Seagull Documentation

Release 1.0.0-beta.4

Lester James V. Miranda

Nov 08, 2020

GENERAL

A Python library for Conway's Game of Life

```
pip install pyseagull
```

This framework allows you to create and simulate various artificial lifeforms and cellular automata easily: simply define your board, add your lifeforms, and execute the *run* command! It also provides a myriad of pre-made lifeforms while allowing you to create your own.

Why name it Seagull? Conway's Game of Life is quite a mouthful, so I just refer to its acronym, CGoL. The word "seagull" is just a pun of that.

- **Free Software**: MIT License
- **Github Repository**: https://github.com/ljvmiranda921/seagull

Conway's Game of Life is considered a zero-player game: you simply set-up your initial configuration on the board and observe how it evolves through time.

Simulate your first lifeforms in a few lines of code:

```
import seagull as sg
from seagull.lifeforms import Pulsar

# Initialize board
board = sg.Board(size=(19,60))

# Add three Pulsar lifeforms in various locations
board.add(Pulsar(), loc=(1,1))
board.add(Pulsar(), loc=(1,22))
board.add(Pulsar(), loc=(1,42))

# Simulate board
sim = sg.Simulator(board)
sim.run(sg.rules.conway_classic, iters=1000)
```

Optionally, you can animate the simulation by running `sim.animate()`:

CHAPTER

ONE

INSTALLATION

To install Seagull, run this command in your terminal:

```
pip install pyseagull
```

This is the preferred method to install Seagull, as it will always install the most recent stable release.

In case you want to install the bleeding-edge version, clone this repo:

```
git clone https://github.com/ljvmiranda921/seagull.git
```

and then run

```
cd seagull
python setup.py install
```

CHAPTER

TWO

CONCEPTS

There are three main components for an artificial life simulation:

- The `Board` or the environment in which the lifeforms will move around
- The `Lifeform` that will interact with the environment, and
- The `rules` that dictate if a particular cell will survive or not

In the classic Conway's Game of Life, there are four rules (taken from @jakevdp's blog post):

- **Overpopulation**: if a living cell is surrounded by more than three living cells, it dies
- **Stasis**: if a living cell is surrounded by two or three living cells, it survives
- **Underpopulation**: if a living cell is surrounded by fewer than two living cells, it dies
- **Reproduction**: if a dead cell is surrounded by exactly three cells, it becomes a live cell

In Seagull, you simply define your `Board`, add your `Lifeform`/s, and run the `Simulator` given a `rule`. You can add multiple lifeforms as you want:

```
import seagull as sg
from seagull import lifeforms as lf

board = sg.Board(size=(30,30))
board.add(lf.Blinker(length=3), loc=(4,4))
board.add(lf.Glider(), loc=(10,4))
board.add(lf.Glider(), loc=(15,4))
board.add(lf.Pulsar(), loc=(5,12))
board.view()  # View the current state of the board
```

Then you can simply run the simulation, and animate it when needed:

```
sim = sg.Simulator(board)
stats = sim.run(sg.rules.conway_classic, iters=1000)
sim.animate()
```

2.1 Adding custom lifeforms

You can manually create your lifeforms by using the `Custom` class:

```
import seagull as sg
from seagull.lifeforms import Custom

board = sg.Board(size=(30,30))
board.add(Custom([[0,1,1,0], [0,0,1,1]]), loc=(0,0))
```

2.2 Obtaining simulation statistics and history

By default, the simulation statistics will always be returned after calling the `run()` method. In addition, you can also obtain the history by calling the `get_history()` method.

```
# The run() command returns the run statistics
stats = sim.run(sg.rules.conway_classic, iters=1000)
# You can also get it using get_history()
hist = sim.get_history()
```

CHAPTER

THREE

EXAMPLES

Below are some examples on how to use the Seagull. They are all in the form of Jupyter notebooks. If you wish to contribute some cool ecosystems to display in our gallery, then please open a Pull Request!

You can also find the notebooks in this link.

3.1 Basic Usage

This notebook illustrates some of the basic features of `Seagull`. Most of these are just animated outputs found in the Documentation. If you wish to add more examples, please create a Jupyter notebook and open up a Pull Request!

[1]:
```
# Some settings to show a JS animation
import matplotlib.pyplot as plt
plt.rcParams["animation.html"] = "jshtml"
```

[2]:
```
import seagull as sg
import seagull.lifeforms as lf
```

3.1.1 Simple Pulsars

In this example, we'll create four (4) `Pulsars` in a 40x40 board.

