Due Date: Friday October 30th, 11:59pm.

- The purpose of this assignment is to practice building, inspecting, and modifying 2D lists effectively.
- This often requires nested for-loops, but not always. It involves thinking of the structure like an N x M matrix of labeled spots, each with a row- and column-index. As before, we are restricting ourselves to the most basic functionalities, and any others that you want, you should implement yourself.
- You will turn in a single python file following our naming convention: `netID_Lab#_P#.py`
- Include the following in comments at the top of your file: Name, G#, lecture and lab sections, and any references or comments we ought to know.
- Use the Piazza discussion forums (and professor/TA office hours) to obtain assistance. Any post with project code in it must be made private, visible to "Instructors" and you alone. Have a specific question ready, and both show what you're thinking and show what you've tried independently before you got stuck. We will prod for more details if needed.

Background:

Two-dimensional lists aren't conceptually more difficult than single-dimension lists, but in practice the nested loops and more complex traversals and interactions merit some extra practice. We will create grids of cells and simulate Conway's "Game of Life", which is not really a game but a cellular automaton.

Procedure

- download this samples file, which has typed out many example `GridString`s and some extra functionality that makes the project animated:
  - http://muddsnyder.com/112/projects/p4provided.py
- Implement the functions described later in this document, using the following testing file as you go.
  - http://muddsnyder.com/112/projects/tester4p.py **(Now Available!)**
  - Invoke it as with prior assignments: `python3 tester4p.py yourcode.py`
  - NEW! limit what's tested – name functions to test (e.g. just `read_coords` and `get_dimensions`):
    `python3 tester4p.py yourcode.py read_coords get_dimensions`

- **Remember: your grade is significantly based on passing test cases – try to completely finish individual functions before moving on. The easiest way to implement many functions is to call the earlier/easier functions, so it will pay off twice to complete functions before moving on. Don't let yourself be "almost done" with a function, but be missing all the test cases!**

Allowed/Disallowed Things

You may only use the following things. You may ask about adding things to a list, but only Dr. Zhong or Dr. Snyder has the authority to add to the list, and we are hesitant to add anything. If you use something disallowed, it's not an honor code violation! You just won't get points for using it.

- no modules may be imported.
- basic statements, variables, operators, del, indexing, slicing, are all allowed
- any form of control flow we've covered is allowed (if/else, loops, etc)
- functions: range(), len(), int(), str()
- methods: s.split(), s.join(), s.pop(), xs.append(), xs.extend(), xs.insert() s.format()
- calling other functions of the project (and your own helper functions). ***Please do this!*** ☺

Conway's Game of Life

The wiki page is a decent introduction, too: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Conway's Game of Life is actually a simulation. We start with a grid of squares (any number of rows and columns). In each spot, we consider it "alive" or "dead". Each cell has 8 neighbors surrounding it (the horizontal, vertical, and diagonal cells). We consider the grid to be one "generation", and the next generation is created by considering whether a cell was living or dead, and how many living neighbors it had, to determine its next state:

- a dead cell must have exactly 3 living neighbors to be born (become alive); otherwise, it remains dead.
- a living cell must have exactly 2 or 3 living neighbors to survive; otherwise, it dies from isolation or overcrowding.

These rules can be described as "B3/S23", meaning "born with 3 living neighbors, survives with 2 to 3 neighbors." We will be using these exact rules, though other rules are possible.

Indexing a Grid

(0,0)	(0,1)	…	**(0,M-1)**
(1,0)	…	…	(1,M-1)
…	…	…	…
(N-1,0)	(N-1,1)	…	**(N-1,M-1)**

When we have a grid, we have two dimensions. The first dimension indicates the row (top row to bottom row), and the second dimension indicates the column (left column to right column). Thus with N rows and M columns, we have a grid with indexes as shown to the right.

