# CA_Numpy_Inheritance

February 23, 2017

Testing and Documentation Notebook for Cellular Automaton Using Inheritance from NumPy NDArray

This document is for testing/demonstrating a cellular automaton class, CAGrid, which inherits much of its functionality from NumPy array objects. The CAGrid class enables use of the array operations (implemented in C) which should make the execution speed better. This document assumes the reader is somewhat familiar with NumPy. It also assumes the user is somewhat familiar with classes and inheritance in Python. If these concepts are not familiar, the following are good resources to consult. - Numpy has a Quick Start Guide. Especially important to understanding this learning module are the concepts of array views, NumPy Data types (dtype), and NumPy structured arrays. - Jessica Hamrick has a great Introduction to Classes and Inheritance (in Python). This is especially good for those who have not used object oriented programing before. - The Python Tutorial describes classes in Python. This includes A First Look, a description of Class and Instance Variables, and a discussion of Inheritance.

Even for those familiar with NumPy arrays and Python classes/inheritance, the inheritance of NumPy arrays is rather unique. The document Subclassing ndarray should be consulted to understand the inheritance of NumPy.ndarray to form the class CAGrid used in this notebook.

## 1 Results

The video below is a game of life simulation using the code in this module.

## 2 Initialization

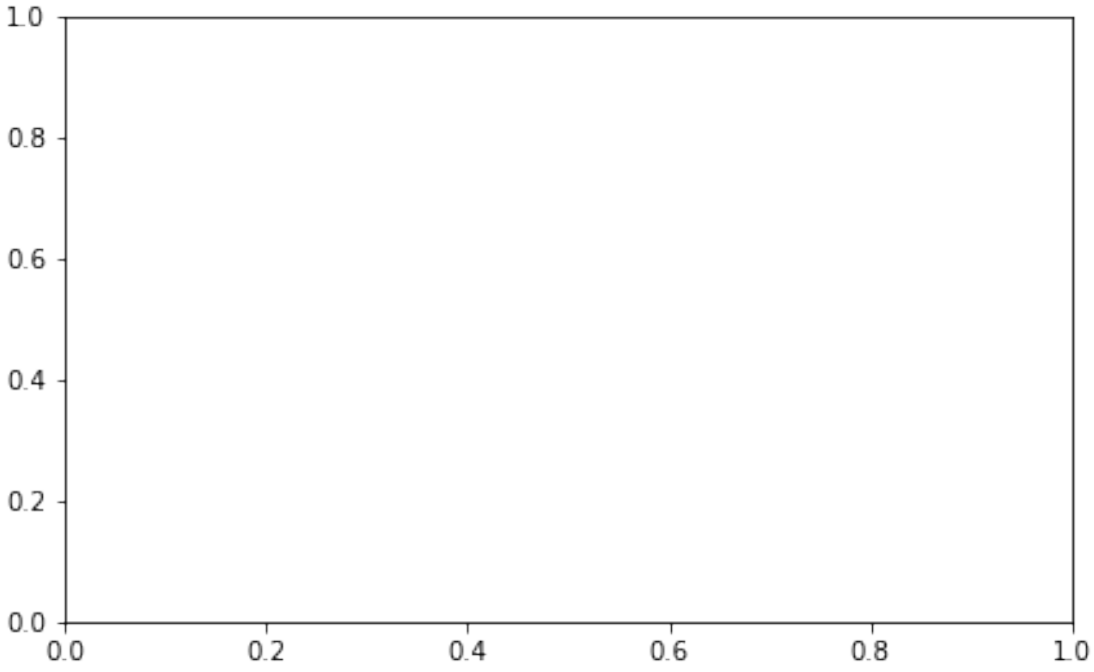
```
In [1]: from CellularAutomaton import *
        from IPython.display import display
        import inspect
        from math import *

        #for plotting in this notebook.
        %matplotlib inline
        import matplotlib
        from matplotlib import pyplot as plt
        from matplotlib import animation
        matplotlib.rc('animation', html='html5')
        matplotlib.rcParams['image.cmap'] = 'jet'    #Set matshow colors to v 1.0 st
        #set the plot aspect ratio and setup a default figure to use.
```

```
goldenratio=1/2*(1+sqrt(5)); fsx=7; fsy=fsx/goldenratio
MyFigure, MyAxes = plt.subplots(1,1,figsize=(fsx,fsy));

#Function to return the function name, parameters and doc string.
def InfoDocString(function):
    """Function to return function definition and document string."""

    code = inspect.getsource(function)
    docstring = inspect.getdoc(function)
    return code[0:code.find(':')+1].lstrip() +'\n' + docstring
```




