**MSCF Python Programming Basics**

**Homework 2**

***Due At 11:59 pm US Eastern Daylight Time,***

***Sunday, July 3, 2022***

1. (40 points) **Formatted Output and lists**

As we’ve seen, the **print()** function is able to display human-readable output for all of the low-level scalar data types. **print()** also displays **list** and **tuple** objects in human-readable form. The arguments to **print()** (like the arguments to any function) must be separated with commas, and by default the **print()** function displays a single space character “in place of” each comma. For example:

**print(12, True, 1/7, 'hello', None)**

dislays:

**12 True 0.14285714285714285 hello None**

Often, we need more precise control over how the output is formatted. For example, we might need to display each value in a field that is 10 characters wide; with numeric values left-justified; **bool**, **str**, and **None** values right-justified; and only 3 digits of precision after the decimal point for a **float** value.

Python 3.9 offers four generations of formatted output “mini languages”. Here, we are just going to introduce part of one, which is reasonably easy to understand and remember, and which handles output field widths, justification, and precision. (The notation is similar to the **printf()** function’s *format strings* in the C programming language.)

The **str** type provides a **format()** method that returns a **str** object with values formatted into the returned **str** according to *format specifications*. The returned **str** object can be assigned to a variable, or displayed on the user’s screen with **print()**, or written to a text output file with *fout***.write()**. Here is a simple example (see file **format\_examples.py**):

**s = '{:10.3f}'.format(1 / 7)**

**print('0123456789')**

**print(s)**

Output:

**0123456789**

**0.143**

Here, the *curly braces* define a *replacement field*, that indicates where in the returned **str** the argument to the **format()** method should be placed. Here we have only one argument—the **float** result of the division **1 / 7**—and only one replacement field. Within the curly braces, **:10.3f** is a *format specification*, saying that the argument should be formatted in a **10**-character wide field, with **3** characters after the decimal point, in **f***ixed point* format (that is, with a decimal point but no exponent part). The variable **s** is assigned to refer to the returned string.

The **print('0123456789')** function call simply displays a character position counting string, as a visual aid. The **print(s)** function call displays the string returned from the **format()** method: we see that there are 5 blank space characters, followed by 5 digit and decimal point characters, for a total width of 10 characters as requested. There is a decimal point character followed by 3 digits (and no exponent part) also as requested. A leading 0 is included by default.

*Notice* that: (a) the displayed value is *rounded* to **0.143** rather than *truncated* to **0.142**; and, (b) the displayed value is *right*-justified in the 10-character output field, which is the default for numeric values.

In these next examples, the **'<'**, **'^'**, and **'>'** characters after the **':'** specify *left-justified*, *centered*, and *right-justified* output, respectively, and the **'e'** (rather than **'f'**) specifies **e***xponential* (scientific notation) output rather than fixed point. If no field width is specified, the “natural” width is the default; if no precision is specified, 6 is the default.

**val = 123 / 7**

**print(val, '\n') # append extra newline**

**print('012345678901234')**

**s = '{:<15.3f}'.format(val)**

**print(s)**

**s = '{:^15.3f}'.format(val)**

**print(s)**

**print('{:>15.3f}'.format(val))**

**print('{:.3f}'.format(val)) # "natural" width**

**print('{:15f}'.format(val)) # 6 after decimal point**

**print('{:f}'.format(val)) # "nat." width, 6 prec.**

**print('{:<15.3e}'.format(val))**

**print('{:^15.3e}'.format(val))**

**print('{:>15.3e}'.format(val))**

Output:

**17.571428571428573**

**012345678901234**

**17.571**

**17.571**

**17.571**

**17.571**

**17.571429**

**17.571429**

**1.757e+01**

**1.757e+01**

**1.757e+01**

For string (**str**) arguments, use the **'s'** format code; for integer (**int**) arguments, use **'d'** for **d***ecimal* (base 10) integer. Field widths make sense for **str** and **int**, but not precisions. Strings are *left*-justified by default; integers are *right*-justified by default. For example:

