PyFormat

Using `%` and `.format()` for great good!

Contribute on GitHub

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.

All examples on this page work out of the box with with Python 2.7, 3.2, 3.3, 3.4, and 3.5 without requiring any additional libraries.

Further details about these two formatting methods can be found in the official Python documentation:

- old style
- new style

If you want to contribute more examples, feel free to create a pull-request on Github!

Table of Contents:

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

```
o n e   t w o
```

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.

This operation is not available with old-style formatting.

New

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

Output

```
t w o   o n e
```

Value conversion

The new-style simple formatter calls by default the `__format__()` method of an object for its representation. If you just want to render the output of `str(...)` or `repr(...)` you can use the `!s` or `!r` conversion flags.

In %-style you usually use `%s` for the string representation but there is `%r` for a `repr(...)` conversion.

Setup

```
class Data(object):

    def __str__(self):
        return 'str'

    def __repr__(self):
        return 'repr'
```

Old

```
'%s %r' % (Data(), Data())
```

New

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

Output

```
s t r   r e p r
```

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

Setup

```
class Data(object):

    def __repr__(self):
        return 'räpr'
```

Old

```
'%r %a' % (Data(), Data())
```

New

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

Output

```
r ä p r   r \ x e 4 p r
```

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

```
          t e s t
```

Align left:

Old

```
'%-10s' % ('test',)
```

New

```
'{:10}'.format('test')
```

Output

```
t e s t
```

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:

> This operation is not available with old-style formatting.

New

```
'{:_<10}'.format('test')
```

Output

```
t e s t _ _ _ _ _ _
```

And also center align values:

> This operation is not available with old-style formatting.

New

```
'{:^10}'.format('test')
```

Output

```
    t e s t
```

When using center alignment where the length of the string leads to an uneven split of the padding characters the extra character will be placed on the right side:

> This operation is not available with old-style formatting.

New

```
'{:^6}'.format('zip')
```

Output

```
 z i p
```

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

```
x y l o p
```

Combining truncating and padding

It is also possible to combine truncating and padding:

Old

```
'%-10.5s' % ('xylophone',)
```

New

```
'{:10.5}'.format('xylophone')
```

Output

```
x y l o p
```

Numbers

Of course it is also possible to format numbers.

Integers:

Old

```
'%d' % (42,)
```

New

```
'{:d}'.format(42)
```

Output

```
4 2
```

Floats:

Old

```
'%f' % (3.141592653589793,)
```

New

```
'{:f}'.format(3.141592653589793)
```

Output

```
3 . 1 4 1 5 9 3
```

Padding numbers

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

Old

```
'%4d' % (42,)
```

New

```
'{:4d}'.format(42)
```

Output

```
  4 2
```

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

```
0 0 3 . 1 4
```

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

```
0 0 4 2
```

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

```
+ 4 2
```

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

```
- 2 3
```

Old

```
'% d' % (42,)
```

New

```
'{: d}'.format(42)
```

Output

```
  4 2
```

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

> This operation is not available with old-style formatting.

New

```
'{:=5d}'.format((- 23))
```

Output

```
-    2 3
```

New

```
'{:=+5d}'.format(23)
```

Output

```
+    2 3
```

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

```
H o d o r  H o d o r !
```

`.format()` also accepts keyword arguments.

This operation is not available with old-style formatting.

New

```
'{first} {last}'.format(first='Hodor', last='Hodor!')
```

Output

```
H o d o r  H o d o r !
```

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:

This operation is not available with old-style formatting.

Setup

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

New

```
'{p[first]} {p[last]}'.format(p=person)
```

Output

```
J e a n - L u c  P i c a r d
```

Setup

```
data = [4, 8, 15, 16, 23, 42]
```

New `'{d[4]} {d[5]}'.format(d=data)`

Output `2 3   4 2`

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

This operation is not available with old-style formatting.

Setup

```
class Plant(object):
    type = 'tree'
```

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

Output `t r e e`

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

This operation is not available with old-style formatting.

Setup

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

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

Output `t r e e :   o a k`

Datetime

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

This operation is not available with old-style formatting.

Setup

```
from datetime import datetime
```

New

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

Output

```
2 0 0 1 - 0 2 - 0 3   0 4 : 0 5
```

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.

Old style formatting also supports some parametrization but is much more limited. Namely it only allows parametrization of the width and precision of the output.

Parametrized alignment and width:

This operation is not available with old-style formatting.

New

```
'{:{align}{width}}'.format('test', align='^', width='10')
```

Output

```
   t e s t
```

Parametrized precision:

Old

```
'%.*s = %.*f' % (3, 'Gibberish', 3, 2.7182)
```

New

```
'{:.{prec}} = {:.{prec}f}'.format('Gibberish', 2.7182, prec=3)
```

Output

```
G i b   =   2 . 7 1 8
```

Width and precision:

Old

```
'%*.*f' % (5, 2, 2.7182)
```

New

```
'{:{width}.{prec}f}'.format(2.7182, width=5, prec=2)
```

Output

```
  2 . 7 2
```

The nested format can be used to replace *any* part of the format spec, so the precision example above could be rewritten as:

This operation is not available with old-style formatting.

New

```
'{:{prec}} = {:{prec}}'.format('Gibberish', 2.7182, prec='.3')
```

Output

```
G i b   =   2 . 7 2
```

The components of a date-time can be set separately:

This operation is not available with old-style formatting.

Setup

```
from datetime import datetime
dt = datetime(2001, 2, 3, 4, 5)
```

New

```
'{:{dfmt} {tfmt}}'.format(dt, dfmt='%Y-%m-%d', tfmt='%H:%M')
```

Output

```
2 0 0 1 - 0 2 - 0 3   0 4 : 0 5
```

The nested formats can be positional arguments. Position depends on the order of the opening curly braces:

This operation is not available with old-style formatting.

New

```
'{:{}{}{}.{}}'.format(2.7182818284, '>', '+', 10, 3)
```

Output

```
         + 2 . 7 2
```

And of course keyword arguments can be added to the mix as before:

This operation is not available with old-style formatting.

New

```
'{:{}{sign}{}.{}}'.format(2.7182818284, '>', 10, 3, sign='+')
```

Output

```
+ 2 . 7 2
```

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.

This operation is not available with old-style formatting.

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'
```

New

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

Output

```
I ' m   a f r a i d   I   c a n ' t   d o   t h a t .
```

You might also like Python strftime reference by Will McCutchen.

Curated by Ulrich Petri & Horst Gutmann

Version: 516238e0e82853567ca7ba20adcbd634b8b8458a (built at 2016-12-27 19:30:23.236195+00:00)