Python String Formatting Guide

John Strickler

Version 1.0, June 2021

Table of Contents

Chapter 1: String Formatting

Objectives

- Understanding the big picture of string formatting
- Doing Simple replacements
- Aligning text and numbers
- Specifying field widths
- Formatting dates
- Learning tips and tricks

Overview

- Strings have a format() method
- Allows values to be inserted in strings
- Values can be formatted
- Add a field as placeholders for variable
- Field syntax: `{SELECTOR:FORMATTING}`
- Selector can be index or keyword
- Formatting controls alignment, width, padding, etc.

Python provides a powerful and flexible way to format data. The string method format() takes one or more parameters, which are inserted into the string via placeholders.

The placeholders, called fields, consist of a pair of braces enclosing parameter selectors and formatting directives.

The selector can be followed by a set of formatting directives, which always start with a colon. The simplest directives specify the type of variable to be formatted. For instance, {1:d} says to format the second parameter as an integer; {0:.2f} says to format the first parameter as a float, rounded to two decimal points.

The formatting part can consist of the following components, which will be explained in detail in the following pages:

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

Parameter Selectors

- Null for auto-numbering
- Can be numbers or keywords
- Start at 0 for numbers

Selectors refer to which parameter will be used in a placeholder.

Null (empty) selectors — the most common — will be treated as though they were filled in with numbers from left to right, beginning with 0. Null selectors cannot be mixed with numbered or named selectors — either all of the selectors or none of the selectors must be null.

Non-null selectors can be either numeric indices or keywords (strings). Thus, {0} will be replaced with the first parameter, {4} will be replaced with the fifth parameter, and so on. If using keywords, then {name} will be replaced by the value of keyword 'name', and {age} will be replaced by keyword 'age'.

Parameters do not have to be in the same order in which they occur in the string, although they typically are. The same parameter can be used in multiple fields.

If positional and keyword parameters are both used, the keyword parameters must come after all positional parameters.

Example

fmt_params.py

```
#!/usr/bin/env python

person = 'Bob'
age = 22

print("{0} is {1} years old.".format(person, age))  ①
print("{0}, {0}, {0} your boat".format('row'))  ②
print("The {1}-year-old is {0}".format(person, age))  ③
print("{name} is {age} years old.".format(name=person, age=age))  ④
print()
print("{} is {} years old.".format(person, age))  ⑤
print("{name} is {} and his favorite color is {}".format(22, 'blue', name='Bob'))  ⑥
```

① Placeholders can be numbered

② Placeholders can be reused

③ They do not have to be in order (but usually are)

④ Selectors can be named

⑤ Empty selectors are autonumbered (but all selectors must either be empty or explicitly numbered)

⑥ Named and numbered selectors can be mixed

fmt_params.py

```
Bob is 22 years old.
row, row, row your boat
The 22-year-old is Bob
Bob is 22 years old.

Bob is 22 years old.
Bob is 22 and his favorite color is blue
```

f-strings

- **f** in front of literal strings
- More readable
- Same rules as `string.format()`

Starting with version 3.6, Python also supports *f-strings*.

The big difference from the `format()` method is that the parameters are inside the `{}` placeholders. Place formatting details after a `:` as usual.

Since the parameters are part of the placeholders, parameter numbers are not used.

All of the following formatting tools work with both `string.format()` and f-strings.

Example

fmt_fstrings.py

```
#!/usr/bin/env python

person = 'Bob'
age = 22

print(f"{person} is {age} years old.")
print(f"The {age}-year-old is {person}.")
print()
```

fmt_fstrings.py

```
Bob is 22 years old.
The 22-year-old is Bob.
```

Data types

- Fields can specify data type
- Controls formatting
- Raises error for invalid types

The type part of the format directive tells the formatter how to convert the value. Builtin types have default formats – 's' for strings, 'd' for integers, 'f' for float.

Some data types can be specified as either upper or lower case. This controls the output of letters. E.g, {:x} would format the number 48879 as 'beef', but {:X} would format it as 'BEEF'.

The type must generally match the type of the parameter. An integer cannot be formatted with type 's'. Integers can be formatted as floats, but not the other way around. Only integers may be formatted as binary, octal, or hexadecimal.

