String Formatting All Versions

When storing and transforming data for humans to see, string formatting can become very important. Python offers a wide variety of string formatting methods which are outlined in this topic.

Examples

Basics of String Formatting

```
foo = 1
bar = 'bar'
baz = 3.14
```

You can use `str.format` to format output. Bracket pairs are replaced with arguments in the order in which the arguments are passed:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

Indexes can also be specified inside the brackets. The numbers correspond to indexes of the arguments passed to the `str.format` function (0-based).

```
print('{0}, {1}, {2}, and {1}'.format(foo, bar, baz))
# Out: "1, bar, 3.14, and bar"
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Out: index out of range error
```

Named arguments can be also used:

```
print("X value is: {x_val}. Y value is: {y_val}.".format(x_val=2, y_val=3))
# Out: "X value is: 2. Y value is: 3."
```

Object attributes can be referenced when passed into `str.format` :

```
class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('My value is: {0.value}'.format(my_value))  # "0" is optional
# Out: "My value is: 6"
```

Dictionary keys can be used as well:

```
my_dict = {'key': 6, 'other_key': 7}
print("My other key is: {0[other_key]}".format(my_dict))  # "0" is optional
# Out: "My other key is: 7"
```

Same applies to list and tuple indices:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

Note: In addition to `str.format` , Python also provides the modulo operator `%` --also known as the *string formatting* or *interpolation operator* (see PEP 3101)--for formatting strings. `str.format` is a successor of `%` and it offers greater flexibility, for instance by making it easier to carry out multiple substitutions.

In addition to argument indexes, you can also include a *format specification* inside the curly brackets. This is an expression that follows special rules and must be preceded by a colon (`:`). An example of format specification is the alignment directive `:~^20` (`^` stands for center alignment, total width 20, fill with ~ character):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

Alignment and padding

Python 2.x ≥2.6

The `format()` method can be used to change the alignment of the string. You have to do it with a format expression of the form `:[fill_char][align_operator][width]` where `align_operator` is one of:

- `<` forces the field to be left-aligned within `width` .
- `>` forces the field to be right-aligned within `width` .
- `^` forces the field to be centered within `width` .
- `=` forces the padding to be placed after the sign (numeric types only).

`fill_char` (if omitted default is whitespace) is the character used for the padding.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# Out: '-00123'
```

Note: you could achieve the same results using the string functions `ljust()` , `rjust()` , `center()` , `zfill()` , however these functions are deprecated since version 2.5.

Format literals (f-string)

Literal format strings were introduced in PEP 498 (Python3.6 and upwards), allowing you to prepend `f` to the beginning of a string literal to effectively apply `.format` to it with all variables in the current scope.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

This works with more advanced format strings too, including alignment and dot notation.

```
>>> f'{foo:^7s}'
'  bar  '
```

Note: The `f''` does not denote a particular type like `b''` for `bytes` or `u''` for `unicode` in python2. The formating is immediately applied, resulting in a normal stirng.

The format strings can also be *nested* :

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'*************$478.23'
```

The expressions in an f-string are evaluated in left-to-right order. This is detectable only if the expressions have side effects:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

String formatting with datetime

Any class can configure its own string formatting syntax through the `__format__` method. A type in the standard Python library that makes handy use of this is the `datetime` type, where one can use `strftime` -like formatting codes directly within `str.format` :

```
>>> from datetime import datetime
>>> 'North America: {dt:%m/%d/%Y}.  ISO: {dt:%Y-%m-%d}.'.format(dt=datetime.now())
'North America: 07/21/2016.  ISO: 2016-07-21.'
```

A full list of list of datetime formatters can be found in the official documenttion .

Float formatting

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
'42.123'

>>> '{0:.5f}'.format(42.12345)
'42.12345'

>>> '{0:.7f}'.format(42.12345)
'42.1234500'
```

Same hold for other way of referencing:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

Floating point numbers can also be formatted in scientific notation or as percentages:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

You can also combine the {0} and {name} notations. This is especially useful when you want to round all variables to a pre-specified number of decimals *with 1 declaration* :

```
>>> s = 'Hello'
>>> a, b, c = 1.12345, 2.34567, 34.5678
>>> digits = 2

>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
'Hello! 1.12, 2.35, 34.57'
```

Format using Getitem and Getattr

Any data structure that supports __getitem__ can have their nested structure formatted:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Object attributes can be accessed using getattr() :

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Named placeholders

Format strings may contain named placeholders that are interpolated using keyword arguments to format .

Using a dictionary:

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Python 3.2+

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

