Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

Conway’s Game of Life
Extemporary U2

Data 7B

Daniel Valdés Artiles

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

What is Conway's Game of Life?

The Game of Life, also known as Life, is a captivating cellular automaton created by the British mathematician John Horton Conway in 1970.

As a zero-player game, Life's evolution is entirely determined by its initial state, without the need for any additional input.
Players engage with the Game of Life by setting up an initial configuration of cells and then observing how these cells evolve over time according to simple rules. Despite its simplicity, Life is remarkable for its ability to exhibit complex and unpredictable behaviors.

It is Turing complete, meaning it can simulate a universal constructor or any other Turing machine, making it a fascinating subject of study in mathematics, computer science, and artificial life.

Why is Cython better in performance than pure python?

Compiling Python code is an advanced method utilized to boost the performance of essential applications, often constituting a dedicated section in the notebook.

Through the utilization of Cython and C Extensions, Python scripts can be transformed into C code, resulting in substantial enhancements in execution speed, particularly for demanding operations and intricate mathematical computations. Cython proves particularly valuable as it permits developers to write Python code with annotations for data types, which are then compiled into native C code.

This process improves execution by reducing Python's dynamic type management overhead, allowing for more direct and swift execution at the machine level. Moreover, C extension modules offer the possibility to implement crucial sections of the code directly in C, ideal for functions necessitating maximum performance optimization.

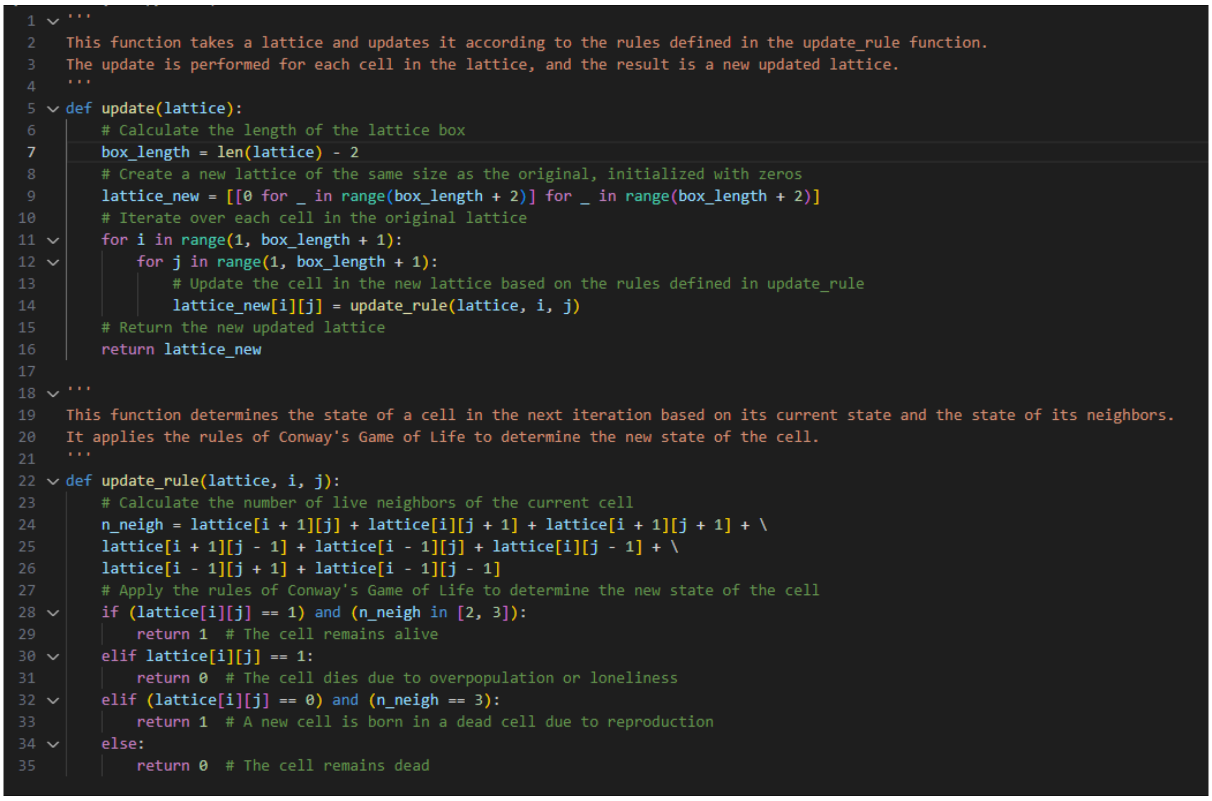
Practical demonstrations in this section encompass loop optimization, where loops conducting intensive computational tasks are rewritten in Cython employing C types to minimize execution time.

Consequently, C compilation facilitates direct access to C libraries, streamlining integration with low-level APIs, and enabling the utilization of optimized algorithms within the C ecosystem.

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

Python pure.

Pure Python is less efficient than Cython due to its interpreted nature, dynamic typing, and associated language overhead. Cython, on the other hand, allows for compiling Python code into C or C++, resulting in faster and optimized code that runs directly on the hardware. Furthermore, Cython enables specifying static types for variables and function arguments, eliminating the need for dynamic type checks at runtime and allowing the compiler to apply more aggressive optimizations.

