BIG DADDY'S PYTHON TOOLBOX For 3.5

Formatting and Substitution

Reference:
print(objects, sep=' ', end='\n')

The Format Specification Mini-Language is a **fixed** series of codes used to format strings and values by the **FORMAT FUNCTION, FORMAT METHOD** and **F-STRINGS**

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

String: s or **none** - string format

Integer types:
b - binary
c - Unicode char
d - base 10 integer
none - same as d
o - Octal format
x - Hex - lower cs
X - Hex - upper cs
n - d with local separators

Float/decimal types:
e - scientific, e - exponent
E - E for exponent
f - fixed point (default 6)
F - fixed, NAN and INF
g - general format
G - g except E for large #
n - g with local separators
% - percentage; * 100; adds "%"
none - for decimal, same as g or G

[. precision] decimal places defaults to 6 if not specified

[_ ,] **thousands** separator - (used with specific types only) float, d, b, o, x, X

[width] min field width in characters

[0 force padding default] forces 0 between sign & number w/o align specified; like '0='; (err with string)

[#-alt form] only for conversion of integer, float, or complex #s, non-decimal integers add prefix, float and complex #s always have a decimal point

[sign] Sign: **" "**(space) - force leading space on +\- sign; **+** - sign all; **-** neg only (sign/align err with strings)

[align] **<** (lft), **>** (rt), **^** (centered) or **=** which forces padding after a sign

[fill] align can pad with any character; default is space

wikipython.com Format mini-language

Special Note: The "printf-style" string formatting, a.k.a, interpolation, using the built in % (modulo) operator is buggy and being deprecated - it is no longer recommended.

Built-in String Format Functions and Methods

.capitalize() -1st letter
.center(width[, fillchar *default: space*])
.ljust(width[, fillchar]) – left justify
.rjust(width[, fillchar]**)** - right justify
.upper() -convert to uppercase
.lower( ) -convert to lowercase
.strip([char list**])** -remove leading and trailing chars, defaults to whitespace
.lstrip([char list**])** –remove leading chars
.rstrip([char list**])** remove trailing chars
.title() -return a titlecased version
.zfill(width**)** – left fill with 0 to width
.swapcase() - upper to lower, visa versa
*separator_str***.join([***string_list***])**
.removeprefix(prefix, **/)**
.removesuffix(suffix, **/)**

Common Examples of Format Specification Formatting

Numeric examples using 0123456.789 and –0123456.789 for 1st five examples

Format String	Result	Description
".2f"	123456.79 and –123456.79	standard 2 decimal places
",2f"	123,456.79 and -123,456.79	add comma for thousands sep
">15.2f"	123,456.79 and -123,456.79	right align in 15 space field
"*>15,2f"	*****123,456.79 and ****-123,456.79	force fill with leading character
".2e"	1.23e+05 and -1.23e+05	scientific notation
">#8x"	0x100 and -0x100	<- integer 256/-256 to alt hex
"<10s"	00123456 |<-end of field	string "00123456" in 10 space field

format Function: easy to use (**no** substitution fields) text / number format and conversions:
Syntax: **format(value, *"format string"*)**
ex: print("|" + format(-12345.6789,">12,.2f") + "|") ↳ **| -12,345.68|**

.format Method: format and substitution in one statement **Syntax and example:**
"string w/ **{**[replacement data] [! r/s/a] [":"+*format string*]**}**".**format(sub source(s))**

literal string with embedded replacement placeholders and formatting command substitution sources

```
data_tuple = ("cash","credit card", "check","bit coin", 5.50, 10, 25.00, 100)
print("Paid by {1!s:<}: ${5:>7.2f}. Thank you.".format(*data_tuple))
```
↳ Paid by **credit card**: $ **10**.00. Thank you.

Tuples are unpacked by a single *; **multiple** tuples are numbered in the print string: [tuple #[item #]].

```
dt1 = ("cash","credit card", "check","bit coin")
dt2 = ( 5.50, 10, 25.00, 100)
print("Paid by {0[2]!s:<}: ${1[0]:>5.2f}. Thank you.".format(dt1, dt2))
```
↳ Paid by **check**: $ **5.50**. Thank you.

```
mydict={"paidby":"cash", "amount":5.50}
mydict["paidby"] = "bitcoin"
print("Paid by {paidby:<}: ${amount:>5.2f}. Thank you.".format(**mydict))
```
↳ Paid by **bitcoin**: $ **5.50**. Thank you.

