# A Guide to the Newer Python String Format Techniques – Real Python

realpython.com (https://realpython.com/python-formatted-output/) · by Real Python

Table of Contents

Remove ads (https://realpython.com/account/join/)

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Formatting Python Strings** (https://realpython.com/courses/formatting-python-strings/)

In the previous tutorial (https://realpython.com/python-input-output/#the-string-modulo-operator) in this introductory series, you learned how to format string data using the **string modulo operator**. The string modulo operator is useful, and it's good for you to be familiar with it because you're likely to encounter it in older Python code. However, there are two newer ways that you can use Python to format strings (https://realpython.com/python-strings/) that are arguably more preferable.

**In this tutorial, you'll learn about:**

1. The **string `.format()` method**
2. The **formatted string literal**, or **f-string**

You'll learn about these formatting techniques in detail and add them to your Python string formatting toolkit. Note that there's a standard module called `string` containing a class called `Template` (https://docs.python.org/3.4/library/string.html#template-strings), which provides some string formatting through interpolation. The string modulo operator provides more or less the same functionality, so you won't cover `string.Template` here.

**Free Download:** Get a sample chapter from Python Tricks: The Book (https://realpython.com/bonus/python-tricks-sample-pdf/) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

## The Python String `.format()` Method

The Python string `.format()` method was introduced in version 2.6. It's similar in many ways to the string modulo operator, but `.format()` goes well beyond in versatility. The general form of a Python `.format()` call is shown below:

```
<template>.format(<positional_argument(s)>, <keyword_argument(s)>)
```

Note that this is a method, not an operator. You call the method on `<template>`, which is a string containing **replacement fields**. The `<positional_arguments>` and `<keyword_arguments>` to the method specify values that are inserted into `<template>` in place of the replacement fields. The resulting formatted string is the method's return value.

In the `<template>` string, replacement fields are enclosed in curly braces (`{}`). Anything not contained in curly braces is literal text that's copied directly from the template to the output. If you need to include a literal curly bracket character, like `{` or `}`, in the template string, then you can escape this character by doubling it:

>>>

```
>>> '{{ {0} }}'.format('foo')
'{ foo }'
```

Now the curly braces are included in your output.



## The String `.format()` Method: Arguments

Let's start with a quick example to get you acquainted before you dive into more detail on how to use this method in Python to format strings. For review, here's the first example from the previous tutorial on the string modulo operator (https://realpython.com/python-input-output/#the-string-modulo-operator):

>>>

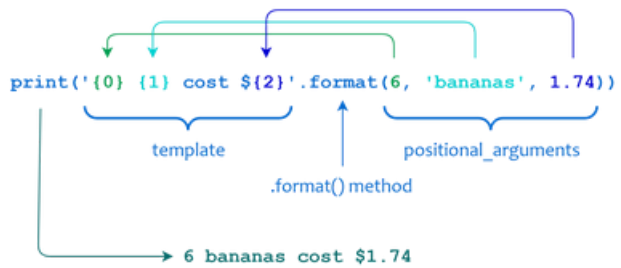
```
>>> print('%d %s cost $%.2f' % (6, 'bananas', 1.74))
6 bananas cost $1.74
```

Here, you used the string modulo operator in Python to format the string. Now, you can use Python's string `.format()` method to obtain the same result, like this:

>>>

```
>>> print('{0} {1} cost ${2}'.format(6, 'bananas', 1.74))
6 bananas cost $1.74
```

In this example, `<template>` is the string `'{0} {1} cost ${2}'`. The replacement fields are `{0}`, `{1}`, and `{2}`, which contain numbers that correspond to the zero-based positional arguments `6`, `'bananas'`, and `1.74`. Each positional argument is inserted into the template in place of its corresponding replacement field:




(https://files.realpython.com/media/t.e6b8525755da.png)

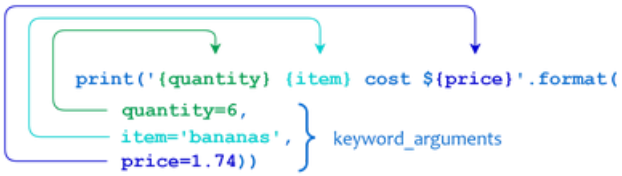
Using The String .format() Method in Python to Format a String With Positional Arguments

The next example uses keyword arguments (https://realpython.com/python-kwargs-and-args/) instead of positional parameters to produce the same result:

>>>

```
>>> print('{quantity} {item} cost ${price}'.format(
...     quantity=6,
...     item='bananas',
...     price=1.74))
6 bananas cost $1.74
```

In this case, the replacement fields are `{quantity}`, `{item}`, and `{price}`. These fields specify keywords that correspond to the keyword arguments `quantity=6`, `item='bananas'`, and `price=1.74`. Each keyword value is inserted into the template in place of its corresponding replacement field:




(https://files.realpython.com/media/t.53be85450e90.png)
Using The String .format() Method in Python to Format a String With Keyword Arguments

You'll learn more about positional and keywords arguments there in the next tutorial in this introductory series, which explores functions and argument passing. For now, the two sections that follow will show you how these are used with the Python `.format()` method.

