April 24, 2024

ICS3U Task 3
- Loops and Building Strings -

The purpose of this task is to practice using ***while-loops***, ***for-loops***, conditions, formatting output, and ***building up strings***. Important when assessing this task will be:

- Completeness and correctness (task complete, meeting specifications, and working)
- Design (minimal duplicate code, good logic and structure, best choices for loop structures)
- Clarity of code (whitespace, good variable names, documentation/comments, etc.)

Submission*: Tuesday May 7th by the end of class*
Submit **`task3_functions.py`** and **`task3_tester.py`** through the Task 3 assignment post in the Google Classroom. ***There will be a written follow-up assessment after this task is submitted.***

Overview
In this task you will write ***7*** functions in a file called **`task3_functions.py`** that use loops (`while` & `for`), nested loops, and building up strings. You will also complete an interactive tester (much like on Task 1), but with much more significant code to complete (again involving loops, etc.).

Compared to Task 1 & 2, the functions will require more problem solving. ***You can't code a solution if you don't understand and plan a solution to the problem first!*** You can work on the parts in any order but may wish to complete the tester elements as you complete the functions. ***Do not overlook the code that needs to be added to the tester file… it is <u>not</u> insignificant.***

Some Important Notes:

- All functions should all have properly formatted *docstrings* and *type hints* You should also have ***more internal documentation*** – comments which describe more complicated elements within the code, or add readability/structure. In addition, there should be a header comment at the top of the file with your name, date of completion, and a brief description of the program.
- You ***cannot*** use *any* programming concepts that we have not discussed yet in the ICS3U notes or which are defined in this handout. This is especially true regarding 'string *methods*', lists, dictionaries, etc. If in doubt, ask Mr. Foster!
- Be sure to **test thoroughly**, ensuring all valid cases are verified.
- Remember the important style elements we have addressed: variable names, whitespace (like paragraphs), etc. Look back at previous tasks' evaluation sheets, and the valuable feedback given.
- You should use the most appropriate type of loop (`while`/`for`) for the given situation and doing so will be part of your evaluation. You should also consider efficiency in terms of not doing more processing of values than necessary.
- Mr. Foster *wants* to help you! Ask good questions, show meaningful attempts, and get help when needed & warranted.
- As a guideline, Mr. Foster's ***entire*** commented & white-spaced solution for the 7 functions file was over 300 lines (not including any testing code), and ***a lot*** of that was in the two 'Braille' functions. The completed tester file was itself over 100 lines (excluding the large 'TO DO' comments, which you can remove once done).

Part 1 – `square_root()`

Have you ever wondered how it is that a calculator can calculate a square root, especially when the result is usually irrational (not a whole number or fraction)? To put it simply, they use a loop that repeatedly gets more and more precise, narrowing in on the true square root until it is 'close enough'. Surprisingly, this will only take a few steps to get more than 10-decimals of precision!

The following describes a >2000-year-old procedure to calculate the square root of a ***number***:

- Start with any positive number as an initial ***estimate***
- Calculate $\frac{estimate + \frac{number}{estimate}}{2}$ and replace your estimate with this new estimate
- Repeat the calculation above until the desired accuracy is reached

What should the ***starting estimate*** be? You could choose just **1**, but that can lead to quite a few extra steps as the number gets larger. A better estimate would be:

$$estimate = 5 \times 10^{\left(\frac{\log_{10}(number)}{2} - 1\right)}$$

To calculate $\log_{10}(number)$, use `math.log(number, 10)`.

What is the ***desired accuracy***? There is a function in the math module called **`isclose()`** that takes two numeric arguments and returns boolean `True` when the two values are 'relatively close' or `False` otherwise. 'Relatively close' means that their ratio is 0.000000001 away from 1.

To use this function here, pass it the current $estimate$, and the value of $\frac{number}{estimate}$.

You will continue to adjust your estimate as long as `isclose()` does not return `True`.

Write the **`square_root()`** function that takes a single float value as an argument and will calculate & return the square root of that number using the algorithm described above.

Part 2 – `digit_product_sum()`

What would happen if you added up the product of every digit in a number with every digit in that same number? *Lets find out!*

Write a function called **`digit_product_sum()`** that will take a single non-negative integer as an argument and will return the result of adding up each possible 'digit product', which would be each digit of the number multiplied by each digit.

For example, if the number was **93861**, we would need to add the total sum of these 25 products:

9 x 9	9 x 3	9 x 8	9 x 6	9 x 1
3 x 9	3 x 3	3 x 8	3 x 6	3 x 1
8 x 9	8 x 3	8 x 8	8 x 6	8 x 1
6 x 9	6 x 3	6 x 8	6 x 6	6 x 1
1 x 9	1 x 3	1 x 8	1 x 6	1 x 1

… which is *obviously* **729**.