### 2.0.1 Functions to help with testing

In [2]:
```
MyDtype = numpy.dtype([('Value','f'),('State',bool)])

def setstates(grid, rows, columns, states):
    i=0
    for y in range(rows):
        for x in range(columns):
            grid["State"][y][x]=states[i]
            i = i + 1
    grid.SetValue()
    grid.SetBoundary()
```

In [3]:
```
def setrandomstate(grid,ProbTrue):
```

```
a=numpy.random.randint(0,100,grid.shape,dtype='i8')
numpy.copyto(grid["State"],ProbTrue>a)
```

## 3 Background

This document introduces cellular automaton calculations. It demonstrates how cellular automaton calculations could be performed using a class which inherits NumPy arrays. This class implements rules for the Game of Life. This class may be used for other cellular automaton simulations by either inheriting this class and changing the update rules or by editing the update function in the CAGrid code directly. ## Game of Life Update Rules In the game of life, the cellular automaton grid is a 2D grid in which each cell is either 'alive' or 'dead.' The cellular automaton progresses through time by updating each cell's state based on the state of its neighbor cells in the previous time step. The game of life treats diagonal cells as neighbors so there are eight nearest neighbors. Let $Count = \sum_{Neighbors} State_{n-1}$, each cell's value at the 'n' time step is then: - $State_n = False\, if\, Count > 3$ (Too much competition, cell dies)

- $State_n = True\, if\, Count = 3$ (Cell birth or cell stays Alive)
- $State_n = False\, if\, Count < 2$ (Not enough neighbors, cells dies or stays dead)
- $State_n = Unchanged\, if\, Count = 2$

