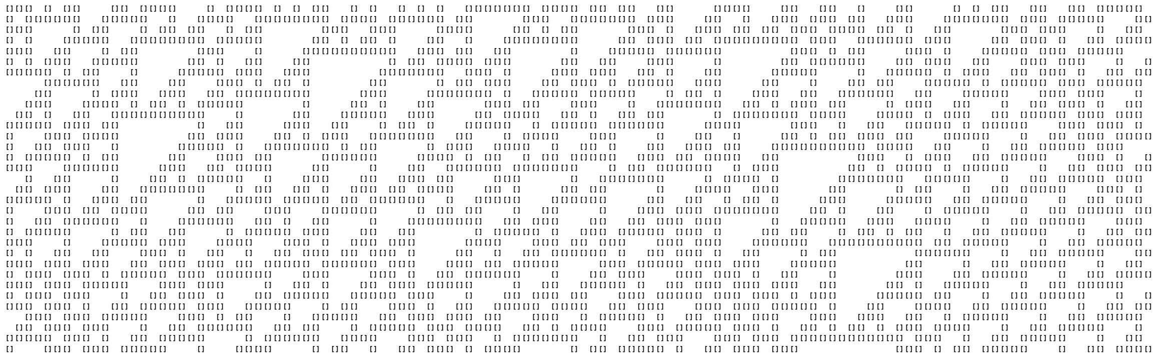
CS1110 Summer 2021 — Assignment 4

Due Saturday 7/17 at 11:59pm

Read this before you begin

In this assignment, you will eventually produce a program that simulates the growth of cells along a single line. The way one generation of cells is computed from the next can be described using simple rules. Nevertheless, the eventual behavior of the lines of cells can show some interesting patterns.[1] One example of such a pattern is:



Learning Goals The main goal of this assignment is to let you get some practice with lists (Units 13 and 14) and loops (Unit 15). We will also make use of other features discussed before, including expressions over integers and booleans (Unit 2) and conditionals (Units 9 and 10). Finally, this assignment should also help you get some more experience with raising and catching errors (Unit 12). Along the way, please be sure to use your debugging skills (Unit 11), particularly reading error messages, to help solve issues you run into.

Submission For this assignment, you will submit two files: `a4.py` and `a4app.py`

Submission within 24 hours after the deadline will be accepted with a 10% late penalty. Even if only one file (of several) is submitted late, the assignment as a whole will be marked late. Assignment submission on Canvas closes 24 hours after the deadline and no further submission will be allowed.

Do not put your name in the files that you will submit, but **do** include your NetID. The reason is to help promote fairness in grading. Thank you for heeding our request, even if it sounds a little strange.

Group work and academic integrity You must work either on your own or with one partner.

If you work with a partner, you must first register as a group in Canvas before you submit your work.

For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is *certainly* not OK to copy code from another person or from other sources.

If you feel that you cannot complete the assignment on you own, please seek help from the course staff.

[1]If you are interested in the principles behind these patterns, you can consult the Wikipedia article on "Rule 110". Please note, however, that you should not need to understand the topics discussed there to complete this assignment.

Your task

For this assignment, you are given a skeleton with three files: `a4.py`, `a4app.py`, and `a4test.py`. You will turn in the first two files. In the first three parts, you will work in `a4.py`. In the last part, you will work in `a4app.py`.

The purpose of `a4test.py` is to test the code that you have written.[2] After each part, you can run this file as a script (i.e., type `python a4test.py`), and it will check your functions for any easy mistakes, testing the examples that appear in the docstrings. When you start out, the output of the test script should be as follows:

```
Testing byte_to_number
-> You have not implemented byte_to_number yet.
Testing number_to_byte
-> You have not implemented number_to_byte yet.
Testing next_generation
-> You have not implemented next_generation yet.
Testing print_generation
-> You have not implemented print_generation yet.
All tests (that ran) succeeded!
```

When you are done, the output should look like this:

```
Testing byte_to_number
-> All tests for byte_to_number succeeded!
Testing number_to_byte
-> All tests for number_to_byte succeeded!
Testing next_generation
-> All tests for next_generation succeeded!
Testing print_generation
-> All tests for print_generation succeeded!
All tests (that ran) succeeded!
```

Part of the idea behind providing this test script is to drive home the point that tests are a great tool to have as a programmer. In the next exercise, you will have to write your own test script again.