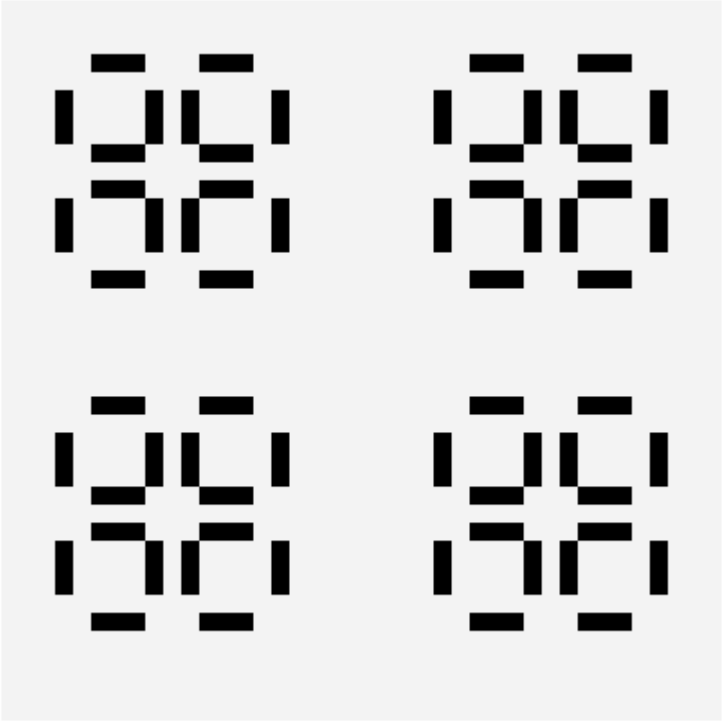
[3]:
```
# Initialize board
board = sg.Board(size=(40,40))

# Add three Pulsar lifeforms in various locations
board.add(lf.Pulsar(), loc=(1,1))
board.add(lf.Pulsar(), loc=(1,22))
board.add(lf.Pulsar(), loc=(20,1))
board.add(lf.Pulsar(), loc=(20,22))
```

The view command allows us to see the current state of the `Board`

[4]:
```
board.view()
```

[4]:
```
(<Figure size 360x360 with 1 Axes>,
 <matplotlib.image.AxesImage at 0x7f2d7a292d10>)
```



Running the simulation returns a set of statistics that characterizes your run:

[5]:
```
# Simulate board
sim = sg.Simulator(board)
sim.run(sg.rules.conway_classic, iters=100)
```

```
2020-11-08 02:22:02.781 | INFO     | seagull.simulator:compute_statistics:128 -␣
↪Computing simulation statistics...
```

[5]:
```
{'peak_cell_coverage': 0.18,
 'avg_cell_coverage': 0.1463366336633663,
 'avg_shannon_entropy': 3.0217796046186556,
 'peak_shannon_entropy': 3.2433182601909962}
```

The animation command returns a `matplotlib.FuncAnimation` that you can use to show or save your animation. Note that sometimes, saving to GIF or MP4 might require other dependencies such as `ffmpeg` or `ImageMagick`. For more information, please check this blog post.

[6]:
```
%%capture
anim = sim.animate()
```

```
2020-11-08 02:22:02.805 | INFO     | seagull.simulator:animate:183 - Rendering␣
↪animation...
```

[7]:
```
anim
```

[7]:
```
<matplotlib.animation.FuncAnimation at 0x7f2d77f6a1d0>
```

3.1.2 Small ecosystem

In this example, we'll demonstrate the diversity of our `Lifeforms` and see how they interact with one another!

[8]:
```
board = sg.Board(size=(30,30))
board.add(lf.Glider(), loc=(4,4))
board.add(lf.Glider(), loc=(10,4))
board.add(lf.Glider(), loc=(15,4))
board.add(lf.Pulsar(), loc=(5,12))
board.add(lf.Blinker(length=3), loc=(22,4))
board.add(lf.Blinker(length=3), loc=(22,8))
```

[9]:
```
%%capture
# Simulate board
sim = sg.Simulator(board)
stats = sim.run(sg.rules.conway_classic, iters=100)
anim = sim.animate()
```

```
2020-11-08 02:22:06.050 | INFO     | seagull.simulator:compute_statistics:128 -␣
↪Computing simulation statistics...
2020-11-08 02:22:06.060 | INFO     | seagull.simulator:animate:183 - Rendering␣
↪animation...
```

[10]:
```
anim
```

[10]:
```
<matplotlib.animation.FuncAnimation at 0x7f2d77ecc950>
```

3.1.3 Custom lifeform

Lastly, we'll create our very first custom lifeform! Defining it is just the same as instantiating any other pre-made lifeform. However in this example, we'll save the instance in a variable `custom_lf` so that we can place it easily without writing the whole matrix again and again.