Note that the rule for $Count = 2$ needs no code to implement. The cell state can be just left unchanged from the $n-1$ state. # CAGrid Class Structure The class CAGrid (or cellular automaton grid) inherits a NumPy ndarry and adds extra functionality to specialize the array for cellular automaton simulations. In this implementation, each cell element is an array location. The class also creates arrays for the neighbors in each direction. In addition to the structure and functionality provided by NumPy arrays, the array structure should allow faster simulation calculations as array operations can be executed in C using BLAS subroutines rather than interpreted Python code. The class doc string, shown below, describes the class.

```
In [4]: print(InfoDocString(CAGrid))

class CAGrid(numpy.ndarray):
Class for implementing cellular automaton grid simulations.

Provides a framework for implementing 2D cellular automaton (CA) simulations using
Each instance of the class has a NumPy array for the cell data.  The class automati
boundary cells surrounding the simulation grid.  It also creates views into the sim
boundary cells, and neighbor cells for use with NumPy array calculation calls.

Since this class inherits numpy.ndarray, several unique approaches must be used (es
class instance creation.)  The document at
https://docs.scipy.org/doc/numpy/user/basics.subclassing.html explains using ndarra
subclass and these special approaches.

Also important is the concept of an array view.  A view to an array does not use ne
memory, but simply 'views' the data which already exists.  This is explained some a
```

```
https://docs.scipy.org/doc/numpy-dev/user/quickstart.html#copies-and-views
Because of this efficiency, this class creates many views into the grid.  These vie
used later for efficient calculation through NumPy calls rather than iterating over
through interpreted python code.

Class methods and attributes Inherited:
All the instance variables associated with Numpy ndarray such as .shape, .dtype
A complete list of ndarray methods and attributes is available at
https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.ndarray.html#nump


The new class methods are:
__new__ -> class initialization method.
            The array (or grid) shape must be provided.  The constructor will not t
            It creates an array 2 cells (or nodes) larger (in each direction) than
            argument passed.  This allows for boundary cells in each direction.  Th
            constructor returns a NumPy array for the main simulation data. This is
            a view into the data of the bigger array which also contains boundary c
            The bigger array is available through .Base or .base. The constructor a
            creates view arrays for the neighbors to the top, top right, left, bott

__array_finalize_ ->  Always called after initialization.  It does not currently do

__init__ -> Not used since it is not always called for subclasses of ndarray.

Update ->   Implements the rules of the cellular automaton
            Currently implements a simulation using 'game of life rules.'
            To implement other cellular automaton simulations, either:
                1.) Change the code in this file (especially Update)
                2.) Create a new class which inherits this class with
                    class NewClass(CAGrid):
                    and include in that class an update function which will replace
                    one here.

FinishUpdate -> Completes tasks after update
                After cell values are updated, there are several tasks which must b
                to prepare the CAGrid object for display and/or the next call to up
                Currently, this includes calling SetValue and SetBoundary.

SetValue -> Sets the 'Value' field for the structured array.
            This function is somewhat tied to the Game of Life Simulation.
            The purpose of this function is do a fast conversion to float values as
            expected by matplotlib.matshow.

SetBoundary ->  Sets the boundary conditions by copying cells from main simulation
                to the boundary cells.

New Data Attributes or Instance Variables are:
```

```
.Base or .base ->   The large array which includes the boundary cells.
                    .base is the normal attribute for a NumPy array.
                        The array returned by the constructor is actually a view in
                        array.  So NumPy automatically creates this reference.
                    .Base is created in the constructor.  It is probably not needed
                        needed to keep the larger array in memory.

.Neighbors[]  ->    List of arrays which are views which give the cell's neighbor i
                    In other words, one of the neighbors is TopLeft.  Accessing Top
                    returns the grid node at [2][3].
                    Diagonal neighbor arrays are included in the list depending on
                    the class variable CAGrid.IncludeDiagonalNeighbors.  The defaul
                    variable is True.

.Old    ->          Attribute which has the same attributes as the CAGrid object.
                    storing the old state of the object before update if necessary.
.TrueArray  ->      Array of True values for use in mask/array operations.  It is t
                    as the simulation grid, not the base grid.  Used for update cal
                    Declaring .TrueArray makes the code run faster because memory i
                    with each update.  Since it is an instance variable, it is crea
                    initialized only once per simulation.
.count      ->      Used by the game of life update to count number of alive neighb
.UpdateCount    ->  Tracks the number of times Update is called.
```

### 3.1 Update Routine

The Update method should be called each time step. It updates the cell (array) values to the new values. The complete code for the update function is below. Note that rather than stepping through each value of the grid, Update determines the new values using calls to numpy.place. The numpy.place routine is mostly compiled code so it executes much faster than iterating through the cells using Python loops.

```
In [5]: print(inspect.getsource(CAGrid.Update))

    def Update(self):
        """Implements a single time step using Game of Life Rules

        Method currently takes no arguments.

        Method sums the 'Value' of the neighbor cells and then applies the game of
        Method uses array operations for calculations rather than iterators so that
            full advantage of underlying C and BLAS routines rather than slower Pyt

        Some of these routines are listed at:
        https://docs.scipy.org/doc/numpy/reference/routines.indexing.html#inserting
        In this method, numpy.place is used.  It is documented at:
        https://docs.scipy.org/doc/numpy/reference/generated/numpy.place.html#numpy
```

```
        Method also calls self.FinishUpdate
        """

        # Determine number of neighbors
        numpy.place(self.count,self.TrueArray,0)
        for n in self.Neighbors:
            self.count = self.count + n['Value']

        # Implement Game of Life Rules
        # Note:  Self.Old is actually not needed for Game of Life.
        #       It might not be needed for other Update methods either.
        # numpy.copyto(self.Old,self)                          # Default is no change
        numpy.place(self['State'], self.count >3, False)    # If count > 3: new valu
        numpy.place(self['State'], self.count ==3, True)    # If count = 3: new valu
        numpy.place(self['State'], self.count <2, False)    # If count < 2: new valu
        self.FinishUpdate()
        self.UpdateCount += 1
```

## 4 Tests and Simulations

In the cells below, we do some testing of the game of life rules and then run some large simulations demonstrating the game of life rules. These tests give us confidence that our code operates as we expect. ## Displaying Array Data Matplotlib has two function for displaying array data. They are matshow and imshow. - matshow takes a single 2D numpy array. It does automatic scaling so that the colormap range goes from the minimum to maximum value.