If we saw the grid below (left), with live cells in blue, then it has live cells at (1,2), (1,3), (2,1), (2,2), and (3,2). The next generation is shown to the right of it. Think through the B3/S23 rules for a few cells.

original grid

(0,0)				(0,4)
		(1,2)	(1,3)	
	(2,1)	(2,2)		
(3,0)		(3,2)		(3,4)

next generation (applying B3/S23 to original grid)

(0,0)				(0,4)
	(1,1)	(1,2)	(1,3)	
	(2,1)			
(3,0)	(3,1)	(3,2)		(3,4)

When we watch a grid from generation to generation, sometimes it quickly dies out; other times, it settles into an oscillating pattern (often very few patterns in the cycle); other times, it goes on seemingly forever before finally doing something less chaotic. The Grid above (with blue live cells), which is named the "r-pentomino", has pretty chaotic-seeming behavior for many steps. If we give it a large enough grid (so that it doesn't hit the edges and mis-calculate), we can get many unique patterns. On a 10x12 board (10 rows tall, 12 columns wide), **shown on the *last* page of this document**, we can see the patterns of each successive generation, playing by our chosen rules of B3/S23. The edge eventually causes some trouble, (not enough places coming alive, not as many neighbors, and some die-offs result), and 52 generations later the grid is finally extinct.

Goal

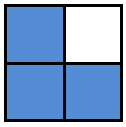
We will implement a series of functions that build up grids and calculate a next generation. Along the way, we will be creating grids (from scratch or based on previous grids), inspecting grids, and modifying grids.

First – Some Definitions

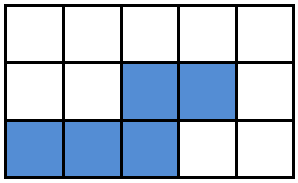
We define a `Grid` and a `GridString`. They are just 2D lists and strings with particular structure inside, but the names will be convenient in describing the purpose of the functions we need to write.

Grid: a list of lists of Booleans. We expect all inner lists to be the same length. The outer (first) dimension represents the rows (from top to bottom), and the inner lists represent the items in that row, from left to right. The boolean value represents if the current cell is alive; True means alive, False means dead.

- Grids are zero-indexed for both dimensions. No negative indexes are allowed for them; only use zero and up as the indexes.
- Examples:
  - `tiny_grid = [[True,False], [True, True]]`



  -
```
sample_grid = [[False, False, False, False, False],
               [False, False, True,  True,  False],
               [True,  True,  True,  False, False]]
```



GridString: a string of periods (`.`), `O`'s, and newlines (`'\n'`). When printed, it would look like a rectangle of periods (dead cells) and `O`'s (living cells), and we think of this as the grid representation.

- Though a `GridString` is supposed to have the same number of items in each row, our program will carefully check this fact when we interact with `GridString` values.
- blank lines in a `GridString` are ignored. This helps us create them with triple-quote strings.
- We can reliably convert between a `GridString` and a `Grid`, in either direction. They store the same information in different ways.
- triple-quote strings can represent newlines by actually typing the ENTER key. Removing empty lines (assumedly at the beginning/end) lets us create `GridString` values like the examples below.
- **Examples** (all in the samples file **p4provided.py**)

```
blinkerA = """
.....
..O..
..O..
..O..
.....
"""

blinkerB = """
.....
.....
.OOO.
.....
.....
"""
```

```
boat = """
.....
.OO..
.O.O.
..O..
.....
"""
```

```
r_pentomino = """
............
............
............
............
........OO..
.......OO...
........O...
............
............
............
"""
```

Functions

- **`read_coords(s):`** Given a **`GridString`**, read through it and create a list of **`int`** pairs for all live cells. Each pair is a **`(row, column)`** coordinate. If the rows don't all have the same number of spots indicated, or if any unexpected characters are present, this function returns **`None`**. Must be ordered by lowest row, and lowest column when rows match.
  - **Assume:** **`s`** is a **`GridString`**.
  - **Hint**: **`split()`** is your friend. What should you split by?
  - `read_coords("O..\n.OO\n")` → `[(0,0), (1,1), (1,2)]`
  - `read_coords("\n\nO..\n.OO\n\n")` → `[(0,0), (1,1), (1,2)]` `# ignore blank lines`
  - `read_coords(".....\n.....\n")` → `[]`

- **`get_dimensions(s):`** Given a **`GridString`**, find out how many rows and columns are represented in the **`GridString`**, and return them as a tuple: **`(numrows, numcols)`**. Remember that any blank lines must be ignored (skipped). If the rows don't all have the same number of items in them, or any unexpected characters are present, this function returns **`None`**.
  - **Assume:** **`s`** is a **`GridString`**.
  - `get_dimensions("O..\n.OO\n")` → `(2,3)`
  - `get_dimensions("OOOOO\n...\nOO\n.....")` → `None` *`# not rectangular!`*
  - `get_dimensions("\nOO\n..\n.O\nO.\n..\n\n")` → `(5,2)` *`# note ignored blank lines`*