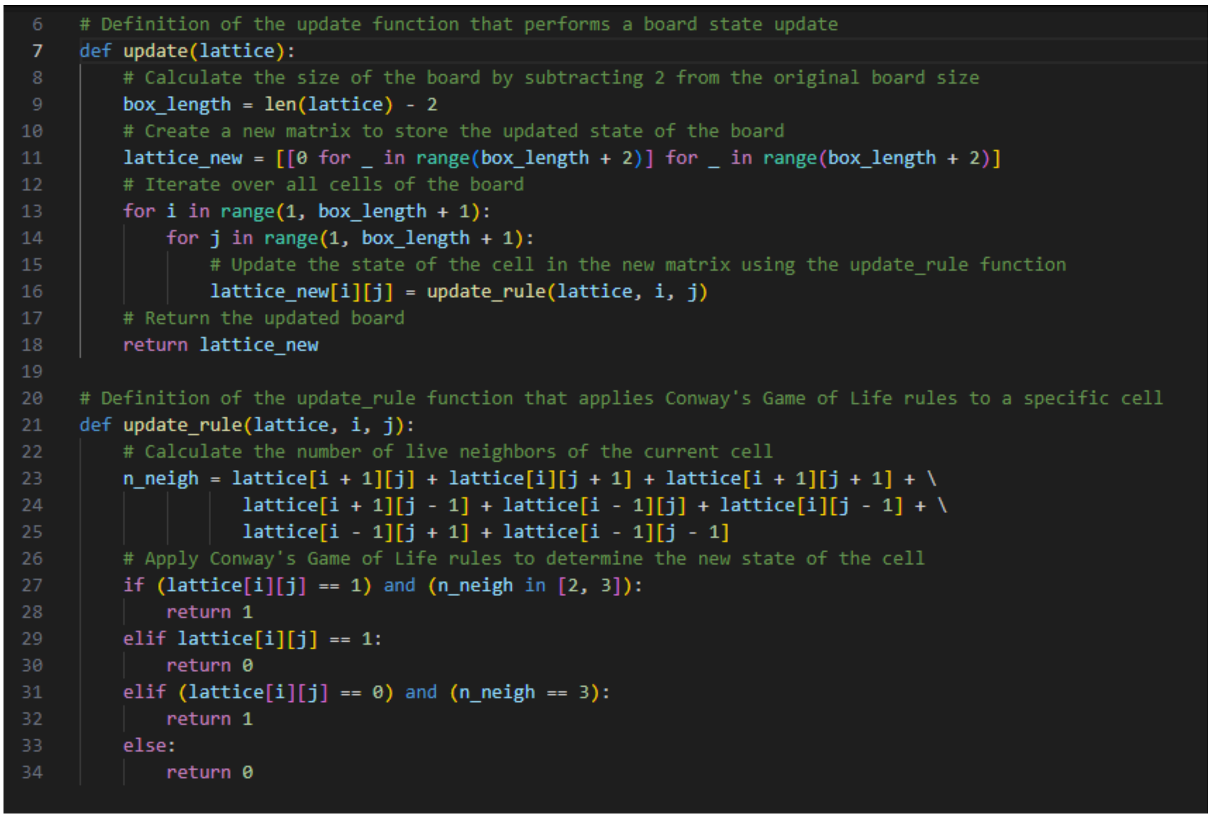


```python
'''
This function takes a lattice and updates it according to the rules defined in the update_rule function.
The update is performed for each cell in the lattice, and the result is a new updated lattice.
'''
def update(lattice):
    # Calculate the length of the lattice box
    box_length = len(lattice) - 2
    # Create a new lattice of the same size as the original, initialized with zeros
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]
    # Iterate over each cell in the original lattice
    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            # Update the cell in the new lattice based on the rules defined in update_rule
            lattice_new[i][j] = update_rule(lattice, i, j)
    # Return the new updated lattice
    return lattice_new

'''
This function determines the state of a cell in the next iteration based on its current state and the state of its neighbors.
It applies the rules of Conway's Game of Life to determine the new state of the cell.
'''
def update_rule(lattice, i, j):
    # Calculate the number of live neighbors of the current cell
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
    lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
    lattice[i - 1][j + 1] + lattice[i - 1][j - 1]
    # Apply the rules of Conway's Game of Life to determine the new state of the cell
    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1  # The cell remains alive
    elif lattice[i][j] == 1:
        return 0  # The cell dies due to overpopulation or loneliness
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1  # A new cell is born in a dead cell due to reproduction
    else:
        return 0  # The cell remains dead
```

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

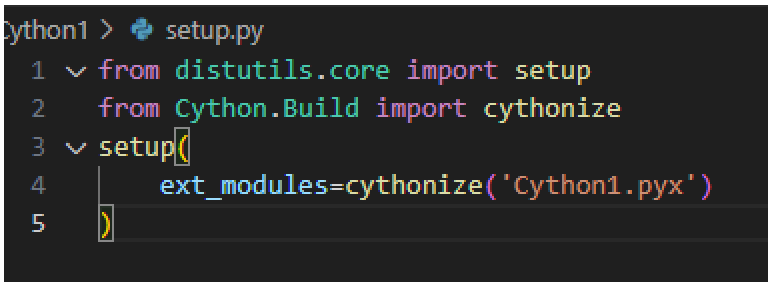
Cython Solution 1.