## Positional Arguments

Positional arguments are inserted into the template in place of **numbered replacement fields**. Like list (https://realpython.com/courses/lists-tuples-python/) indexing, the numbering of replacement fields is zero-based. The first positional argument is numbered `0`, the second is numbered `1`, and so on:

>>>

```
>>> '{0}/{1}/{2}'.format('foo', 'bar', 'baz')
'foo/bar/baz'
```

Note that replacement fields don't have to appear in the template in numerical order. They can be specified in any order, and they can appear more than once:

>>>

```
>>> '{2}.{1}.{0}/{0}{0}.{1}{1}.{2}{2}'.format('foo', 'bar', 'baz')
'baz.bar.foo/foofoo.barbar.bazbaz'
```

When you specify a replacement field number that's out of range, you'll get an error (https://realpython.com/courses/python-exceptions-101/). In the following example, the positional arguments are numbered `0`, `1`, and `2`, but you specify `{3}` in the template:

```
>>>
```

```
>>> '{3}'.format('foo', 'bar', 'baz')
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    '{3}'.format('foo', 'bar', 'baz')
IndexError: tuple index out of range
```

This raises an `IndexError` exception (https://realpython.com/python-exceptions/).

Starting with Python 3.1, you can omit the numbers in the replacement fields, in which case the interpreter assumes sequential order. This is referred to as **automatic field numbering**:

```
>>>
```

```
>>> '{}/{}/{}'.format('foo', 'bar', 'baz')
'foo/bar/baz'
```

When you specify automatic field numbering, you must provide at least as many arguments as there are replacement fields:

```
>>>
```

```
>>> '{}{}{}{}'.format('foo','bar','baz')
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    '{}{}{}{}'.format('foo','bar','baz')
IndexError: tuple index out of range
```

In this case, there are four replacement fields in the template but only three arguments, so an `IndexError` exception occurs. On the other hand, it's fine if the arguments outnumber the replacement fields. The excess arguments simply aren't used:

```
>>>
```

```
>>> '{}{}'.format('foo', 'bar', 'baz')
'foobar'
```

Here, the argument `'baz'` is ignored.

Note that you can't intermingle these two techniques:

```
>>>
```

```
>>> '{1}{}{0}'.format('foo','bar','baz')
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    '{1}{}{0}'.format('foo','bar','baz')
ValueError: cannot switch from manual field specification to automatic field
 numbering
```

When you use Python to format strings with positional arguments, you must choose between either automatic or explicit replacement field numbering.



## Keyword Arguments

Keyword arguments are inserted into the template string in place of **keyword replacement fields with the same name**:

>>>

```
>>> '{x}/{y}/{z}'.format(x='foo', y='bar', z='baz')
'foo/bar/baz'
```

In this example, the values of the keyword arguments `x`, `y`, and `z` take the place of the replacement fields `{x}`, `{y}`, and `{z}`, respectively.

If you refer to a keyword argument that's missing, then you'll see an error:

>>>

```
>>> '{x}/{y}/{w}'.format(x='foo', y='bar', z='baz')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'w'
```

Here, you specify replacement field `{w}`, but there's no corresponding keyword argument named `w`. Python raises a `KeyError` exception (https://realpython.com/python-keyerror/).

While you have to specify positional arguments in sequential order, but you can specify keyword arguments in any arbitrary order:

>>>

```
>>> '{0}/{1}/{2}'.format('foo', 'bar', 'baz')
'foo/bar/baz'
>>> '{0}/{1}/{2}'.format('bar', 'baz', 'foo')
'bar/baz/foo'

>>> '{x}/{y}/{z}'.format(x='foo', y='bar', z='baz')
'foo/bar/baz'
>>> '{x}/{y}/{z}'.format(y='bar', z='baz', x='foo')
'foo/bar/baz'
```

You can specify both positional and keyword arguments in one Python `.format()` call. Just note that, if you do so, then all the **positional arguments must appear before** any of the keyword arguments:

>>>

```
>>> '{0}{x}{1}'.format('foo', 'bar', x='baz')
'foobazbar'
>>> '{0}{x}{1}'.format('foo', x='baz', 'bar')
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

In fact, the requirement that all positional arguments appear before any keyword arguments doesn't apply only to Python format methods. This is generally true of any function or method call. You'll learn more about this in the next tutorial in this series, which explores functions and function calls.

In all the examples shown so far, the values you passed to the Python `.format()` method have been literal values, but you may specify variables (https://realpython.com/python-variables/) as well:

>>>

```
>>> x = 'foo'
>>> y = 'bar'
>>> z = 'baz'
>>> '{0}/{1}/{s}'.format(x, y, s=z)
'foo/bar/baz'
```

In this case, you pass the variables `x` and `y` as positional parameter values and `z` as a keyword parameter value.

## The String `.format()` Method: Simple Replacement Fields

As you've seen, when you call Python's `.format()` method, the `<template>` string contains **replacement fields**. These indicate where in the template to insert the arguments to the method. A replacement field consists of three components:

```
{[<name>][!<conversion>][:<format_spec>]}
```

These components are interpreted as follows:

Component Meaning `<name>` Specifies the source of the value to be formatted `<conversion>` Indicates which standard Python function to use to perform the conversion `<format_spec>` Specifies more detail about how the value should be converted

Each component is optional and may be omitted. Let's take a look at each component in more depth.



### The `<name>` Component

The `<name>` component is the first portion of a replacement field:

```
{ [<name>] [!<conversion>][:<format_spec>]}
```

`<name>` indicates which argument from the argument list is inserted into the Python format string in the given location. It's either a number for a positional argument or a keyword for a keyword argument. In the following example, the `<name>` components of the replacement fields are `0`, `1`, and `baz`, respectively:

```
>>>
```

```
>>> x, y, z = 1, 2, 3
>>> '{0}, {1}, {baz}'.format(x, y, baz=z)
'1, 2, 3'
```

If an argument is a list, then you can use indices with `<name>` to access the list's elements:

```
>>>
```

```
>>> a = ['foo', 'bar', 'baz']
>>> '{0[0]}, {0[2]}'.format(a)
'foo, baz'
>>> '{my_list[0]}, {my_list[2]}'.format(my_list=a)
'foo, baz'
```

Similarly, you can use a key reference with `<name>` if the corresponding argument is a dictionary (https://realpython.com/courses/dictionaries-python/):

```
>>>
```

```
>>> d = {'key1': 'foo', 'key2': 'bar'}
>>> d['key1']
'foo'
>>> '{0[key1]}'.format(d)
'foo'
>>> d['key2']
'bar'
>>> '{my_dict[key2]}'.format(my_dict=d)
'bar'
```

You can also reference **object attributes** from within a replacement field. In the previous tutorial (https://realpython.com/python-variables/#object-references) in this series, you learned that virtually every item of data in Python is an **object**. Objects may have **attributes** assigned to them that are accessed using dot notation:

```
obj.attr
```

Here, `obj` is an object with an attribute named `attr`. You use dot notation to access the object's attribute. Let's see an example. Complex numbers (https://realpython.com/python-complex-numbers/) in Python have attributes named `.real` and `.imag` that represent the real and imaginary portions of the number. You can access these using dot notation as well:

```
>>>
```

```
>>> z = 3+5j
>>> type(z)
<class 'complex'>
>>> z.real
3.0
>>> z.imag
5.0
```

There are several upcoming tutorials in this series on object-oriented programming, in which you'll learn a great deal more about object attributes like these.

The relevance of object attributes in this context is that you can specify them in a Python `.format()` replacement field:

```
>>>
```

```
>>> z
(3+5j)
>>> 'real = {0.real}, imag = {0.imag}'.format(z)
'real = 3.0, imag = 5.0'
```

As you can see, it's relatively straightforward in Python to format components of complex objects using the `.format()` method.

## The `<conversion>` Component

The `<conversion>` component is the middle portion of a replacement field:

```
{[<name>] [!<conversion>] [:<format_spec>]}
```

Python can format an object as a string using three different built-in functions:

1. `str()`
2. `repr()`
3. `ascii()`

By default, the Python `.format()` method uses `str()`, but in some instances, you may want to force `.format()` to use one of the other two. You can do this with the `<conversion>` component of a replacement field. The possible values for `<conversion>` are shown in the table below:

Value Meaning `!s` Convert with `str()` `!r` Convert with `repr()` `!a` Convert with `ascii()`

The following examples force Python to perform string conversion using `str()`, `repr()`, and `ascii()`, respectively:

```
>>>
```

```
>>> '{0!s}'.format(42)
'42'
>>> '{0!r}'.format(42)
'42'
>>> '{0!a}'.format(42)
'42'
```

In many cases, the result is the same regardless of which conversion function you use, as you can see in the example above. That being said, you won't often need the `<conversion>` component, so you won't spend a lot of time on it here. However, there are situations where it makes a difference, so it's good to be aware that you have the capability to force a specific conversion function if you need to.



## The `<format_spec>` Component

The `<format_spec>` component is the last portion of a replacement field:

```
{[<name>][!<conversion>] [:<format_spec>] }
```

`<format_spec>` represents the guts of the Python `.format()` method's functionality. It contains information that exerts fine control over how values are formatted prior to being inserted into the template string. The general form is this:

```
:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<prec>][<type>]
```

The ten subcomponents of `<format_spec>` are specified in the order shown. They control formatting as described in the table below:

Subcomponent Effect `:` Separates the `<format_spec>` from the rest of the replacement field `<fill>` Specifies how to pad values that don't occupy the entire field width `<align>` Specifies how to justify values that don't occupy the entire field width `<sign>` Controls whether a leading sign is included for numeric values `#` Selects an alternate output form for certain presentation types `0` Causes values to be padded on the left with zeros instead of ASCII space characters `<width>` Specifies the minimum width of the output `<group>` Specifies a grouping character for numeric output `.<prec>` Specifies the number of digits after the decimal point for floating-point presentation types, and the maximum output width for string presentations types `<type>` Specifies the presentation type, which is the type of conversion performed on the corresponding argument

These functions are analogous to the components you'll find in the string modulo operator's conversion specifier (https://realpython.com/python-input-output/#conversion-specifiers), but with somewhat greater capability. You'll see their capabilities explained more fully in the following sections.

### The `<type>` Subcomponent

Let's start with **`<type>`**, which is the final portion of `<format_spec>`. The `<type>` subcomponent specifies the presentation type, which is the type of conversion that's performed on the corresponding value to produce the output. The possible values are shown below:

Value Presentation Type `b` Binary integer `c` Single character `d` Decimal integer `e` or `E` Exponential `f` or `F` Floating point `g` or `G` Floating point or Exponential `o` Octal integer `s` String `x` or `X` Hexadecimal integer `%` Percentage

These are like the conversion types (https://realpython.com/python-input-output/#conversion-type) used with the string modulo operator, and in many cases, they function the same. The following examples demonstrate the similarity:

>>>

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

>>> '%f' % 2.1
'2.100000'
>>> '{:f}'.format(2.1)
'2.100000'

>>> '%s' % 'foobar'
'foobar'
>>> '{:s}'.format('foobar')
'foobar'

>>> '%x' % 31
'1f'
>>> '{:x}'.format(31)
'1f'
```