So: `print(`**`digit_product_sum(93861)`**`)` would output **729**

Keep in mind that although the function must take an integer as an argument, you cannot iterate through an integer… but you *can* iterate through a string.

Part 3 – `text_to_braille()` and `braille_to_text()`

Braille is an alphabet used by visually impaired people. The 'letters' are represented as a set of raised bumps on a surface, which can be 'read' by passing the reader's fingers over them. This truly impressive system was invented by 15-year-old *Louis Braille* in 1824, who lost how own sight in an accident.
To keep it simple, each lowercase letter maps to a single Braille symbol, and vise-versa. We will ignore uppercase letters, numbers, and punctuation. These Braille symbols all have special Unicode values.

Mapping of each character to its Braille version								
Letter	**Braille**	**Escape Sequence**	**Unicode Value**		**Letter**	**Braille**	**Escape Sequence**	**Unicode Value**
a	⠁	\u2801	10241		**n**	⠝	\u281d	10269
b	⠃	\u2803	10243		**o**	⠕	\u2815	10261
c	⠉	\u2809	10249		**p**	⠏	\u280f	10255
d	⠙	\u2819	10265		**q**	⠟	\u281f	10271
e	⠑	\u2811	10257		**r**	⠗	\u2817	10263
f	⠋	\u280b	10251		**s**	⠎	\u280e	10254
g	⠛	\u281b	10267		**t**	⠞	\u281e	10270
h	⠓	\u2813	10259		**u**	⠥	\u2825	10277
i	⠊	\u280a	10250		**v**	⠧	\u2827	10279
j	⠚	\u281a	10266		**w**	⠺	\u283a	10298
k	⠅	\u2805	10245		**x**	⠭	\u282d	10285
l	⠇	\u2807	10247		**y**	⠽	\u283d	10301
m	⠍	\u280d	10253		**z**	⠵	\u2835	10293
**All space characters will remain as space characters*								

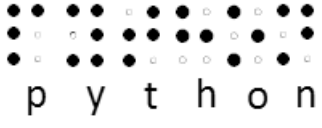
Write a function called **`text_to_braille()`** that will take a single string as an argument and will return the Braille version of it. You may assume that the string argument only contains lowercase letters and spaces (and do not need to verify this).

Also, write a function called **`braille_to_text()`** that will take a single string of Braille characters as an argument and will return the text version of it. You may assume that the string argument only contains Braille characters and spaces (and do not need to verify this).

Make sure you are careful don't make any careless mistakes with the mapping of characters to their values. Test this thoroughly.

Examples:

```
test = text_to_braille("try this out")
print(test)                          ← prints  ⠞⠗⠽ ⠞⠓⠊⠎ ⠕⠥⠞
print(braille_to_text(test))         ← prints  try this out
```

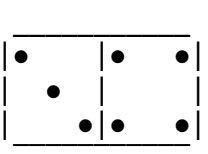


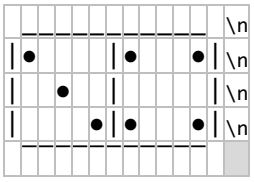
Part 4 – `domino_str()`

This function will take two integer arguments and will return a string that looks like a 'domino' when printed. The first argument represents the left-side of the domino, and the second argument represents the right-side of the domino. You may assume the argument values will be any integer from 0 to 6 *(inclusive)*. This domino string will always be the exact same size, regardless of the argument values.

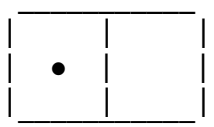
- The 'pips' (dots) are made of `'●'` using escape sequence `'\u25cf'` or `chr(9679)`
- The vertical lines are made using the vertical-bar character `'|'`
- The top line is made using underscores `'_'`
- The bottom line is made of `'‾'` using escape sequence `'\u203e'` or `chr(8254)`

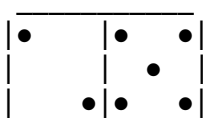
The exact format of a domino is shown here. The 'grid' version is to help you see the characters more clearly, including the newlines and blank spaces. This would result from calling **`domino_str(3, 4)`**:

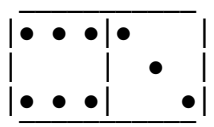




Other examples: **`domino_str(1, 0)`** **`domino_str(2, 5)`** **`domino_str(6, 3)`**:





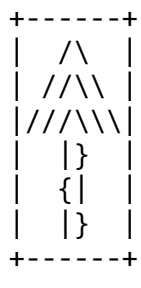


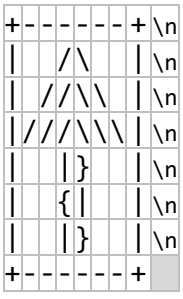
Part 5 – `tree_box()`

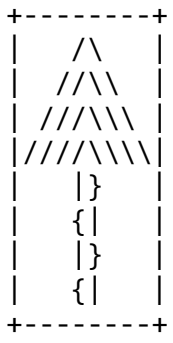
This function will take a single integer argument representing a 'size' of a ***tree*** and return a string that when printed looks somewhat like a pine tree inside a box. The tree has a top-half of 'foliage' and a bottom-half of trunk. Each of these halves is the same number of rows, which is equal to the size argument. You may assume the size will not be *negative* and will be an integer.

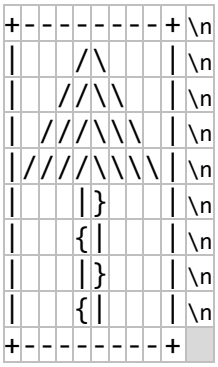
- The top half of the tree is made with `'/'` and `'\'` characters.
- The trunk is made with vertical bars `'|'`, `'}'` and `'{'` alternating on each row, starting with }.
- The entire tree is 'framed' in a box made of vertical bars `'|'`, plus-signs `'+'`, and dashes `'-'`.

Samples of `tree_box(3)` and `tree_box(4)` are shown below to show the exact formatting required (including spaces and newlines).



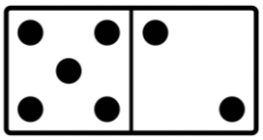






Consider approaching this function one element at time until it is complete. Start by making the top half of the tree. Then, make the alternating rows of the trunk. Finally, alter your code to add the frame around the whole thing.

Part 6 – `domino_stack()`



In the hot new 1-player game of '***domino stack***', a domino consists of a rectangular block with two numbers on it between 1 and 5 (inclusive). There are *infinite* copies of each domino in the '*vault*' with each possible number pairing. To start the game a domino is chosen to be the ***base*** and placed face-up on the table (called the ***stack***). This base domino can be specified by the player and isn't chosen at random. The player then selects a *random* domino from the vault. As long as that domino has a matching number on *at least* one side, it can be placed on top of the stack, ***lining up*** the matching number(s). If the player can play the domino, they do so and select another from the vault. If they cannot play the domino, the game is over.

Points are scored with each domino placed on the stack. When *both* numbers match, ***5*** points are scored. When only *one* number matches, ***2*** points are scored.

Write a function called **`domino_stack()`** that takes three integer arguments:

- the left-value of the base domino
- the right value of the base domino
- the target number of points the user wants to earn in a game

This function will build-up and return a string showing detailed results of each game as the user attempts to reach (or exceed) their target points. Each game will show the dominos played on the stack (including the orientation of the dominos). Once the game is over (the player's domino cannot be played) a *game summary* line is added to the bottom showing the game # and how many points were earned that game. The player will continue re-starting new games with the same base and with 0 points until the target number of points is reached (or surpassed) in a single game, at which point the entire summary string showing all the results is returned. The dominos are represented based on how `domino_str()` is defined, and will build 'up' on a stack, where base domino is at the bottom and the last domino is at the top.

Examples:
```
results = (domino_stack(1, 4, 10))
print(results)
```
may result in this →

```
 _____
|•  •|•  •|
|     |    |
|•  •|•  •|
 _____

 _____
|     |•  •|
|  •  |    |
|     |•  •|
 _____

Game #1 Points: 2
 _____
|•  •|•  •|
|  •  |    |
|•  •|•  •|
 _____

 _____
|     |•  •|
|  •  |    |
|     |•  •|
 _____

Game #2 Points: 2
 _____
|•    |•    |
|  •  |     |
|    •|     •|
 _____

 _____
|•    |•  •|
|  •  |  •  |
|    •|•  •|
 _____

 _____
|     |•  •|
|  •  |  •  |
|     |•  •|
 _____

 _____
|     |•  •|
|  •  |     |
|     |•  •|
 _____

 _____
|     |•  •|
|  •  |     |
|     |•  •|
 _____

Game #3 Points: 11
```

In the sample run here, notice the following:

- the base domino for each game (1, 4) is part of the output, and is the bottom one shown in the output
- each played domino is added ***above*** the previous, and is oriented so that the matching numbers 'line-up' with the one above it
- each game summary line shows the game number and the final number of points
- the final number of points could possibly go well over the target, since the game only ends once a domino cannot be played
- sometimes it could take *hundreds* of games before the target is reached

Interactive Tester

You are provided with a basic interactive tester file (**`task3_tester.py`**). In that file you have some simple menu interaction provided, some input statements, and some standard calls to the functions. However, you must complete some important ***validation*** code that repeatedly asks for input until the value(s) meet the specified criteria.