Solution 1 using Cython shows us that with the same code we use in pure Python, we can optimize efficiency by translating it into compiled C language, without the need for modifications in the raw code. Simply ceasing to use the interpreter language improves the performance of the created algorithms almost twofold.



```python
# Definition of the update function that performs a board state update
def update(lattice):
    # Calculate the size of the board by subtracting 2 from the original board size
    box_length = len(lattice) - 2
    # Create a new matrix to store the updated state of the board
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]
    # Iterate over all cells of the board
    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            # Update the state of the cell in the new matrix using the update_rule function
            lattice_new[i][j] = update_rule(lattice, i, j)
    # Return the updated board
    return lattice_new

# Definition of the update_rule function that applies Conway's Game of Life rules to a specific cell
def update_rule(lattice, i, j):
    # Calculate the number of live neighbors of the current cell
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]
    # Apply Conway's Game of Life rules to determine the new state of the cell
    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1
    elif lattice[i][j] == 1:
        return 0
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1
    else:
        return 0
```

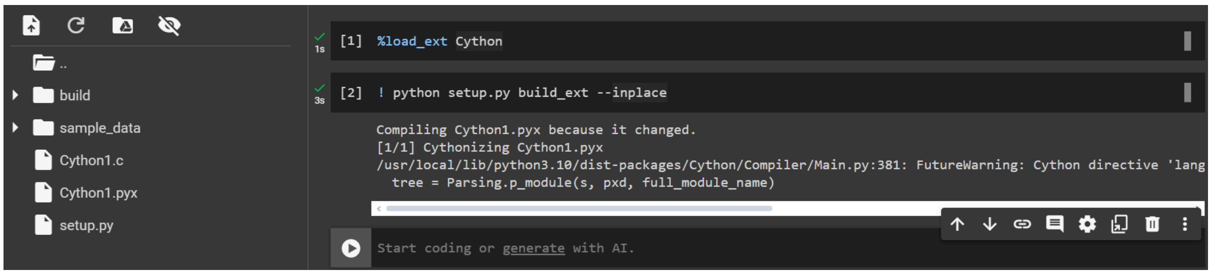
This is the setup.py script used to compile Cython code



ython1 > setup.py

```python
from distutils.core import setup
from Cython.Build import cythonize
setup(
    ext_modules=cythonize('Cython1.pyx')
)
```

Compilation process:
Code ! python setup.py build_ext --inplace run the setup file and generate the Cython.c file in colab environment



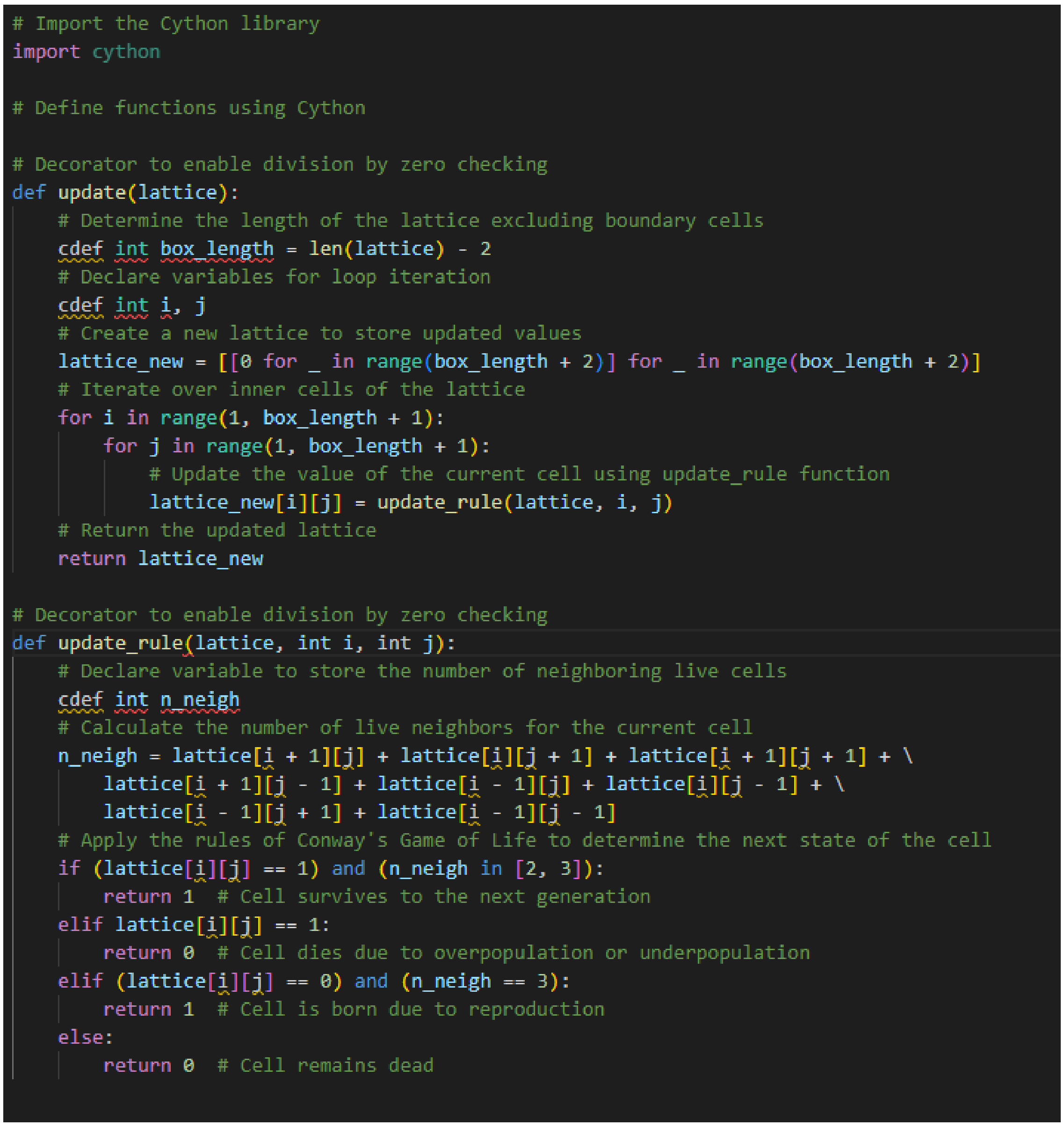
```
[1] %load_ext Cython

[2] ! python setup.py build_ext --inplace

Compiling Cython1.pyx because it changed.
[1/1] Cythonizing Cython1.pyx
/usr/local/lib/python3.10/dist-packages/Cython/Compiler/Main.py:381: FutureWarning: Cython directive 'lang
  tree = Parsing.p_module(s, pxd, full_module_name)
```

