Cheatography

Python F-Strings Number Formatting Cheat Sheet

by Brian Allan (BrianAllan) via cheatography.com/133708/cs/27382/

F-String Template

```
f" text {replacement_field} text ... "
```

- Inside the quotes the f-string consists of two kinds of parts: (1) regular string literals, i.e., **text**, and (2) **replacement fields** containing Python expressions for evaluation along with formatting control.
- Double quotes are used here but single or triple quotes could also be used.
- F-strings may consist of just a replacement field:
  `f"{r epl ace men t_f iel d}"`

Replacement Field

```
{f-expression = !conversion:format_specifier}
```

- A replacement field is signaled by a pair of curly braces: `{ }`
- A replacement field consists of a Python expression for evaluation with optional debugging mode (=), type conversion (!), and format specification (:).
- Substituting into the f-string template:

```
f" text {f-exp ression = !conve rsi on: for mat - _sp eci fier} text ..."
```

Format Specifier

```
:fill align sign # 0 width sep .precision type
```

Brief Summary of the Format Specification Mini-Language

`fill` : the padding character
`align` : alignment of text within the space
`sign` : how + and - are used preceding numbers
`#` : alternate presentation format for some number types
`0` : sign-aware, zero-padding on numbers
`width` : the minimum total field width
`sep` : the separator character for numbers (',' or '_')
`.precision` : how many digits displayed for floats; maximum field width for strings
`type` : the type of presentation to use based on data type

Note: `sign`, `#`, `0`, `sep`, `precision`, and `type` are of particular interest for **number formatting**, the focus of this cheatsheet. For a general treatment of F-Strings, see my cheatsheet **Python F-Strings Basics**.

Format Specifier: Options

fill	align	sign	# 0	width	sep	.prec	type
char	<	+		digit(s)	_	digit(s)	*string*: `s`
	>	-			,		*number*: `n`
	^	''					*integer*: `d`, `b`, `o`, `x`, `X`, `c`
	=						*float*: `e`, `E`, `f`, `F`, `g`, `G`, `%`

Sign and Separator Options

Sign	
+	a sign is used for both positive and negative numbers
-	a sign is used only for negative numbers (default)
space	leading space for positive numbers and minus sign for negative
Separator	
,	comma. decimal integers and floats: thousands separator
_	underscore. decimal integers and floats: thousands separator; b, o, x, and X types: separates every four digits

Examples: Sign and Separator

# sign (+): a sign is attached to both positive and negative numbers	
`f"{3:+} or {-3:+} "`	'+3 or -3'
# sign (-): a sign is only used for negative numbers	
`f"{3:-} or {-3:-} "`	'3 or -3'
# no sign is the same as sign (-)	
`f"{3} or {-3}"`	'3 or -3'
# sign (space): a leading space for positive numbers and a minus sign for negative	
`f"{3: } or {-3: }"`	' 3 or -3'
# sign (+), separator (,)	
`f"{1 000 0:+ ,}"`	'+10,000'
# fill (?), align (<), sign (+), width (11), sep (,)	
`f"{1 000 0:? <+1 1,} "`	'+10,000????'

Sign Aware Padding

Using the **= alignment option** forces the padding determined by the fill and width specifications to be placed after the sign (if any) but before the digits. This is useful for number representations like '+000042'. This alignment option is only available for numeric types.

The **0-option** produces sign aware zero-padding just like the = alignment option with a fill of '0'. It is produced by preceding the field width specification with '0'.

```
# fill (?), align (=), sign (+), width (5)
f"{3 :?= +5} "          '+????3'
# fill (0), align (=), sign (+), width (5)
f"{3 :0= +5} "          '+0003'
# sign (+), 0-option, width (5)
f"{3 :+0 5}"            '+0003'
# 0-option, width (5)
f"{3 :05 }"             '00003'
```

Integer Presentation Types

	Decimal Integer Types
`d`	Decimal Integer: the number in base 10
None	Same as `d`
`n`	Number. Same as `d`, except uses current locale setting for the appropriate number separator character
	Others
`b`	Binary: the number in base 2.
`o`	Octal: the number in base 8.
`x`, `X`	Hex: the number in base 16. `x` vs `X` signals lower vs upper case for digits > 9
`c`	Unicode Character. Converts decimal integer to the corresponding unicode character

