Python f-string tips & cheat sheets

Trey Hunner

13 min. read • Python 3.8—3.11 • April 12, 2022 Share

Tags Strings

This is **a cheat sheet to string formatting in Python** with explanations of each "cheat".

Feel free to jump straight to the cheat sheets if that's what you're here for.

Not sure what string formatting is? See string concatenation and string interpolation in Python.

What are we talking about?

Python's string formatting syntax allows us to inject objects (often other strings) into our strings.

```
>>> name = "Trey"
>>> print(f"My name is {name}. What's your name?")
My name is Trey. What's your name?
```

We can even embed expressions:

```
>>> name = "Trey"
>>> print(f"{name}, which starts with {name[0]}")
Trey, which starts with T
```

But Python's string formatting syntax also allows us to control the formatting of each of these string components.

There is **a lot of complexity** in Python's string formatting syntax. If you're just for quick answers, skip to the cheat sheets section.

Definitions

Let's start with some definitions. The Python documentation around string formatting uses a lot of terminology that you probably haven't heard before.

Replacement field: each of the curly brace components (between `{` and `}`) in an f-string is called a replacement field

Conversion field: this "converts" the object within a replacement field using a specific converter and it's preceded by an exclamation mark (`!`)

Format specification: this controls the formatting of a replacement field object (basically *how* it is converted to a string) and it's preceded by a colon (`:`)

Self-documenting expressions: replacement fields designed for print-debugging, which are followed by an equals sign (`=`)

Modifier: this is an unofficial term I'm using to refer to the combination of conversion fields, format specifications, and self-documenting expressions. There isn't an official umbrella term for all three ideas, so "modifier" is the one I'm using.

These terms probably don't make much sense yet, but examples will help.

Example f-strings

Let's start with these variables:

```
>>> full_name = "Trey Hunner"
>>> costs = [1.10, 0.30, 0.40, 2]
```

Table of Contents

Mark Read

A Python Tip Every Week ↑

```
>>> print(f"My name is {full_name}.")
My name is Trey Hunner.
```

This f-string has **two replacement fields**:

```
>>> print(f"My name is {full_name} which is {len(full_name)} characters long.")
My name is Trey Hunner which is 11 characters long.
```

This f-string has one replacement field which uses a **format specification** (note the `:`):

```
>>> print(f"The total cost is ${sum(costs):.2f}")
The total cost is $3.80
```

This f-string has two replacement fields and one of them uses a **format specification** (note the `:` in the second replacement field):

```
>>> n_count = full_name.count("n")
>>> print(f"My name, {full_name}, is {n_count/len(full_name):.0%} n's.")
My name, Trey Hunner, is 18% n's.
```

This f-string has one replacement field which uses a **conversion field** (note the `!`):

```
>>> print(f"The variable 'full_name' contains {full_name!r}.")
The variable 'full_name' contains 'Trey Hunner'.
```

This f-string has two **self-documenting expressions** (note the `=` suffix in each replacement field):

```
>>> print(f"Variables: {full_name=}, {costs=}")
Variables: full_name='Trey Hunner', costs=[1.1, 0.3, 0.4, 2]
```

How do these different **f-string modifiers** work and what can you do with them?

String formatting works differently on different types of objects. Let's look at some common objects you might want to use string formatting on.

Formatting numbers

It's *very* common to see format specifications used with numbers in Python.

Below are the most useful string format specifications for numbers. You can test some of these out from your browser here.

N digits after the decimal point (fixed-point notation)

The `.Nf` format specifier (where `N` is a whole number) will format a number to show `N` digits after the decimal point. This is called **fixed-point notation** (yet another term you don't need to remember).

Given these two numbers:

```
>>> from math import pi
>>> n = 4
```

We can show one digit after the decimal point with `.1f`:

```
>>> print(f"One digit: {n:.1f} and {pi:.1f}")
One digit: 4.0 and 3.1
```

Two digits with `.2f`:

```
>>> print(f"Two digits: {n:.2f} and {pi:.2f}")
Two digits: 4.00 and 3.14
```

Or zero digits with `.0f`:

This fixed-point notation format specification rounds the number the same way Python's `round` function would.