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

Cython Solution 2

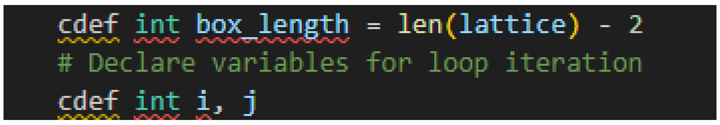
This solution uses the "cdef" statements which allow us to establish variables of a specific type. This way, we can convert Python interpreter code into compiled C code more efficiently. This, combined with the use of compilers instead of interpreters when executing the algorithm, makes the second Cython solution slightly more efficient than the first one.

This is the full code.

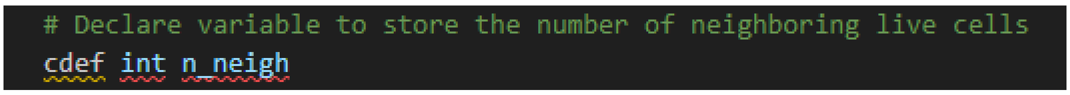


```python
# Import the Cython library
import cython

# Define functions using Cython

# Decorator to enable division by zero checking
def update(lattice):
    # Determine the length of the lattice excluding boundary cells
    cdef int box_length = len(lattice) - 2
    # Declare variables for loop iteration
    cdef int i, j
    # Create a new lattice to store updated values
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]
    # Iterate over inner cells of the lattice
    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            # Update the value of the current cell using update_rule function
            lattice_new[i][j] = update_rule(lattice, i, j)
    # Return the updated lattice
    return lattice_new

# Decorator to enable division by zero checking
def update_rule(lattice, int i, int j):
    # Declare variable to store the number of neighboring live cells
    cdef int n_neigh
    # Calculate the number of live neighbors for the current cell
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
        lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
        lattice[i - 1][j + 1] + lattice[i - 1][j - 1]
    # Apply the rules of Conway's Game of Life to determine the next state of the cell
    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1  # Cell survives to the next generation
    elif lattice[i][j] == 1:
        return 0  # Cell dies due to overpopulation or underpopulation
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1  # Cell is born due to reproduction
    else:
        return 0  # Cell remains dead
```

Here you can see the static variables defined using cdef



```python
cdef int box_length = len(lattice) - 2
# Declare variables for loop iteration
cdef int i, j
```



```python
# Declare variable to store the number of neighboring live cells
cdef int n_neigh
```

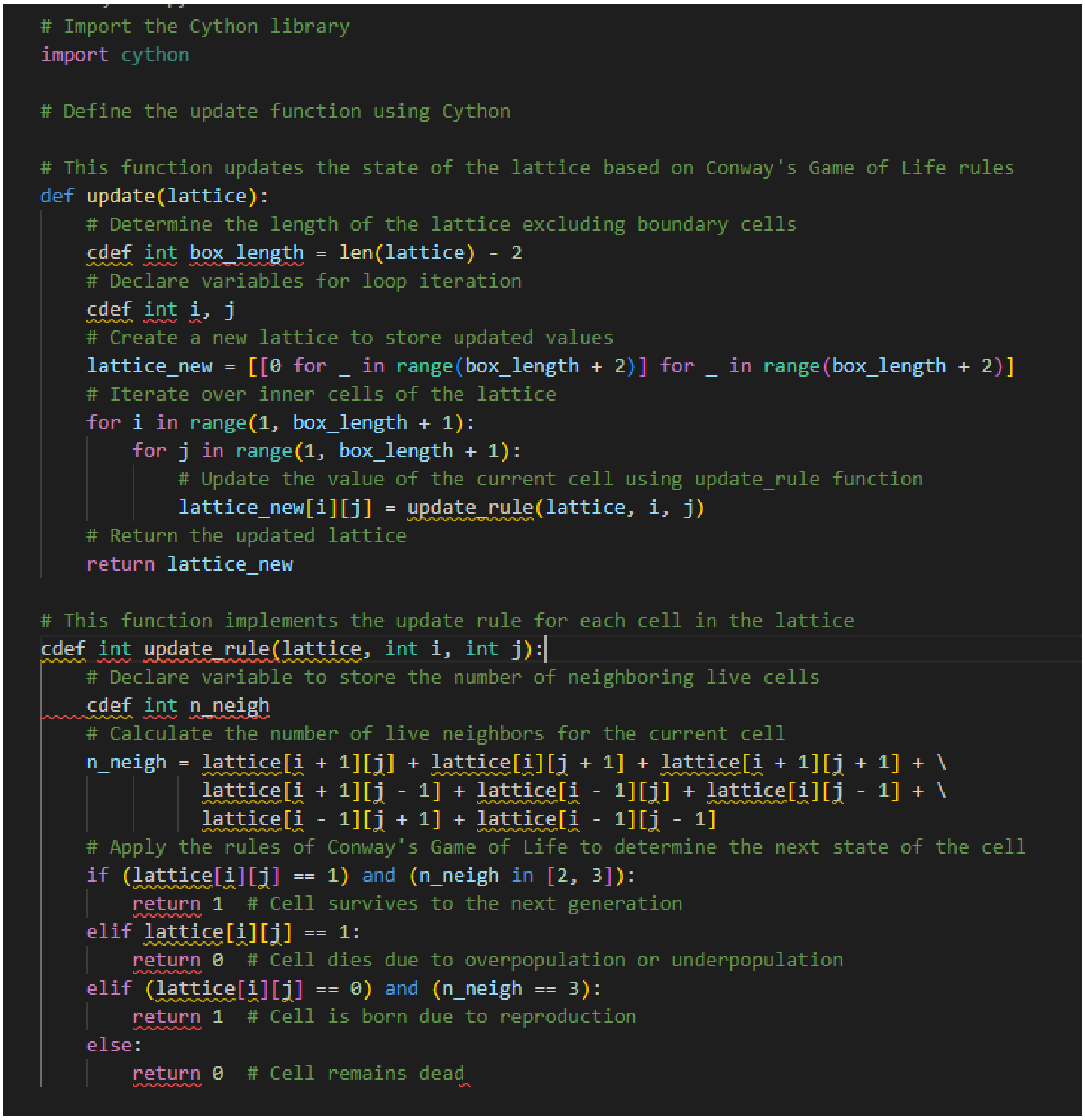
Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