- imshow is similar to matshow, but has a few more options. Of primary importance is that we can pass an array of touples to specify the rgb or rgba color values. This would allow us to specif the color at each grid location.

For these game of life simulations, matshow will work well. For more complicated cellular automaton in which each cell has a more complicated state, imshow might work better. ## Test a 3 x 3 Checker Board The checker board pattern should be stable. All cells have exactly three neighbors.

First we create and show the grid.

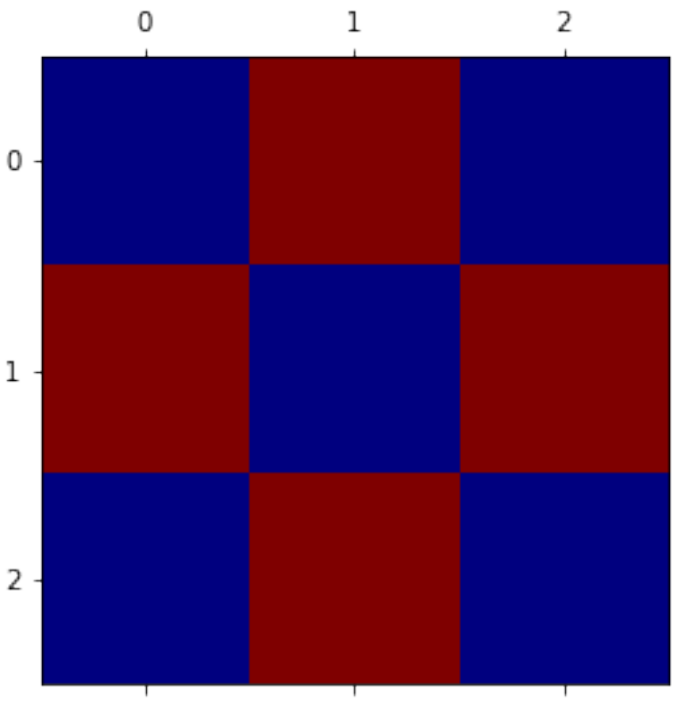
```
In [6]: rows = 3; columns = 3; MyGrid=CAGrid((rows,columns),MyDtype)
        setstates(MyGrid, rows, columns, (0,1,0,1,0,1,0,1,0))
        print(MyGrid['State']); print(MyGrid['Value'])
```

```
[[False  True False]
 [ True False  True]
 [False  True False]]
[[ 0.  1.  0.]
 [ 1.  0.  1.]
```

```
 [ 0.  1.  0.]]
```

```
In [7]: MyAxes.matshow(MyGrid['Value']); MyFigure
```