Please bear in mind that we will still grade for style (10 points) and other mistakes, so even if your code passes all of our tests you may still see points deducted if you do not follow the Style Guide, or if you make mistakes that are not picked up by the test script. If you want, you can also write your own test script.

Important: be sure to read the docstring for each function carefully.

Part A: Binary numbers [30 points]

Background When a computer stores numbers, it does not do this in the decimal notation that you are used to. Instead, numbers are stored in *binary representation*, or zeroes and ones. Just like a decimal number is a sequence of digits from 0 to 9, a binary number is a sequence of zeroes and ones. As an example, the decimal number 17 corresponds to the binary number **`10001`**, and the binary number **`1101`** is the same as the decimal number 13.[3]

To see how this work in general, consider a decimal number like 123. We can split it up in hundreds, tens and singles: 1 times a hundred, 2 times ten, and 3 times one. Or, perhaps more explicitly, we can write

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

Binary numbers work the same way, except that we use powers of two instead of powers of ten. So,

$$\mathtt{\mathbf{1101}} = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$$

In Python, we can represent a binary number using a list of booleans, where **`1`** is represented by `True`, and **`0`** is represented by `False`. So, for example, **`1101`** becomes `[True, True, False, True]`.

[2]You should not have to edit or even read `a4test.py`; it contains some advanced techniques that we have not discussed yet.

[3]In an effort to prevent any confusion, we typeset binary numbers in bold monospace, **`like this`**.

Implementation In this part, you will need to implement two functions:

- `byte_to_number`, which takes a list of booleans and converts it to the corresponding number.

  Hint: one way to accomplish this is as follows. If the list `byte` is n bits long, you should check if `byte`[i] is true, and if so add the right power of two to the accumulator, for each index i counting up from zero.

  You should use a loop to check each of the elements of the input list; *do not* improvise with `if`-statements.

- `number_to_byte`, which takes a number from 0 to 255 and returns a list of exactly eight booleans representing that number (with `False` added to the front if necessary). Again, you should use a loop.

  Hint: one of several possible ways to do this is as follows. For each of the powers of two, from 2^7 down to 2^0, check if the power fits inside the number. If this is the case, subtract it from the input number and append `True` to the accumulator list, otherwise append `False` to the accumulator list.

Note that each of these functions also has some additional requirements in terms of error handling that are specified in the docstring. Be sure to run the test script after implementing each function, to see if you are on the right track.

A warning If you search for "conversion to binary" or "conversion from binary" online, you will find many results. It is okay to look up the general principle of how to do this for reference, and even to consult explanations by other people. However, you should *never* copy-paste code from the internet. We have very sophisticated methods in place to catch this, and we *always* find out, so spare yourself the consequences of an Academic Integrity violation.

Part B: Simulating a Line of Cells [30 points]

Background Consider a one-dimensional world consisting of single cells arranged in a line. Each of these cells can either be alive or dead. For instance, in the following world there are 10 cells:

```
[][]--------[][][]--                (1)
```

The first two cells (drawn using open-close square brackets `[]`) are alive, the next four cells (drawn as `--`) are dead, then there are three live cells, and at the very end there is another dead cell.

Once every second, each cell looks at its two neighbors, to get a *pattern* consisting of three cells. For example:

- The pattern for the second cell in the above example is `[][]--`: one living cell (the neighbor to the left), another living cell (the cell itself!), and a dead cell (the neighbor to the right).
- The seventh cell in the line above has pattern `--[][]`: a dead cell (its neighbor to the left), a living cell (the cell itself), and another living cell (its neighbor to the right).

The first and last line are also neighbors (we say the line "wraps around"). More precisely, the left neighbour of the first cell is the last cell, and the right neighbor of the last cell is the first cell. So, in line (1):

- The pattern for the first cell is `--[][]`, consisting of the neighbor to the left (which is the last cell), followed by the first cell itself, and its neighbor to the right (the second cell).
- The pattern for the last cell is `[]--[]`, which has the neighbor to the left (the second-to-last cell), the last cell itself, and its neighbor to the right (which is the first cell).

The pattern of a cell determines whether it will be alive in the next generation, according to a certain *rule*. There are many different rules, but let's look at one possibility. *Rule 184* specifies that each of the following patterns leads to a dead or alive cell in the next generation, as determined by the table that follows.

Pattern	Next gen.	Pattern	Next gen.
`[][][]`	`[]`	`--[][]`	`[]`
`[][]--`	`--`	`--[]--`	`--`
`[]--[]`	`[]`	`----[]`	`--`
`[]----`	`[]`	`------`	`--`

To obtain the next generation of cells under *Rule 184*, find the pattern for each cell; then, look in the table above to see whether the cell is dead or alive in the next generation.

Worked example Let's apply the steps for *Rule 184* to the line in (1):

- The pattern for the first cell is `--[][]`, so this cell becomes `[]` in the next generation.
- The pattern for the second cell is `[][]--`, so this cell becomes `--` in the next generation.
- The pattern for the third cell is `[]----`, so this cell becomes `[]` in the next generation.
- The pattern for the fourth and fifth cell is `------`, so these cells become `--` in the next generation.
- The pattern for the sixth cell is `----[]`, so this cell becomes `--` in the next generation.
- The pattern for the seventh cell is `--[][]`, so this cell becomes `[]` in the next generation.
- The pattern for the eighth cell is `[][][]`, so this cell becomes `[]` in the next generation.
- The pattern for the ninth cell is `[][]--`, so this cell becomes `--` in the next generation.
- The pattern for the last (tenth) cell is `[]--[]`, so this cell becomes `[]` in the next generation.

If we put all of these new cells in a row, we obtain the next generation, which is as follows:

```
[]--[]------[][]--[]                                   (2)
```

There exists a rule for each number from 0 up to 255. We have implemented a function `next_cell` that uses the functions you wrote in Part A to create a table like the one above for each number. This function takes (1) a list of three booleans representing the pattern for a cell and (2) a rule number from 0 to 255; it outputs True if the cell is alive in the next generation, and False if it is not. You can look at the docstring for more information.

Implementation Your task is to implement the function `next_generation`. This function takes two arguments: a list of booleans representing the current line, and a rule number. Your code should do all of the following:

- For each cell, figure out its pattern, keeping the description above in mind.
- Call `next_cell` with the pattern and rule number to look up the state of that cell in the next generation.
- Accumulate all results in a *new* list of booleans, exactly as long as the input, and return that.

Note that this function also has additional requirements written in the docstring that you should implement.

Part C: Talking About My Generation [10 points]

In this part, you will write `print_generation`. This function takes a list of booleans representing a line of cells, and prints out a line where each living cell is `[]`, and each dead cell is two spaces[4], all on the same line.

As always, you should look at the docstring for further details about how to implement this function.

Part D: Putting It All Together [20 points]

Now, it is time to open up `a4app.py`, and write an application that serves as a user interface to what you just wrote. Specifically, this script should do the following:

1. Ask the user for a rule number, and convert it to an integer.

   Use `try` and `except` (like we talked about in Lecture 11) to print out an error message if the input is not a valid integer. After the message, you should also stop the program by calling `exit()`.

2. Ask the user for the length of the line they want to simulate, and convert it to an integer.

   Just like the rule number, you should wrap this conversion inside `try` and use `except` to handle any errors that come up, calling `exit()` to stop your script in that case.

3. Generate a random starting line of that length using the function `random_generation` from `a4`.

4. Print out the first 100 generations, using a loop and calling the function `next_generation` that you implemented in Part B, as well as `print_generation` from Part C.

When you are done, test your script on some inputs. Make sure the line fits in your terminal window; 30 or 40 cells should be a good number to start with. Interesting rule numbers to try are 110, 90, 30, and 184.

[4]Note how this is slightly different from the way dead cells are drawn above!