Cython Solution 3

This code is more efficient than code 2 due to the use of the “cdef” declaration for the update_rule function and the “n_neigh” variable. By declaring the “update_rule” function with “cdef”, we are instructing Cython that this function should be compiled as a C function, which can lead to faster execution compared to a pure Python function.

Furthermore, by declaring the “n_neigh” variable as an integer using “cdef”, we are specifying a static type for this variable, allowing Cython to generate more efficient code by eliminating the overhead associated with dynamic typing in Python.

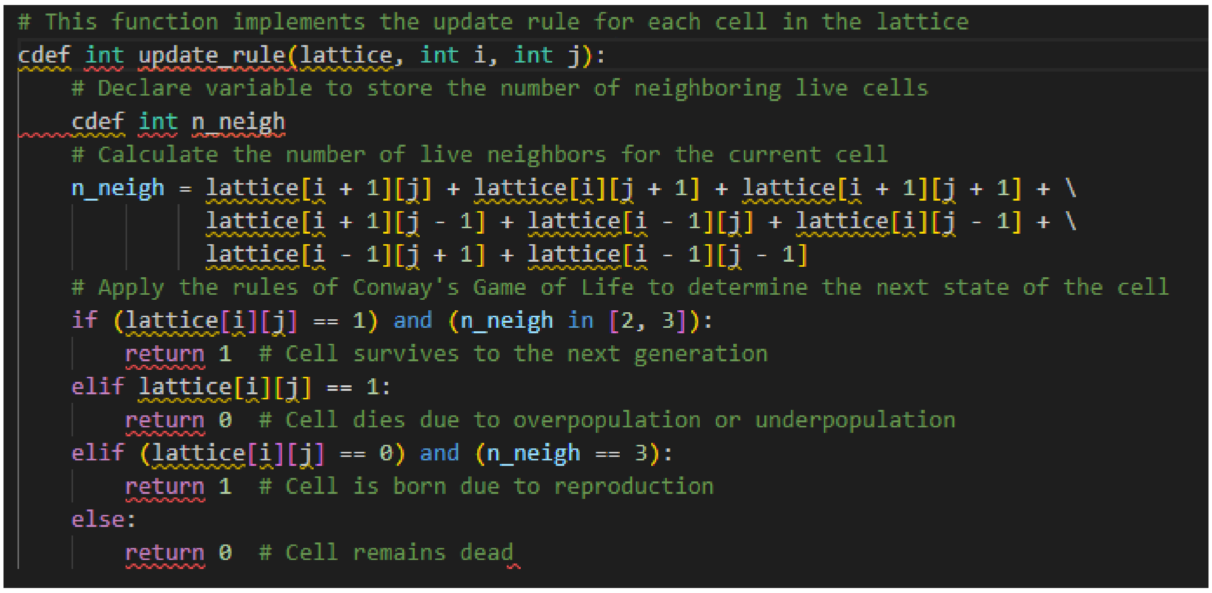
Full code.



```
# Import the Cython library
import cython

# Define the update function using Cython

# This function updates the state of the lattice based on Conway's Game of Life rules
def update(lattice):
    # Determine the length of the lattice excluding boundary cells
    cdef int box_length = len(lattice) - 2
    # Declare variables for loop iteration
    cdef int i, j
    # Create a new lattice to store updated values
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]
    # Iterate over inner cells of the lattice
    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            # Update the value of the current cell using update_rule function
            lattice_new[i][j] = update_rule(lattice, i, j)
    # Return the updated lattice
    return lattice_new

# This function implements the update rule for each cell in the lattice
cdef int update_rule(lattice, int i, int j):
    # Declare variable to store the number of neighboring live cells
    cdef int n_neigh
    # Calculate the number of live neighbors for the current cell
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]
    # Apply the rules of Conway's Game of Life to determine the next state of the cell
    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1  # Cell survives to the next generation
    elif lattice[i][j] == 1:
        return 0  # Cell dies due to overpopulation or underpopulation
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1  # Cell is born due to reproduction
    else:
        return 0  # Cell remains dead
```

Function typed using cdef to be more efficient.