- **`build_empty_grid(height, width):`** Given positive **`int`** values for the **`height`** and **`width`** of a grid, create a **`Grid`** that has **`False`** values at each location (representing dead cells).
  - **Assume:** **`height`** and **`width`** are positive integers.
  - `build_empty_grid(2,3)` → `[[False, False, False], [False, False, False]]`
  - `build_empty_grid(1,4)` → `[[False, False, False, False]]`
  - `build_empty_grid(3,1)` → `[[False], [False], [False]]`
  - `build_empty_grid(1,3)` → `[[False, False, False]]`

- **`build_grid(s):`** Given a **`GridString`**, determine the dimensions, build a grid of that size, and make alive each cell that should be alive.
  - **Assume:** **`s`** is a **`GridString`**.
  - **Hint**: try to use **`read_coords`**, **`get_dimensions`**, and **`build_empty_grid`** in your solution.
  - `build_grid("O..\n.OO\n")` → `[[True, False, False], [False, True, True]]`
  - `build_grid("..\n.O\nOO\n")` → `[[False,False], [False, True], [True, True]]`
  - `build_grid("OO.O..")` → `[[True, True, False, True, False, False]]`

- **`show_grid(grid, live='O', dead='.'):`** Given a **`Grid`**, and the option indicate what representation to use for live and dead cells, create the **`GridString`** that has no blank lines in it and represents the indicated grid. (The last character should be a newline).
  - **Assume:** **`grid`** is a **`Grid`**; **`live`** and **`dead`** are strings.
  - **Hint**: You'll likely want to call **`print(show_grid(someGrid))`** when trying it out.
  - `show_grid([[True,False,False], [False,True,True]])` → `"O..\n.OO\n"`
  - `show_grid([[False,False],[True,True]])` → `"..\nOO\n"`
  - `show_grid([[False,False],[True,True]],'A','#')` → `"##\nAA\n"`
  - `show_grid([])` → `""`

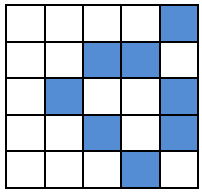
- **`count_living(grid):`** Given a **`Grid`**, count how many live cells there are; return that number.
  - **Assume:** **`grid`** is a **`Grid`**.
  - `count_living([[True,False,False], [True,False,True], [False,False,True]])` → `4`
  - `count_living([[True,True]])` → `2`

- **`any_living(grid):`** Given a **`Grid`**, determine if any live cells are present. Return the Boolean answer.
  - **Assume:** **`grid`** is a **`Grid`**.
  - `any_living([[True,False,False], [True,False,True]])` → `True`
  - `any_living([[False,False,False],[False,False,False]])` → `False`

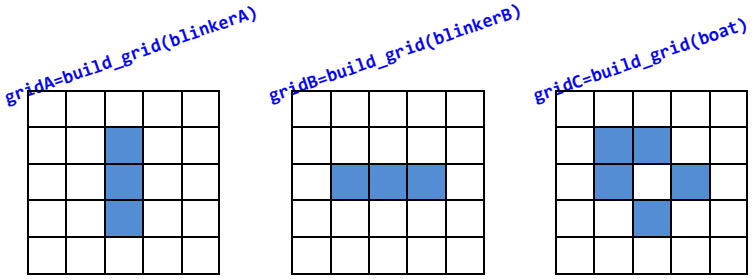
- **`on_grid(grid, r, c):`** Given a **`Grid`** and two integers indicating row/column position, determine if **`(r,c)`** is on the grid and return the answer as a boolean.
  - **Assume:** **`grid`** is a **`Grid`**; **`r`** and **`c`** are integers.
  - **Hint**: Remember, the grid is zero-indexed, and negative indexes are ***not*** to be used.
  - Examples use ***`<MxN grid>`*** as non-code placeholders for **`Grid`**s, so that the examples are small.
  - `on_grid(<3x5 grid>, 0, 0)` → `True`
  - `on_grid(<3x5 grid>, -1,-1)` → `False` `# negatives not allowed`
  - `on_grid(<3x5 grid>, 2, 4)` → `True` `# bottom-right corner`
  - `on_grid(<1x3 grid>, 1, 3)` → `False` `# zero-indexing, remember?`