Example

fmt_types.py

```
#!/usr/bin/env python

person = 'Bob'
value = 488
bigvalue = 3735928559
result = 234.5617282027

print('{:s}'.format(person))    ①
print('{name:s}'.format(name=person))    ②
print('{:d}'.format(value))    ③
print('{:b}'.format(value))    ④
print('{:o}'.format(value))    ⑤
print('{:x}'.format(value))    ⑥
print('{:X}'.format(bigvalue))    ⑦
print('{:f}'.format(result))    ⑧
print('{:.2f}'.format(result))    ⑨
```

① String

② String

③ Integer (displayed as decimal)

④ Integer (displayed as binary)

⑤ Integer (displayed as octal)

⑥ Integer (displayed as hex)

⑦ Integer (displayed as hex with uppercase digits)

⑧ Float (defaults to 6 places after the decimal point)

⑨ Float rounded to 2 decimal places

fmt_types.py

```
Bob
Bob
488
111101000
750
1e8
DEADBEEF
234.561728
234.56
```

Table 1. Formatting Types

b	Binary – converts number to base 2
c	Character – converts to corresponding character, like chr()
d	Decimal – outputs number in base 10
e, E	Exponent notation. 'e' prints the number in scientific notation using the letter 'e' to indicate the exponent. 'E' is the same, except it uses the letter 'E'
f,F	Floating point. 'F' and 'f' are the same.
g	General format. For a given precision p >= 1, rounds the number to p significant digits and then formats the result in fixed-point or scientific notation, depending on magnitude. This is the default for numbers
G	Same as g, but upper-cases 'e', 'nan', and 'inf"
n	Same as d, but uses locale setting for number separators
o	Octal – converts number to base 8
s	String format. This is the default type for strings
x, X	Hexadecimal – convert number to base 16; A-F match case of 'x' or 'X'
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

Field Widths

- Specified as {0:width.precision}
- Width is really minimum width
- Precision is either maximum width or # decimal points

Fields can specify a minimum width by putting a number before the type. If the parameter is shorted than the field, it will be padded with spaces, on the left for numbers, and on the right for strings.

The precision is specified by a period followed by an integer. For strings, precision means the maximum width. Strings longer than the maximum will be truncated. For floating point numbers, precision means the number of decimal places displayed, which will be padded with zeros as needed.

Width and precision are both optional. The default width for all fields is 0; the default precision for floating point numbers is 6.

It is invalid to specify precision for an integer.

Example

fmt_width.py

```
#!/usr/bin/env python

name = 'Ann Elk'
value = 10000
airspeed = 22.347
# note: [] are used to show blank space, and are not part of the formatting
print('[{:s}]'.format(name))     ①
print('[{:10s}]'.format(name))   ②
print('[{:3s}]'.format(name))    ③
print('[{:3.3s}]'.format(name))  ④
print()
print('[{:8d}]'.format(value))       ⑤
print('[{:8f}]'.format(value))       ⑥
print('[{:8f}]'.format(airspeed))    ⑦
print('[{:.2f}]'.format(airspeed))   ⑧
print('[{:8.3f}]'.format(airspeed))  ⑨
```

① Default format — no padding

② Left justify, 10 characters wide

③ Left justify, 3 characters wide, displays entire string

④ Left justify, 3 characters wide, truncates string to max width

⑤ Right justify, decimal, 8 characters wide (all numbers are right-justified by default)

⑥ Right justify int as float, 8 characters wide

⑦ Right justify float as float, 8 characters wide

⑧ Right justify, float, 3 decimal places, no maximum width

⑨ Right justify, float, 3 decimal places, maximum width 8

fmt_width.py

```
[Ann Elk]
[Ann Elk   ]
[Ann Elk]
[Ann]

[   10000]
[10000.000000]
[22.347000]
[22.35]
[  22.347]
```

Alignment

- Alignment within field can be left, right, or centered
  - < left align
  - > right align
  - ^ center
  - = right align but put padding after sign

You can align the data to be formatted. It can be left-aligned (the default), right-aligned, or centered. If formatting signed numbers, the minus sign can be placed on the left side.

Example

fmt_align.py

```
#!/usr/bin/env python

name = 'Ann'
value = 12345
nvalue = -12345

①
print('[{0:10s}]'.format(name))    ②
print('[{0:<10s}]'.format(name))   ③
print('[{0:>10s}]'.format(name))   ④
print('[{0:^10s}]'.format(name))   ⑤
print()
print('[{0:10d}] [{1:10d}]'.format(value, nvalue))    ⑥
print('[{0:>10d}] [{1:>10d}]'.format(value, nvalue))  ⑦
print('[{0:<10d}] [{1:<10d}]'.format(value, nvalue))  ⑧
print('[{0:^10d}] [{1:^10d}]'.format(value, nvalue))  ⑨
print('[{0:=10d}] [{1:=10d}]'.format(value, nvalue))  ⑩
```

① note: all of the following print in a field 10 characters widedd

② Default (left) alignment

③ Explicit left alignment

④ Right alignment

⑤ Centered

⑥ Default (right) alignment

⑦ Explicit right alignment

⑧ Left alignment

⑨ Centered

⑩ Right alignment, but pad *after* sign

fmt_align.py

```
[Ann       ]
[Ann       ]
[       Ann]
[   Ann    ]

[     12345] [    -12345]
[     12345] [    -12345]
[12345     ] [-12345    ]
[  12345   ] [  -12345  ]
[     12345] [-    12345]
```

Fill characters

- Padding character must precede alignment character
- Default is one space
- Can be any character except }

By default, if a field width is specified and the data does not fill the field, it is padded with spaces. A character preceding the alignment character will be used as the fill character.