Zero-padding

The `0Nd` format specifier (where `N` is a whole number) will format a number to be `N` digits long (by zero-padding it on the left-hand side).

Here's a list of numbers formatted to be 2 digits long (with `02d`):

```
>>> tracks = [(1, "Harlem"), (3, "Grandma's Hands"), (10, "Moanin' and Groanin'")]
>>> for n, title in tracks:
...     print(f"{n:02d}. {title}")
...
01. Harlem
03. Grandma's Hands
10. Moanin' and Groanin'
```

Zero-padding can be helpful if you're trying to line numbers up (in a table column for example).

Important note: `0N` is a shorthand for `02d` on numbers, but it does something different on strings (while `0Nd` raises an exception on strings):

```
>>> n = 3
>>> print(f"{n:04}")
0003
>>> n = 'Hi'
>>> print(f"{n:04}")
Hi00
>>> print(f"{n:04d}")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'd' for object of type 'str'
```

So I prefer `0Nd` over `0N` because it's a bit more explicit.

That trailing `d` in `0Nd` stands for *decimal integer* and only works on integer types.

Space-padding

The `Nd` format specifier (where `N` is a whole number) will pad an integer with space characters until it's `N` digits long.

Here's a list of numbers formatted to be space-padded to 2 characters (with `2d`):

```
>>> tracks = [(1, "Harlem"), (3, "Grandma's Hands"), (10, "Moanin' and Groanin'")]
>>> for n, title in tracks:
...     print(f"{n:2d}. {title}")
...
 1. Harlem
 3. Grandma's Hands
10. Moanin' and Groanin'
```

Space-padding can be helpful if you're lining up numbers in a fixed-width setting (in a command-line program for example).

If you prefer, you can add a space before the `Nd` specifier (so it'll look more like its sibling, the `0Nd` modifier):

```
>>> n = 3
>>> print(f"{n: 4d}")
   3
```

If you'd like to space-pad floating point numbers, check out `>N` in the section on strings below.

The `.N%` format specifier (where `N` is a whole number) formats a number as a percentage. Specifically `.N%` will multiply a number by `100`, format it to have `N` digits after the decimal sign and put a `%` sign after it.

Here's the percentage specifier used with `0`, `1`, and `2` digits after the decimal point:

```
>>> purple = 675
>>> total = 1000
>>> print(f"They were {purple/total:.0%} purple")
They were 68% purple
>>> print(f"They were {purple/total:.1%} purple")
They were 67.5% purple
>>> print(f"They were {purple/total:.2%} purple")
They were 67.50% purple
```

Thousands separators

The `,` format specifier formats a number to include commas as a thousands separator:

```
>>> population = 9677225658
>>> print(f"Earth's population peaked at {population:,}.")
Earth's population peaked at 9,677,225,658.
```

The `_` format specifier formats a number to use underscore as a thousands separator:

```
>>> print(f"Earth's population peaked at {population:_}.")
Earth's population peaked at 9_677_225_658.
```

And the `n` format specifier formats a number in a locale-aware way (using period, comma, or another appropriate thousands separator based on the locale):

```
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, "en_IN.utf-8")
'en_IN.utf-8'
>>> print(f"The population peaked at {population:n}.")
The population peaked at 9,67,72,25,658.
>>> locale.setlocale(locale.LC_NUMERIC, "en_US.utf-8")
'en_US.utf-8'
>>> print(f"The population peaked at {population:n}.")
The population peaked at 9,677,225,658.
```

Hexadecimal, binary, and more

Python also has format specifiers for representing numbers in binary or hexadecimal.

The `b` format specifier represents a number in binary notation:

```
>>> n = 140503780278544
>>> print(f"In binary: {n:b}")
In binary: 1111111110010011001010111101001110000100010000
```

And the `x` format specifier represents a number in hexadecimal:

```
>>> n = 140503780278544
>>> print(f"In hex: {n:x}")
In hex: 7fc995e9c110
```

You can also put a `#` before these to add a `0b` or `0x` prefix:

```
>>> print(f"In hex: {n:#x}")
In hex: 0x7fc995e9c110
```

And uppercasing the `X` customizes the hexadecimal notation to use uppercase letters:

```
>>> print(f"In hex: {n:#X}")
In hex: 0X7FC995E9C110
```

Multiple format specifiers can often be combined.

