*Py*Format Using *%* and *.format()* for great good!

Source : http://pyformat.info/

Python has had awesome string formatters for many years but the documentation on them is far too theoretic and technical. With this site we try to show you the most common use-cases covered by the old and new style string formatting API with practical examples.

Table of Contents:

1. Basic formatting

Simple positional formatting is probably the most common use-case. Use it if the order of your arguments is not likely to change and you only have very few elements you want to concatenate.

Since the elements are not represented by something as descriptive as a name this simple style should only be used to format a relatively small number of elements.

Old : `'%s %s' % ('one', 'two', )`
New : `'{} {}'.format('one', 'two')`
Output : `one two`

Old : `'%d %d' % (1, 2)`
New : `'{} {}'.format(1, 2)`
Output : `1 2`

With new style formatting it is possible (and in Python 2.6 even mandatory) to give placeholders an explicit positional index.

This allows for re-arranging the order of display without changing the arguments.

Old : (not available with old style formatting)

New : `'{1} {0}'.format('one', 'two')`
Output : `two one`

2. Padding and aligning strings

By default values are formatted to take up only as many characters as needed to represent the content. It is however also possible to define that a value should be padded to a specific length.
Unfortunately the default alignment differs between old and new style formatting. The old style defaults to right aligned while for new style it's left.

Align right:

Old : `'%10s' % ('test', )`
New : `'{:>10}'.format('test')`
Output : `      test`

Align left:

Old : `'%-10s' % ('test', )`
New : `'{:10}'.format('test')`
Output : `test`

Again new style formatting surpasses the old variant by providing more control over how values are padded and aligned.
You are able to choose the padding character:

Old : (not available with old style formatting)
New : `'{:*<10}'.format('test')`
Output : `test******`

And also center align values:

Old : (not available with old style formatting)
New : `'{:^10}'.format('test')`
Output : `   test`

3. Truncating long strings

Inverse to padding it is also possible to truncate overly long values to a specific number of characters.
The number behind a . in the format specifies the precision of the output. For strings that means that the output is truncated to the specified length. In our example this would be 5 characters.

Old : `'%.5s' % ('xylophone', )`
New : `'{:.5}'.format('xylophone')`
Output : `xylop`

4. Numbers

Of course it is also possible to format numbers.

Integers:

Old : `'%d' % (42, )`
New : `'{:d}'.format(42)`
Output : `42`

Floats:

Old : `'%f' % (3.141592653589793, )`
New : `'{:f}'.format(3.141592653589793)`
Output : `3.141593`

5. Padding numbers

Similar to strings numbers can also be constrained to a specific width.

Old : `'%4d' % (42, )`
New : `'{:4d}'.format(42)`
Output : `  42`

Again similar to truncating strings the precision for floating point numbers limits the number of positions after the decimal point.
For floating points the padding value represents the length of the complete output. In the example below we want our output to have at least 6 characters with 2 after the decimal point.

Old : `'%06.2f' % (3.141592653589793, )`
New : `'{:06.2f}'.format(3.141592653589793)`
Output : `003.14`

For integer values providing a precision doesn't make much sense and is actually forbidden in the new style (it will result in a ValueError).

Old : `'%04d' % (42, )`
New : `'{:04d}'.format(42)`

Output : `0042`

6. Signed numbers

By default only negative numbers are prefixed with a sign. This can be changed of course.

Old : `'%+d' % (42, )`
New : `'{:+d}'.format(42)`
Output : `+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.

Old : `'% d' % (-23, )`
New : `'{: d}'.format(-23)`
Output : `-23`

Old : `'% d' % (42, )`
New : `'{: d}'.format(42)`
Output : ` 42`

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

Old : (not available with old style formatting)
New : `'{:=5d}'.format(-23)`
Output : `-  23`

7. Named placeholders

Both formatting styles support named placeholders.

Setup :

```
data = {
    'first': 'Hodor',
    'last': 'Hodor!',
}
```

Old : `'%(first)s %(last)s' % data`
New : `'{first} {last}'.format(**data)`
Output : `Hodor Hodor!`

`.format()` also accepts keyword arguments.

Old : (not available with old style formatting)

New :
```
'{first} {last}'.format(first="Hodor", last="Hodor!")
```
Output :
```
Hodor Hodor!
```

8. Getitem and Getattr

New style formatting allows even greater flexibility in accessing nested data structures. It supports accessing containers that support __getitem__ like for example dictionaries and lists:

Setup :
```
person = {
    'first': 'Jean-Luc',
    'last': 'Picard',
}
```

Old : (not available with old style formatting)

New :
```
'{p[first]} {p[last]}'.format(p=person)
```
Output :
```
Jean-Luc Picard
```

Setup :
```
data = [4, 8, 15, 16, 23, 42]
```
New :
```
'{d[4]} {d[5]}'.format(d=data)
```
Output :
```
23 42
```

As well as accessing attributes on objects via getattr():

Setup
```
class Plant(object):
    type = "tree"
```

Old : (not available with old style formatting)

New :
```
'{p.type}'.format(p=Plant())
```
Output :
```
tree
```

Both type of access can be freely mixed and arbitrarily nested:

Setup
```
class Plant(object):
    type = "tree"
    kinds = [{
        'name': "oak",
    }, {
        'name': "maple"
    }]
```

Old : (not available with old style formatting)

New :
```
'{p.type}: {p.kinds[0][name]}'.format(p=Plant())
```
Output :
```
tree: oak
```

9. Datetime

Additionally new style formatting allows objects to control their own rendering. This for example allows datetime objects to be formatted inline:

Setup :
```
from datetime import datetime
```

Old : (not available with old style formatting)

New :
```
'{:%Y-%m-%d %H:%M}'.format(datetime(2001, 2, 3, 4, 5))
```

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

10. Custom objects

The above 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.

Setup
```
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"
```

Old : (not available with old style formatting)

New :
```
'{:open-the-pod-bay-doors}'.format(HAL9000())
```

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