However, there are some minor differences between some of the Python `.format()` presentation types and the string modulo operator conversion types:

Type `.format()` Method String Modulo Operator **b** Designates binary integer conversion Not supported **i, u** Not supported Designates integer conversion **c** Designates character conversion, and the corresponding value must be an integer Designates character conversion, but the corresponding value may be either an integer or a single-character string **g, G** Chooses between floating point or exponential output, but the rules governing the choice are slightly more complicated Chooses between floating point or exponential output, depending on the magnitude of the exponent and the value specified for `<prec>` **r, a** Not supported (though the functionality is provided by the `!r` and `!a` conversion components in the replacement field) Designates conversion with `repr()` or `ascii()`, respectively **%** Converts a numeric argument to a percentage Inserts a literal `'%'` character into the output

Next, you'll see several examples illustrating these differences, as well as some of the added features of the Python `.format()` method presentation types. The first presentation type you'll see is `b`, which designates **binary integer conversion**:

>>>

```
>>> '{:b}'.format(257)
'100000001'
```

The modulo operator doesn't support binary conversion type at all:

>>>

```
>>> '%b' % 257
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: unsupported format character 'b' (0x62) at index 1
```

However, the modulo operator does allow **decimal integer conversion** with any of the `d`, `i`, and `u` types. Only the `d` presentation type indicates decimal integer conversion with the Python `.format()` method. The `i` and `u` presentation types aren't supported and aren't really necessary.

Next up is **single character conversion**. The modulo operator allows either an integer or single character value with the `c` conversion type:

>>>

```
>>> '%c' % 35
'#'
>>> '%c' % '#'
'#'
```

On the other hand, Python's `.format()` method requires that the value corresponding to the `c` presentation type be an integer:

>>>

```
>>> '{:c}'.format(35)
'#'
>>> '{:c}'.format('#')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'c' for object of type 'str'
```

When you try to pass a value of a different type, you'll get a `ValueError`.

For both the string modulo operator and Python's `.format()` method, the **`g` conversion type** chooses either floating-point or exponential output, depending on the magnitude of the exponent and the value specified for `<prec>`:

>>>

```
>>> '{:g}'.format(3.14159)
'3.14159'

>>> '{:g}'.format(-123456789.8765)
'-1.23457e+08'
```

The exact rules (https://docs.python.org/3.4/library/string.html#format-specification-mini-language) governing the choice are slightly more complicated with `.format()` than they are with the modulo operator. Generally, you can trust that the choice will make sense.

**`G`** is identical to `g` except for when the output is exponential, in which case the `'E'` will be in uppercase:

>>>

```
>>> '{:G}'.format(-123456789.8765)
'-1.23457E+08'
```

The result is the same as in the previous example, but this time with an uppercase `'E'`.

**Note:** There are a couple of other situations where you'll see a difference between the `g` and `G` presentations types.

Under some circumstances, a floating-point operation can result in a value that's essentially **infinite**. The string representation of such a number in Python is `'inf'`. It may also happen that a floating-point operation produces a value that can't be represented as a number. Python represents this with the string `'NaN'`, which stands for **Not a Number**.

When you pass these values to Python's `.format()` method, the `g` presentation type produces lowercase output, and `G` produces uppercase output:

>>>

```
>>> x = 1e300 * 1e300
>>> x
inf

>>> '{:g}'.format(x)
'inf'
>>> '{:g}'.format(x * 0)
'nan'

>>> '{:G}'.format(x)
'INF'
>>> '{:G}'.format(x * 0)
'NAN'
```

You'll see similar behavior with the `f` and `F` presentations types as well:

>>>

```
>>> '{:f}'.format(x)
'inf'
>>> '{:F}'.format(x)
'INF'

>>> '{:f}'.format(x * 0)
'nan'
>>> '{:F}'.format(x * 0)
'NAN'
```

For more information on floating-point representation, `inf`, and `NaN`, check out the Wikipedia page on IEEE 754-1985 (https://en.wikipedia.org/wiki/IEEE_754-1985).

The modulo operator supports **`r` and `a` conversion types** to force conversion by `repr()` and `ascii()`, respectively. Python's `.format()` method doesn't support `r` and `a` presentation types, but you can accomplish the same thing with the `!r` and `!a` conversion components in the replacement field.

Finally, you can use the `%` **conversion type** with the modulo operator to insert a literal `'%'` character into the output:

```
>>>
```

```
>>> '%f%%' % 65.0
'65.000000%'
```

You don't need anything special to insert a literal `'%'` character into the Python `.format()` method's output, so the `%` presentation type serves a different handy purpose for **percent output**. It multiplies the specified value by 100 and appends a percent sign:

```
>>>
```

```
>>> '{:%}'.format(0.65)
'65.000000%'
```

The remaining parts of `<format_spec>` indicate how the chosen presentation type is formatted, in much the same way as the string modulo operator's width and precision specifiers (https://realpython.com/python-input-output/#width-and-precision-specifiers) and conversion flags (https://realpython.com/python-input-output/#conversion-flags). These are described more fully in the following sections.