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly



```
# This function implements the update rule for each cell in the lattice
cdef int update_rule(lattice, int i, int j):
    # Declare variable to store the number of neighboring live cells
    cdef int n_neigh
    # Calculate the number of live neighbors for the current cell
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]
    # Apply the rules of Conway's Game of Life to determine the next state of the cell
    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1  # Cell survives to the next generation
    elif lattice[i][j] == 1:
        return 0  # Cell dies due to overpopulation or underpopulation
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1  # Cell is born due to reproduction
    else:
        return 0  # Cell remains dead
```

Cython Solution 4

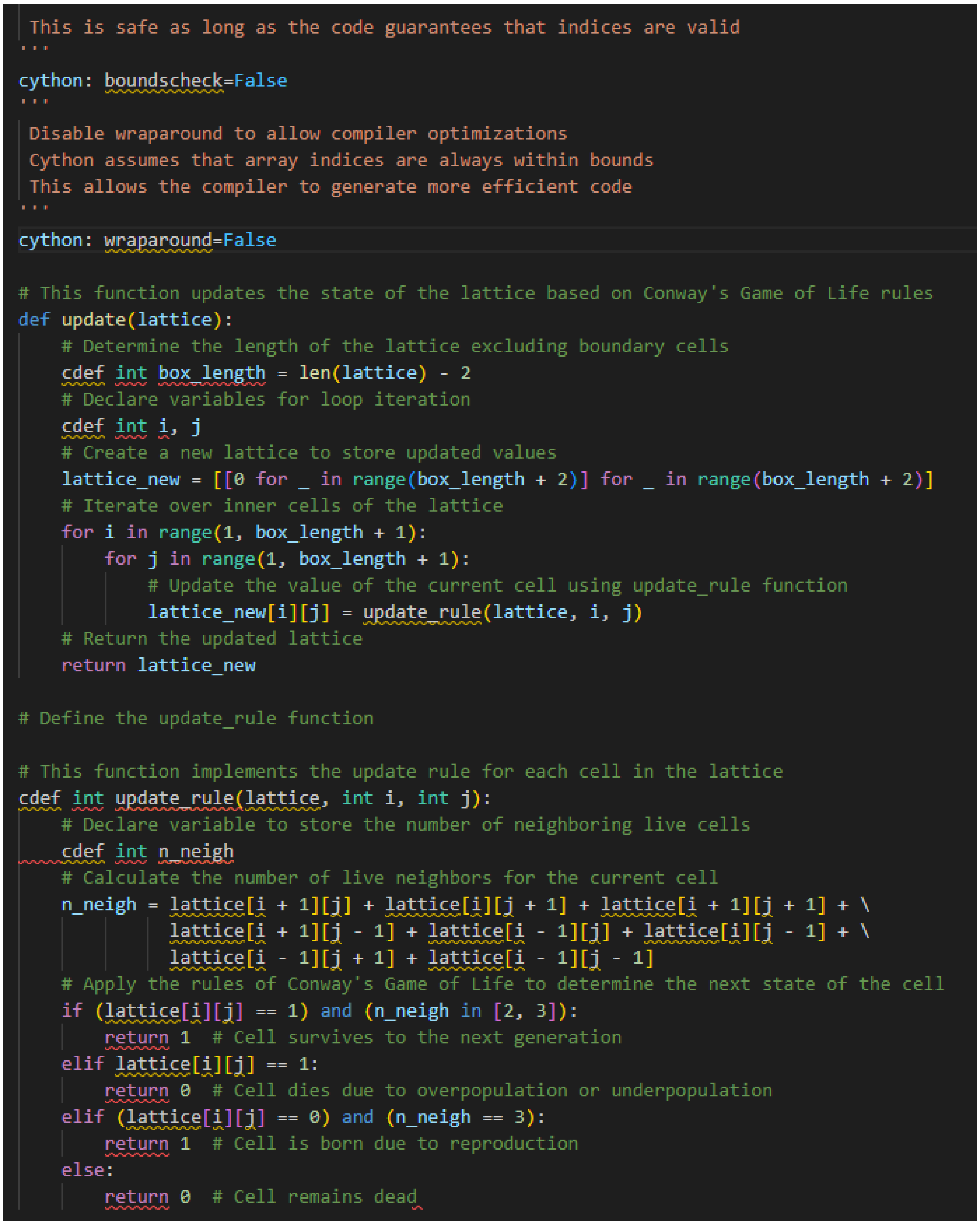
This solution is essentially similar to solution3, but includes two additional Cython directives: boundscheck=False and wraparound=False.

The boundscheck=False directive disables boundary checking in loops, meaning Cython assumes that array indices are always within bounds. This can allow the compiler to generate more efficient code by eliminating the overhead associated with boundary checking at runtime.

The wraparound=False directive also disables a form of boundary checking in loops, but instead of checking if indices are within bounds, this directive allows the compiler to optimize the code by assuming that indices are always within bounds. This can lead to greater efficiency by enabling the compiler to perform more optimizations.

