

Output: Format()

There are four **basic** things you can do to data in Python:

1. Acquire, retrieve or create it.
2. Manipulate and apply it.
3. Store it.
4. Present it, which is usually the point of 1-3.

The first three have many commands and functions to utilize. The last has mainly two: **print()** and **format()**. Of these two, **95% of what we can do to affect output is with format()** and 95% of format() is learning the highly structured **mini-language** that it employs.

For Reference:

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

To avoid confusion before getting deeply into format(), there are two basic **but separate "jobs"** that format does, **AND** there are **two forms of the syntax** that you can use in Python 3.5+. For a new user, conflicting examples can play with your head. **The 2 jobs are:**

- (1) Allow the programmer, with great flexibility and detail, to **format** the visual output of strings and numbers.
- (2) Allow the programmer to **substitute** or **order** text and numbers in various string or numeric statements.

For formatting, examples of the forms of syntax will serve to demonstrate them. Assume either of the statements below are in a print or string assignment and variable **number=12345.67890**:

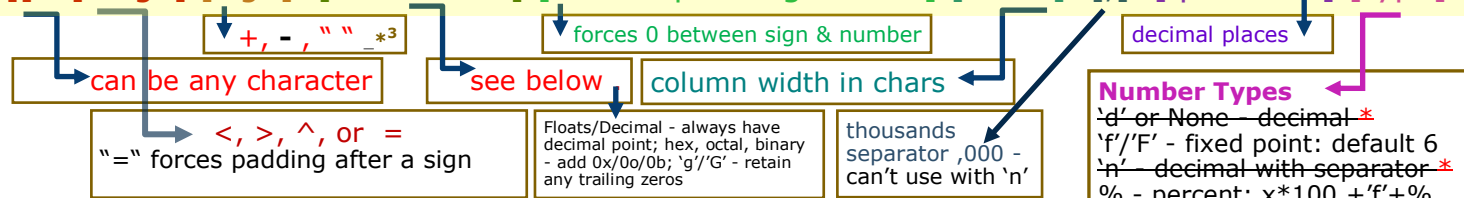
`format(number, '0=+20,.3f')` #<- characters in teal are the mini-language
or `'{:0=+20,.3f}'.format(number)` #<- Note the {: sequence to start and } to end
...will yield the same result - in both cases we used **THE SAME FORMAT SPEC**

+000,000,012,345.679
+000,000,012,345.679

What these symbols do: '{' - open format string & '0' - fill number with zeros;
'=' - pad after sign but before number; '+' - use a sign; '20' - width in characters; ',' - use , to show thousands; '.3' - 3 digit precision; 'f' - floating point number; '}' - close

This is how the mini-language statements are ordered and structured in general: *(Note: they must be in this order!)*

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



There are at least 8,448 variations or structures of possible **numeric formats** if we do not count multiple values for fill characters, width, and precision. Here are just a few examples of more likely desired outputs: assume variables **aNumber=12345.6789** and **bNumber=12345.6721** with an output resulting from **printnums** (see below) unless otherwise noted. As a reference, output with **print()** yields: 12345.6789 -12345.6721. Where Anum and Bnum are as noted and Afor and Bfor are format strings shown in red below.

formatstring	Yields	Description
">15.2f"	Column Border -> 12345.68 -12345.67	two fixed decimal places *1
">15,.2f"	Column Border -> 12,345.68 -12,345.67	s.a.b add comma separators
">+15.2f"	Column Border -> +12345.68 -12345.67	force sign with all numbers
"0>+15.0f"	Column Border -> 000000000+12346 000000000-12346	fill with zeros, no decimals
"0=+15.0f"	Column Border -> +00000000012346 -00000000012346	force pad between sign/num
",.1%"	Column Border -> 1,234,567.9% -1,234,567.2%	%, 1 dec place, no col width *2
"g"	Column Border -> 12345.7 -12345.7	General format, no options
"15e"	Column Border -> 1.234568e+04 -1.234567e+04	Exponential, col width 15 spc

Number Types

'd' or None - decimal *
'f'/'F' - fixed point: default 6
'n' - decimal with separator *
'%' - percent: x*100 + 'f' + %
'b' - binary
'o' - octal
'x'/'X' - hex - lower/upper cs
'g'/'G' - general format
'e'/'E' - exponent notation

String Types

's' - string; may be omitted
'c' - character - converts number to character
In our testing 'n' and 'd' do NOT work with windows. d yields "unknown code" and n yields scientific notation.