### The `<fill>` and `<align>` Subcomponents

`<fill>` and `<align>` control how formatted output is padded and positioned within the specified field width. These subcomponents only have meaning when the formatted field value doesn't occupy the entire field width, which can only happen if a minimum field width is specified with `<width>`. If `<width>` isn't specified, then `<fill>` and `<align>` are effectively ignored. You'll cover `<width>` later on in this tutorial.

Here are the possible values for the `<align>` subcomponent:

- `<`
- `>`
- `^`
- `=`

A value using the less than sign (`<`) indicates that the output is left-justified:

```
>>>
```

```
>>> '{0:<8s}'.format('foo')
'foo     '
>>> '{0:<8d}'.format(123)
'123     '
```

This behavior is the default for string values.

A value using the greater than sign (`>`) indicates that the output should be right-justified:

```
>>>
```

```
>>> '{0:>8s}'.format('foo')
'     foo'
>>> '{0:>8d}'.format(123)
'     123'
```

This behavior is the default for numeric values.

A value using a caret (`^`) indicates that the output should be centered in the output field:

```
>>>
```

```
>>> '{0:^8s}'.format('foo')
'  foo   '
>>> '{0:^8d}'.format(123)
'  123   '
```

Finally, you can also specify a value using the equals sign (`=`) for the `<align>` subcomponent. This only has meaning for numeric values, and only when a sign is included in the output.

When numeric output includes a sign, it's normally placed directly to the left of the first digit in the number, as shown above. If `<align>` is set to the equals sign (`=`), then the sign appears at the left edge of the output field, and padding is placed in between the sign and the number:

```
>>>
```

```
>>> '{0:+8d}'.format(123)
'    +123'
>>> '{0:=+8d}'.format(123)
'+    123'

>>> '{0:+8d}'.format(-123)
'    -123'
>>> '{0:=+8d}'.format(-123)
'-    123'
```

You'll cover the `<sign>` component in detail in the next section.

`<fill>` specifies how to fill in extra space when the formatted value doesn't completely fill the output width. It can be any character except for curly braces (`{}`). (If you really feel compelled to pad a field with curly braces, then you'll just have to find another way!)

Some examples of the use of `<fill>` are shown below:

```
>>>
```

```
>>> '{0:->8s}'.format('foo')
'-----foo'

>>> '{0:#<8d}'.format(123)
'123#####'

>>> '{0:*^8s}'.format('foo')
'**foo***'
```

If you specify a value for `<fill>`, then you should also include a value for `<align>` as well.

### The `<sign>` Subcomponent

You can control whether a **sign** appears in numeric output with the `<sign>` component. For example, in the following, the plus sign (`+`) specified in the `<format_spec>` indicates that the value should always be displayed with a leading sign:

>>>

```
>>> '{0:+6d}'.format(123)
'  +123'
>>> '{0:+6d}'.format(-123)
'  -123'
```

Here, you use the plus sign (`+`), so a sign will always be included for both positive and negative values. If you use the minus sign (`-`), then only negative numeric values will include a leading sign, and positive values won't:

>>>

```
>>> '{0:-6d}'.format(123)
'   123'
>>> '{0:-6d}'.format(-123)
'  -123'
```

When you use a single space (`' '`), it means a sign is included for negative values, and an ASCII space character for positive values:

>>>

```
>>> '{0:*> 6d}'.format(123)
'** 123'

>>> '{0:*>6d}'.format(123)
'***123'

>>> '{0:*> 6d}'.format(-123)
'**-123'
```

Since the space character is the default fill character, you'd only notice the effect of this if an alternate `<fill>` character is specified.

Lastly, recall from above that when you specify the equals sign (`=`) for `<align>` and you include a `<sign>` specifier, the padding goes between the sign and the value, rather than to the left of the sign.

### The `#` Subcomponent

When you specify a hash character (`#`) in `<format_spec>`, Python will select an alternate output form for certain presentation types. This is analogous to the `#` conversion flag for the string modulo operator. For binary, octal, and hexadecimal presentation types, the hash character (`#`) causes inclusion of an explicit base indicator to the left of the value:

```
>>>
```

```
>>> '{0:b}, {0:#b}'.format(16)
'10000, 0b10000'
>>> '{0:o}, {0:#o}'.format(16)
'20, 0o20'
>>> '{0:x}, {0:#x}'.format(16)
'10, 0x10'
```

As you can see, the base indicator can be `0b`, `0o`, or `0x`.

For floating-point or exponential presentation types, the hash character forces the output to contain a decimal point, even if the output consists of a whole number:

```
>>>
```

```
>>> '{0:.0f}, {0:#.0f}'.format(123)
'123, 123.'
>>> '{0:.0e}, {0:#.0e}'.format(123)
'1e+02, 1.e+02'
```

For any presentation type other than those shown above, the hash character (`#`) has no effect.