Example

fmt_fill.py

```
#!/usr/bin/env python

name = 'Ann'
value = 123

print('[{:>10s}]'.format(name))    ①
print('[{:.>10s}]'.format(name))   ②
print('[{:->10s}]'.format(name))    ③
print('[{:.10s}]'.format(name))    ④
print()
print('[{:10d}]'.format(value))    ⑤
print('[{:010d}]'.format(value))   ⑥
print('[{:_>10d}]'.format(value))  ⑦
print('[{:+>10d}]'.format(value))  ⑧
```

① Right justify string, pad with space (default)

② Right justify string, pad with '.'

③ Right justify string, pad with '-'

④ Left justify string, pad with '.'

⑤ Right justify number, pad with space (default

⑥ Right justify number, pad with zeroes

⑦ Right justfy, pad with '_' ('>' required)

⑧ Right justfy, pad with '+' ('>' required)

fmt_fill.py

```
[       Ann]
[.......Ann]
[-------Ann]
[Ann]

[       123]
[0000000123]
[_______123]
[+++++++123]
```

Signed numbers

- Can pad with any character except '{}'
- Sign can be '+', '-', or space
- Only appropriate for numeric types

The sign character follows the alignment character, and can be plus, minus, or space.

A plus sign means always display + or – preceding non-zero numbers.

A minus sign means only display a sign for negative numbers.

A space means display a – for negative numbers and a space for positive numbers.

Example

fmt_signed.py

```
#!/usr/bin/env python

values = 123, -321, 14, -2, 0

for value in values:
    print("default: |{:d}|".format(value))  ①
print()

for value in values:
    print("   plus: |{:+d}|".format(value))  ②
print()

for value in values:
    print("  minus: |{:-d}|".format(value)) ③
print()

for value in values:
    print("  space: |{: d}|".format(value)) ④
print()
```

① default (pipe symbols just to show white space)

② plus sign puts '+' on positive numbers (and zero) and '-' on negative

③ minus sign only puts '-' on negative numbers

④ space puts '-' on negative numbers and space on others

fmt_signed.py

```
default: |123|
default: |-321|
default: |14|
default: |-2|
default: |0|

   plus: |+123|
   plus: |-321|
   plus: |+14|
   plus: |-2|
   plus: |+0|

  minus: |123|
  minus: |-321|
  minus: |14|
  minus: |-2|
  minus: |0|

  space: | 123|
  space: |-321|
  space: | 14|
  space: |-2|
  space: | 0|
```

Parameter Attributes

- Specify elements or properties in template
- No need to repeat parameters
- Works with sequences, mappings, and objects

When specifying container variables as parameters, you can select elements in the format rather than in the parameter list. For sequences or dictionaries, index on the selector with []. For object attributes, access the attribute from the selector with . (period).

Example

fmt_attrib.py

```
#!/usr/bin/env python

from datetime import date

fruits = 'apple', 'banana', 'mango'
values = [5, 18, 27, 6]
dday = date(1944, 6, 6)
pythons = {'Idle': 'Eric', 'Cleese': 'John', 'Gilliam': 'Terry',
           'Chapman': 'Graham', 'Palin': 'Michael', 'Jones': 'Terry'}

print('{0[0]} {0[2]}'.format(fruits))  ①
print('{f[0]} {f[2]}'.format(f=fruits))  ②
print()
print('{0[0]} {0[2]}'.format(values))  ③
print()
print('{0[Palin]} {0[Cleese]}'.format(pythons))  ④
print('{names[Palin]} {names[Cleese]}'.format(names=pythons))  ⑤
print()
print('{0.month}-{0.day}-{0.year}'.format(dday))  ⑥
```

① select from tuple

② named parameter + select from tuple

③ Select from list

④ select from dict

⑤ named parameter + select from dict

⑥ select attributes from date

fmt_attrib.py

```
apple mango
apple mango

5 27

Michael John
Michael John

6-6-1944
```

Formatting Dates

- Special formats for dates
- Pull appropriate values from date/time objects

To format dates, use special date formats. These are placed, like all formatting codes, after a colon. For instance, {0:%B %d, %Y} will format a parameter (which must be a datetime.datetime or datetime.date) as "Month DD, YYYY".