For example if you wanted a number represented as hexadecimal with padding to make it 2-digits long, you could combine `02d` with `x` to make `02x`:

```
>>> bits = 13
>>> print(f"{bits:02x}")
0d
```

The `d` stands for *decimal*, while `x` stands for *hexadecimal* and `b` stands for *binary*. So that trailing letter is just specifying the number system we'd like to use for our integer.

Adding a `#` in front to add the `0x` prefix also works:

```
>>> print(f"{bits:#02x}")
0xd
```

We could also combine the `_` thousands separator with the `b` binary format (the separator is used for groups of 4 in binary though):

```
>>> bits = 560
>>> print(f"{bits:_b}")
10_0011_0000
```

Or we could combine the `,` thousands separator with the `.Nf` format:

```
>>> print(f"${amount:,.2f}")
$4,780.00
```

We've actually already seen combining types: `.N%` is just a variation of the `.Nf` type that works on floating point numbers and integers.

```
>>> n = .48
>>> print(f"{n*100:.2f}")
48.00
>>> print(f"{n:.2%}")
48.00%
```

Formatting strings

The format specifiers for strings are all about alignment. I use these pretty rarely (mostly when lining-up data in a command-line interface).

The `>N` format specifier (where `N` is a whole number) right-aligns a string to `N` characters. Specifically, this formats the resulting substring to be `N` characters long with spaces padding the left-hand side of the string.

Here the second replacement field is formatted to 25 characters long and right-aligned:

```
>>> tracks = [(1, "Harlem", "3:23"), (3, "Grandma's Hands", "2:00"), (10, "Moanin'
and Groanin'", "2:59")]
>>> for n, title, length in tracks:
...     print(f"{n:02}. {title:>25} {length}")
...
01.                    Harlem 3:23
03.           Grandma's Hands 2:00
10.      Moanin' and Groanin' 2:59
```

Likewise, the `<N` format specifier left-aligns a string to `N` characters: Here the second replacement field is formatted to 25 characters long and left-aligned:

```
...
01. Harlem                    3:23
03. Grandma's Hands           2:00
10. Moanin' and Groanin'      2:59
```

The `^N` format specifier center-aligns a string to `N` characters:

```
>>> for n, title, length in tracks:
...     print(f"{n:02}. {title:^25} {length}")
...
01.           Harlem           3:23
03.       Grandma's Hands      2:00
10.    Moanin' and Groanin'    2:59
```

By default, alignment uses a space character. Putting a character just before the `<` or `>` signs will customize the alignment character used. For example the middle replacement field here `.`-pads our title name and the last replacement field here `0`-pads our string:

```
>>> for n, title, length in tracks:
...     print(f"{n:02}. {title:.<25} {length:0>5}")
...
01. Harlem..................... 03:23
03. Grandma's Hands.......... 02:00
10. Moanin' and Groanin'..... 02:59
```

Formatting `datetime` objects

It's nifty that we can customize the format of strings and numbers within an f-string. But what about other types? What else supports format specifiers?

Python's `datetime.datetime` class (along with `datetime.date` and `datetime.time`) supports string formatting which uses the same syntax as the `strftime` method on these objects.

```
>>> import datetime
>>> a_long_long_time_ago = datetime.date(1971, 5, 26)
>>> print(f"It was {a_long_long_time_ago:%B %d, %Y}.")
It was May 26, 1971.
```

I often prefer using string formatting for `datetime.date` and `datetime.datetime` objects rather than calling their `strftime` method.

Python's `ipaddress` module also supports custom string formatting for `IPv4Address` and `IPv6Address` objects.

All the objects built-in to Python that support custom string formatting within Python are:

1. Numbers
2. Strings
3. `datetime` and `date` objects
4. `IPv4Address` and `IPv6Address` objects

Third-party libraries can also add their own custom string formatting support by adding a `__format__` method to their objects.