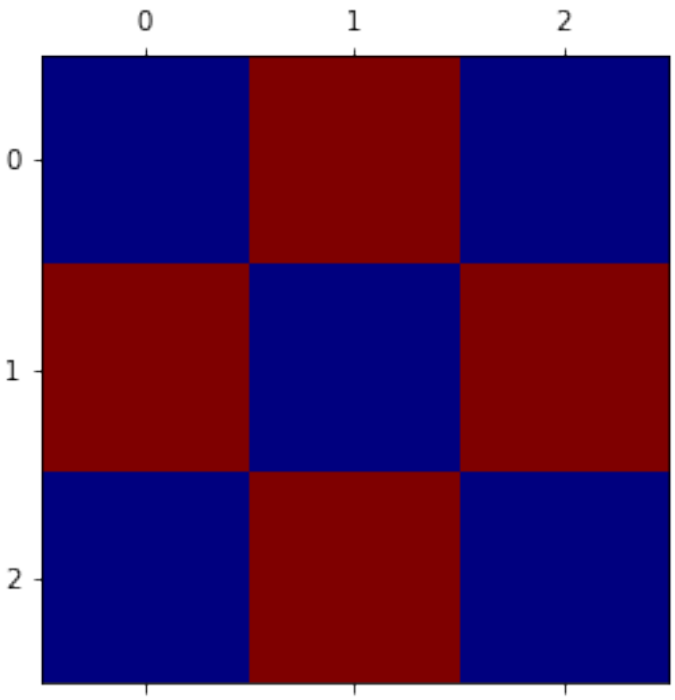
```
Out[7]:
```




Now we update and show the grid.

```
In [8]: MyGrid.Update(); MyAxes.matshow(MyGrid['Value']); MyFigure
```

```
Out[8]:
```




We will update and show the grid one more time. The grid is in a steady state as it should as each 'alive' cell has exactly two neighbors that are also 'alive,' and each 'dead' neighbor is surrounded by four 'alive' neighbors so it always stays 'dead.'

```
In [9]: MyGrid.Update(); print(MyGrid['State']); print(MyGrid['Value'])
```

```
[[False  True False]
 [ True False  True]
 [False  True False]]
[[ 0.  1.  0.]
 [ 1.  0.  1.]
 [ 0.  1.  0.]]
```

Continuous Line

With periodic boundary conditions, a continuous line grows and then dies because of overpopulation.

```
In [10]: rows = 3; columns = 3; MyGrid=CAGrid((rows,columns),MyDtype)
         setstates(MyGrid, rows, columns, (0,0,0,1,1,1,0,0,0))
         print(MyGrid['Value'])
```

```
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 0.  0.  0.]]
```

```
In [11]: MyGrid.Update(); print(MyGrid['Value'])

[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]


In [12]: MyGrid.Update(); print(MyGrid['Value'])

[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Blinker
If we surround the line with empty cells, it is a blinker.

```
In [13]: rows = 4; columns = 4; MyGrid=CAGrid((rows,columns),MyDtype);
         setstates(MyGrid, rows, columns, (0,0,0,0, 0,1,1,1, 0,0,0,0, 0,0,0,0));
         print(MyGrid['Value'])

[[ 0.  0.  0.  0.]
 [ 0.  1.  1.  1.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]


In [14]: MyGrid.Update(); print(MyGrid['Value'])

[[ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  0.]]


In [15]: MyGrid.Update(); print(MyGrid['Value'])

[[ 0.  0.  0.  0.]
 [ 0.  1.  1.  1.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]


In [16]: MyGrid.Update(); print(MyGrid['Value'])