#-option: using the `#`-option adds prefixes

`#b`: adds prefix `0b` `#o`: adds prefix `0o`

`#x`: adds prefix `0x` `#X`: adds prefix `0X`

Examples: Integer Types

```
i = 13
# representing decimal 13 in binary, octal, and hexadecimal
f"bi nary: {i:b}; octal: {i:o}; hex: {i:x} or {i:X}"
                        'binary: 1101; octal: 15; hex: d or D'
# the #-option adds prefixes
f"bi nary: {i:#b}; octal: {i:#o}; hex: {i:#x} or {i:#X} "
                        'binary: 0b1101; octal: 0o15; hex: 0xd or 0XD'
# binary with zero padding
f"{i :08 b}"    '00001101'
# binary with prefix and zero padding
f"{i :#0 -      '0b001101'
8b} "
# fill (0), align (=), #-option, width (8), type (b)
f"{i :0= #8b }"
                        '0b001101'
# returns the unicode character with decimal representation 36
f"{3 6:c }"     '$'
# printing symbols with unicode decimal representations 33-41
print( ''.j oi n([ f"{x :c} " for x in range( -
33, 42)]))
                        !"#$%&'()
# separator (,) and type (d)
f"{1 000 -      '10,000'
0:, d}"
# type d is the default for integers, so unnecessary
f"{1 000 -      '10,000'
0:, }"
# separator (_), type (b)
f"{1 -          '100_1011_0000'
200 :_b }"
# to use locale specific number formatting, set the locale
import locale
locale.se tlo cal e(l oca le.L C_ALL, '')
# then use format type 'n'
f"{1 0** -      '1,000,000'
6:n }"
```

Cheatography

Python F-Strings Number Formatting Cheat Sheet

by Brian Allan (BrianAllan) via cheatography.com/133708/cs/27382/

Float Presentation Types

Type	Name	For a given precision p...
e, E	Scientific Notation	The coefficient has 1 digit before and p digits after the decimal point.
f, F	Fixed-Point Notation	There are p digits following the decimal point.
g, G	General Format	Rounds number to p significant digits, then formats in either fixed-point or scientific notation depending on magnitude.
%	Percentage	Multiplies the number by 100 and displays in fixed f format, followed by a percent sign.

- Default for float is p = 6. Default for decimal.Decimal is p large enough to show all coefficient digits.
- Lower case versus upper case: Determines whether scientific notation shows 'e' or 'E' and whether to use 'nan' versus 'NAN' and 'inf' versus 'INF'.

Examples: Simple Float Comparisons

```
# %-style with precision=1
f"{1 /2:.1% }"                          '50.0%'
# Lower case e versus upper case E
f"{1 000 :.1e} vs {1000:.1E }"          '1.0e+03 vs 1.0E+03'
# Lower case f versus upper case F
f"{1 000 :.1f} vs {1000:.1F }"          '1000.0 vs 1000.0'
# Lower case g versus upper case G
f"{1 000 :.1g} vs {1000:.1G }"          '1e+03 vs 1E+03'
# Lower versus upper case NaN representation
nan = float( 'nan')
f"{n an:f} vs {nan:F }"                 'nan vs NAN'
```

General Format (g or G) Rule

Given the number y and the precision specification p:

(a) Let exp = the exponent of y in scientific notation.

(b) Is -4 <= exp < p ?

General Format (g or G) Rule (cont)

"Yes": Format y using fixed-point notation and a new precision: p' = p - (1 + exp) where p represents the number of significant digits before any trailing zeros are removed. (In fixed-point format, the precision, here p', determines the number of digits following the decimal point. When it is positive, the value 1 + exp can be interpreted as the total number of digits before the decimal; otherwise, it represents the total number of zeros after the decimal point but before the significant digits.)

"No": Format y using scientific notation and a new precision: p' = p - 1 where p represents the total number of significant digits in the coefficient before any trailing zeros are removed. (In scientific format, the precision determines the number of digits displayed after the decimal point of the coefficient. So, the total number of digits displayed will be: precision + 1 = p' + 1 = p .)

(c) With the #-option (#g, #G): Keep any trailing zeros and the decimal point.
Without the #-option (g, G): If the result has insignificant trailing zeros, remove them as well as an unflanked decimal point.

Note: In both cases in (b), the original precision specification p determines the effective number of significant digits up until insignificant trailing zeros are removed in (c).

General Format (g or G): Illustration

#.4g	.4g	exp	format type
(keep trailing zeros)	(no trailing zeros)		
1.000e-06	1e-06	-6	e
1.000e-05	1e-05	-5	e
0.0001000	0.0001	-4	f
0.001000	0.001	-3	f
0.01000	0.01	-2	f
0.1000	0.1	-1	f
1.000	1	0	f
10.00	10	1	f
100.0	100	2	f
1000.	1000	3	f
1.000e+04	1e+04	4	e

In this illustration, the precision specification is p = 4; so, for exp from -4 to 3, fixed-point notation (f) is used; otherwise, scientific notation (e) is used.