### The `0` Subcomponent

If output is smaller than the indicated field width and you specify the digit zero (`0`) in `<format_spec>`, then values will be padded on the left with zeros instead of ASCII space characters:

```
>>>
```

```
>>> '{0:05d}'.format(123)
'00123'
>>> '{0:08.1f}'.format(12.3)
'000012.3'
```

You'll typically use this for numeric values, as shown above. However, it works for string values as well:

```
>>>
```

```
>>> '{0:>06s}'.format('foo')
'000foo'
```

If you specify both `<fill>` and `<align>`, then `<fill>` overrides `0`:

```
>>>
```

```
>>> '{0:*>05d}'.format(123)
'**123'
```

`<fill>` and `0` essentially control the same thing, so there really isn't any need to specify both. In fact, `0` is really superfluous, and was probably included as a convenience for developers who are familiar with the string modulo operator's similar `0` conversion flag.

### The `<width>` Subcomponent

`<width>` specifies the minimum width of the output field:

>>>

```
>>> '{0:8s}'.format('foo')
'foo     '
>>> '{0:8d}'.format(123)
'     123'
```

Note that this is a **minimum field width**. Suppose you specify a value that's longer than the minimum:

>>>

```
>>> '{0:2s}'.format('foobar')
'foobar'
```

In this case, `<width>` is effectively ignored.

### The `<group>` Subcomponent

`<group>` allows you to include a **grouping separator character** in numeric output. For decimal and floating-point presentation types, `<group>` may be specified as either a comma character ( `,` ) or an underscore character ( `_` ). That character then separates each group of three digits in the output:

>>>

```
>>> '{0:,d}'.format(1234567)
'1,234,567'
>>> '{0:_d}'.format(1234567)
'1_234_567'

>>> '{0:,.2f}'.format(1234567.89)
'1,234,567.89'
>>> '{0:_.2f}'.format(1234567.89)
'1_234_567.89'
```

A `<group>` value using an underscore ( `_` ) may also be specified with the binary, octal, and hexadecimal presentation types. In that case, each group of four digits is separated by an underscore character in the output:

>>>

```
>>> '{0:_b}'.format(0b111010100001)
'1110_1010_0001'
>>> '{0:#_b}'.format(0b111010100001)
'0b1110_1010_0001'

>>> '{0:_x}'.format(0xae123fcc8ab2)
'ae12_3fcc_8ab2'
>>> '{0:#_x}'.format(0xae123fcc8ab2)
'0xae12_3fcc_8ab2'
```

If you try to specify `<group>` with any presentation type other than those listed above, then your code will raise an exception.

#### The `.<prec>` Subcomponent

`.<prec>` specifies the number of digits after the decimal point for floating point presentation types:

```
>>>
```

```
>>> '{0:8.2f}'.format(1234.5678)
' 1234.57'
>>> '{0:8.4f}'.format(1.23)
'  1.2300'

>>> '{0:8.2e}'.format(1234.5678)
'1.23e+03'
>>> '{0:8.4e}'.format(1.23)
'1.2300e+00'
```

For string types, `.<prec>` specifies the maximum width of the converted output:

```
>>>
```

```
>>> '{:.4s}'.format('foobar')
'foob'
```

If the output would be longer than the value specified, then it will be truncated.



## The String `.format()` Method: Nested Replacement Fields

Recall that you can specify (https://realpython.com/python-input-output/#width-and-precision-specifiers) either `<width>` or `<prec>` by an asterisk with the string modulo operator:

```
>>>
```

```
>>> w = 10
>>> p = 2
>>> '%*.*f' % (w, p, 123.456)  # Width is 10, precision is 2
'    123.46'
```

The associated values are then taken from the argument list. This allows `<width>` and `<prec>` to be evaluated dynamically at run-time, as shown in the example above. Python's `.format()` method provides similar capability using **nested replacement fields**.

Inside a replacement field, you can specify a nested set of curly braces ( `{}` ) that contains a name or number referring to one of the method's positional or keyword arguments. That portion of the replacement field will then be evaluated at run-time and replaced using the corresponding argument. You can accomplish the same effect as the above string modulo operator example with nested replacement fields:

>>>

```
>>> w = 10
>>> p = 2
>>> '{2:{0}.{1}f}'.format(w, p, 123.456)
'    123.46'
```

Here, the `<name>` component of the replacement field is `2` , which indicates the third positional parameter whose value is `123.456` . This is the value to be formatted. The nested replacement fields `{0}` and `{1}` correspond to the first and second positional parameters, `w` and `p` . These occupy the `<width>` and `<prec>` locations in `<format_spec>` and allow field width and precision to be evaluated at run-time.

You can use **keyword arguments with nested replacement fields** as well. This example is functionally equivalent to the previous one:

>>>

```
>>> w = 10
>>> p = 2
>>> '{val:{wid}.{pr}f}'.format(wid=w, pr=p, val=123.456)
'    123.46'
```

In either case, the values of `w` and `p` are evaluated at run-time and used to modify the `<format_spec>` . The result is effectively the same as this:

>>>

```
>>> '{0:10.2f}'.format(123.456)
'    123.46'
```

The string modulo operator only allows `<width>` and `<prec>` to be evaluated at run-time in this way. By contrast, with Python's `.format()` method you can specify any portion of `<format_spec>` using nested replacement fields.