[[ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  0.]]
```

### 4.1 Small Grid Animation Test

We will now run a small simulation to demonstrate this game of life simulation. The final result of this test is an animation of the simulation. Matplotlib has a built in animation module. This notebook describes the necessary background as well well as gives several examples.

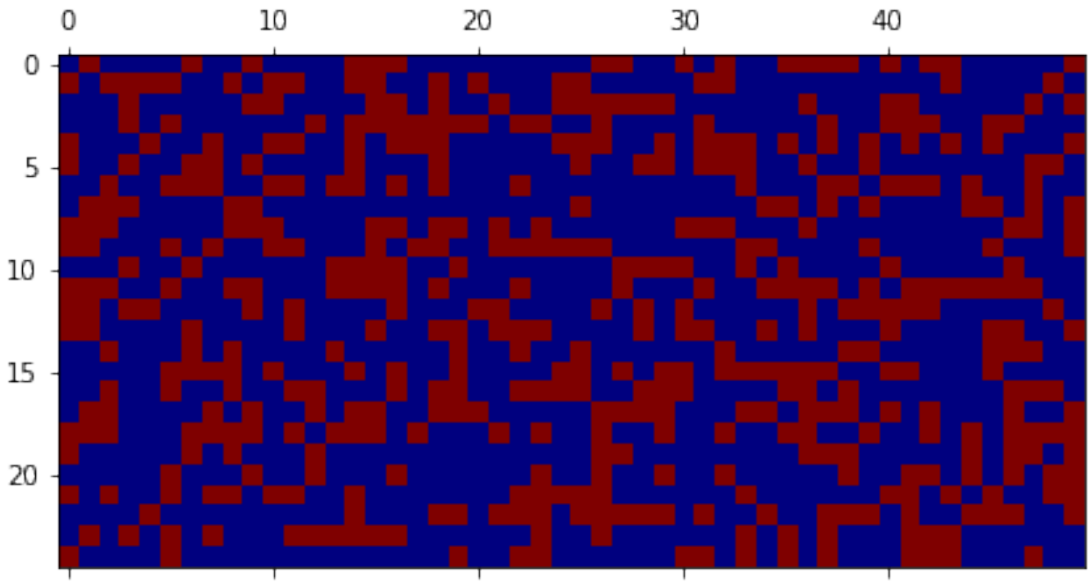
The matplotlib function animation method requires a callable function which updates the figure. This function must take one argument, the sequence number of the image being produced. The `af(n)` function defined below is this function for this simulation.

In the cells below, we will save the animation to an external file, and then include it in the notebook with the html video tag.

In [17]:
```
def af(n):
    MyGrid.Update()
    MyAxes.matshow(MyGrid['Value'])
```

In [18]:
```
rows = 25; columns = 50;
MyGrid=CAGrid((rows,columns),MyDtype)   #initilized with whatever random me
setrandomstate(MyGrid,38)               #set 38% (3/8) cells as 'alive.'
MyGrid.FinishUpdate()
MyAxes.matshow(MyGrid['State'])
MyFigure
```

Out[18]:




In [19]:
```
AnimationFrames = 400  #total number of frames in animation.
DelayBetweenFrames = 33 #in msec (20 gives 50 fps) (33.3 gives 30 fps)
AnimationTime = AnimationFrames * DelayBetweenFrames/1000
print ("The animation will be {:.1f} seconds long.".format(AnimationTime))
```

```
The animation will be 13.2 seconds long.
```

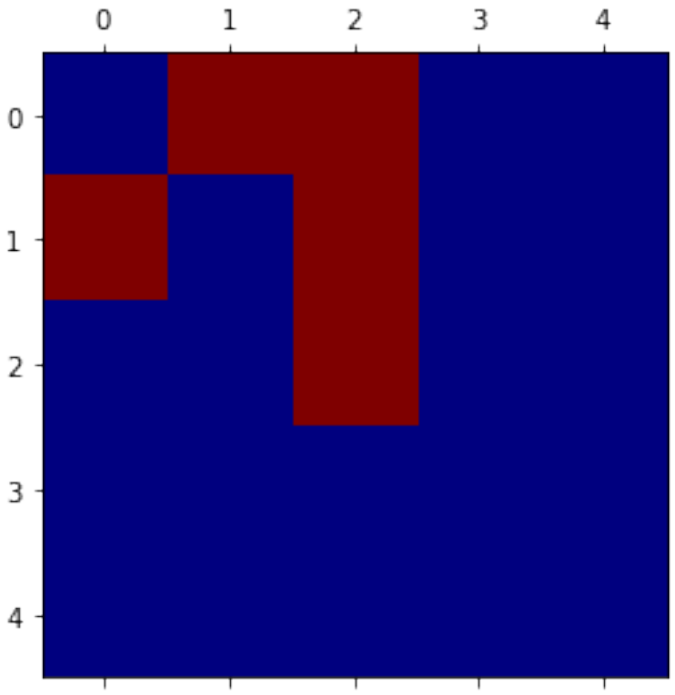
```
In [20]: MyAnimation = animation.FuncAnimation(MyFigure, af,
                                               frames=AnimationFrames, interval=DelayBetwe
         MyAnimation.save('GOL_Small_Grid_test.mp4',extra_args=['-vcodec', 'h264'])
```

Glider Tests
A good test for the game of life simulation is if a glider will act correctly.
Small Grid Example

```
In [21]: rows = 5; columns = 5
         MyGrid=CAGrid((rows,columns),MyDtype)
         #Set all values to zero or 'dead' or False
         numpy.copyto(MyGrid['State'],numpy.zeros((rows,columns),dtype=numpy.dtype
         #Set specific cells to 'alive' or True
         MyGrid['State'][0][1] = True
         MyGrid['State'][1][1] = True
         MyGrid['State'][1][2] = True
         MyGrid['State'][2][0] = True
         MyGrid['State'][2][2] = True