Function to generate the results:

def printnums(Anum,Afor,Bnum,Bfor):
`print("Column Border ->|"+format(Anum,Afor)+"|"+format(Bnum,Bfor)+"|")`

*1 Note rounding - always up

*2 Note rounding: traditional: 0-5 down, 5+ up

The substitution command structure is in three parts:

1. A set of positional or formatting values (one or the other but not both) enclosed in curly brackets inside a quoted **single string**. A string variable can be used.

"{variable to output} | {numeric format}..." .format('string' or numeric values...)

2. **".format()"** - the command for the built-in function, and
3. A set of variables from which the positional values are selected and output.

If the first character inside a bracket is a colon, format recognizes a format definition. If any object is numeric, every object included in the format parens MUST have a set of curly format brackets, related in the same order, these can be empty for strings in the sequence or contain a format definition for strings or numerics. (The use of a position value is illegal in mixed groups.)

If all the variables to be formatted are strings, the {curley brackets} can contain a number to designate the parens value to be selected for that sub variable's order of print. There can be any number of curly brackets in the string. Each must contain a number from 0 to the number of objects in the format parens minus 1, i.e., the objects are numbered starting with zero. Not all the values have to be used, but values can be repeated.

Format() Substitution

More Notes:

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

Conversion Flags: !s and !r apply str() and repr(adds ')
'{0[x]}' selects the xth value in a tuple which format names: ex: print('{0[3]}' .format(mytuple))

Dictionary values can be selected by key reference: mydict={'a':1,'b':2000}; print('{[b]}' .format(mydict)) yields: 2000

Can nest dictionary selection and formatting: mydict={'a':1000,'b':2000}; print('{:.2f}' .format(float('{[b]}' .format(mydict)))) ..yields: 2000.00

.format allows string additions preceding or intermixed with items. Ex: print('{<18}{>8.2f}{>10}{<12}' .format(champs[i][0],champs[i][1],champs[i][2], "+champs[i][3]))

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

....yields: 01/19/2018 SEE WWW.WIKIPYTHON.COM

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

Data on Disk

Structure, method, syntax and examples **assume the following.** (1) Variables “pyFile”, “pyDir”, “pyPath”, “txtList” & “dataList” hold data file names [i.e., **Mary.txt** or **myCsv.csv**], directory path [**D:\\Pyfiles**], **pyDir+pyfile**, and strings of text in a list, **respectively**. (2) File names and directory locations are hard coded by the programmer. (3) Read, write, append or attribute retrieval below is **done inside a “with” command structure, not shown**, except where noted. (4) Data written and retrieved is data type “text” unless otherwise specified. (5) The file object created is called “myFile” (6) you import modules as needed.

Create and Open a New File Object **with** `open(pyPath,'w+')` as myFile
Append to an existing file **with** `open(pyPath,'a')` as myFile

Write Text Strings to a File Object **for line in txtList:**
`myFile.write(line)`

Create a New Empty File Object **with** `open(pyPath,'a')` as emptyfile
`emptyfile.close` # just being cautious-**with** closes file automatically

Create/Open/Write **not** using **with**
`myFile= open(pyPath, 'w')`
`myFile.write("This is line 1. \n")`
`myFile.close()` **#MUST manually close the file!**

Ways to Read Access a File
(1) **Loop** through it (2) `.readline()`
(3) `.read()` (4) `.readlines()` (5) `.list()`

Read by Looping **for line in myFile:**
`print(line, end="")`
(process line here)

String == compare must include “\n” at the end for compare to work.