**print('012345678901234')**

**print('{:15s}'.format('hello')) # left-just. default**

**print('{:<15s}'.format('hello'))**

**print('{:^15s}'.format('hello'))**

**print('{:>15s}'.format('hello'))**

**print('{:15d}'.format(4321)) # right-just. default**

**print('{:<15d}'.format(4321))**

**print('{:^15d}'.format(4321))**

**print('{:>15d}'.format(4321))**

**print('{:>15.3d}'.format(4321))**

Output:

**012345678901234**

**hello**

**hello**

**hello**

**hello**

**4321**

**4321**

**4321**

**4321**

**...**

**ValueError: Precision not allowed ...**

A **bool** value can be displayed as an integer: 1 for **True** or 0 for **False**. There is no format code for **None**. A **bool** value or **None** can be explicitly converted to **str** and then displayed using the **'s'** format code. For example:

**print('0123456789')**

**print('{:<10d}'.format(True))**

**print('{:^10d}'.format(False))**

**print('{:>10s}'.format(str(True)))**

**print('{:<10s}'.format(str(False)))**

**print('{:^10s}'.format(str(None)))**

Output:

**0123456789**

**1**

**0**

**True**

**False**

**None**

The format string can contain *multiple* replacement fields; the number of arguments to the **format()** method must exactly match the number of replacement fields. Characters in the format string *not* contained within curly braces are preserved in the returned string as-is. For example:

**print('000000000011111111112222222222')**

**print('012345678901234567890123456789')**

**print('{:10s}{:^10s}{:>10s}'.format('how',**

**'are', 'you?'))**

**print('\nPrice of {:s} is ${:.2f}'.format(**

**'AAPL', 204.23))**

**# a list of tuples**

**t\_and\_p = [('GS', 207.90), ('AAPL', 204.23),**

**('X', 14.75), ('AMAT', 43.98)]**

**print('\n{:10s}{:>10s}'.format('Ticker', 'Price'))**

**for t, p in t\_and\_p:**

**print('{:10s}{:10.2f}'.format(t, p))**

Output:

**000000000011111111112222222222**

**012345678901234567890123456789**

**how are you?**

**Price of AAPL is $204.23**

**Ticker Price**

**GS 207.90**

**AAPL 204.23**

**X 14.75**

**AMAT 43.98**

*We have covered only a tiny fraction of the formatting facilities available in Python 3.9,*

and yet you can get a lot of mileage out of this formatting “micro mini language”. In

general, a *replacement field* within the *format string* for the **str** type’s **format()** method

can look like:

**{:***format\_spec***}**

where *format\_spec* can look like:

[**<^>**][*width*[**.***precision*]][**dfes**]

One of **<**, **^**, or **>** is optional; *width* is optional; **.***precision* (for **float**) is optional; and

one of the *format codes* **d**, **f**, **e**, or **s** specifies **d**ecimal integer, **f**ixed point, **e**xponential,

or **s**tring formatting.

1. (continued) **Formatted Output, File Input, and lists**
   1. **Processing expense records**

Using the Integrated Development Environment (IDE) of your choice (Spyder, PyCharm, IDLE, …) open the program file **hw2\_1\_a.py**. This program defines a **list** of **str** values named **expenses**, describing business expenses during February and March of 2017. The program uses a **for** loop to display the values (records) in **expenses**, one per line. Run the program to confirm this behavior.

The top of the file has comments giving the name of the program file and the names of the file’s authors. Add your homework team members to the authors comment. (In this course, every code file you create should always have the **File:** and **Author(s):** comments at the top.)

**# File: hw2\_1\_a.py**

**# Author(s):** *... team member names here ...*

Next, modify your program so that each record from **expenses** is displayed preceded by its record number and one space character. Record numbers start from 1 (for the column name header record), and each record number should be displayed in a field 4 characters wide. The first and last lines of output should look like:

1 Amount:Category:Date:Description

...

31 8.98:supply:20170325:Flair pens

You will notice that the fields in the **expenses** records are separated with colons, rather than commas, because the descriptions of many expenses (like “dinner, Tavern64”) include commas. (Also, we use triple single quotes for each record in **expenses**, because both single quote and double quote characters occur in some expense descriptions.)