```
'Hodor Hodor!'
```

`str.format_map` allows to use dictionaries without having to unpack them first. Also the class of `data` (which might be a custom type) is used instead of a newly filled `dict`.

Without a dictionary:

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

Nested formatting

Some formats can take additional parameters, such as the width of the formatted string, or the alignment:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

Those can also be provided as parameters to `format` by nesting more `{}` inside the `{}`:

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

In the latter example, the format string `'{:{}{}{}}'` is modified to `'{:*^15}'` (i.e. "center and pad with * to total length of 15") before applying it to the actual string `'foo'` to be formatted that way.

This can be useful in cases when parameters are not known beforehand, for instances when aligning tabular data:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Custom formatting for a class

Note:

> Everything below applies to the `str.format` method, as well as the `format` function. In the text below, the two are interchangeable.

For every value which is passed to the `format` function, Python looks for a `__format__` method for that argument. Your own custom class can therefore have their own `__format__` method to determine how the `format` function will display and format your class and it's attributes.

This is different than the `__str__` method, as in the `__format__` method you can take into account the formatting language, including alignment, field width etc, and even (if you wish) implement your own format specifiers, and your own formatting language extensions. 1

```
object.__format__(self, format_spec)
```

For example :

```
# Example in Python 2 - but can be easily applied to Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Implement special semantics for the 's' format specifier """
        # Reject anything that isn't an s
        if format_spec[-1] != 's':
            raise ValueError('{} format specifier not understood for this object', format_spec[:

        # Output in this example will be (<a>,<b>,<c>)
        raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
        # Honor the format language by using the inbuilt string format
        # Since we know the original format_spec ends in an 's'
        # we can take advantage of the str.format method with a
        # string argument we constructed above
        return "{r:{f}}".format( r=raw, f=format_spec )
```

```
inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# out :              (1,2,3)
# Note how the right align and field width of 20 has been honored.
```

Note:

> If your custom class does not have a custom `__format__` method and an instance of the class is passed to the `format` function, **Python2** will always use the return value of the `__str__` method or `__repr__` method to determine what to print (and if neither exist then the default `repr` will be used), and you will need to use the `s` format specifier to format this. With **Python3** , to pass your custom class to the `format` function, you will need define `__format__` method on your custom class.

Format integers to differens bases (hex, oct, binary)

```
>>> '{0:x} or {0:X}'.format(42) # Hexadecimal
'2a or 2A'

>>> '{:o}'.format(42) # Octal
'52'

>>> '{:b}'.format(42) # Binary
'101010'

>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # With prefix
'0b101010, 0o52, 0x2a'

>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # Add zero padding
'8 bit: 00101010; Three bytes: 00002a'
```

Use formatting to convert an RGB float tuple to a color hex string:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Only integers can be converted:

```
>>> '{:x}'.format(42.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'x' for object of type 'float'
```

Formatting Numerical Values

The `.format()` method can interpret a number in different formats, such as:

```
>>> '{:b}'.format(10)    # base 2
'1010'
>>> '{:c}'.format(65)    # Unicode character
'A'
>>> '{:d}'.format(0x0a)  # base 10
'10'
>>> '{:o}'.format(10)    # base 8
'12'
>>> '{:x}'.format(10)    # base 16, lowercase
'a'
>>> '{:X}'.format(10)    # base 16, uppercase
'A'
>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Padding and truncating strings, combined

Say you want to print variables in a 3 character column.

Note: doubling `{` and `}` escapes them.

```
s = """
```

```
pad
{{:3}}            :{a:3}:

truncate
{{:.3}}           :{e:.3}:

combined
{{:>3.3}}         :{a:>3.3}:
{{:3.3}}          :{a:3.3}:
{{:3.3}}          :{c:3.3}:
{{:3.3}}          :{e:3.3}:
"""

print (s.format(a="1"*1, c="3"*3, e="5"*5))
```

Output:

```
pad
{:3}            :1  :

truncate
{:.3}           :555:

combined
{:>3.3}         :  1:
{:3.3}          :1  :
{:3.3}          :333:
{:3.3}          :555:
```

Syntax

```
"{}".format(42) ==> "42"
"{0}".format(42) ==> "42"
"{0:.2f}".format(42) ==> "42.00"
"{0:.0f}".format(42.1234) ==> "42"
"{answer}".format(no_answer=41, answer=42) ==> "42"
"{answer:.2f}".format(no_answer=41, answer=42) ==> "42.00"
"{[key]}".format({'key': 'value'}) ==> "value"
"{[1]}".format(['zero', 'one', 'two']) ==> "one"
"{answer} = {answer}".format(answer=42) ==> "42 = 42"
' '.join(['stack', 'overflow']) ==> "stack overflow"
```

Parameters

Remarks

- Should check out PyFormat.info for a very thorough and gentle introduction/explanation of how it works.