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

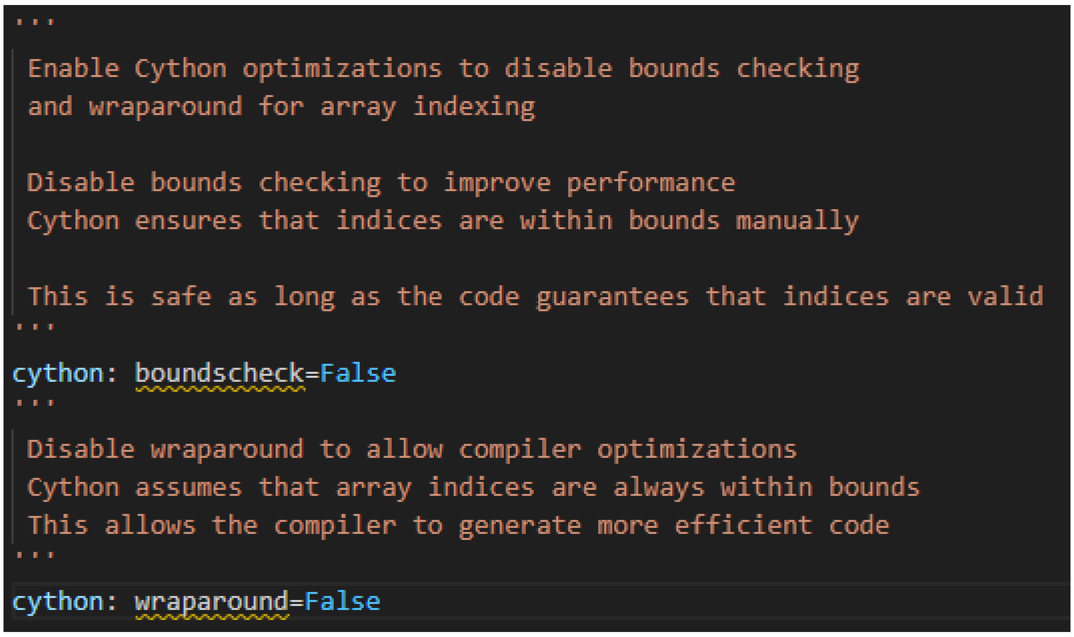
Full Code.



```
 This is safe as long as the code guarantees that indices are valid
'''
cython: boundscheck=False
'''
 Disable wraparound to allow compiler optimizations
 Cython assumes that array indices are always within bounds
 This allows the compiler to generate more efficient code
'''
cython: wraparound=False

# This function updates the state of the lattice based on Conway's Game of Life rules
def update(lattice):
    # Determine the length of the lattice excluding boundary cells
    cdef int box_length = len(lattice) - 2
    # Declare variables for loop iteration
    cdef int i, j
    # Create a new lattice to store updated values
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]
    # Iterate over inner cells of the lattice
    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            # Update the value of the current cell using update_rule function
            lattice_new[i][j] = update_rule(lattice, i, j)
    # Return the updated lattice
    return lattice_new

# Define the update_rule function

# This function implements the update rule for each cell in the lattice
cdef int update_rule(lattice, int i, int j):
    # Declare variable to store the number of neighboring live cells
    cdef int n_neigh
    # Calculate the number of live neighbors for the current cell
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]
    # Apply the rules of Conway's Game of Life to determine the next state of the cell
    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1  # Cell survives to the next generation
    elif lattice[i][j] == 1:
        return 0  # Cell dies due to overpopulation or underpopulation
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1  # Cell is born due to reproduction
    else:
        return 0  # Cell remains dead
```

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

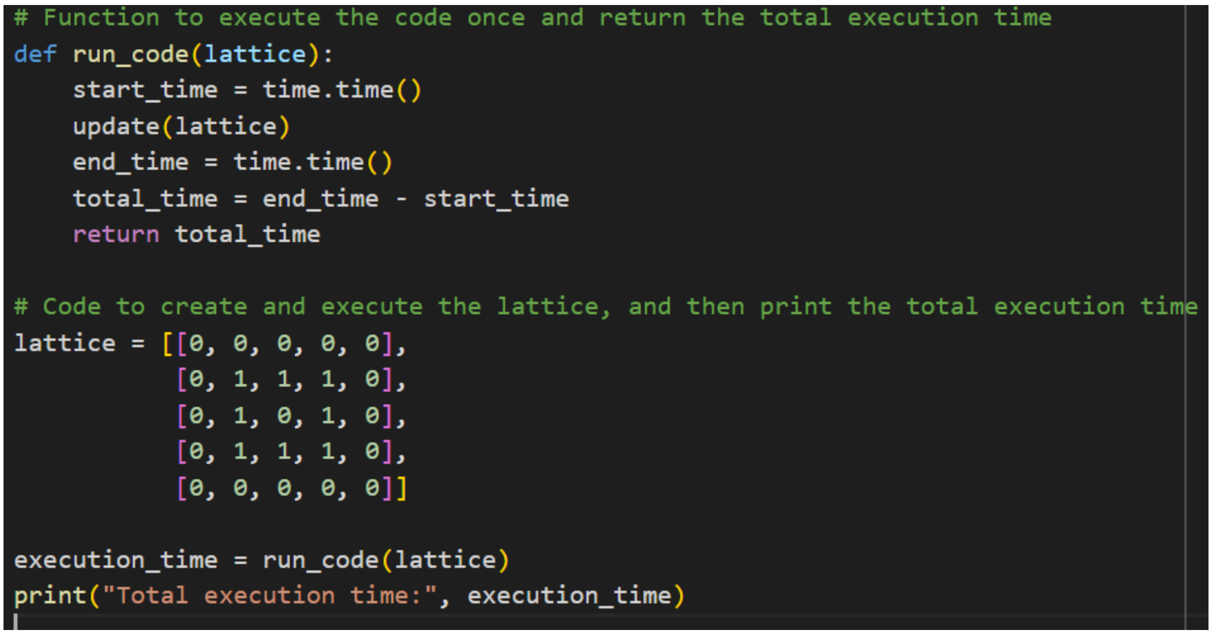
Directives added:



```
'''
 Enable Cython optimizations to disable bounds checking
 and wraparound for array indexing

 Disable bounds checking to improve performance
 Cython ensures that indices are within bounds manually

 This is safe as long as the code guarantees that indices are valid
'''
cython: boundscheck=False
'''
 Disable wraparound to allow compiler optimizations
 Cython assumes that array indices are always within bounds
 This allows the compiler to generate more efficient code
'''
cython: wraparound=False
```

Benchmarking.