By **Brian Allan** (BrianAllan)
cheatography.com/brianallan/

Published 16th March, 2022.
Last updated 16th March, 2022.

Cheatography

Python F-Strings Number Formatting Cheat Sheet

by Brian Allan (BrianAllan) via cheatography.com/133708/cs/27382/

Types with Conditional Formatting

Type	Interpretation
`g`, `G`	= `e` or `f` (`E` or `F`) according to the General Format Rule
none	= `s` for strings
	= `d` for integers
	= `g` for floats — When fixed-point notation is used, it always includes at least one digit past the decimal point. The precision varies to represent the given value faithfully.
`n`	= `d` for integers
	= `g` for floats — Uses the current locale setting for separator formatting.

Examples: Both Integers and Floats

```
si = 55  # small integer
li = 666666  # large integer
sf = 7.77777  # small float
lf = 8888.88  # large float
sc = 9e6  # scientific notation
```

```
# none -- no type specified
f"{si} {li} {sf} {lf} {sc}"
```

```
'55 666666 7.77777 8888.88 9000000.0'
```

```
# n -- number type
f"{si:n} {li:n} {sf:n} {lf:n} {sc:n} "
```

```
'55 666666 7.77777 8888.88 9e+06'
```

```
# n with #-option
f"{s i:#n} {li:#n} {sf:#n} {lf:#n} {sc:#n }"
```

```
'55 666666 7.77777 8888.88 9.00000e+06'
```

```
# n with precision = 3 (precision not allowed for integers)
f"{s i:.3n} {li:.3n} {sf:.3n} {lf:.3n} {sc:.3 n}"
```

```
☹
```

```
# e with precision = 3 (forces 3 digits after decimal point)
f"{s i:.3e} {li:.3e} {sf:.3e} {lf:.3e} {sc:.3 e}"
```

```
# including the #-option gives the same result here
f"{s i:#.3e} {li:#.3e} {sf:#.3e} {lf:#.3e}
{sc:#.3e}"
```

```
'5.500e+01 6.667e+05 7.778e+00 8.889e+03 9.000e+06'
```

```
# f with precision = 3 (forces 3 digits after decimal point)
f"{s i:.3f} {li:.3f} {sf:.3f} {lf:.3f} {sc:.3 f}"
```

Examples: Both Integers and Floats (cont)

```
# including the #-option gives the same result here
f"{s i:#.3f} {li:#.3f} {sf:#.3f} {lf:#.3f}
{sc:#.3f}"
```

```
'55.000 666666.000 7.778 8888.880 9000000.000'
```

```
# g with precision = 3 (shows no unnecesary zeros or decimal
points)
f"{s i:.3g} {li:.3g} {sf:.3g} {lf:.3g} {sc:.3 g}"
```

```
'55 6.67e+05 7.78 8.89e+03 9e+06'
```

```
# g with #-option (forces showing 3 significant digits)
f"{s i:#.3g} {li:#.3g} {sf:#.3g} {lf:#.3g}
{sc:#.3g}"
```

```
'55.0 6.67e+05 7.78 8.89e+03 9.00e+06'
```

```
# In float types generally, the #-option forces a decimal point
occurrence even when no digits follow it
f"{1 0:#.0e} {10:#.0f} {10:#.0 g} "
```

```
'1.e+01 10. 1.e+01'
```

References

- "A Guide to the Newer Python String Format Techniques" by John Sturtz at *Real Python*: https://realpython.com/python-formatted-output/#the-python-formatted-string-literal-f-string
- "Python 3's f-Strings: An Improved String Formatting Syntax (Guide)" by Joanna Jablonski at *Real Python*: https://realpython.com/python-f-strings/
- "Format String Syntax" including "Format Specification Mini-Language" from the page "`string` -- Common string operations": https://docs.python.org/3/library/string.html#format-string-syntax
- "2.4.3. Formatted string literals" from the page "2. Lexical analysis": https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals
- "PEP 498 -- Literal String Interpolation": https://www.python.org/dev/peps/pep-0498/
- "Python String Format Cookbook": https://mkaz.blog/code/python-string-format-cookbook/

By **Brian Allan** (BrianAllan)
cheatography.com/brianallan/

Published 16th March, 2022.
Last updated 16th March, 2022.