[11]:
```
board = sg.Board(size=(30,30))

# Our custom lifeform
custom_lf = lf.Custom([[0, 0, 1, 1, 0],
                       [1, 1, 0, 1, 1],
                       [1, 1, 1, 1, 0],
                       [0, 1, 1, 0, 0]])

board.add(custom_lf, loc=(1,1))
board.add(custom_lf, loc=(10,1))
board.add(custom_lf, loc=(20,1))
board.add(custom_lf, loc=(5, 10))
board.add(custom_lf, loc=(15, 10))
board.add(custom_lf, loc=(10, 20))
```

[12]:
```
%%capture
sim = sg.Simulator(board)
stats = sim.run(sg.rules.conway_classic, iters=100)
anim = sim.animate()
```

```
2020-11-08 02:22:09.362 | INFO     | seagull.simulator:compute_statistics:128 -␣
↪Computing simulation statistics...
2020-11-08 02:22:09.372 | INFO     | seagull.simulator:animate:183 - Rendering␣
↪animation...
```

[13]:
```
anim
```

[13]:
```
<matplotlib.animation.FuncAnimation at 0x7f2d7667f890>
```

3.2 Sprator

In this example, we'll try cloning Sprator using Seagull. The idea for Sprator is fun and simple:

1. Generate a 4x8 random noise
2. Change the state according to the Conway Rule
3. Repeat steps twice
4. Flip the 4x8 image to create an 8x8 one

[1]:
```
# Some settings to show a JS animation
import matplotlib.pyplot as plt
plt.rcParams["animation.html"] = "jshtml"
%matplotlib inline
```

[2]:
```
import seagull as sg
import seagull.lifeforms as lf
```

3.2.1 Creating the lifeform

First, we'll setup the board, of the size 4x8. Then, we'll create a Custom lifeform using some random noise. Lastly, we'll add the lifeform onto the board.

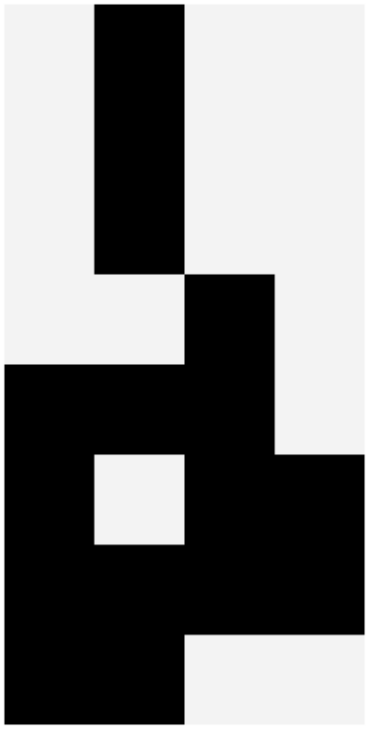
[3]:
```
board = sg.Board(size=(8,4))
```

[4]:
```
import numpy as np
np.random.seed(42)
noise = np.random.choice([0,1], size=(8,4))
custom_lf = lf.Custom(noise)
```

[5]:
```
board.add(custom_lf, loc=(0,0))
```

[6]:
```
board.view()
```

[6]:
```
(<Figure size 360x360 with 1 Axes>,
 <matplotlib.image.AxesImage at 0x7f1f5cb0dcf8>)
```



3.2.2 Creating a custom rule

The rule of Sprator is a bit different from Conway's Game of Life, so we'll create a custom function instead. What's cool about the Sprator rule is that all dead cells stay dead and all other live cells die in the next generation. Interesting concept!