         MyGrid.FinishUpdate()
         MyAxes.matshow(MyGrid['State'])
         MyFigure
```

```
Out[21]:
```



```
In [22]: AnimationFrames = 20  #total number of frames in animation.
         DelayBetweenFrames = 1000 #in msec (20 gives 50 fps) (33.3 gives 30 fps)
         AnimationTime = AnimationFrames * DelayBetweenFrames/1000
         print ("The animation will be {:.1f} seconds long.".format(AnimationTime))
```

```
The animation will be 20.0 seconds long.
```

```
In [23]: MyAnimation = animation.FuncAnimation(MyFigure, af,
                                               frames=AnimationFrames, interval=DelayBetwe
         MyAnimation.save('GOL_Glider_test.mp4',extra_args=['-vcodec', 'h264'])
```

It does indeed return to its initial state.

### 4.1.1 Large Grid Glider Test with Multiple Gliders

Let us put several gliders onto the same grid and let them run. When they collide they produce different patterns.

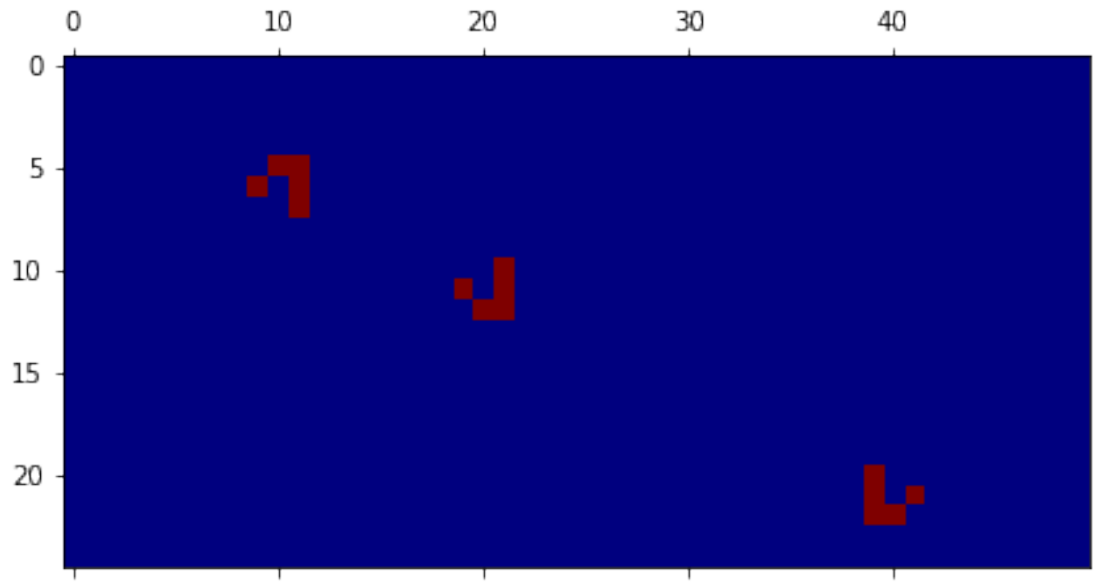
```
In [24]: rows = 25; columns = 50
         MyGrid=CAGrid((rows,columns),MyDtype)
         numpy.copyto(MyGrid['State'],numpy.zeros((rows,columns),dtype=numpy.dtype
         MyGrid['State'][5][10] = True
         MyGrid['State'][6][10] = True
         MyGrid['State'][6][11] = True
         MyGrid['State'][7][9]  = True
         MyGrid['State'][7][11] = True

         MyGrid['State'][12][20] = True
         MyGrid['State'][11][20] = True
         MyGrid['State'][11][21] = True
         MyGrid['State'][10][19] = True
         MyGrid['State'][10][21] = True

         MyGrid['State'][22][40] = True
         MyGrid['State'][21][40] = True
         MyGrid['State'][21][39] = True
         MyGrid['State'][20][41] = True
         MyGrid['State'][20][39] = True