The **str** type’s **split()** method splits a string into a **list** of substrings, separated with the character given as an argument to **split()**. (The splitting character itself is *not* included in the substrings.) Use **split(':')** to split each record from **expenses** into a **list** of 4 substrings, then for each record display:

the record number in a field 4 characters wide, followed by one space;

the amount in a field 8 characters wide, followed by one space;

the category in a field 10 characters wide, followed by one space;

the date in a field 10 characters wide, followed by one space;

and finally the description in a field of its “natural” width.

The first two and last two lines of output should look like:

1 Amount Category Date Description

2 5.25 supply 20170222 box of staples

...

30 284.23 util 20170323 Peoples Gas

31 8.98 supply 20170325 Flair pens

The dollar amounts and categories for the expenses do not look very good, since strings are left-justified in the output by default. Modify the output so that the amount and category columns are right-justified instead. The first two and last two lines of output should look like:

1 Amount Category Date Description

2 5.25 supply 20170222 box of staples

...

30 284.23 util 20170323 Peoples Gas

31 8.98 supply 20170325 Flair pens

***Make sure to save*** your **hw2\_1\_a.py** code file to submit as part of this homework.

* 1. **Simple Statistics on a list**

***Save a copy*** of your **hw2\_1\_a.py** as **hw2\_1\_b.py**. Modify the file name in the comment at the top of **hw2\_1\_b.py** to give the correct file name.

Modify your program so that, instead of producing the output described in part (1a) above, your program constructs a **list** of **float** values of all the expenses described in **expenses**. Display this **list** on the screen using a single **print()** function call. The output should look like this (we have left out all but the first 3 and last 3 values in this output):

[5.25, 79.81, 43.0, ..., 119.56, 284.23, 8.98]

Next, define the following functions (for this homework you ***must*** define these functions yourself: you ***may not*** use equivalent built-in functions or functions from some imported module. You ***may*** use the **len()** function to obtain the length of an iterable such as a **list**, and you may assume the iterable contains at least 2 items, and that all items are numeric):

**sum\_of\_vals()** that takes an *iterable* (such as a **list**) as its argument and returns the *sum* of the values from the iterable.

**mean\_val()** that takes an iterable as its argument and returns the *mean* of the values from the iterable.

**stdev\_of\_vals()** that takes an iterable as its argument and returns the *sample standard deviation* of the values from the iterable. (Be sure to compute and return the *sample* standard deviation, not the *population* standard deviation.)

**median\_val()** that takes an iterable as its argument and returns the *median* of the values from the iterable.

**min\_max\_vals()** that takes an iterable as its argument and returns a **tuple** in which the first item is the *minimum* value from the iterable, and the second item is the *maximum* value from the iterable.

Modify your program to display output in this format, based on the expense values from the **expenses** **list**:

Num of values: *NN*

Sum of values: *XXXXX.XX*

Mean value: *XXXXX.XX*

Std Deviation: *XXXXX.XX*

Median value: *XXXXX.XX*

Minimum value: *XXXXX.XX*

Maximum value: *XXXXX.XX*

Here, *NN* is an integer and *XXXXX.XX* is a fixed point number, 8 characters wide with 2 digits after the decimal point.

***Save*** your **hw2\_1\_b.py** code file to submit as part of this homework.

1. (60 points) **list**, **tuple**, **set**, and **dict**

Create a program file named **hw2\_2.py**. At the top of your file, in comment lines, specify the name of the file, the names of the authors, and the date on which you finished coding this file, like this:

**# File: hw2\_2.py**

**# Authors:** *... team member names here ...*

* 1. Create a variable **v1** that refers to a **list** containing these values, in this order: **1**, **2**, **3**, **4**, **5**, **6**. Use **print()** to display **v1**. The output should look like:

**[1, 2, 3, 4, 5, 6]**

* 1. Create a variable **t1** that refers to a **tuple** containing these values, in this order: **9**, **8**, **7**, **6**. Use **print()** to display **t1**. Output should be:

**(9, 8, 7, 6)**

* 1. Using *subscript notation* on both **v1** and **t1**, assign the 2nd item of **t1** to the 4th item of **v1** with. Use **print()** to display **v1**. Output should be:

**[1, 2, 3, 8, 5, 6]**

* 1. Using a *named method* of the **list** type, put the value **-3** at the end of **v1** (that is, at the end of the **list** to which **v1** is a reference). Display **v1**. Output should be:

**[1, 2, 3, 8, 5, 6, -3]**

* 1. Using the *repetition operator* (**\***), create a variable **v2** that refers to a **list** containing five **3**’s. Display **v2**. Output should be:

**[3, 3, 3, 3, 3]**

* 1. Using *slice notation* and the *append operator* (**+**), create a variable **v3** that refers to this combination of parts of **v1** and **v2**:

**[5, 6, -3, 3, 3, 3, 3, 3, 1, 2, 3, 8]**

Display **v3** to confirm this result.

* 1. Using a *named method* of the **list** type, **print()** the number of occurrences of the value **3** in **v3**.
  2. Using a **while** loop and a *named method* of the **list** type, eliminate all occurrences of **3** from **v3** *without* causing a runtime error. Display **v3** when done. Output should be:

**[5, 6, -3, 1, 2, 8]**

* 1. Using **t1** and *assignment to an empty slice* of **v3**, modify **v3** to contain:

**[5, 6, 9, 8, 7, 6, -3, 1, 2, 8]**

Display **v3** to confirm the change.

* 1. Using a *named method* of the **list** type and an appropriately defined **range()**, modify **v3** to contain:

**[5, 6, 9, 8, 7, 6, -3, 1, 2, 8, 1, 3, 5, 7]**

Display **v3** to confirm the change.

* 1. Using a *named method* of the **list** type, find the position (or subscript, or index) of the **9** in **v3**; create a variable named **p9** to refer to that position value. Find the position of the **2** in **v3**, and save that in a variable named **p2**. Then, using *assignment to a slice*, delete the values in **v3** from the **9** up to but not including the **2**. Display **v3** when done. Output should be:

**[5, 6, 2, 8, 1, 3, 5, 7]**

* 1. Sort the items in **v3** in ascending (technically, non-descending) order and display **v3**.
  2. Reverse the order of the items in **v3** and display **v3**.
  3. Using a **for** loop, iterate through the items in **v3** and, if an item is even, insert it at the front of **v3**. Display **v3** when done. Output should be:

**[2, 6, 8, 8, 7, 6, 5, 5, 3, 2, 1]**

* 1. Create a variable **t3** that refers to a **tuple** containing just the single item **4**. Display **t3** when done. (What should the output look like?)
  2. Using *multiple assignment*, write a single statement that swaps the first and last items in the **list** **v3**. (*Hint*: You do *not* need to know the length of **v3** in order to know the index of **v3**’s last item.) Display **v3** when done. Output should be:

**[1, 6, 8, 8, 7, 6, 5, 5, 3, 2, 2]**

* 1. Write a loop that tests whether each integer value in the range from **0** through **9** (inclusive) is in the **list** **v3**. For each integer value in the range, display a line of output such as:

**0: Not in v3**

**1: Is in v3**

* 1. Create an *empty* **set** named **s1**. Display **s1** when done. (What should the output be?)
  2. Using a loop on **list** **v3** and a *named method* of the **set** type, add each value in **v3** to the set **s1**. Display s1 when done. Output *may* (except for order) look like:

**{1, 2, 3, 5, 6, 7, 8}**

* 1. Create a **set** **s2** using these values: **0**, **2**, **4**, **7**, **8**, **9**, **10**. Display **s2** when done. Output *may* look like:

**{0, 2, 4, 7, 8, 9, 10}**

* 1. Using a *named method* of the **set** type, add the value **-2** to **s2**. Display **s2** when done. Output *may* look like:

**{0, 2, 4, 7, 8, 9, 10, -2}**

* 1. Using a *named method*, eliminate the value **8** from **s2**. Make sure not to cause a runtime error if **8** is not an element of **s2**. Then, using the same named method, eliminate the value **-1** from **s2**, being sure not to cause a runtime error. Display **s2** when done. Output *may* look like:

**{0, 2, 4, 7, 9, 10, -2}**

* 1. Using a *symbolic operator*, create a **set** **s1us2** containing the *union* of **s1** with **s2**. Display **s1us2** when done. Output *may* look like:

**{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -2}**

* 1. Using a *symbolic operator*, create a **set** **s1is2** containing the *intersection* of **s1** with **s2**. Display **s1is2** when done. Output *may* look like:

**{2, 7}**

* 1. Using a *symbolic operator*, create a **set** **s1ms2** containing the *set difference* of **s1** minus **s2**. Display **s1ms2** when done. Output *may* look like:

**{1, 3, 5, 6, 8}**

* 1. Using a *symbolic operator*, create a **set** **s2ms1** containing the *set difference* of **s2** minus **s1**. Display **s2ms1** when done. Output *may* look like:

**{0, 4, 9, 10, -2}**

* 1. Using a *symbolic operator*, create a **set** **s1sds2** containing the *symmetric difference* (“exclusive or”) of **s1** and **s2**. Display **s1sds2** when done. Output *may* look like:

**{0, 1, 3, 4, 5, 6, 8, 9, 10, -2}**

* 1. Define a function **are\_disjoint(sa, sb)** that takes references to two **set** objects as arguments, and that returns **True** if the two **set**s are disjoint, otherwise **False**. Add these tests of your function in your code, below the function definition:

**print('s1us2 and s1is2 are disjoint?',**

**are\_disjoint(s1us2, s1is2))**

**print('s1ms2 and s1is2 are disjoint?',**

**are\_disjoint(s1ms2, s1is2))**

**print('s1us2 and s2ms1 are disjoint?',**

**are\_disjoint(s1us2, s2ms1))**

**print('s1ms2 and s2ms1 are disjoint?',**

**are\_disjoint(s1ms2, s2ms1))**

* 1. Add these **print()** function calls to your code, with appropriate tests added using *symbolic* **set** operators. The first test is done for you; you will need to complete the rest.

**print('4 is an element of s1:', 4 in s1)**

**print('3 is NOT an element of s2:', ... )**

**print('s1is2 is a proper subset of s1us2:', ... )**

**print('the union of s1ms2 with s2ms1 is equal\n'**

**' to s1us2 minus s1is2:', ... )**

* 1. Create an *empty* **dict** named **c2count** (“character to count”). Display **c2count** when done. Output should be:

**{}**

* 1. Define a function, **str\_to\_c2count**, that takes a reference to a **str** as its argument, and that returns a **dict** mapping from each one-character substring of the argument to the count of occurrences of that character. *Hint*: recall the **in** operator for testing whether a key does or does not (yet) exist in a **dict**. Test your **str\_to\_c2count** function with this code:

**ret = str\_to\_c2count('this is a test')**

**print(ret)**

Output should be:

**{'t': 3, 'h': 1, 'i': 2, 's': 3, ' ': 3, 'a': 1, 'e': 1}**

* 1. Define a function, **str\_list\_to\_c2count**, that takes a reference to a **list**-of-**str** as its argument, and that returns a **dict** mapping from each one-character substring of the argument to the count of occurrences of that character. Test your **str\_list\_to\_c2count** function with this code:

**ret = str\_list\_to\_c2count(['hello', 'world'])**

**print(ret)**

Output should be:

**{'h': 1, 'e': 1, 'l': 3, 'o': 2, 'w': 1, 'r': 1, 'd': 1}**

* 1. Make a copy of the **expenses** **list** of **str** values from part 1 of this homework into your **hw2\_2.py** file. Test your **str\_list\_to\_c2count** function with this code:

**ret = str\_list\_to\_c2count(expenses)**

**print(ret)**

***And Finally***

Create a zip archive named **HW2\_Team\_*N*.zip**, where ***N*** is your team number, containing your **hw2\_1\_a.py**, **hw2\_1\_b.py**, and **hw2\_2.py** files. One team member should upload this zip archive to Canvas.