`.readline()` - process a line at a time
`getAline="initialize the variable"`
while not(getAline==""):
`getAline=myFile.readline()`
if ...etc....:

`.read()` - into list of strings `.seek(0)`
`txtList=[myFile.read()]` *exhausts pointer - now at EOF. Reads all lines into a single joined line with newline (“\n”) dividers
`myFile.seek(0)` reset pointer to start
`txtList=myFile.read()` *reads individual letters of strings to list items
`myFile.seek(0)` reset pointer to start
To remove \n in read use `.splitlines()`
`txtList=myFile.read().splitlines()` - will put discrete lines without newline as items in txtList

`.readlines()` or `list()`
- get whole file at once
`txtList=myFile.readlines()` or
`txtList=list(myFile)`
will put discrete lines with newline as items in txtList; no `splitlines()` support

Basic Modes: ‘r’-read only ‘r+’-read or write ‘w’-write only, overwrites existing file ‘w+’ read or overwrite ‘a’-append ‘a+’-append or read; add ‘b’ to anything to invoke binary format; w,wb,w+,wb+,a,ab,a+,ab+ create new file if file does not exist

Useful File/Path/Directory Tools

Get current Directory
`holdCurDir=os.getcwd()`
Change directory to ‘path’
`os.chdir('path')`
Check for current filepath
if `os.path.exists(pyDir):`
Check for current file
if `os.path.isfile(pyPath)`
Get list of directories in path
`dirList=os.listdir(path)`
Get list of entries in path directory
`os.listdir(path)`

import CSV-Comma Separated Values

Use a standard “with open” as “myFile” structure, then create a csv.reader object or a csv.writer object.

`csv.reader (csv file object [, dialect='excel'] [, optional parameter override])` - iterates the file and then....
`__next__()` reads the subsequent lines (rows/records) of data
`.line_num` returns number of lines read
`csv.writer (csv file object, dialect='excel', formatting parameters)` -writing series of lists, each a record/table row
`.writerow(row)` write a row formatted by dialect (on existing file overwrite or append)
`.writerows(rows)` write all rows

Create csv reader obj (example)
`mycsvr=csv.reader(myFile, dialect= 'excel')`
Get next row of data
`Row1=mycsvr.__next__()`
Create csv writer obj (example)
`Mycsvr=csv.writer(myFile, dialect='excel')`
Write or Append csv data
`mycsvr.writerow(dataList)` or
`Mycsvr.writerows(dataList)`

Dictionary Read/Write (csv module) - see <https://docs.python.org>
`csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwargs)`
`csv.DictWriter(csvfile, fieldnames, restval="", extrasaction='raise', dialect='excel', *args, **kwargs)`

import pickle - serialized objects

Uses standard “with open” structure - **must be opened for binary operations**
To **dump** (save) an object/file:
`pickle.dump(object-to-pickle, save-to-file, protocol=3, fix_imports=True)`
EX: `pickle.dump(someList,myFile)`
To **load** (retrieve) an object/file:
`pickle.load(file-to-read [, fix_imports= True] [, encoding="ASCII"][, errors= "strict"])`
EX: `myList = pickle.load(myFile)`

import sqlite3 - SQL database

Create **connection** object
`sq3con = sqlite3.connect('mysqlFile.db' [,detect_types])` **also:**
`sq3con = sqlite3.connect(":memory:")`
A few connection object methods:
`.cursor` (see below) `.commit()`
`.rollback()` `.close()` `.iterdump()`
Create **cursor** object
`CurObj = sq3con.cursor()`
A cursor object methods and attributes:
`.fetchone()` `.fetchmany(size)`
`.fetchall()` `.close()` `.rowcount` `.lastrowid`
`.execute("sql [,parameters]")`
EX: `CurObj.execute("CREATE TABLE table_name (col_name data_type,...)")`
`.executemany("sql [,parameters]")`
Notes: sql statements are case insensitive. Multiple statements are separated by semicolons (;). SQL ignores white space. Parameters are separated by commas but a comma after the last parameter causes a error.

Create database
Connection creates if it does not exist
A few SQL commands to **.execute**
CREATE TABLE
ALTER TABLE
DROP TABLE
INSERT INTO table_name VALUE(vals.)
REPLACE search_str, sub_str, rep_with
UPDATE table_name SET col_name = new_value WHERE limiting conditions
DELETE FROM col_name WHERE
SELECT col_name FROM table WHERE
Data types (**Python:SQL**)
None:NULL **int:**INTEGER **float:**REAL
str:TEXT **bytes:**BLOB