- **`count_neighbors(grid, r, c):`** Given a **`Grid`** and two integers indicating row/column position, count how many living neighbors there are. When a cell is on the edge or corner of our **`Grid`**, treat all non-existent neighbor positions as dead (they don't contribute to the returned count).
  - **Assume:** **`grid`** is a **`Grid`**; live and **`dead`** are strings of length one.
  - **Hint**: Use your **`on_grid`** definition!
  - Examples all use the grid **`g`** drawn to the right.
    - `count_neighbors(g, 0, 0)` → `0`
    - `count_neighbors(g, 0, 4)` → `1`
    - `count_neighbors(g, 2, 4)` → `2`
    - `count_neighbors(g, 3, 2)` → `2`
    - `count_neighbors(g, 2, 3)` → `5`

`g:`



- **`next_gen(grid):`** Given a **`Grid`**, create and return a new **`Grid`** that represents the next generation. Note that **you will not be modifying the original Grid value** – each cell's next state is dependent on its neighbors' previous state, so updating one cell at a time would incorrectly mix generation statuses.
  - **Assume:** **`grid`** is a **`Grid`**.
  - **Hint**: Use previous definitions!
  - Examples based on the **`GridString`**s shown in **`GridString`** definition.
  - `next_gen(gridA)` → $\texttt{gridB}_{\text{copy}}$
  - `next_gen(gridB)` → $\texttt{gridA}_{\text{copy}}$
  - `next_gen(gridC)` → $\texttt{gridC}_{\text{copy}}$



  - Notes
    - **`gridA`** and **`gridB`** are two parts of an "oscillator"; a stable cycle of grids. This one is called "blinker", and was included in the Definitions section.
    - **`gridC`** is a "still life" example, a **`grid`** that equals its next generation. This one is called "boat".
    - In each case, we should be generating a new **`Grid`** value – hence the $_{\text{copy}}$ annotations.

- **`n_gens(grid, n=20):`** Given a **`Grid`** and a positive integer of how many generations to store, utilize your **`next_gen`** function to create a list with **`n`** generations in it, where the given **`grid`** is included as the first generation.
  - **Assume: `grid`** is a **`Grid`**; **`n`** is a positive integer that may default to **`20`** when not given.
  - **Hints**:
    - Use your **`next_gen`** definition!
    - you are returning a list of **`Grid`** values – it is a list that is 3 levels deep.
  - The examples would get pretty large – r_n indicates the same numbered generations from the page of **`r_pentomino`** steps; **`gridA`**, **`gridB`**, and **`gridC`** are also defined above (blinker and boat).

```
n_gens(r_pentomino,3)  →  [r0, r1, r2]    # see the last page
n_gens(gridA,5)        →  [gridA, gridB, gridA, gridB, gridA]
n_gens(gridB,3)        →  [gridB, gridA, gridB]
n_gens(gridC,4)        →  [gridC, gridC, gridC, gridC]
n_gens([[False]])      →  [[[False]], [[False]], [[False]], [[False]], [[False]], [[False]],
                           [[False]], [[False]], [[False]], [[False]], [[False]], [[False]],
                           [[False]], [[False]], [[False]], [[False]], [[False]], [[False]],
                           [[False]], [[False]]]
```

- **`is_still_life(grid, limit=100):`** Given a single **`Grid`**, determine if its sequence of generations becomes a "still life", where after some point, each generation matches the previous generation. Note that the chosen cut-off number of generations can change the answer (if a **`Grid`** takes 30 generations to settle into a still life, but we only look at **`10`** generations, we won't realize it, and report **`False`**).
  - **Assumes: `grid`** is a **`Grid`**, and **`limit`** is a positive integer that may default to **`100`**.
  - **Hint**: try comparing two different **`Grid`** values with the **`==`** operator.
  - Examples use names of provided **`GridString`**s.

```
is_still_life(build_grid(boat),5)          →  True
is_still_life(build_grid(boat),1)          →  False # only one grid: can't see still-life.
is_still_life(build_grid(blinkerA),10)     →  False # period is 2; this isn't still.
is_still_life(build_grid(r_pentomino),50)  →  False
is_still_life(build_grid(r_pentomino),55)  →  True  # all-dead equals all-dead.
```

