COMP 202

Fall 2022

Assignment 2

Due: Friday, October 21st, 11:59 p.m.

Please read the entire PDF before starting. You must do this assignment individually.

Question 1:	15 points
Question 2:	85 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, allowing the TAs to provide better feedback and not waste time on administrative details.

To get full marks, you must follow all directions below:

- Make sure that all file names and function names are **spelled exactly** as described in this document. Otherwise, a 50% penalty per question will be applied.
- Make sure that your code **runs without errors**. Code with errors will receive a very low mark.
- Write your name and student ID in a comment at the top of all `.py` files you hand in.
- Name your variables appropriately. The purpose of each variable should be obvious from the name.
- Avoid writing repetitive code, but rather call helper functions! You are welcome to add additional functions if you think this can increase the readability of your code.
- Lines of code should NOT require the TA to scroll horizontally to read the whole thing. If the line is getting way too long, then it's best to split it up into two different statements.
- Vertical spacing is also important when writing code. Separate each block of code (also within a function) with an empty line.
- **Up to 30% can be removed for bad indentation of your code, omission of docstrings, and/or poor coding style as discussed in our lectures.**

Hints & tips

- **Start early.** Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. **Submit early and often**—a good rule of thumb is to submit every time you finish writing and testing a function.
- Write your code **incrementally**. Don't try to write everything at once. That never works well. Start off with something small and make sure that it works, then add to it gradually, making sure that it works every step of the way.
- Read these instructions and make sure you understand them thoroughly before you start. Ask questions if anything is unclear!
- Seek help when you get stuck! Check our discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already.

Talk to a TA during office hours if you are having difficulties with programming. Go to the instructor's office hours if you need extra help with understanding a part of the course material.

  - At the same time, beware not to post anything on the discussion board that might give away any part of your solution—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved. If you cannot think of a way to ask your question without giving away part of your solution, then please drop by our office hours.

- If you come to see us in office hours, please do not ask "Here is my program. What's wrong with it?" We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. Reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.

  - However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried in order to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

Revisions

None yet!

Part 1 (0 points): Warm-up

*Do **NOT** submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions. You are responsible for knowing all of the material in these questions.*

Warm-up Question 1 (0 points)

Write a function `swap` which takes as input two int values `x` and `y`. Your function should do 3 things:

1. Print the value of `x` and `y`
2. Swap the values of the variables `x` and `y`, so that whatever was in `x` is now in `y` and whatever was in `y` is now in `x`
3. Print the value of `x` and `y` again.

Here is an example of what your function should print when called (in the shell):

```
>>> swap(3, 4)
inside swap: x is:3 y is:4
inside swap: x is:4 y is:3
```

Warm-up Question 2 (0 points)

Consider the program you have just written. Create two global integer variables in the main body of your program. Call them `x` and `y`. Assign values to them and call the swap function you wrote in the previous part using `x` and `y` as input parameters.

After calling the `swap()` function —inside the main body— print the values of `x` and `y`. Are they different than before? Why or why not?

Warm-up Question 3 (0 points)

Create a function called `counting` that takes as input a positive integer and counts up to that number. For example:

```
>>> counting(10)
Counting up to 10: 1 2 3 4 5 6 7 8 9 10
```

Warm-up Question 4 (0 points)

Modify the last function by adding an additional input that represents the step size by which the function should be counting. For example:

```
>>> counting(25, 3)
Counting up to 25 with a step size of 3: 1 4 7 10 13 16 19 22 25
```

Warm-up Question 5 (0 points)

Write a function `replace_all` which takes as input a string and two characters. If the second and third input string do not contain exactly one character the function should raise a `ValueError`. Otherwise, the function returns the string composed by the same characters of the given string where all occurrences of the first given character are replaced by the second given character. For example, `replace_all("squirrel", "r" , "s")` returns the string `"squissel"`, while `replace_all("squirrel", "t", "a")` returns the string `"squirrel"`. Do not use the method `replace` to do this.