A dictionary with reference keys in the print string is unpacked with **.

More examples:

Objects in the following examples

```
OrderString = '{1}, {0}, {2}'
ShirtTuple = ('red', 'white', 'blue', 'purple')
StoogeDict = {'Straight':'Larry', 'Dunce':'Moe',
'Foil':'Curley', 'Boob':'Don'}
PetDict = {1 : "cow", 2 : "dog", 3 : "fish"}
StoogeTuple = ('Larry','Moe','Curley','Don')
```
 # Simple selection and ordering of values with literals
```
mystring = "The tourney ranking: {1}, {3}, {0}".format /('Larry','Moe','Curley','Donald')
print(mystring)
```
↳ 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: {First}, {Second}".format (First = "Bob", Second ="Don"))
```
↳ Winners: Bob, Don
 # Use * to unpack a single tuple (BUT not a list— for lists use 0[val])
```
print("The stooges are: {2}, {1}, and {0}.".format (*StoogeTuple))  # note * & sub syntax
```
↳ The stooges are: Curley, Moe, and Larry.
 # Use the {index[value index]} without having to use * or for a list
```
print("My favorite stooge is {0[1]}.".format(StoogeTuple))
```
↳ My favorite stooge is Moe. The use of index allows sub of multiple tuples and lists.
 # The '0[1]' 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 with unpacking
```
print("The stooges are: {Straight}, {Foil}, {Dunce}.".format(**StoogeDict))
```
↳ The stooges are: Larry, Curley, Moe.
 # Select a single dictionary item by unpacking
```
print("My favorite stooge is {Foil}.".format(**StoogeDict))
```
↳ My favorite stooge is Curley.
 # 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'))
```

BIG DADDY'S PYTHON TOOLBOX For 3.5

...plus a LOT more at: www.wikipython.com

Formatting and Substitution Options

Class Attributes and .format substitution

```
class Flowers(object):
    def __init__(self, center, petals):
        self.center=center
        self.petals=petals
Daisy = Flowers("black","yellow")
Dogwood = Flowers ("brown","white")
print("Daisy petals are bright {0.petals}, its center {0.center}, while the \
Dogwood petals are {1.petals}." .format(Daisy, Dogwood))
```

↳ Daisy petals are bright yellow, its center black, while the Dogwood petals are white.

The String Constants Module: was the basis for much of the format function and method including the format mini-language. It contains some useful constants and **Template strings** which "provide simpler string substitutions as described in PEP 292" supporting $-based subs. **from String import Template** Module example:

```
s = Template('$who likes $what')
s.substitute(who = "Tim", what = "kung pao")
```

↳ 'Tim likes kung pao'

Helper function: string.**capwords**(s, sep=None)

Formatting Dates: Datetime, time, and calendar are vast modules with many methods and class objects. Their instances expose a built-in special format method called **strftime()**. Strftime uses modulo-plus-letter **format codes** (like "%m" for a 2 digit month) to display date and time attributes of their class objects. The full list of code "directives" is at: https://docs.python.org/3/library/datetime.html. These use the mondulo operator but are **not part of the old interpolation formatting**. Conversely, a properly formatted string can be used to create a datetime object using the class method called **strptime().** Since format can access instance attributes we can use it with dates, though it may not be our best choice.

For example, given:

```
from datetime import *
today = date.today()  #today's date assigned to variable "today"
```

(see abbreviated strftime codes list at page lower right)

we could use the instance attributes:

```
months =('0', 'Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec')
days = ('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday')
print("Today is {1!s}, {2!s} {0.day}, {0.year}.".format/
   (today, days[today.weekday()], months[today.month]))
```

↳ **Today is Saturday, Jan 1, 2022.**

or we can get about the same result more easily with strftime:

```
print("Today is", today.strftime("%A, %b %d, %Y."))
```

↳ **Today is Saturday, Jan 01, 2022.**

New in version [3.6]: f-strings - formatted string literals - prefixed with letters f or F

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

Text except }, {, or NULL - {{ & }} are replaced with single braces

"s" | "r" | "a" str(), repr(), ascii()