- ***Option 1*****: square root**
  An input statement is given for the number that the user wants to find the square root. You must add code that validates that the number is ***positive***.
  The remaining code for this option is complete and should not be edited.

- ***Option 2*****: digit product sum**
  Two input statements are given for the low and high end of the range of numbers to test `digit_product_sum()`. You must add code that first validates the low-end value to ensure it is not negative. You must then add validation so that the high-end value is at least as big as the low-end value.
  Next, you will call `digit_product_sum()` for every integer in the specified range. You will need to add code that will check for and store the ***highest*** digit product sum and its corresponding number. The final output should be formatted exactly as follows *(input & output shown in bold)*:

```
Enter a low number for the range: 9000
Enter a high number for the range: 12000
From 9000 to 12000, 9999 had the highest digit product sum (1296)
```

- ***Option 3*****: Braille**
  A sub-menu string and an input statement for the submenu are provided. You must add code that repeatedly asks for a menu entry as long as the user has not entered '1' or '2' for the sub-menu choice. *The remaining code for this option is complete and should not be edited.*

- ***Option 4*****: domino**
  There is no user input for this option. You will add code that will automatically generate and print all 49 possible dominoes that use the numbers 0 through 6 for each of the left and right sides.

- ***Option 5*****: tree box**
  An input statement is given for the size of the tree. You must add code that validates that the number is ***not negative***.
  The remaining code for this option is complete and should not be edited.

- ***Option 6*****: domino stack**
  There are 3 input statements given. After each input statement you must add validation code that ensures that the **left** number on the base domino is between 1 and 5 (inclusive), the **right** number is between 1 and 5, and the **target** points are a positive number.
  The remaining code for this option is complete and should not be edited.

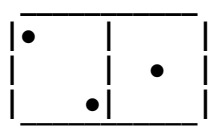
Sample Tester Run:

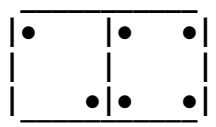
Take careful notice of the formatting of the output below. The user's input and program output are noted in **bold**. You can watch the sample video to see more cases where the validation of bad input takes place.

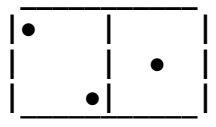
```
Test which?
1. Square Root
2. Digit Product Sum
3. Braille
4. Show All Dominos
5. Trees
6. Domino Stack
7. QUIT
Choice: 1
Number to square root: 456.789
Square root of 456.789 is 21.372622674814643

                    <menu omitted to save space in handout>
Choice: 2
Enter a low number for the range: 87
Enter a high number for the range: 1234
From 87 to 1234, 999 had the highest digit product sum (729)

                    <menu omitted to save space in handout>
Choice: 3

1. text to Braille
2. Braille to text
--> 1
Enter text: ics task three
To Braille: ⠊⠉⠎⠀⠞⠁⠎⠅⠀⠞⠓⠗⠑⠑
Back To Text: ics task three


                    <menu omitted to save space in handout>
Choice: 5
Tree size: 3
Your size-7 tree:
+--------------+
|      /\      |
|     //\\     |
|    ///\\\    |
|   ////\\\\   |
|  /////\\\\\  |
| //////\\\\\\ |
|///////\\\\\\\|
|      |}      |
|      {|      |
|      |}      |
|      {|      |
|      |}      |
|      {|      |
|      |}      |
+--------------+
```

<menu omitted to save space in handout>

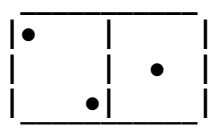
```
Choice: 6
Base domino left # (1 to 5): 2
Base domino right # (1 to 5): 1
Enter a target number of points: 8
```



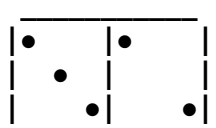


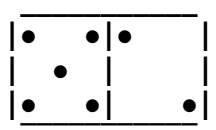


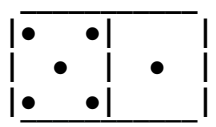
Game #1 Points: 4

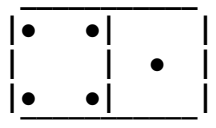


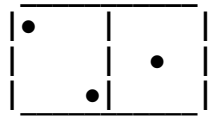
Game #2 Points: 0











Game #3 Points: 8

<menu omitted to save space in handout>

```
Choice: 7
```