Warm-up Question 6 (0 points)

Write a module with the following global variables:

```
lower_alpha = "abcdefghijklmnopqrstuvwxyz"
upper_alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

In this module write a function `make_lower` which takes a string as input and returns a string containing the same characters as the input string, but all in lower case. For example, `make_lower("AppLE")` returns the string `"apple"`. Do not use the method `lower` to do this. Hint: note that characters from the English alphabet appear in the same position in the two global variables.

Part 2

The questions in this part of the assignment will be graded.

The main learning objectives for this assignment are:

- Correctly define and use simple functions.
- Solidify your understanding of the difference between `return` and `print`.
- Generate and use random numbers inside a program.
- Understand how to test functions that contain randomness.
- Correctly use loops and understand how to choose between a while and a for loop.
- Solidify your understanding of how to work with strings: how to check for membership, how to access characters, how to build a string with accumulator patterns.
- Learn how to use functions you have created in a different module.

Note that this assignment is designed for you to be practicing what you have learned in our lectures up to and including Lecture 14 (Debugging). For this reason, you are NOT allowed to use anything seen after that lecture or not seen in class at all. You will be heavily penalized if you do so.

For full marks, in addition to the points listed on page 1, make sure to add the appropriate documentation string (docstring) to *all* the functions you write. The docstring must contain the following:

- The type contract of the function.
- A description of what the function is expected to do.
- At least three (3) examples of calls to the function. You are allowed to use *at most one* example per function from this PDF.

Examples

For each question, we provide several **examples** of how your code should behave. All examples are given as if you were to call the functions from the shell.

When you upload your code to codePost, some of these examples will be run automatically to test that your code outputs the same as given in the example. However, **it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples.** When the time comes to grade your assignment, we will run additional, private tests that will use inputs not seen in the examples. You should make sure that your functions work for all the different possible scenarios. Also, when testing your code, know that mindlessly plugging in various different inputs is not enough—it's not the quantity of tests that matters, it's having tests that cover all of the possible scenarios, and that requires thinking about possible scenarios.

Furthermore, please note that your code files **should not contain any function calls in the main body of the program** (i.e., outside of any functions). Code that contains function calls in the main body **will automatically fail the tests on codePost and thus be heavily penalized**. It is OK to place function calls in the main body of your code for testing purposes, but if you do so, make certain that you remove them before submitting. You can also test your functions by calling them from the shell. Please review what you have learned in our lecture on Modules if you'd like to add code to your modules which executes only when you run your files.

Safe Assumptions

For all questions on this assignment, you can safely assume that the **type** of the inputs (both to the functions and those provided to the program by the user) will always be correct. For example, if a function takes as

input a string, you can assume that a string will always be provided to it during testing. The same goes for user input. At times you will be required to do some input validation, but this requirement will always be clearly stated. Otherwise, your functions should work with any possible input that respect the function's description. For example, if the description says that the function takes as input a positive integer, then it should work with all integers greater than 0. If it mentions an integer, then it should work for *any* integer. Make sure to test your functions for edge cases!

Code Repetition

One of the main principles of software development is DRY: Don't Repeat Yourself. One of the main ways we can avoid repeating ourselves in code is by writing functions, then calling the functions when necessary, instead of repeating the code contained within them. Please pay careful attention in the questions of this assignment to not repeat yourself, and instead call previously-defined functions whenever appropriate. As always, you can also add your own helper functions if need be, with the intention of reducing code repetition as much as possible.

Question 1: Dungeons! (15 points)

Remember how it was like living in the 80s? I don't but I heard a lot about it. This was a magical time when there was no internet as we know it and most computers could not even display graphics on screen. They could only display text. Doesn't it sound great?

I know your first question – 'but how can I play games if there is no graphics?' Well, it's great that you asked that, and it's also a big coincidence, because we are going to learn all about that in this question.