[7]:
```
import scipy.signal


def count_neighbors(X) -> np.ndarray:
    """Count neighbors for each element in an array"""
    return scipy.signal.convolve2d(X, np.ones((3, 3)), mode="same", boundary="fill") -
↪ X


def custom_rule(X) -> np.ndarray:
    """Custom sprator rule"""
    # Count the neighbors for each cell
    n = count_neighbors(X)
    dead_with_less_one_neighbor = (X == 0) & (n <= 1)
    alive_with_two_three_neighbors = (X == 1) & ((n == 2) | (n == 3))
    return dead_with_less_one_neighbor | alive_with_two_three_neighbors
```

3.2.3 Running the simulation

Now that we have a board, what's left is to just run the simulation!

[8]:
```
sim = sg.Simulator(board)
stats = sim.run(custom_rule, iters=1)  # 1 iteration seems to give better results
```

```
2020-03-22 20:54:54.626 | INFO     | seagull.simulator:compute_statistics:128 -␣
↪Computing simulation statistics...
```

3.2.4 Create the Sprite!

In order to create the Sprite, we should get the final step of the simulator (from the history), flip the array, and concatenate them into an 8x8 sprite!

[9]:
```
final = sim.get_history()[-1]
```

[10]:
```
sprator = np.hstack([final, np.fliplr(final)])
```

[11]:
```
sprator = np.pad(sprator, mode="constant", pad_width=1, constant_values=1)
```

[12]:
```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(5,5))
ax = fig.add_axes([0, 0, 1 , 1], xticks=[], yticks=[], frameon=False)
im = ax.imshow(sprator, cmap=plt.cm.binary, interpolation="nearest")
im
```

[12]:
```
<matplotlib.image.AxesImage at 0x7f1f2b6b4e48>
```

3.2.5 Let's add some colors!

Let's put some colors in our sprites by assigning a `fill_color` and an `line_color`

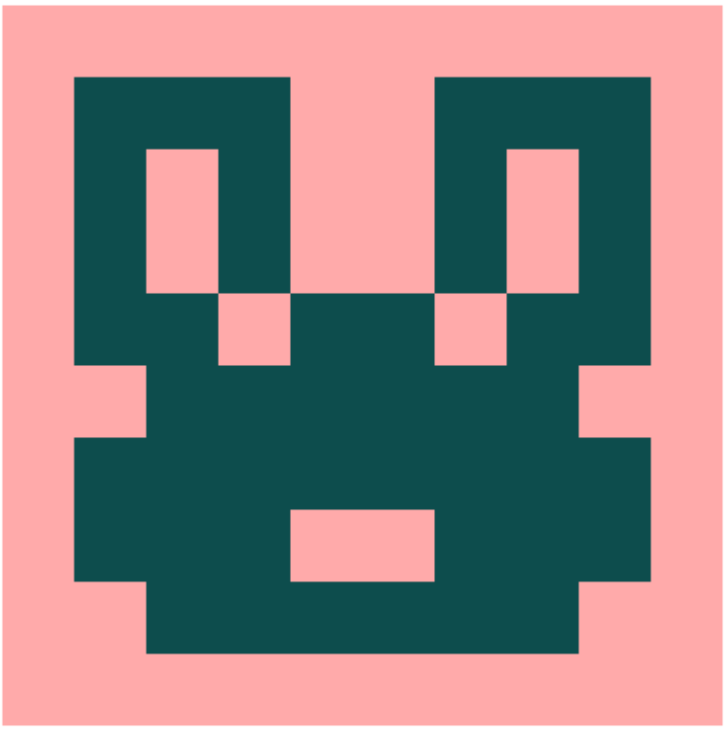
[13]:
```
def hex_to_rgb(h):
    """Convert a hex code to an RGB tuple"""
    h_ = h.lstrip("#")
    return tuple(int(h_[i:i+2], 16) for i in (0, 2, 4))
```

[14]:
```
def apply_color(sprite, fill_color, line_color):
    """Apply color to the sprite"""
    sprite_color = [
        (sprite * line) + (np.invert(sprite) * fill)
        for line, fill in zip(hex_to_rgb(line_color), hex_to_rgb(fill_color))
    ]
    return np.rollaxis(np.asarray(sprite_color), 0, 3)
```

[15]:
```
sprite_color = apply_color(sprator, "#0d4d4d", "#ffaaaa")
```

[16]:
```
fig = plt.figure(figsize=(5,5))
ax = fig.add_axes([0, 0, 1 , 1], xticks=[], yticks=[], frameon=False)
im = ax.imshow(sprite_color)
im
```

