

For reference the print() syntax:
print(objects, sep=' ', end='\n')

Formatting Options

INTERPOLATION: Old Style formatting operator % (modulo) - **to be deprecated**, widely used, **has bugs**

Note: [in brackets] means optional, ↵ means yields or returns There are **2 syntax formats**:

(1) "a string with format/insert (%) spec(s)" % (values to insert) or (2) "format string" % value to format

The interpolation format string is constructed of 2 required and 4 optional parameters; "length modifier" is never used.

Start of
format
specifier

: alternate form, 0 : zero padded,
- : left adjusted, ' ' (space) : space before
pos numbers, + : a sign +/- is required

Precision starts with a
decimal point followed by an
integer specifying places

[ten mod] was planned but
not implemented

% [(dict key)] [conversion flags] [minimum field width [*]] [precision: .## or [*]] **conversion type**

Mapping key in
parens for a
dictionary value

Examples of format strings:
%-14.4f left adj, min 14
char, 4 decimal places, float
-ing point or %("key1")s
use dict value for a string

An integer specifying
the minimum field
width

i/d : signed integer decimal; o : signed octal; x : signed hex
lower case; X signed hex upper case; e flt pt exponential lower
case; E flt pt exp upper case; f : floating point decimal format;
r : string using repr(); s : string using str(); more @:
<https://docs.python.org/2/library/stdtypes.html#string-formatting>

Examples:

print ("The cost of %d widgets is \$% .2f each" % (10, 202.95)) ↵ The cost of 10 widgets is \$ 202.95 each

sft="%14.4f" #sft is a string variable to hold the format spec statement - min field width (14), precision(.) 4 digits(4), floating point
print((sft)%(-7.51298701254)) ↵ -7.5130

The format() FUNCTION and STRING FORMAT: recommended over interpolation. There are **two forms of the format syntax**, both using the same format **mini-language** (available in Python 3.5) in a string or string variable. For a new user, conflicting examples can play with your head. The two forms of formatting are:

(1) **"format()" syntax:** `format(string/number, '0=+20,.3f')` <- **Teal** format string is **mini-language**. `format()` form is the easiest but **does not support any replacement or substitution fields** - only string and number **format and conversion**

(2) **"string format" syntax:** `'{:0=+20,.3f}'.format(string/number)` <- **can contain replacement fields**

In the examples above, assume the statements are in a print or variable assignment and the variable `number=12345.67890`:

...both yield the same result - both cases use **THE SAME FORMAT SPEC** both ↵ `+000,000,012,345.679`

What the format symbols mean: '{:' use format string (string
format only); 0 fill number with this character; = pad after sign but
before number; + use a sign; 20 required width in characters; ,
use commas to show thousands; .3 set (3 in this case) digit
precision; f number type, 'f' is floating point number; } - close

The examples on this page used the **"format()"** form as
shown in the function which generated the outputs listed.

The **"string format"** form is addressed on the **reverse**

side - page 2 of this toolbox. www.wikipython.com

How the **mini-language** statements are ordered and structured in general: (Note: they must be in the order as shown below!)

[[fill]] align [sign] [# -alt form] [0 force padding default] [width] [,] [.precision] [type]

+, -(neg only), " " space pos, - neg *1

IF align used; can be any character

<, >, ^ (centered), or =
'=' forces padding after a sign

forces 0 between sign
& number when no align
specified; like '0='
Floats/Decimal - always have dec point;
integers -> hex, oct, binary - add 0x/0o/0b;
'g'/'G' - retain any trailing zeros

width in chars

thousands
separator ,000 -
can't use with 'n'

decimal places

Integer types:

b - binary; c - Unicode char;
d - base 10 integer; o - Octal;
x - Hex base 16 - lower cs;
X - Hex base 16 - upper cs;
n - like d but uses local
separator definitions

Float/decimal types:

e - scientific, 'e' for exponent;
E - 'E' for exponent; f - fixed
point (default 6); F - fixed,
NaN and INF; g - general
format, rounds and formats; n
- same as g but uses local
separator definitions; % -
percentage, * 100, adds "%";
None - g except 1 num > .
String: s - string format, can
be omitted

8M+ variations of possible **numeric formats** not counting
multiple values for fill, width, and precision. A few output
examples: `aNum=12345.6789` and `bNum=-12345.6721`
and format string `myForm` shown in red below. Output re-
sulting from `printrnums` unless otherwise noted. (Output
with `print()` yields: 12345.6789 -12345.6721)

Function to generate the results:

```
def printrnums (aNum,bNum,myForm):
    print("Column Border->|" + format \
    (aNum, myForm) + "|" + format \
    (bNum, myForm) + "|")
```

Format string	myForm=this string	Yields	Description
"%.3f"	Column Borders ->	12345.679 -12345.672	fixed, 3 places, float
".2f"	Column Borders ->	12,345.68 -12,345.67	comma sep, fixed, 2 places, float
"15,.2f"	Column Borders ->	12,345.68 -12,345.67	width=15, comma sep, fixed, 3 places, float
"^15,.3f"	Column Borders ->	12,345.679 -12,345.672	center, width=15, comma, fixed, 3 places, float
"^15,e"	Column Borders ->	1.234568e+04 -1.234567e+04	center, set width (15), scientific
"~^15,.0f"	Column Borders ->	~~~~12,346~~~~ ~~~~-12,346~~~~	fill ~, ctr, width=15, commas, 0 dec places, float
"+,.0f"	Column Borders ->	+12,346 -12,346	sign, comma sep, fixed, no dec places, float
"_ =+15,.0f"	Column Borders ->	+ 12,346 - 12,346	pad _ after sign, sign, comma, no dec plcs, float
The following examples use aNum=17 and bNum=256			
"x^10d"	Column Borders ->	xxxx17xxxx xxx256xxxx	fill w/ x, center, width=10, integer (base 10)
">#12X"	Column Borders ->	0X11 0X100	right align, width=12, hex (uppercase)
"b"	Column Borders ->	10001 100000000	Binary conversion

*1 With sign: + sign all, - neg only,
space force leading space on + and
sign neg

String Format() functions
continued on reverse side.

...plus a LOT LOT more at:

www.wikipython.com

Formatting Options

"string.format()" form syntax: Ordering or Substituting text and numbers in statements

Substitution, ordering or format: this syntax is in 3 PARTS: **Part 1** is **either** a way to identify which value is referenced by the literal or data container between the parens of .format(), for example '{1}' to select the 2nd value, **or** a format spec designated by following the opening '{' with ':' ; for example '{:0=+20,.3f}'. **Part 2** is the command - .format(). **Part 3** is the literal strings or data containers referenced inside the .format parens. Look at it like this -

print string with {selection values}[{x}{x}...].format(-/*/**- source for selection/insertion)

a string with embedded values in {} brackets holding a selection index or format specification : in mini-language

literal values; a tuple to unpack preceded by a single *; multiple tuple items coded in the print string: [tup#[item#]]; a dictionary to reference for keys coded in print string, preceded by **

literal values; a tuple to unpack preceded by a single *; multiple tuple items coded in the print string: [tup#[item#]]; a dictionary to reference for keys coded in print string, preceded by **

Examples using the order and replacement functions of str.format()

Objects to use in the following examples

```
OrderString = '{1}, {0}, {2}'; Stoogetuple = ('Larry', 'Moe', 'Curley')
ShirtTuple = ('red', 'white', 'blue', 'purple') # {index-of-tuple-in-format-list[index of item]}
PetDict = {'Straightman': 'Larry', 'Numskull': 'Moe', 'Foil': 'Curley', 'Looser': 'Don'}
PetDict = {'1': 'cow', '2': 'dog', '3': 'goldfish'}
class Flowers(object):
    def __init__(self, center, petals):
        self.center=center
        self.petals=petals
Daisy = Flowers("black", "yellow")
```

```
# Simple selection and ordering of values with literals
aPrintString = "The tourney ranking: {1}, {3}, {0}".format /
('Larry', 'Moe', 'Curley', 'Donald')
print(aPrintString) # The tourney ranking is: Moe, Donald, Larry
# String holding substitution/replacement selections
print("The tourney rank is: "+ OrderString.format /
('Abe', 'Bob', 'Cal', 'Don')) # The tourney rank is: Bob, Abe, Cal
# Named items
print("Winners: {FirstPlace}, {SecondPlace}".format /
(FirstPlace = "Bob", SecondPlace="Don")) # Winners: Bob, Don
# Use * to unpack a single tuple (but not a list)
print("The stooges are: {2}, {1}, and {0}.".format /
(*Stoogetuple)) # note * & sub syntax
# The stooges are: Curley, Moe, and Larry.
# Use the {0[value index]} without having to use *
print("My favorite stooge is {0[0]}.".format(Stoogetuple))
# My favorite stooge is Larry.
```

```
# The '[0[]]' structure enables us to select from multiple tuples
print("I saw {0[1]} in a {1[2]} shirt.".format(Stoogetuple, /
ShirtTuple)) # I saw Moe in a blue shirt.
# Use ** to access dictionary values by their keys
print("The stooges are: {Straightman}, {Foil}, {Numskull}. /
".format(**Stoogetuple)) # note ** "dictionary is external"
# The stooges are: Larry, Curley, Moe.
# Select a single dictionary item
print("My favorite stooge is {Foil}.".format(**Stoogetuple)) /
# My favorite stooge is Curley.
# A single dictionary item using the {x[]} format and keyword
print("One stooge is {0[Foil]}.".format(Stoogetuple))
# One stooge is Curley.
# Select multiple items from multiple dictionaries using keywords
print("It look like {1[Straightman]} has a {0[1]} and a {0 /
[2]}.".format(PetDict, Stoogetuple))
# It look like Larry has a cow and a dog
# Refer to an object's attribute # combine with your class - very powerful
print("Its petals are bright {0.petals}.".format(Daisy))
# Its petals are bright yellow.
# using !r and !s - example borrowed from
https://docs.python.org/3/library/string.html#formatspec
print("repr() shows quotes: {!r}; str() doesn't: {!s}.".format /
('test1', 'test2')) #best possible example we could imagine
# repr() shows quotes: 'test1'; str() doesn't: test2
```

Built-in String Format Methods

```
.capitalize() -1st letter
.center(width[, fillchar default: space])
.ljust(width[, fillchar]) -justify
.rjust(width[, fillchar]) -right justify
.upper() -converted to uppercase
.lower() -convert to lowercase
.strip([chars]) -remove leading and trailing chars
.lstrip([chars]) -remove leading chars
.rstrip([chars]) -remove trailing chars
.title() -return a titlecased version
.zfill(width) - left fill with 0 to width
.swapcase()
```

Template strings: A simple substitution function imported from the **string module**. (from **string import Template**) To keep it simple: (1) use the **Template** function to build a variable with named objects preceded by \$ to be replaced with subs, (2) then use **substitute(map object, **kws)** on that variable to define replacement values and build the string. (\$\$ escapes and yields \$)

```
from string import Template
stoogetuple = {'L': "Larry", "M": "Moe", "C": "Curley"}
funnyStr = Template("$C handed the goat to $L and butted $M.")
funnyStr = funnyStr.substitute(stoogetuple)
print(funnyStr)
# Curley handed the goat to Larry and butted Moe.
....put together more succinctly
print(Template("$M and $C butted $L's goat.").substitute
(stoogetuple))
# Moe and Curley butted Larry's goat.
```

Template strings are easy, but VERY slow to execute!

.format dates: the easy way
import datetime
d = datetime.datetime(2018, 1, 19);
print('{:%m/%d/%Y}'.format(d))
01/19/2018

New in version 3.6: f-strings - formatted string literals - prefixed with f/F - much like string.format() above

https://docs.python.org/3/reference/lexical_analysis.html#f-strings (2.4.3 Formatted string literals) ALSO: see PEP 498

