Lab 05: A Maze Without Walls (part 1)

In this puzzle, you are given a grid of numbers, presented as a list of lists. the number in each grid square tells you how many steps you can move in one direction. For example, from a square with a number 2, you can move two squares up, two squares down, two square right, or two squares left. The goal, like in a maze is get from a start position to an end position.

Represent the Puzzle

When dealing with a list of lists, if can get a little confusing about how to access individual elements. If we have `L = [[1,2], 3,4]` , then `L[0][1]` is `2` . The first index is the row, starting from zero. Increasing this index moves down the grid (this is also how matrices are indexed). The second index is the column number, starting at zero. As a result, it makes sense to indicate a position in the grid using a tuple `(r, c)` , where `r` is the row and `c` is the column.

We will build a puzzle class with the following ADT.

- `onboard(position)` - return `True` if the (r,c)-pair `position` is within the bounds of the grid. Return `False` otherwise.
- `__getitem__(position)` - return the number stored in the (r,c)-pair `position` .
- `rdsolve(start, end)` - return `True` if it is possible to get from the position `start` to the position `end` using only right and down moves. Return `False` otherwise.
- `solve(start, end)` - return `True` if it is possible to get from the position `start` to the position `end` . Return `False` otherwise.

Start by writing a class called `Puzzle` and store it in a file called `puzzle.py` . Write an `__init__` method that takes a list of lists of integers. It should store these lists internally. The initializer should `raise ValueError` if the lists do not all have the same length (only rectangular puzzles are allowed).

Next, implement the `onboard` method to check if a given row and column (given as a tuple of ints) is within the bounds of the input lists.

`__getitem__` is great!

Often, we know roughly how we want code to work, before we write it. In this case, we want to make

a puzzle and then we'd like to look at the cells using a nice notation. We'd like code like the following to work.

```
puz = Puzzle([[1,1,2],[1,1,0]])
assert(puz[0,2] == 2)
```

Note, we're putting the row and column in the square brackets. Python automatically turns pairs like that into tuples, so `puz[0,2]` is the same as writing `puz[(0,2)]`, but it looks nicer without the parentheses. To use square brackets with your own classes, you implement a magic method called `__getitem__`. Here is a toy example.

```
class Foo:
    def __getitem__(self, index):
        return index + 1

f = Foo()
print(f[139])
print(f[2.14])
```

In our case, the type of `index` will be `tuple` and it will have both a row and a column. A nice way to **unpack** a tuple is to do the following.

```
mytuple = (5, 12, 13)
side1, side2, hypotenuse = mytuple
```

The above assignment creates three variables and assigns them the values of the three elements of the tuple. In the case of row and column positions as tuples, we might write `r, c = position`. This splits the tuple into two variables and is much cleaner than using indices: `position[0]` and `position[1]`.

The Only Right and Down Version

To warm, up we're going to solve the problem when it's only allowed to move right and down. This will be computed in the `rdsolve` method that tests if there exists a path from `start` to `end` using only right and down moves.

As with all recursive algorithms, start with the base case.

Don't forget that you're stuck if you land on a zero that is not the end.

Now, the heart of any recursive solution is the realization that the problem can be **reduced** to a *smaller* instance of the same problem. In this case, there are two possible positions you can get to from the start position (by moving right or down). If there is a solution from either of these positions, then there is a solution from start. This is a natural choice for recursion.

Memoize

Now, implement `solve`.
This will be trickier than `rdsolve` because there is no chance of getting stuck in a loop if you only move right and down. We will want to have a way to make sure that we don't get into the recursion version of an infinite loop (which generally results in a `RecursionError`). A handy trick to solve this problem is to store a `set` that has all the positions we have already visited. Then, each time we visit a new cell, we add it to the set. Each time we make a recursive call, we pass along this set of visited positions. (Be careful to modify the set rather than making copies.) If the start position is already in the `visited` set, we need not continue our search.

This approach is called **memoization**. That's not a typo. See the recursion chapter of the book for more info.

The `solve` method will take the `visited` set as an argument. The default value should be set to `None`.

Important: The following is bad. Don't do it.

```
def solve(self, start, end, visited = set()):
    ...
```

The use of `visited = set()` is a common mistake. It might seem like a handy way to initialize `visited` to an empty set, but it doesn't work twice. The problem is that this initialization happens only when the function is defined and not every time the function is called without the `visited` argument. The second time it is called, it will reuse the set from the previous call and you generally don't want that. Instead, set the default to `None` and initialize `visited` to an empty set if it is `None`.

A note about testing exceptions

Since the `__init__` method should raise an exception if the input list of lists is not square, you might be curious to know how to test this.

In the supplied test code you will find the following test.

```
def testinitcheckforrect(self):
    with self.assertRaises(ValueError):
        Puzzle([[1,2], [1,2,3]])
```

This test uses the `with` keyword. This is an example of a **context manager**. The idea is to introduce a block of code where some other code is guaranteed to be executed before and afterwards. In this case, the `assertRaises` function gives a context manager that will catch the expected exception and signal a failed test if it doesn't find it. The test passes if the exception is raised.

You will also see context managers when reading files. In that case, the context manager is used to make sure that files are closed properly after being opened.

Summary

For this lab, you should implement all the methods in the puzzle ADT given above. The methods `solve` and `rdsolve` should be recursive and should run in linear time.