Example

fmt_dates.py

```
#!/usr/bin/env python

from datetime import datetime

event = datetime(2016, 1, 2, 3, 4, 5)

print(event)  ①
print()

print("Date is {0:%m}/{0:%d}/{0:%y}".format(event))  ②
print("Date is {:%m/%d/%y}".format(event))  ③
print("Date is {:%A, %B %d, %Y}".format(event))  ④
```

① Default string version of date

② Use three placeholders for month, day, year

③ Format month, day, year with a single placeholder

④ Another single placeholder format

fmt_dates.py

```
2016-01-02 03:04:05

Date is 01/02/16
Date is 01/02/16
Date is Saturday, January 02, 2016
```

Table 2. Date Formats

Directive	Meaning	See note
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%f	Microsecond as a decimal number [0,999999], zero-padded on the left	1
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	2
%S	Second as a decimal number [00,61].	3
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	4
%w	Weekday as a decimal number [0(Sunday),6].	
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	4
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%z	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive).	5
%Z	Time zone name (empty string if the object is naive).	
%%	A literal '%' character.	

1. When used with the strptime() method, the %f directive accepts from one to six digits and zero pads on the right. %f is an extension to the set of format characters in the C standard (but

implemented separately in datetime objects, and therefore always available).

2. When used with the strptime() method, the %p directive only affects the output hour field if the %I directive is used to parse the hour.
3. The range really is 0 to 61; according to the Posix standard this accounts for leap seconds and the (very rare) double leap seconds. The time module may produce and does accept leap seconds since it is based on the Posix standard, but the datetime module does not accept leap seconds instrptime() input nor will it produce them in strftime() output.
4. When used with the strptime() method, %U and %W are only used in calculations when the day of the week and the year are specified.
5. For example, if utcoffset() returns timedelta(hours=-3, minutes=-30), %z is replaced with the string '-0330'.

Run-time formatting

- Use parameters to specify alignment, precision, width, and type
- Use {} placeholders for runtime values for the above

To specify formatting values at runtime, use a {} placeholder for the value, and insert the desired value in the parameter list. These placeholders are numbered along with the normal placeholders.

Example

fmt_runtime.py

```
#!/usr/bin/env python

FIRST_NAME = 'Fred'
LAST_NAME = 'Flintstone'
AGE = 35

print("{0} {1}".format(FIRST_NAME, LAST_NAME))

WIDTH = 12
print("{0:{width}s} {1:{width}s}".format(  ①
    FIRST_NAME,
    LAST_NAME,
    width=WIDTH,
))

PAD = '-'
WIDTH = 20
ALIGNMENTS = ('<', '>', '^')

for alignment in ALIGNMENTS:
    print("{0:{pad}{align}{width}s} {1:{pad}{align}{width}s}".format(  ②
        FIRST_NAME,
        LAST_NAME,
        width=WIDTH,
        pad=PAD,
        align=alignment,
    ))
```

① value of WIDTH used in format spec

② values of PAD, WIDTH, ALIGNMENTS used in format spec

fmt_runtime.py

```
Fred Flintstone
Fred         Flintstone
Fred---------------- Flintstone----------
----------------Fred ----------Flintstone
--------Fred-------- -----Flintstone-----
```

Miscellaneous tips and tricks

- Adding commas to large numbers {n:,}
- Auto-converting parameters to strings (!s)
- Non-decimal prefixes

- Adding commas to large numbers {n:,}

You can add a comma to the format to add commas to numbers greater than 999.

Using a format type of !s will call str() on the parameter and force it to be a string.

Using a # (pound sign) will cause binary, octal, or hex output to be preceded by '0b', '0o', or '0x'. This is only valid with type codes b, o, and x.

Example

fmt_misc.py

```
#!/usr/bin/env python

'''Demonstrate misc formatting'''

big_number = 2303902390239

print("Big number: {:,d}".format(big_number))  ①
print()

value = 27

print("Binary: {:#010b}".format(value))  ②
print("Octal:  {:#010o}".format(value))  ③
print("Hex:    {:#010x}".format(value))  ④
print()
```

① Add commas for readability

② Binary format with leading 0b

③ Octal format with leading 0o

④ Hexadecimal format with leading 0x

fmt_misc.py

```
Big number: 2,303,902,390,239

Binary: 0b00011011
Octal:  0o00000033
Hex:    0x0000001b
```

Index

s