Games back in the day were completely text-based. One popular style of game was known as 'interactive fiction.' These games would tell a story in which you were the main character and progressed through a narrative by typing in commands. It was sort of like a Choose Your Own Adventure book, but instead of only having the choice to flip to different pages, the player could type in all kinds of commands and different things would happen, as long as the programmer had written code to handle them.

One of the first interactive fiction games ever made was called *Colossal Cave Adventure*, written by Will Crowther for his children in 1976. In the game, you explore a cave system. The cave is described to you through text, and you play the game by entering commands in natural language. You could type 'west' to move in a westward direction, perhaps to a different part of the cave; you could type 'get lamp' to pick up a lamp nearby (and could then subsequently light it, letting you see more of the cave); you could type 'throw eggs at troll' to vanquish an enemy blocking your path. The game had 66 rooms in total that you could navigate through, and 193 different verbs ('get', 'throw', etc.) you could use in different situations.

Crowther worked on the game as a hobby. His main job was in developing the ARPANET, one of the first computer networks and a precursor to the Internet. One day he went on vacation, leaving a copy of the game on the computer mainframe he had been working on. While he was away, people from across the network, from other companies and universities, began to find the game and play it. It was a massive hit. Don Woods, a graduate student at Stanford, would build upon the code and extend the game. It could be found everywhere in the late 1970s, and is considered to this day one of the most influential video games ever made.[1]

Wow! That was a big tangent. Anyway, for this question, you will write your own small interactive fiction game. To reduce the scope, we are going to write an 'escape room' interactive fiction game. Surely you know of escape rooms. Those places you go to with your friends and try to solve puzzles in a closed room in order to escape. Well - now we can play an escape room without leaving the comfort of our own home!

You will design your own escape room by writing a function `escape_room`, a void function that takes no inputs, in a file **`dungeon.py`**. You will come up with your own scenario, and implement a number of valid commands that the player could type in. Your function must satisfy the following requirements:

- When the function runs, a textual description of the escape room should be displayed to the screen. The description should list various objects in the room.
- The user should then be prompted to enter a command that interacts with an object in some way (e.g., 'examine table' or 'push button').
- If the user enters a valid command, some message relating to the object should display that contains some hint about how to escape the room; then the user should be prompted for another command.
- If the user enters an invalid command, a message should display alerting them that they have entered an invalid command; then the user should be prompted for another command.
- If the user enters a special command, deduced from the hints given to them by interacting with objects, then a message should print stating that they have escaped the room, and the function should return (i.e., the game should end).
- If the user enters 'list commands', then all valid commands should be displayed.

[1] You can read more about Colossal Cave Adventure, also known simply as *Adventure*, at the following link (this same site also contains articles on many other influential interactive fiction games): `https://if50.substack.com/p/1976-adventure`

- Commands should be able to be expressed in different ways. For instance, 'examine book', 'look at book', 'read book' should all be valid for interacting with a book. Furthermore, the commands should be able to be inferred by the room description (e.g., if an object is mentioned in the room, you should be able to examine it, possibly pick it up, and so on, depending on the object).
- Commands typed in the user by should be case-insensitive. That is, 'Examine book', 'EXAMINE BOOK' and 'eXaMiNe BoOk' should all do the same thing.
- There must be at least three objects in the room that the player can interact with in some way (by examining, etc.).
- Note that although this function is open-ended, we will as usual be paying close attention to code style. Make sure to follow our style guidelines as discussed in class, including providing a docstring in the usual format and avoiding repetition as much as possible.

Finally, at the top of your code file, define three global variables: `ROOM_NAME`, a string for the name of your escape room; `AUTHOR`, a string for your name or nickname; and `PUBLIC`, a boolean indicating whether you would be comfortable for the room to be displayed publicly in some fashion along with your `AUTHOR` name.

Any submission meeting these requirements will obtain full marks, but you are encouraged to go beyond them. Our TAs will be showcasing their favourite submissions in the Slack.

Here is an example of an escape room. Note that this example is too simple for our requirements – it has only two objects. Also, do not simply copy the example; you should come up with your own idea!

```
>>> escape_room()
You have walked deep into a cave, looking for treasure. You wander down a passage that turns
out to be a dead-end, and all of a sudden you hear the rumble of rocks from behind you. You
are trapped!

In front of you, the rock face is smooth and glistens with moisture. Moss of many different
hues grows on the surface. There is an eerie silence, punctuated only by the drops of water
from the ceiling into a small pond next to you. What do you do?

> list commands
examine moss
examine pond

> examine moss
You look at the rock wall closely. Clumps of moss grow in strange formations. Most are green,
but there is a small purple area. It's a very strange shape, almost like a keyhole.

> examine pond
You kneel down and peer into the pond. In the reflection of the water, you see yourself with
a key in your hand. Your reflection then smiles and puts the key in their pocket. Startled,
you stand up and feel inside your pocket -- there is a key there! If only you knew where to
use it!

> put key in keyhole
A secret door opens in the rock wall...you climb through and escape!
```

Question 2: Treasure! (85 points)

In this question, we will go on a hunt for treasure!!! We will follow the trail in a treasure map, which we will represent as a big string. Here is an example treasure map string:

```
'..vv.<>..'
```

A treasure map string represents a rectangular grid comprised of rows and columns. A map string will have a width and height that will be provided alongside it. For instance, if we are told the above map string has width 3 and height 3, then the map contained in the string would be laid out like this:

```
..v
v.<
>..
```

That is, the first set of three characters of the map string forms the first row, the second set of three characters forms the second row, and the third set of three characters forms the third row.

We call this type of treasure map string (`'..vv.<>..'`) a **two-dimensional map** because it has a width and height (rows and columns). We can also construct a **three-dimensional (or 3D) map**. A 3D map has a width, height and depth. A 3D map string comprises several 2D map strings. For instance, here is a 3D map:

```
'..vv.<>....vv.<>..'
```

In this case, it is simply two copies of our 2D map from before, one after the other, in the same string. We can say this 3D map has width 3, height 3 and depth 2. The map can be displayed as follows (each 2D map printed one after the other, with an empty line between each):

```
..v
v.<
>..

..v
v.<
>..
```

We can also have a **four-dimensional (or 4D)** treasure map. A 4D treasure map is....no, just joking about that. :) We will only have 2D and 3D maps for now.

A treasure map can be of any rectangular size (not necessarily 3x3 nor square), and can contain different symbols. There are movement symbols, an empty symbol, a treasure symbol and a breadcrumb symbol. Our goal when given a treasure map is to follow the trail in it, character by character, and the symbols will tell us how to follow the trail.

Here are the different kinds of symbols we may encounter in a map:

- Movement symbols, typically the `'>'`, `'<'`, `'v'` and `'^'` characters, tell us in what direction to move the next. If we see one of these characters, we will move to the symbol to the right, to the left, below, or above the current character, respectively. If moving in this direction would cause us to go 'off the map' (e.g., trying to go right when you're already on the right-most column of the map), then we will instead 'wrap around', meaning that we will go to the first column of the same row if we tried to go too far to the right, or the last column of the same row if we tried to go too far to the left, etc.

  There are also two more movement symbols for 3D maps (described further below).

- The empty symbol, typically the `'.'` character, means to continue going in the same direction that we were going in when we got to this character.

- The treasure symbol, typically the `'+'` character, operates the same as the empty symbol, except that we also add one to our count of the treasure we have found while following the trail.
- Finally, the breadcrumb symbol, typically the `'X'` character, is a character that we will put in the map: as we follow the trail, we will replace movement symbols by this symbol after we visit them, to indicate that we have been there. When following a trail, if we encounter a breadcrumb, then we know we need to stop, because otherwise we will start going in circles. (Note: We only want to change movement symbols that we have visited into breadcrumb symbols; we don't want to change any other symbols.)

Let's say we wanted to follow the trail in the following treasure map, and decided to start in the top-left corner (or row 0 column 0 if we start counting at zero, like programmers should):

```
>.v
v+<
>..
```

The character at that position is a `'>'`, so we would start moving right, past the `'.'`, to the `'v'`, then we would move down, then left twice (picking up a treasure along the way), then down, then right twice until we got to the bottom-right corner. At this point, since there is a `'.'`, which means continue moving in the same direction, we would move again to the right, which would take us off the map. Therefore, we will 'wrap around' and come back to the `'>'` character (row 2 column 0).

At this point, since we have reached a tile that we have previously encountered (and we know that we have since we had changed it into a breadcrumb after we had visited it), we will stop. We will have picked up one treasure in total. And our map will look like this at the end:

```
X.X
X+X
X..
```

In this case, we visited all the movement symbols in the map, so they were all changed to breadcrumbs. But in general, we may visit only a few movement symbols – it all depends on the map.

With **3D treasure maps**, we introduce two more movement symbols, typically the `'*'` and `'|'` characters. A `'*'` symbol represents a hole. If we get to a `'*'`, then we should 'fall through' to the next map of the 3D map (i.e., one map deeper in the string), keeping our same row and column position. Likewise, a `'|'` symbol represents a ladder. If we get to a `'|'`, then we should 'climb up' to the previous map of the 3D map (i.e., the map at one depth greater).

Here is an example of following a trail in a 3D map. First, consider the following map string:

`'>+v..*..<...v.<*.|vvv+v+>.|'`

If we say that this map has a width of 3, height of 3 and depth of 3, then we can display it as follows:

```
>+v
..*
..<

...
v.<
*.|

vvv
+v+
>.|
```

If we start following this trail at map 0 row 0 col 0, meaning the top-left corner of the first map, then we will see a `'>'`, go right twice, go down, then fall down a hole to map 1 row 1 col 2 (again, starting counting at zero). That character on map 1 is a `'<'`, so we will go left twice, down, fall down the hole, go right twice (now on map 2), encounter a ladder (`'|'`), climb up, climb up again, then go left twice, wrap around back to the `'<'` and stop since we have been there already. So we will pick up two treasures in total and our completed map will look as follows:

```
X+X
..X
..X

...
X.X
X.X

vvv
+v+
X.X
```

(You'll note in this case we did not end up visiting all the movement symbols – some in map 2 remain as they are and are not changed to breadcrumb symbols because we did not visit them.)

Now that we know a bit about treasure map strings and following trails, we can start to write our code. We will begin by writing some small helper functions, in a file **`treasure_utils.py`**.

First, in the file, define the following global variables which you should use when appropriate in your functions:

```
MOVEMENT_SYMBOLS = '><v^'

EMPTY_SYMBOL = '.'

TREASURE_SYMBOL = '+'

BREADCRUMB_SYMBOL = 'X'

MOVEMENT_SYMBOLS_3D = '*|'
```

As in Assignment 1, we may decide to modify the values of these variables in our tests. Your code should work for any characters, not just the initial values set above. What you can assume is that the position of the character in each string will always correspond to its meaning. That is, e.g., the first character of `MOVEMENT_SYMBOLS` will always be the symbol that means to move to the right, though we may change it from a `'>'` character to something else.

Then write the following functions. The first few will deal only with 2D treasure maps, while the later ones will deal with 3D ones.

Note: Many of these functions take width and height (and depth) inputs in addition to a treasure map string. You can assume that the width, height and depth will always match the number of characters in the map string. You can also assume that the string will be a valid treasure map string of non-negative length containing only characters contained in the global variables.

- `get_nth_row_from_map`: takes a treasure map string, integer n and integer width and height as inputs. Returns the n'th row of the treasure map (remember that we start counting at zero, so $n = 0$ means the first row in the treasure map string). If n is not a valid row (i.e., outside the bounds of the number of rows), return an empty string.

  Hint: For this and some subsequent functions, it can help to draw on paper a string, section it into different rows, and think about what the indices for each row will be.

```
>>> get_nth_row_from_map('^..>>>..v', 1, 3, 3)
'>>>'

>>> get_nth_row_from_map('......', 0, 2, 3)
'..'
```

- **print_treasure_map**: takes a treasure map string and integer width and height as inputs and returns nothing. Prints out the treasure map with each row on its own line.

```
>>> print_treasure_map('<..vvv..^', 3, 3)
<..
vvv
..^

>>> print_treasure_map('<..vvv..', 2, 4)
<.
.v
vv
..
```

- **change_char_in_map**: takes a treasure map string, integer row and column index, character c, and integer width and height as inputs. Returns a copy of the given treasure map string but with the character at the given row and column index replaced by c. If either or both of the indices are out of bounds of the map, return the input string unchanged.

```
>>> change_char_in_map('.........', 1, 1, 'X', 3, 3)
'....X....'
```

- **get_proportion_travelled**: takes a treasure map string as input, and returns as a float the percentage (between 0 and 1) of the map that was travelled (i.e., the number of breadcrumb symbols in the map), rounded to two decimal places.

```
>>> get_proportion_travelled('.X..X.XX.')
0.44
```

- **get_nth_map_from_3D_map**: takes a 3D treasure map string, integer n and integer width, height and depth as inputs. Returns the n'th map of the 3D treasure map. (When $n = 0$, e.g., return the first map in the string.) If n is not a valid map index (i.e., outside the bounds of the number of maps), return an empty string.

```
>>> get_nth_map_from_3D_map('.X.XXX.X..v.vXv.v.', 0, 3, 3, 2)
'.X.XXX.X.'

>>> get_nth_map_from_3D_map('.X.XXX.X..v.vXv.v.', 1, 3, 3, 2)
'.v.vXv.v.'
```

- `print_3D_treasure_map`: takes a 3D treasure map string and integer width, height and depth as inputs and returns nothing. Prints out the treasure map with each row on its own line, and each map separated by a blank line. Make sure that there is **no blank line** at the end of your output.

```
>>> print_3D_treasure_map('.X.XXX.X..v.vXv.v.', 3, 3, 2)
.X.
XXX
.X.

.v.
vXv
.v.
```

- `change_char_in_3D_map`: takes a 3D treasure map string, integer row, column and depth index, character *c*, and integer width, height and depth as inputs. Returns a copy of the given 3D treasure map string but with the character at the given row, column and depth index replaced by *c*. If any of the indices are out of bounds of the map, return the input string unchanged.

```
>>> change_char_in_3D_map('.X.XXX.X..v.vXv.v.', 0, 0, 0, '#', 3, 3, 2)
'#X.XXX.X..v.vXv.v.'
```

We are now ready to write functions that deal with generating treasure maps and following trails. Write the following functions in a file **`treasure.py`**.

- `generate_treasure_map_row`: takes a positive integer width and a boolean indicating if a 3D map is being generated as inputs. Creates and returns a single row of the given width (as a string) of a treasure map. Each character of the string should have a $\frac{5}{6}$ chance to be one of the four basic movement symbols and a $\frac{1}{6}$ chance to be a breadcrumb symbol. If the boolean input is `True`, then there should be a 50% chance that one character in the row (only one, not more) is replaced by a 3D movement symbol (the hole or the ladder).

  Note: Your function does not need to produce the exact output as shown in the examples below for this function and for the subsequent generation function. That is because, although we set the random seed, the random numbers produced are determined not only by the seed, but by the particular functions called from the random module, the order in which they are called, and the arguments given to them. We cannot specify all these things to you without giving you the exact code for the function. Therefore, we simply provide these examples to show you the kind of output the function should produce, but we will not check that your function returns the same. Instead, we will check that whatever is returned by your function satisfies the criteria above.

```
>>> random.seed(9001)
>>> generate_treasure_map_row(10, False)
'^>^>..^>v.'

>>> random.seed(9001)
>>> generate_treasure_map_row(10, True)
'^>^>..^>v|'
```

- `generate_treasure_map`: takes an integer width and height and a boolean indicating if a 3D map is being generated as inputs. Creates and returns a treasure map of the given width and height (as a string). The occurrence probabilities of the characters in each row should be as is written above. The first character (row 0, col 0) of the map must be a right-pointing movement symbol.

```
>>> random.seed(9001)
>>> generate_treasure_map(3, 3, False)
'>>^>..^>v'

>>> random.seed(9001)
>>> generate_treasure_map(9, 8, True)
'>>^>..^|v|<<v.^^>>^v^<^^^>v<>>^v^|.^v*<<<^<<><.^^v.v.*v^<<<^<>>^.>v>.|>.'
```

- `generate_3D_treasure_map`: takes a positive integer width, height and depth as inputs. Creates and returns a 3D treasure map of the given width, height and depth (as a string). The occurrence probabilities of the characters in each row of each map should be as is written above. The first character (row 0, col 0) of the first map (map 0) must be a right-pointing movement symbol.

```
>>> random.seed(9001)
>>> generate_3D_treasure_map(3, 3, 3)
'>>|^>|.*v>|^v.*^v*>^^^v<.>|'

>>> random.seed(9001)
>>> generate_3D_treasure_map(4, 6, 3)
'>>^>^>v|<<<v|^>>^v^<^^^>><|><^*.v*<<>*^.><*^v<*v><<*><v^>.<|..^^vv>^<>^<'
```

- `follow_trail`: takes a 3D treasure map string, starting row, column and depth index (all integers), integer width, height and depth of the map, and number of tiles to travel (integer) as inputs. Follows the trail in the map, starting at the given row, column and depth index. Stops when encountering a tile that has been previously encountered, or when the specified number of tiles has been travelled (whichever comes first), prints the number of treasures collected in the format `Treasures collected:` n (where n is the number collected) and the number of symbols visited in the format `Symbols visited:` n and returns the travelled map (i.e., where all movement symbols which were followed are replaced by breadcrumb symbols).

  If the number of tiles to travel is `-1`, then do not stop following the trail until encountering a tile that has been previously encountered.

  If any of the starting indices are out of bounds of the map, return the input string unchanged.

  Hint: This is a complex function. To begin, you could consider writing the function to only follow trails in 2D maps (or even 1D rows). Then, when you have that working, you can extend it to 3D maps.

  Hint: Due to the complexity of the function, you can imagine that there are many different kinds of treasure maps that could be given to this function. We provide four examples below, but it is quite important for you to come up with many of your own examples to make sure your function works with any input that satisfies the instructions, not just the ones below; this is true for all functions in our assignments, particularly so for this one!

```
>>> follow_trail('>+....', 0, 0, 0, 3, 2, 1, 3)
Treasures collected: 1
Symbols visited: 3
'X+....'

>>> follow_trail('>>v..v', 0, 0, 0, 3, 2, 1, 1)
Treasures collected: 0
```

```
Symbols visited: 1
'X>v..v'

>>> follow_trail('>>v..v', 0, 0, 0, 3, 2, 1, 2)
Treasures collected: 0
Symbols visited: 2
'XXv..v'

>>> follow_trail('>>v..v', 0, 0, 0, 3, 2, 1, 100)
Treasures collected: 0
Symbols visited: 5
'XXX..X'
```

What To Submit

You must submit all your files on codePost (https://codepost.io/). The files you should submit are listed below. Any deviation from these requirements may lead to lost marks.

`dungeon.py`
`treasure_utils.py`
`treasure.py`
`README.txt` In this file, you can tell the TA about any issues you ran into while doing this assignment.
Remember that this assignment like all others is an `individual` assignment and must represent the entirety of your own work. You are permitted to verbally discuss it with your peers, as long as no written notes are taken. If you do discuss it with anyone, please make note of those people in this `README.txt` file. If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.

You may make as many submissions as you like, but we will only grade your final submission (all prior ones are automatically deleted).

Note: If you are having trouble, make sure the names of your files are exactly as written above.

Assignment debriefing

In the week(s) following the due date for this assignment, you will be asked to meet with a TA for a 10-20 minute meeting (exact duration TBD). In this meeting, the TA will grade your submission and discuss with you what you should improve for future assignments.

Only your code will determine your grade. You will not be able to provide any clarifications or extra information in order to improve your grade. However, you will have the opportunity to ask for clarifications regarding your grade.

You may also be asked during the meeting to explain portions of your code. Answers to these questions will not be used to determine your grade, but inability to explain your code may be used as evidence to support a charge of plagiarism later in the term.

Details on how to schedule a meeting with the TA will be shared with you in the days following the due date of the assignment.

If you do not attend a meeting, you will receive a 0 for your assignment.