In the following example, the presentation type `<type>` is specified by a nested replacement field and determined dynamically:

>>>

```
>>> bin(10), oct(10), hex(10)
('0b1010', '0o12', '0xa')
>>> for t in ('b', 'o', 'x'):
...     print('{0:#{type}}'.format(10, type=t))
...
0b1010
0o12
0xa
```

Here, the grouping character `<group>` is nested:

```
>>>
```

```
>>> '{0:{grp}d}'.format(123456789, grp='_')
'123_456_789'
>>> '{0:{grp}d}'.format(123456789, grp=',')
'123,456,789'
```

Whew! That was an adventure. The specification of the template string is virtually a language unto itself!

As you can see, string formatting can be very finely tuned when you use Python's `.format()` method. Next, you'll see one more technique for string and output formatting that affords all the advantages of `.format()`, but with more direct syntax.

## The Python Formatted String Literal (f-String)

In version 3.6, a new Python string formatting syntax was introduced, called the **formatted string literal.** These are also informally called **f-strings**, a term that was initially coined in PEP 498 (https://www.python.org/dev/peps/pep-0498), where they were first proposed.



### f-String Syntax

An f-string (https://realpython.com/courses/python-3-f-strings-improved-string-formatting-syntax/) looks very much like a typical Python string except that it's prepended by the character `f`:

```
>>>
```

```
>>> f'foo bar baz'
'foo bar baz'
```

You can also use an uppercase `F`:

```
>>>
```

```
>>> s = F'qux quux'
>>> s
'qux quux'
```

The effect is exactly the same. Just like with any other type of string, you can use single, double, or triple quotes to define an f-string:

```
>>>
```

```
>>> f'foo'
'foo'
>>> f"bar"
'bar'
>>> f'''baz'''
'baz'
```

The magic of f-strings is that you can embed Python expressions directly inside them. Any portion of an f-string that's enclosed in curly braces ( `{}` ) is treated as an **expression**. The expression is evaluated and converted to string representation, and the result is interpolated into the original string in that location:

>>>

```
>>> s = 'bar'
>>> print(f'foo.{s}.baz')
foo.bar.baz
```

The interpreter treats the remainder of the f-string—anything not inside curly braces—just as it would an ordinary string. For example, escape sequences are processed as expected:

>>>

```
>>> s = 'bar'
>>> print(f'foo\n{s}\nbaz')
foo
bar
baz
```

Here's the example from earlier using an f-string:

>>>

```
>>> quantity = 6
>>> item = 'bananas'
>>> price = 1.74
>>> print(f'{quantity} {item} cost ${price}')
6 bananas cost $1.74
```

This is equivalent to the following:

>>>

```
>>> quantity = 6
>>> item = 'bananas'
>>> price = 1.74
>>> print('{0} {1} cost ${2}'.format(quantity, item, price))
6 bananas cost $1.74
```

Expressions embedded in f-strings can be almost arbitrarily complex. The examples below show some of the possibilities:

- Variables:

  >>>

  ```
  >>> quantity, item, price = 6, 'bananas', 1.74
  >>> f'{quantity} {item} cost ${price}'
  '6 bananas cost $1.74'
  ```

- Arithmetic expressions:

```
>>>
```

```
>>> quantity, item, price = 6, 'bananas', 1.74
>>> print(f'Price per item is ${price/quantity}')
Price per item is $0.29

>>> x = 6
>>> print(f'{x} cubed is {x**3}')
6 cubed is 216
```

- Objects of composite types:

```
>>>
```

```
>>> a = ['foo', 'bar', 'baz']
>>> d = {'foo': 1, 'bar': 2}

>>> print(f'a = {a} | d = {d}')
a = ['foo', 'bar', 'baz'] | d = {'foo': 1, 'bar': 2}
```

- Indexing, slicing, and key references:

```
>>>
```

```
>>> a = ['foo', 'bar', 'baz']
>>> d = {'foo': 1, 'bar': 2}

>>> print(f'First item in list a = {a[0]}')
First item in list a = foo

>>> print(f'Last two items in list a = {a[-2:]}')
Last two items in list a = ['bar', 'baz']

>>> print(f'List a reversed = {a[::-1]}')
List a reversed = ['baz', 'bar', 'foo']

>>> print(f"Dict value for key 'bar' is {d['bar']}")
Dict value for key 'bar' is 2
```

- Function and method calls:

```
>>>
```

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> print(f'List a has {len(a)} items')
List a has 5 items

>>> s = 'foobar'
>>> f'--- {s.upper()} ---'
'--- FOOBAR ---'

>>> d = {'foo': 1, 'bar': 2}
>>> print(f"Dict value for key 'bar' is {d.get('bar')}")
Dict value for key 'bar' is 2
```

- Conditional expressions:

```
>>>
```

```
>>> x = 3
>>> y = 7
>>> print(f'The larger of {x} and {y} is {x if x > y else y}')
The larger of 3 and 7 is 7

>>> age = 13
>>> f'I am {"a minor" if age < 18 else "an adult"}.'
'I am a minor.'
```

- Object attributes:

  >>>

  ```
  >>> z = 3+5j
  >>> z
  (3+5j)

  >>> print(f'real = {z.real}, imag = {z.imag}')
  real = 3.0, imag = 5.0
  ```

To include a literal curly brace in an f-string, escape it by doubling it, the same as you would in a template string for Python's `.format()` method:

>>>