Forcing a programmer-readable representation

Python's format strings also support a conversion field, which can force a replacement field (the thing between `{` and `}`) to use a specific string representation.

By default, string formatting uses the human-readable representation of an object (`str` instead of `repr`).

Most Python objects have the same `str` and `repr` representations, so this distinction often isn't noticeable. But some objects have different representations for `str` and `repr`, such as strings and `datetime.date` objects:

```
>>> name = "Trey"
>>> print("str:", str(nye), str(name))
str: 1999-12-31 Trey
>>> print("repr:", repr(nye), repr(name))
repr: datetime.date(1999, 12, 31) 'Trey'
>>> print(f"f-string: {nye} {name}")
f-string: 1999-12-31 Trey
```

Python's format strings use the human-readable representation by default, but you can change which representation is used by suffixing your replacement field with `!r`, `!a`, or `!s`:

```
>>> sparkles = "\u2728"
>>> print(f"!s {nye!s} {name!s} {sparkles!s} -- like str (the default)")
!s 1999-12-31 Trey ✨ -- like str (the default)
>>> print(f"!r {nye!r} {name!r} {sparkles!r} -- like repr")
!r datetime.date(1999, 12, 31) 'Trey' '✨' -- like repr
>>> print(f"!a {nye!a} {name!a} {sparkles!a} -- like ascii (escapes unicode)")
!a datetime.date(1999, 12, 31) 'Trey' '\u2728' -- like ascii (escapes unicode)
```

The `!r` and `!a` conversion fields are especially helpful for implementing a class's `__repr__` method.

This class has a string representation that doesn't look quite like code (you can't run `Person(Trey Hunner)` as code).

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return f"Person({self.name})"
...
>>> Person("Trey Hunner")
Person(Trey Hunner)
```

Whereas this class has a string representation that represents valid code (you *could* run `Person('Trey Hunner')`):

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return f"Person({self.name!r})"
...
>>> Person("Trey Hunner")
Person('Trey Hunner')
```

Try running that `!r` conversion example yourself.

Self-documenting expressions & debugging

Format strings also include a very neat feature that was added in Python 3.8: self-documenting expressions.

You can put suffix your replacement fields with an `=` sign to make a self-documenting expression:

```
>>> print(f"FYI: {name=}, {color=}")
FYI: name='Trey', color='purple'
```

With self-documenting expressions, the resulting replacement string includes the original replacement field expression as well as the result with an `=` sign separating them.

Note that the `repr` representation is used by default. You can change that by adding an explicit `!s` (or even `!a`).

A Python Tip Every Week ↑

Though I usually prefer `repr` format when using `=` anyway because I'm usually using it for debugging purposes.

As usual, you can use any expression you'd like in your replacement field and the whole expression will be shown:

```
>>> costs = [1.10, 0.30, 0.40, 2]
>>> print(f"{sum(costs)=}")
sum(costs)=3.8000000000000003
```

You can even add a format specifier to format the expression result:

```
>>> print(f"{sum(costs)=:.2f}")
sum(costs)=3.80
```

Note that the `=` should be at the end of the replacement field but it must be before the format specifier or conversion field (if there are any).

You can also put spaces around the `=` sign to add spaces around the `=` sign in the resulting string:

```
>>> print(f"{costs = }")
costs = [1.1, 0.3, 0.4, 2]
>>> print(f"{sum(costs) = :.2f}")
sum(costs) = 3.80
```

Unfortunately this isn't very well documented.

Python's various format specifiers are documented in an odd and very dense format specification mini-language section within the `string` module documentation (the `string` module, not the `str` class). But using `=` to make a self-documenting expression isn't mentioned anywhere on that page. You can see some examples in What's new in Python 3.8, but the only other mention of `=` is in the Lexical Analysis page, which is a very dense page (with few code examples) that describes Python's syntax at a very low level.

The `=` syntax also doesn't seem to have an official name. I've been calling it a **self-documenting expression** but that term is only used a single time in the documentation within a header. It sounded pretty good to me, so I'm using it as if it's an official term. 😅

Cheat sheets

Examples are great, but it's hard to hold many examples in your head at once.