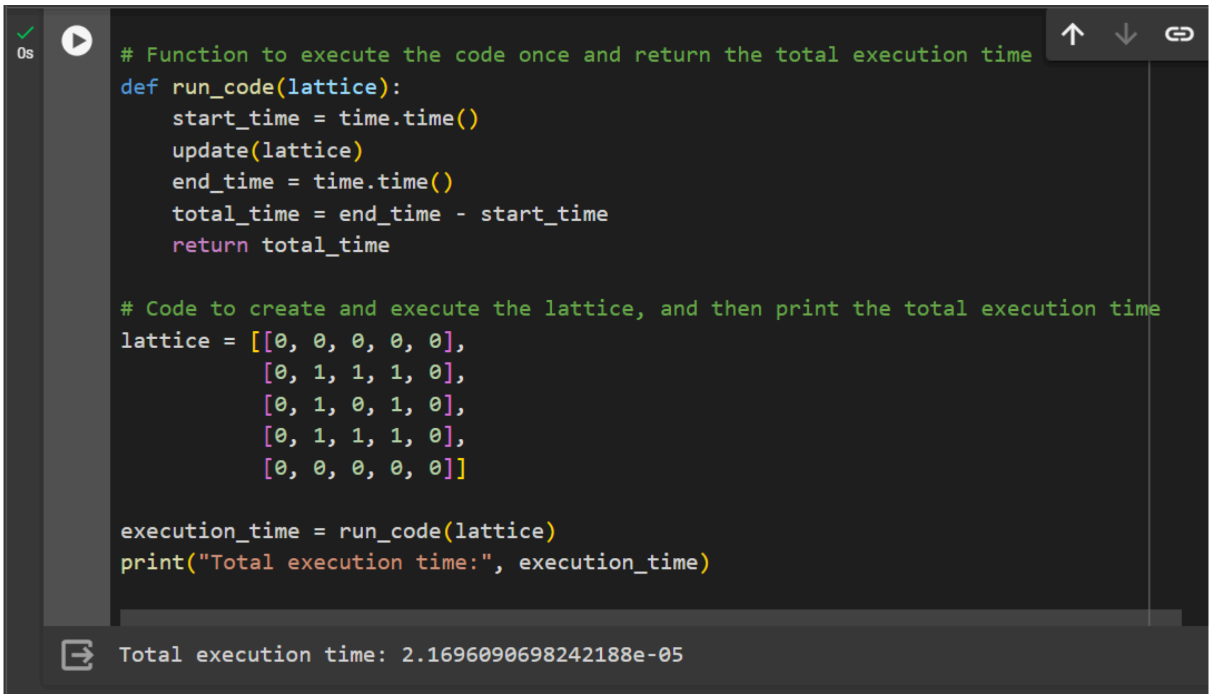
I used the library "Time" with the function run_code(lattice) to measure the execution time.



```
# Function to execute the code once and return the total execution time
def run_code(lattice):
    start_time = time.time()
    update(lattice)
    end_time = time.time()
    total_time = end_time - start_time
    return total_time

# Code to create and execute the lattice, and then print the total execution time
lattice = [[0, 0, 0, 0, 0],
           [0, 1, 1, 1, 0],
           [0, 1, 0, 1, 0],
           [0, 1, 1, 1, 0],
           [0, 0, 0, 0, 0]]

execution_time = run_code(lattice)
print("Total execution time:", execution_time)
```

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

Python pure execution time.



```
# Function to execute the code once and return the total execution time
def run_code(lattice):
    start_time = time.time()
    update(lattice)
    end_time = time.time()
    total_time = end_time - start_time
    return total_time

# Code to create and execute the lattice, and then print the total execution time
lattice = [[0, 0, 0, 0, 0],
           [0, 1, 1, 1, 0],
           [0, 1, 0, 1, 0],
           [0, 1, 1, 1, 0],
           [0, 0, 0, 0, 0]]

execution_time = run_code(lattice)
print("Total execution time:", execution_time)
```

```
Total execution time: 2.1696090698242188e-05
```

Note: Image was taken from Visual Code Studio, that have some bad configurations, and mark errors, when code was used on Google collab, it works properly

References:

[1] Contributors to Wikimedia projects. “Conway's Game of Life - Wikipedia”. Wikipedia, the free encyclopedia. Accedido el 26 de abril de 2024. [En línea]. Disponible: https://en.wikipedia.org/wiki/Conway's_Game_of_Life#:~:text=It%20is%20a%20zero-player,or%20any%20other%20Turing%20machine.

[2] D. Gamboa, *Cython*. Merida, Yucatan: pag:(25 - 35), 2024.

[3] Kumar, S. (2022, January 5). Speed-up your Numpy Operations with Num Expr Package. Medium. https://towardsdatascience.com/speed-upyour-numpy-operations-with-numexpr-package-1bead7f5b520

[4] NumExpr 2.0 User Guide — numexpr 2.8.5.dev1 documentation. (n.d.). https://numexpr.readthedocs.io/en/latest/userguide.html