See mini-language described on page 1 Example:
format(12345.678,"*=+15,.2f") ↳ +*****12,345.68
In **f string** would be f"{12345.678**:***=+15,.2f}"

f or **F** ● opening quote " ● [literal text] ● {replacement fields [**:**format string]} ● [literal text] ● closing quote "

f_expression : (**conditional_expression** | "*" or_expr) ("," conditional_expression | "," "*" or_expr)* [","] (NO BACKSLASHES IN EXPRESSION PARTS; **Must put LAMBDAS in parens ( ))**

"**{**" f_expression ["**!**" conversion] ["**:**" format_spec] "**}**"

no backslashes* *var*!s** *var*!**r** *var*!**a** ⇨ *'var' is literal variable*

```
# variables for examples     nametup = ("Larry","Curley","Moe")
myindex, Name, width, value, x = 2, 'Curley', 12, 12345.678, 75
state, subpart, subpart2 = 'Mississippi', 'iss', 'x'
lamstate = lambda state: state if subpart in state else "unknown"
intro_string, fmt_str = "Cost: $", "<#8x"
# substitution using an indexed tuple
print(f"He said his name is {nametup[myindex]}.")  ↳ He said his name is Moe.
# substitution and string format
print(f"{Name.upper(): ^10} center & caps!")  ↳ CURLEY  center & caps!
# literal string holding formatted variable : thousand comma, 2 places, float
print(f"{intro_string}{value: {width},.2f} is cheap?")  ↳ Cost: $  12,345.68 is cheap?
# conditional f_expression using lambda, built-in format and column placement
print(f"Going to {(lamstate(state)).upper():^20}!")  ↳ Going to  MISSISSIPPI  !
# conditional f_expression using ternary "if"
print(f"Go to {state if subpart in state else 'unknown'}!")  ↳ Go to Mississippi!
print(f"Going to {state if subpart2 in state else 'unknown'}!")  ↳ Going to unknown!
# conversion of float or integer
print(f"Curley's IQ is about {x!r}.")  ↳ Curley's IQ is about 75.
# Date formatting using strftime:    import datetime
print(f"Today is {datetime.date.today():%m/%d/%Y}.")  ↳ Today is 03/06/2020.
# Conversion to scientific notation: 4 decimal places
print(f"Estimated precision is: {value:.04e}")  ↳ Estimated precision is: 1.2346e+04
# get and print today's date using  datetime strftime values:
from datetime import date
today = date.today()
print(f"Today is {today:%A, %b %d, %Y}.")  ↳ Today is Sunday, Jan 02, 2022.
# print 8 space right aligned colums of tuple values converted to hexadecimal:
val=(1,256,1028, 65536)
print(f"{val[0]:<#8x}{val[1]:<#8x}{val[2]:<#8x}{val[3]:<#8x}")
↳ 0x1   0x100   0x404   0x10000     or alternatively — yielding same result:
for v in range(0, len(val)):
    print(f"{val[v]:{fmt_str}}",end="")  ← note use of {fmt_str}
```

formatter module - deprecated since [3.4]

pprint module—Data Pretty Printer provides a quick, simple way to format complex data consisting of Python literals and make it visually intelligible. See blogs on **www.wikipython.com**

```
import pprint
# now create a pretty printer instance
pp = pprint.PrettyPrinter(indent=1, width=80,
depth=None, stream=none,*, compact=False,
sort_dicts=True, underscore_numbers=False)
```

For output, send the object to be formatted to the method of the object you created.

```
pp.pprint(your_object)
```

There are also several "shortcut functions" in the module:

pprint.pformat() - returns a string holding the formatted representation of the object

pprint.pp() - prints formatted object plus "\n"

...more at: https://docs.python.org/3/library/pprint.html#module-pprint

strftime & strptime() format codes

%a abbreviated weekday	%A weekday, full
%d day of mo, 2 digits	%w weekday #
%b abbreviated month	%B month, full
%m month, 2 digits	%y year w/o century
%Y year with century	%H hour, 0 padded
%M minute, 0 padded	%S second, 0 paded
%j day of year, 0 padded	%Z time zone name
%c locale's date and time	%x locale's date
%X locale's time	%% - literal % char
%U wk of the year, Sunday 1st day	
%W wk of the year, Monday 1st day	
%f microsecond 6 digit zero padded decimal	