```
>>> z = 'foobar'
>>> f'{{ {z[::-1]} }}'
'{ raboof }'
```

You may prefix an f-string with `'r'` or `'R'` to indicate that it is a **raw f-string**. In that case, backslash sequences are left intact, just like with an ordinary string:

>>>

```
>>> z = 'bar'
>>> print(f'foo\n{z}\nbaz')
foo
bar
baz
>>> print(rf'foo\n{z}\nbaz')
foo\nbar\nbaz
>>> print(fr'foo\n{z}\nbaz')
foo\nbar\nbaz
```

Note that you can specify the `'r'` first and then the `'f'`, or vice-versa.



## f-String Expression Limitations

There are a few minor **restrictions** on f-string expression. These aren't too limiting, but you should know what they are. First, an f-string expression **can't be empty**:

>>>

```
>>> f'foo{}bar'
  File "<stdin>", line 1
SyntaxError: f-string: empty expression not allowed
```

It isn't obvious why you'd want to do this. But if you feel compelled to try, then just know that it won't work.

Additionally, an f-string expression **can't contain a backslash ( `\` ) character**. Among other things, that means you can't use a backslash escape sequence in an f-string expression:

```
>>>
```

```
>>> print(f'foo{\n}bar')
  File "<stdin>", line 1
SyntaxError: f-string expression part cannot include a backslash
>>> print(f'foo{\'}bar')
  File "<stdin>", line 1
SyntaxError: f-string expression part cannot include a backslash
```

You can get around this limitation by creating a temporary variable that contains the escape sequence you want to insert:

```
>>>
```

```
>>> nl = '\n'
>>> print(f'foo{nl}bar')
foo
bar
>>> quote = '\''
>>> print(f'foo{quote}bar')
foo'bar
```

Lastly, an expression in an f-string that is triple-quoted **can't contain comments**:

```
>>>
```

```
>>> z = 'bar'

>>> print(f'''foo{
... z
... }baz''')
foobarbaz

>>> print(f'''foo{
... z    # Comment
... }''')
  File "<stdin>", line 3
SyntaxError: f-string expression part cannot include '#'
```

Note, however, that the multiline f-string may contain embedded newlines.

## f-String Formatting

Like Python's `.format()` method, f-strings support extensive **modifiers** that control the final appearance of the output string. There's more good news, too. If you've mastered the Python `.format()` method, then **you already know how to use Python to format f-strings!**

Expressions in f-strings can be modified by a `<conversion>` or `<format_spec>`, just like replacement fields used in the `.format()` template. The syntax is identical. In fact, in both cases Python will format the replacement field using the same internal function. In the following example, `!r` is specified as a `<conversion>` component in the `.format()` template string:

>>>

```
>>> s = 'foo'
>>> '{0!r}'.format(s)
"'foo'"
```

This forces conversion to be performed by `repr()`. You can get essentially the same code using an f-string instead:

>>>

```
>>> s = 'foo'
>>> f'{s!r}'
"'foo'"
```

All the `<format_spec>` components that work with `.format()` also work with f-strings:

>>>

```
>>> n = 123
>>> '{:=+8}'.format(n)
'+    123'
>>> f'{n:=+8}'
'+    123'

>>> s = 'foo'
>>> '{0:*^8}'.format(s)
'**foo***'
>>> f'{s:*^8}'
'**foo***'

>>> n = 0b111010100001
>>> '{0:#_b}'.format(n)
'0b1110_1010_0001'
>>> f'{n:#_b}'
'0b1110_1010_0001'
```

**Nesting** works as well, like nested replacement fields with Python's `.format()` method:

>>>

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> w = 4
>>> f'{len(a):0{w}d}'
'0005'

>>> n = 123456789
>>> sep = '_'
>>> f'{(n * n):{sep}d}'
'15_241_578_750_190_521'
```

This means formatting items can be evaluated at run-time.

f-strings and Python's `.format()` method are, more or less, two different ways of doing the same thing, with f-strings being a more concise shorthand. The following expressions are essentially the same:

```
f'{<expr>!<conversion>:<format_spec>}'
'{0!<conversion>:<format_spec>}'.format(<expr>)
```

If you want to explore f-strings further, then check out Python 3's f-Strings: An Improved String Formatting Syntax (Course) (https://realpython.com/courses/python-3-f-strings-improved-string-formatting-syntax).



## Conclusion

In this tutorial, you mastered two additional techniques that you can use in Python to format string data. You should now have all the tools you need to prepare string data for output or display!

You might be wondering which Python formatting technique you should use. Under what circumstances would you choose `.format()` over the **f-string**? See Python String Formatting Best Practices (https://realpython.com/python-string-formatting/#which-string-formatting-method-should-you-use) for some considerations to take into account.

In the next tutorial, you're going to learn more about **functions** in Python. Throughout this tutorial series, you've seen many examples of Python's **built-in functions**. In Python, as in most programming languages, you can define your own custom **user-defined functions** as well. If you can't wait to learn how then continue on to the next tutorial!

« Basic Input, Output, and String Formatting in Python (https://realpython.com/python-input-output/)
Defining Your Own Python Function » (https://realpython.com/defining-your-own-python-function/)

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Formatting Python Strings** (https://realpython.com/courses/formatting-python-strings/)