         MyGrid.FinishUpdate()
         MyAxes.matshow(MyGrid['Value'])
         MyFigure
```

```
Out[24]:
```




```
In [25]: AnimationFrames = 300  #total number of frames in animation.
         DelayBetweenFrames = 100 #in msec (20 gives 50 fps) (33.3 gives 30 fps)
         AnimationTime = AnimationFrames * DelayBetweenFrames/1000
         print ("The animation will be {:.1f} seconds long.".format(AnimationTime))
```

```
The animation will be 30.0 seconds long.
```

```
In [26]: MyAnimation = animation.FuncAnimation(MyFigure, af,
                                             frames=AnimationFrames, interval=DelayBetwe
         MyAnimation.save('GOL_Glider_M_test.mp4',extra_args=['-vcodec', 'h264'])
```

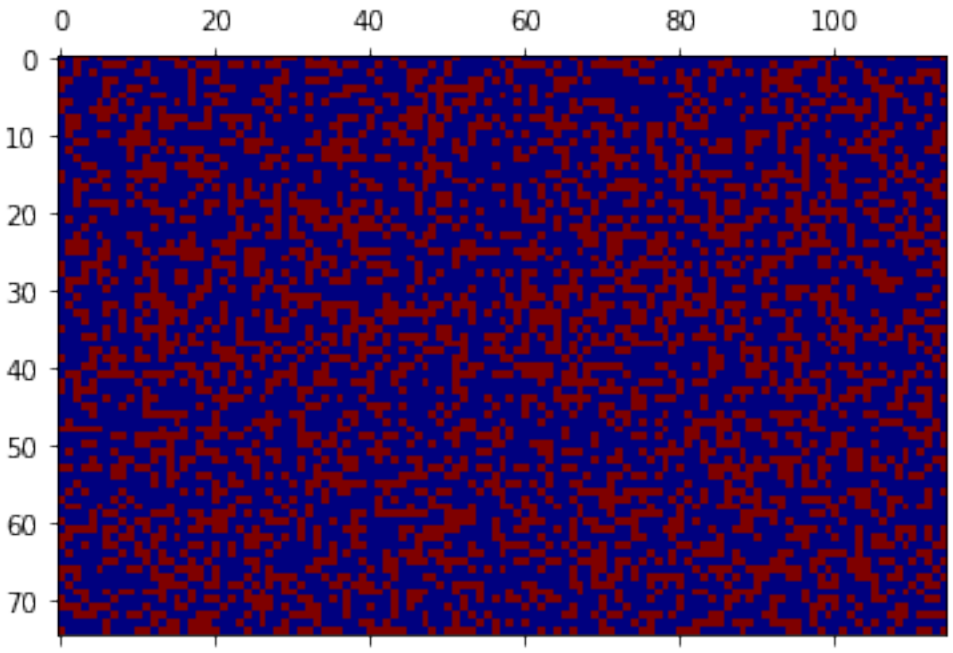
The multiple glider example is below.

### 4.2 Full Game of Life Simulation

Just for fun, let us run a simulation on a large grid.

```
In [27]: rows = 75
         columns = 115
         MyGrid=CAGrid((rows,columns),MyDtype)  #initilized with whatever random me
         setrandomstate(MyGrid,38)
         MyGrid.FinishUpdate()
         MyFigure, MyAxes = plt.subplots()
         MyAxes.matshow(MyGrid['State'])
```

```
Out[27]: <matplotlib.image.AxesImage at 0x8e79b00>
```




In [28]:
```
AnimationFrames = 1000  #total number of frames in animation.
DelayBetweenFrames = 33 #in msec (20 gives 50 fps) (33.3 gives 30 fps)
AnimationTime = AnimationFrames * DelayBetweenFrames/1000
print ("The animation will be {:.1f} seconds long.".format(AnimationTime))
```

```
The animation will be 33.0 seconds long.
```

In [29]:
```
MyAnimation = animation.FuncAnimation(MyFigure, af,
                                      frames=AnimationFrames, interval=DelayBetwe
MyAnimation.save('GOL_Large_Sim.mp4',extra_args=['-vcodec', 'h264'])
```

In [ ]: