))
```

```
0042
```

Signed Numbers

In [40]:
```
print('{:+d}'.format(42))
```

```
+42
```

Use a space character to indicate that negative numbers should be prefixed with a minus symbol and a leading space should be used for positive ones.

In [41]:
```
print('{: d}'.format((- 23)))
```

```
-23
```

In [43]:
```
print('{: d}'.format(42))
```

```
 42
```

New style formatting is also able to control the position of the sign symbol relative to the padding.

In [44]:
```
print('{:=5d}'.format((- 23)))
```

```
-   23
```

In [45]:
```
print('{:=+5d}'.format(23))
```

```
+   23
```

Named placeholders

In [47]:
```
print('{last} {first}'.format(first='HI', last='BYE'))
```

```
BYE HI
```

Getitem and Getattr

In [48]:
```
person = {'first': 'Jean-Luc', 'last': 'Picard'}
print('{p[first]} {p[last]}'.format(p=person))
```

```
Jean-Luc Picard
```

In [49]:
```
data = [4, 8, 15, 16, 23, 42]
print('{d[4]} {d[5]}'.format(d=data))
```

```
23 42
```

In [50]:
```
class Plant(object):
    type = 'tree'
print('{p.type}'.format(p=Plant()))
```

```
tree
```

In [51]:
```
class Plant(object):
    type = 'tree'
    kinds = [{'name': 'oak'}, {'name': 'maple'}]
print('{p.type}: {p.kinds[0][name]}'.format(p=Plant()))
```

```
tree: oak
```

Datetime

In [55]:
```
from datetime import datetime as dt
print('{:%Y-%m-%d %H:%M}'.format(dt.today()))
```

```
2020-06-25 13:40
```

Parametrized formats

Additionally, new style formatting allows all of the components of the format to be specified dynamically using parametrization. Parametrized formats are nested expressions in braces that can appear anywhere in the parent format after the colon.

In [56]:
```
print('{:{align}{width}}'.format('test', align='^', width='10'))
```

```
   test
```

In [57]:
```
print('%.*s = %.*f' % (3, 'Gibberish', 3, 2.7182))
```

```
Gib = 2.718
```

In [59]:
```
print('{:.{prec}} = {:.{prec}f}'.format('Gibberish', 2.7182, prec=3))
```

```
Gib = 2.718
```

In [60]:
```
print('{:{width}.{prec}f}'.format(2.7182, width=5, prec=2))
```

```
 2.72
```

In [61]:
```
print('{:{prec}} = {:{prec}}'.format('Gibberish', 2.7182, prec='.3'))
```

```
Gib = 2.72
```

In [62]:
```
from datetime import datetime
dt = datetime(2001, 2, 3, 4, 5)
print('{:{dfmt} {tfmt}}'.format(dt, dfmt='%Y-%m-%d', tfmt='%H:%M'))
```

```
2001-02-03 04:05
```

In [63]:
```
print('{:{}{}{}.{}}'.format(2.7182818284, '>', '+', 10, 3))
```

```
     +2.72
```

It is the same as:

In [68]:
```
print('{:>+10.3}'.format(2.7182818284, '>', '+', 10, 3))
```

```
     +2.72
```

In [69]:
```
print('{:{}{sign}{}.{}}'.format(2.7182818284, '>', 10, 3, sign='+'))
```

```
     +2.72
```

Custom objects

The datetime example works through the use of the **format**() magic method. You can define custom format handling in your own objects by overriding this method. This gives you complete control over the format syntax used.

In [70]:
```
class HAL9000(object):

    def __format__(self, format):
        if (format == 'open-the-pod-bay-doors'):
            return "I'm afraid I can't do that."
        return 'HAL 9000'
print('{:open-the-pod-bay-doors}'.format(HAL9000()))
```

```
I'm afraid I can't do that.
```