[16]:
```
<matplotlib.image.AxesImage at 0x7f1f2b682898>
```



CHAPTER

FOUR

CHANGELOG

4.1 v1.0.0-a.1 (2019-05-03)

- First alpha release on PyPI
- Not usable, no code yet. I just released it to reserve the name

4.2 v1.0.0-b.0 (2019-05-11)

- First beta release on PyPI
- Includes the most basic set of features: Board, Lifeforms, Simulator
- Add Documentation with Sphinx and ReadTheDocs
- Add Continuous Integration using Azure Pipelines

4.2.1 v1.0.0-b.1 (2019-05-13)

- Update documentation with examples PR#13
- Fix README.md PR#15
- Ensure that board.state will never change PR#20

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/ljvmiranda921/seagull/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it. Those that are tagged with "first-timers-only" is suitable for those getting started in open-source software.

5.1.4 Write Documentation

Seagull could always use more documentation, whether as part of the official Seagull docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ljvmiranda921/seagull/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *seagull* for local development.

1. Fork the *seagull* repo on GitHub.
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/seagull.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
cd seagull/
make venv  # Creates a virtual environment
make dev  # Installs development requirements
```

   For windows users, you can do the following:

```
cd seagull
python -m venv venv
venv\Scripts\activate
pip install pip-tools
pip install -r requirements.txt
pip install -r requirements-dev.txt
```

4. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox. In addition, ensure that your code is formatted using black:

```
flake8 seagull tests
black seagull tests
pytest-v
```

   To get flake8, black, and tox, just pip install them into your virtualenv. If you wish, you can add pre-commit hooks for both flake8 and black to make all formatting easier.

6. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, and above. Check https://dev.azure.com/ljvmiranda/ljvmiranda/_build/latest?definitionId=3&branchName=master and make sure that the tests pass for all supported Python versions.

5.4 Contributing examples

When contributing notebooks, just ensure the following:

1. **All notebooks have clear outputs.** You can click the *Restart and Clear Output* in the toolbar or use a tool like *nbstripout*. Sphinx does the job of executing them before deployment.
2. **Each cell has an execution timeout of 3 minutes.** Take note of that when setting very long iterations. Please note in the PR if the example really reqiures long iterations so the limit can be relaxed properly.
3. **Ensure that the environment can be reproduced easily.** Highly-complex configuration might not be accepted. If the notebook only relies on Seagull, the better.

CHAPTER

SIX

CODE OF CONDUCT

6.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

6.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

6.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

6.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

6.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at ljvmiranda@gmail.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

6.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html.

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq.

CHAPTER

SEVEN

BOARD

The Board represents the environment where lifeforms can evolve, you can initialize a Board by passing a tuple representing its size:

```
import seagull as sg
board = sg.Board(size=(30, 30))  # default is (100, 100)
```

You can add lifeforms to the board by using the `add()` command. You should pass an instance of the lifeform and its location on the board. The *loc* parameter is anchored at the top-left for two-dimensional lifeforms and to the left for one-dimensional lifeforms. It follows numpy's indexing convention

Whenever a lifeform's size exceeds the edge of the board, then Seagull throws a `ValueError`:

```
import seagull as sg
board = sg.Board()
board.add(sg.lifeforms.Blinker(length=3), loc=(0,0))
```

You can always view the board's state by calling the `view()` method. Lastly, you can clear the board with the `clear()` command.

class seagull.board.**Board**(*size=(100, 100)*)

Represents the environment where the lifeforms can grow and evolve

__init__(*size=(100, 100)*)

Initialize the class

Parameters **size** (*array_like of size 2*) – Size of the board (default is *(100, 100)*)

add(*lifeform*, *loc*)

Add a lifeform to the board

Parameters

- **lifeform** (*seagull.lifeforms.base.Lifeform*) – A lifeform that can evolve in the board
- **loc** (*array_like of size 2*) – Initial location of the lifeform on the board

clear()

Clear the board and remove all lifeforms

view(*figsize=(5, 5)*)

View the current state of the board

Parameters **figsize** (*tuple*) – Size of the output figure

Returns Graphical view of the board

Return type (matplotlib.figure.Figure, matplotlib.image.AxesImage)

CHAPTER

EIGHT

SIMULATOR

The Simulator takes in a `seagull.Board`, and runs a simulation given a set number of iterations and a rule. For each iteration, the rule is applied to the Board in order to evolve the lifeforms. After the simulation, run statistics are returned.