- **`is_cycle(grid, limit=100):`** Given a single **`Grid`**, determine if its sequence of generations becomes an "oscillator", where after some point, a **repeating sequence of at least two distinct grids occurs. Note that still life grids are not cycles**! Again, the chosen number of generations to inspect (**`limit`**) may be short enough to prohibit our finding the cycle.
  - **Assumes: `grid`** is a **`Grid`**, and **`limit`** is a positive integer that may default to **`100`**.
  - **Hint**: use your **`n_gens`** function. Also – what **function on a previous project** could be useful??
  - Examples use names of provided **`GridString`**s.

```
is_cycle(build_grid(boat),5)              →  False      # still-lifes ≠ oscillators
is_cycle(build_grid(blinker),10)          →  True
is_cycle(build_grid(r_pentomino),50)      →  False
is_cycle(build_grid(pulsar),5)            →  True
is_cycle(build_grid(pulsar),3)            →  False
```

Some provided definitions (to animate your code!)

- **`PROVIDED CODE: print_gens(gs, live='O', dead='.'):`** Given a list of **`Grid`** values, prints each one and calls **`input()`** with no arguments afterwards to pause for effect. When running this in the terminal, if you mash down the ENTER button, you'll have a very cheap animation effect!
  - **Assumes:** **`gs`** is a list of **`Grid`** values. **`live`** and **`dead`** are strings.
  - `print_gens(n_gens(anyGrid))` → *<try it out!>*

- **`PROVIDED CODE: go(s, limit=100):`** Given a **`GridString`**, this function will build the **`Grid`**, calculate **`limit`** number of generations, and print them all. As this function does printing and user interaction, we're just providing it for you to play with your finished project.
  - **Assumes:** **`s`** is a **`GridString`**; **`limit`** is a positive integer that may default to **`100`**.
  - `go(s,n)` → *<try it out!>*

Extra Credit

Solve this problem for extra credit (up to +5%) and good practice.

- **`next_gen_growable(grid):`** Re-implement the **`next_gen`** function so that the border never interacts with the simulation. We'll do this by extending the size of the **`Grid`** whenever there's a live cell in the border. This means we must add an extra row or column of dead cells. We need to check for live border cells both before calculating the next generation to avoid incorrect calculations, and also after calculating the next generation, so that we never report a grid with live border cells. This means your **`Grid`** will *possibly* grow, and never shrink, in dimension. It also may stay the same size when all the action is in the center.
  - **Assume:** **`grid`** is a **`Grid`**.
  - `next_gen_growable([[True]])` → `[[False,False,False], [False,False,False], [False,False,False]]`
    `# original True required addition of borders, and then that cell died a lonely death.`
  - ```
    >>> print(show_grid(next_gen_growable(build_grid("""
    .O...
    .O..O
    ...OO
    .O..O
    """))))
                # ← end of the multi-line command. size was 4x5.
    .......
    .......
    ..OOO..
    ..OOOO.
    ...OO..
    .......     # ← new size is 6x7. New top/bottom rows and right column added before,
                and new right column added after.
    ```

- **`n_gens_growable(grid, n=20):`** Re-implement **`n_gens`** to allow the grid to grow from generation to generation.
  - **Assumes:** **`grid`** is a **`Grid`**; **`n`** is a positive integer that may default to **`20`** when not given.
  - `print_gens(n_gens_growable(anyGrid))` → *<try it out!>*

- **`PROVIDED CODE: go_growable(s, limit=100):`** Given a **`GridString`**, this function will build the grid, calculate **`limit`** number of growable generations, and print them all. As this function does printing and user interaction, we're just providing it for you to play with your finished project.
  - **Assumes:** **`s`** is a **`GridString`**; **`limit`** is a positive integer that may default to **`100`**.
  - `go(s,n)` → *<try it out!>*
    ```

Grading Rubric

Code passes shared tests:	90	
Well-documented/submitted:	10	
TOTAL:	100	*+5 extra credit*

Reminders on Turning It In:

No work is accepted more than 48 hours after the initial deadline, regardless of token usage. Tokens are automatically applied whenever they are available.

You can turn in your code as many times as you want; we only grade the last submission that is <=48 hours late. If you are getting perilously close to the deadline, it may be worth it to turn in an "almost-done" version about 30 minutes before the clock strikes midnight. If you don't solve anything substantial at the last moment, you don't need to worry about turning in code that may or may not be runnable, or worry about being late by just an infuriatingly small number of seconds – you've already got a good version turned in that you knew worked at least for part of the program.

You can (and should) check your submitted files. If you re-visit BlackBoard and navigate to your submission, you can double-check that you actually submitted a file (it's possible to skip that crucial step and turn in a no-files submission!), you can re-download that file, and then you can re-test that file to make sure you turned in the version you intended to turn in. It is your responsibility to turn in the correct file, on time, to the correct assignment.

Use a backup service. Do future you an enormous favor, and just keep all of your code in some automatically synced location, such as a Dropbox or Google Drive folder. Every semester someone's computer is lost/drowned/dead, or their USB drive is lost/destroyed, or their hard drive fails. Don't give these situations the chance to doom your project work!

Many iterations of r_pentomino