You can use the below tables as cheat sheets to help you during your string formatting adventures. Or use the fstring.help/cheat to quickly find these reference tables.

Floating point numbers *and* integers

Here's a summary of the various options within the format specification field for generic number formatting.

These string formatting techniques work an all numbers (both `int` and `float`):

Fill	Width	Grouping	Precision	Type	All Together	Example Output
			.2	f	{num:.2f}	'4125.60'
		,	.2	f	{num:,.2f}	'4,125.60'
0	8		.2	f	{num:08.2f}	'04125.60'
	8		.2	f	{num: 8.2f}	' 4125.60'

			`.0`	`%`	`{num:.0%}`	`'50%'`

There's also `g`, `G`, `e`, `E`, `n`, and `F` types for floating point numbers and `c`, `o`, and `n` types for integers which I haven't shown but which are documented in the format specification mini-language documentation.

Integers

These format specifications work only on integers (`int`):

Alt	Fill	Width	Grouping	Type	All Together	Example Output
	`0`	`2`		`d`	`{number:02d}`	`'09'`
		`3`		`d`	`{number: 3d}`	`' 9'`
			`,`		`{number:,}`	`'9'`
				`b`	`{number:b}`	`'1001'`
				`x`	`{number:x}`	`'9'`
				`X`	`{number:X}`	`'9'`
`#`				`x`	`{number:#x}`	`'0x9'`
`#`	`0`	`2`		`x`	`{number:#02x}`	`'0x9'`
	`0`	`8`	`_`	`b`	`{number:08_b}`	`'000_1001'`

An empty type is synonymous with `d` for integers.

Strings

These format specifications work on strings (`str`) and most other types (any type that doesn't specify its own custom format specifications):

Fill Char	Align	Width	All Together	Example Output
	`>`	`15`	`{string:>6}`	`' Trey'`
	`<`	`15`	`{string:<6}`	`'Trey '`
	`^`	`15`	`{string:^6}`	`' Trey '`
`0`	`>`	`8`	`{string:0>8}`	`'0000Trey'`

All objects

All the above options were about the format specifications (the part after a `:` within a format string replacement field). Format specifications are object-specific (`str`, `int`, `float`, `datetime` all support a different syntax).

The below modifiers are special syntaxes which are supported by all object types. For clarity, I've added some format specifications below to show how to mix and match these syntaxes with the `:` syntax above.

These conversion field and self-documenting syntaxes are supported by all objects:

	!s		{expression!r}	'Hi! ✨'
	!r		{expression!r}	"'Hi! ✨'"
	!a		{expression!a}	"'Hi! \\u2728'"
	!r	<10	{expression!r:<10}	"'Hi! ✨' "
=			{expression=}	"name='Trey'"
=			{expression = }	"name = 'Trey'"
=	!s		{expression=!s}	"name=Trey"
=		.2f	{expression=:.2f}	'len(name)=4.00'

An empty conversion field is synonymous with `!s`, unless a self-documenting expression is used. When a self-documenting expression is used, an empty conversion field uses `!r`.

Summary

String formatting can be simple and it can be complex.

When using string formatting, most of the time all you'll need to do is embed your variable or expression within the replacement field on an f-string:

```
>>> description = "fairly simple"
>>> print(f"This example is {description}.")
This example is fairly simple.
```

But sometimes you'll find yourself wishing for more control over the format of your strings.

You may need to control how many digits show up after the decimal point. You might need to format a number as a percentage.

```
>>> ratio = 1/3
>>> print(f"{ratio:.2f}")
0.33
>>> print(f"{ratio:.0%}")
33%
```

Or you might want to control the format of your `datetime` objects.

```
>>> from datetime import datetime
>>> print(f"Happy {datetime.now():%A}!")
Happy Friday!
```

Whatever your need may be, it's worth looking up the magic incantation that will convince your f-string to show the output you need.

And remember that learning happens from *doing*, not reading. Try copy-pasting some of these code samples and playing around. Or try playing with [this string formatting code with datetime and self-documenting expressions].

Happy string formatting!