```
import seagull as sg

board = sg.Board()
board.add(Blinker(), loc=(0,0))

# Initialize a simulator
sim = sg.Simulator(board)
stats = sim.run(sg.rules.conway_classic, iters=1000)
```

You can always get the history of the whole simulation by calling the *get_history()* method. The length of the history will always be equal to `iters + 1` since we include the initial state

---

Note: Running a simulation does not change the `state` attribute of the board. Internally, the simulator makes a copy of that layout and updates that instead. This is to avoid unintended behaviour when running simulations again and again.

Various statistics such as entropy, peak cell coverage, and the like are returned as a dictionary. This gives us an idea on the characteristics of the simulation experiment.

Note: Some statistics are highly-dependent on the size of the board and the number of iterations. For example, peak cell coverage (pertaining to the max. amount of active cells during the whole run) depends on board size. If you have better ideas for computing these statistics, please open-up an Issue!

The `run()` method only computes the progress of the board for the whole simulation, but it does not animate it yet. To create an animation, call the `animate()` method:

```
sim.animate()
```

This returns a `matplotlib.animation.FuncAnimation` that you can turn into an interactive animation in your notebook or exported as a GIF.

Note: When exporting to GIF, it is required to have the ffmpeg backend installed.

---

class seagull.simulator.**Simulator**(*board*)

__init__ (*board*)
Initialize the class

Parameters **board** (*seagull.Board*) – The board to run the simulation on

animate (*figsize=(5, 5)*, *interval=100*)
Animate the resulting simulation

Parameters

- **figsize** (*tuple*) – Size of the output figure
- **interval** (*int*) – Interval for transitioning between frames

Returns Animation generated from the run

Return type matplotlib.animation.FuncAnimation

compute_statistics (*history*)
Compute various statistics for the board

Parameters **history** (*list or numpy.ndarray*) – The simulation history

Returns Compute statistics

Return type dict

get_history (*exclude_init=False*)
Get the simulation history

Parameters **exclude_init** (*bool*) – If True, then excludes the initial state in the history

Returns Simulation history of shape `(iters+1, board.size[0], board.size[1])`

Return type numpy.ndarray

run (*rule*, *iters*, ***kwargs*)
Run the simulation for a given number of iterations

Parameters

- **rule** (*callable*) – Callable that takes in an array and returns an array of the same shape.
- **iters** (*int*) – Number of iterations to run the simulation.

Returns Computed statistics for the simulation run

Return type dict

CHAPTER

NINE

LIFEFORMS

Lifeforms represent the evolving patterns whenever a rule is applied. In Seagull, lifeforms are first-class citizens: you can add them to the board, view them independently, compose, customize, and the like. This library provides a collection of pre-made lifeforms that you can play around.

Lifeforms are arranged into categories based on their configurations (excluding the Base and Custom lifeforms):

seagull.lifeforms.gliders	Gliders are lifeforms that oscillate but move while oscillating
seagull.lifeforms.growers	Growers are lifeforms that exhibit asymptotically unbounded growth
seagull.lifeforms.oscillators	Oscillators are lifeforms that returns to its initial configuration after some time
seagull.lifeforms.methuselahs	Methuselahs are long-lived lifeforms which evolve rapidly before reaching a steady state after many cycles.
seagull.lifeforms.random	Random lifeforms are generated on-the-fly without specific configuration
seagull.lifeforms.static	Static lifeforms do not oscillate nor move given classic Conway rules

9.1 Base

Base class for all Lifeform implementations. All lifeforms found in this library inherits from the *seagull.lifeforms.base.Lifeform* class. You can use this to implement your own lifeforms or contributing new lifeforms to Seagull:

```
class MyNewLifeform(Lifeform):

    def __init__(self, arg1=1, arg2=2):
        super(MyNewLifeform, self).__init__()
        self.arg1 = arg1
        self.arg2 = arg2

