BIG DADDY'S

PYTHON TOOLBOX For 3.5

Output: Format()

Four **basic** things to do to data :
1. Acquire,retrieve or create it.
2. Manipulate and apply it.
3. Store it.
4. Present it.

The first three employ many commands and functions. **Presentation** has mainly two: **print()** and **format()**. Of these, 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 the print() syntax:
print(objects, sep=' ', end='\n')

There are two basic **but separate "jobs" or functions** that format does, **AND** there are **two forms of the format syntax** that you can use in Python 3.5+. For a new user, conflicting examples can play with your head. The **jobs** are:

(1) **Format** the visual output of numbers and strings.
(2) Allow the programmer to **substitute** or **order** text and numbers in various string or numeric statements .

For formatting, examples of the 2 forms of syntax: (Both use the same commands.) Assume the statements below are in a print or string assignment and variable number=12345.67890:

```
(1) format(number,'0=+20,.3f')  #<- characters in teal are the mini-language
(2) '{:0=+20,.3f}'.format(number) # <-Note the {: sequence to start and } to end
...both yield the same result - both cases used THE SAME FORMAT SPEC
+000,000,012,345.679
+000,000,012,345.679
```

Can use string variable as format string.

What these symbols do: '**{:** open format string; **0** fill number with this character; **=** pad after sign but before number; **+** use a sign; **20** width in characters; **,** use commas to show thousands; **.3** 3 digit precision; **f** number type, 'f' is 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]

- **+, -(neg only) ," "space for +, - 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

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 rsulting from **printnums** unless otherwise noted. (Output with print() yields: 12345.6789 -12345.6721)

Function to generate the results:

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

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

Format string	Yields	Description
">15.2f"	Column Border ->| 12345.68| -12345.67|	two fixed decimal places *2
">15,.2f"	Column Border ->| 12,345.68| -12,345.67|	add comma separators
"> 15.2f"	Column Border ->| +12345.68| -12345.67|	Space before +, - negative
"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 *3
"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

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 ommited

String Format Substitution

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 curley 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 **curley 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 illeagle 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 curley 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.

```
(EX; positional)  print('{3}, {0}, {1}'.format('Abe','Bob','Cal','Dug'))
↳ Dug, Abe, Bob
(EX: numeric format) number = 100.1
print("{:<15e},{:<10g},{:<10.02%}".format(number, number, number))
↳   1.001000e+02   ,100.1     ,10010.00%
```

*1 With sign: + sign all, - neg only, space force leading space on + and sign neg
Conversion Flags: !s and !r apply str() and repr(*adds* '); str() does not add ''. **ex: print('{!r} had a little {!s}.'.format\('Mary','lamb'))**
↳ **'Mary' had a little lamb.** <-Note 'Mary'
'{0[x]}' selects the xth value in a tuple which formats names: **ex:** print ('**{0[3]}**'.format (**mytup**))
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 preceeding 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

BIG DADDY'S PYTHON TOOLBOX For 3.5

...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**", "**txList**" & "**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 txList:
    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)
txList=[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
txList=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 Seperated 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)

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 FRROM col_name WHERE
SELECT col_name FROM table WHERE
Data types (**Python**:SQL)
None:NULL **int**:INTEGER **float**:REAL
str:TEXT **bytes**:BLOB

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

comments and suggestions appreciated: john@johnoakey.com