```
>>> go(r_pentomino)
0
.....
.....
.....
.....
........00..
.......00...
........0...
.....
.....
.....

1
.....
.....
.....
.....
.......000..
.......0....
.......00...
.....
.....
.....

2
.....
.....
.....
........0...
.......00...
......0..0..
.......00...
.....
.....
.....

3
.....
.....
.....
.......00...
.......000..
......0..0..
.......00...
.....
.....
.....

4
.....
.....
.....
.......0.0..
......0..0..
......0..0..
.......00...
.....
.....
.....

5
.....
.....
.....
........0...
......00.00.
......0..0..
.......00...
.....
.....
.....

6
.....
.....
.....
.....
.......000..
......00.00.
......0..00.
.......00...
.....
.....
.....

7
.....
.....
.....
........0...
......00.00.
......0.....
......0...0.
.......000..
.....
.....
.....

8
.....
.....
.....
.......000..
......0000..
.....00..00.
......0.00..
.......000..
........0...
.....
.....
.....

9
.....
.....
........0...
......0..0..
.....0......
.....0....0.
.....00.....
.....
.......000..
.....
.....
.....

10
.....
.....
.....
.....00.....
....00......
.....00.....
......000...
........0...
........0...
.....
.....
.....

11
.....
.....
.....
....000.....
....0.......
....0.......
.....00.0...
........00..
.....
.....
.....
.....

12
.....
.....
.....
.....0......
....00......
...00.......
....0.......
.....0.000..
.......000..
.....
.....
.....

13
.....
.....
.....
....00......
...0.0......
...0........
...000..0...
......00.0..
......00.0..
........0...
.....
.....
.....

14
.....
.....
.....
....00......
...0.0......
..00.0......
...000000...
....0....0..
......0..0..
.......00...
.....
.....
.....

15
.....
.....
.....
....00......
..00.00.....
..0....0....
..0...000...
...00....0..
.......0.0..
.......00...
.....
.....
.....

16
.....
.....
.....
...0000.....
..00.00.....
.00..0..0...
..0...000...
...0..0..0..
.......0.0..
.......00...
.....
.....
.....

17
.....
.....
....00......
..00..0.....
.0.....0....
.0..00..0...
.000.00.00..
......0..0..
......00.0..
.......00...
.....
.....
.....

18
.....
.....
...000......
..00000.....
.0.00000....
00.000..00..
.000..0.00..
..0......00.
......0..0..
......000...
.....
.....
.....

19
....0.......
..0...0.....
........0....
00.....00...
0........0..
0....0.0....
.000...0..0.
......0..00.
......000...
.......0....
.....
.....
.....

20
.....
.....
.0....000...
00.....00...
0.....00....
0.0...0.0...
.00....0000.
..0...0..00.
......0.00..
......000...
.....
.....
.....
```

<many more…>

```
51
..0.........
..0.........
.....
.....
.....
.....
.....
.....
.....
.....
.....

52
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

<extinction!
all blanks for
the rest of
time>