    @property
    def layout(self) -> np.ndarray:
        return np.array([[0, 0, 1, 1]])
```

When contributing your new lifeform, we highly-recommend to set sensible defaults when initializing it. This is because the current test running simply runs the `inspect` module to get all classes and run the same set of tests.

If you wish to pass a custom lifeform to the board, I recommend using the *seagull.lifeforms.custom.Custom* class.

class seagull.lifeforms.base.**Lifeform**
Base class for all Lifeform implementation

abstract property layout
Lifeform layout or structure

Type numpy.ndarray

Return type ndarray

property size
Size of the lifeform

Type tuple

Return type Tuple[int, int]

view(*figsize=(5, 5)*)
View the lifeform

Returns Graphical view of the lifeform

Return type matplotlib.axes._subplots.AxesSubplot

9.2 Custom

The Custom lifeform allows you to easily pass any arbitrary array as a *seagull.lifeforms.base.Lifeform* to the Board. However, it is important that the array passes two conditions:

- It must be a 2-dimensional array. For lines such as Blinkers, we often use an array of shape `(2, 1)`.
- It must be a binary array where `True` represents active cells and `False` for inactive cells. You can also use 0s and 1s as input.

If any of these conditions aren't fulfilled, then Seagull will raise a `ValueError`

Here's an example in creating a custom lifeform:

```
import seagull as sg
from seagull.lifeforms import Custom

board = sg.Board(size=(30,30))
board.add(Custom([[0,1,1,0], [0,0,1,1]]))
```

class seagull.lifeforms.custom.**Custom**(*X*)
Create custom lifeforms

__init__(*X*)
Initialize the class

Parameters **X** (*array_like*) – Custom binary array for the lifeform

property layout
Lifeform layout or structure

Type numpy.ndarray

Return type ndarray

validate_input_shapes(*X*)
Check if input array is of size 2

validate_input_values(*X*)
Check if all elements are binary

9.3 All Lifeforms

This section contains all lifeforms currently implemented in the library. It is not yet comprehensive, so it would be really nice if you can help me add more!

9.3.1 Gliders

Gliders are lifeforms that oscillate but move while oscillating

`seagull.lifeforms.gliders.Glider`()	

9.3.2 Growers

Growers are lifeforms that exhibit asymptotically unbounded growth

`seagull.lifeforms.growers.Unbounded`()	A lifeform with asymptotically unbounded growth

9.3.3 Oscillators

Oscillators are lifeforms that returns to its initial configuration after some time

`seagull.lifeforms.oscillators.Blinker`([length])	A horizontal Blinker lifeform
`seagull.lifeforms.oscillators.Toad`()	A Toad lifeform oscillator
`seagull.lifeforms.oscillators.Pulsar`()	A Pulsar lifeform oscillator

9.3.4 Random

Random lifeforms are generated on-the-fly without specific configuration

`seagull.lifeforms.random.RandomBox`([shape, seed])	A random box with arbitrarily-set shape

9.3.5 Static

Static lifeforms do not oscillate nor move given classic Conway rules

`seagull.lifeforms.static.Box()`	A static Box
`seagull.lifeforms.static.Seed()`	A static Seed
`seagull.lifeforms.static.Moon()`	A static Moon
`seagull.lifeforms.static.Kite()`	A static Kite

CHAPTER

TEN

RULES AND UTILITIES

10.1 Rules

Rules determine how the evolution of the lifeforms will progress. In Seagull, rules are implemented as a function that takes in a 2-dimensional array of a given shape then returns the updated array with the rule applied

`seagull.rules.`**`conway_classic`**(*X*)
The classic Conway's Rule for Game of Life (B3/S23)

Return type ndarray

`seagull.rules.`**`life_rule`**(*X*, *rulestring*)
A generalized life rule that accepts a rulestring in B/S notation

Rulestrings are commonly expressed in the B/S notation where B (birth) is a list of all numbers of live neighbors that cause a dead cell to come alive, and S (survival) is a list of all the numbers of live neighbors that cause a live cell to remain alive.

Parameters

- **X** (*np.ndarray*) – The input board matrix
- **rulestring** (*str*) – The rulestring in B/S notation

Returns Updated board after applying the rule

Return type np.ndarray

10.2 Utilities

This module contains various utility functions to help with processing

10.2.1 Statistics

Statistics contain various computations to characterize a board state

`seagull.utils.statistics.`**`cell_coverage`**(*state*)
Compute for the live cell coverage for the whole board

Parameters **state** (numpy.ndarray) – The board state to compute statistics from

Returns Cell coverage

Return type float

seagull.utils.statistics.**shannon_entropy**(*state*)

Compute for the shannon entropy for the whole board

Parameters **state** (numpy.ndarray) – The board state to compute statistics from

Returns Shannon entropy

Return type float

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

G

L

M

R

S

V