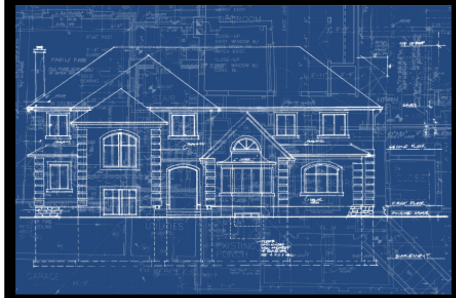
Classes & Objects

A class combines (and abstracts) data and functions

An object is an instantiation of a class

Classes & Objects



class

object

Classes & Objects

A class combines (and abstracts) data and functions

An object is an instantiation of a class

List is a built-in class, *append* is a method

Int is a built-in class, + is a operator

We can define our own classes

Classes & Objects

```
b = Ball(10.0, 15.0, 0.0, -5.0)

constructor:
```

Classes & Objects

```
b = Ball(10.0, 15.0, 0.0, -5.0)

constructor:
 • allocate memory for a Ball object
```

Classes & Objects

```
b = Ball(10.0, 15.0, 0.0, -5.0)

constructor:
 • allocate memory for a Ball object
 • initializes the Ball object with values
```

Classes & Objects

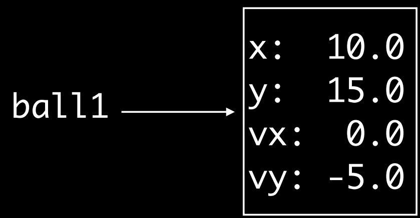
```
b = Ball(10.0, 15.0, 0.0, -5.0)

constructor:
 • allocate memory for a Ball object
 • initializes the Ball object with values
 • returns address of the Ball object
```

Classes & Objects

```
b = Ball(10.0, 15.0, 0.0, -5.0)

constructor:
 • allocate memory for a Ball object
 • initializes the Ball object with values
 • returns Ball instance
 • similar to a list
```

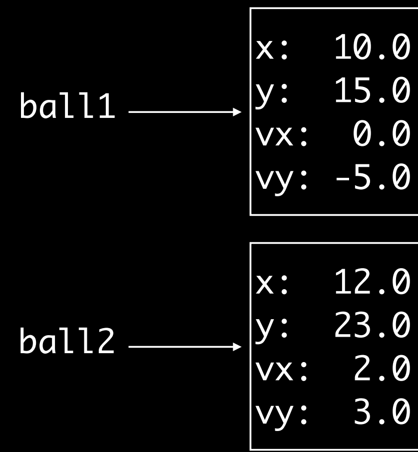
Classes & Objects

```
ball1 = Ball(10.0, 15.0, 0.0, -5.0)
```



Classes & Objects

```
ball1 = Ball(10.0, 15.0, 0.0, -5.0)
ball2 = Ball(12.0, 23.0, 2.0, 3.0)
```



Classes & Objects

```
ball1 = Ball(10.0, 0.0, 1.0, 1.0) # x,y,vx,vy
ball2 = Ball(-10.0, 0.0, 1.0, 1.0)


print( "the x-coordinate is ", ball1.x)
```

Classes & Objects

```
ball1 = Ball(10.0, 0.0, 1.0, 1.0) # x,y,vx,vy
ball2 = Ball(-10.0, 0.0, 1.0, 1.0)

print( ball1.x)
10.0
print( ball2.x)
-10.0

ball1.update_position() # x = x + vx

print( ball1.x)
11.0
print( ball2.x)
-10.0
```

```
D = draw.Drawing(200, 200, origin='center') # define drawing canvas
EARTH_GRAVITY_ACCELERATION = -9.8   # acceleration due to gravity, m/sec^2
BALL_RADIUS = 10  # radius of the ball in pixels

class Ball:
    def __init__(self, start_x, start_y, start_v_x, start_v_y, color='blue'):
        # Ball location, velocity, and color
        self.x = start_x
        self.y = start_y
        self.v_x = start_v_x
        self.v_y = start_v_y
        self.color = color

    def update_position(self, timestep=1):
        self.x = self.x + timestep * self.v_x
        self.y = self.y + timestep * self.v_y

    def update_velocity(self, timestep=1):
        self.v_y = self.v_y + timestep * EARTH_GRAVITY_ACCELERATION

    def animate_step(self, timestep=1):
        self.update_position(timestep)
        self.update_velocity(timestep)

    def draw(self):
        D.append(draw.Circle(self.x, self.y, BALL_RADIUS, fill=self.color))
```


```
D = draw.Drawing(200, 200, origin='center') # define drawing canvas
EARTH_GRAVITY_ACCELERATION = -9.8   # acceleration due to gravity, m/sec^2
BALL_RADIUS = 10  # radius of the ball in pixels

class Ball:
    def __init__(self, start_x, start_y, start_v_x, start_v_y, color='blue'):
        # Ball location, velocity, and color
        self.x = start_x
        self.y = start_y
        self.v_x = start_v_x
        self.v_y = start_v_y
        self.color = color

    def update_position(self, timestep=1):
        self.x = self.x + timestep * self.v_x
        self.y = self.y + timestep * self.v_y

    def update_velocity(self, timestep=1):
        self.v_y = self.v_y + timestep * EARTH_GRAVITY_ACCELERATION

    def animate_step(self, timestep=1):
        self.update_position(timestep)
        self.update_velocity(timestep)

    def draw(self):
        D.append(draw.Circle(self.x, self.y, BALL_RADIUS, fill=self.color))
```

```
D = draw.Drawing(200, 200, origin='center') # define drawing canvas
EARTH_GRAVITY_ACCELERATION = -9.8   # acceleration due to gravity, m/sec^2
BALL_RADIUS = 10  # radius of the ball in pixels

class Ball:
    def __init__(self, start_x, start_y, start_v_x, start_v_y, color='blue'):
        # Ball location, velocity, and color
        self.x = start_x
        self.y = start_y
        self.v_x = start_v_x
        self.v_y = start_v_y
        self.color = color

    def update_position(self, timestep=1):
        self.x = self.x + timestep * self.v_x
        self.y = self.y + timestep * self.v_y

    def update_velocity(self, timestep=1):
        self.v_y = self.v_y + timestep * EARTH_GRAVITY_ACCELERATION

    def animate_step(self, timestep=1):
        self.update_position(timestep)
        self.update_velocity(timestep)

    def draw(self):
        D.append(draw.Circle(self.x, self.y, BALL_RADIUS, fill=self.color))
```

```
D = draw.Drawing(200, 200, origin='center') # define drawing canvas
EARTH_GRAVITY_ACCELERATION = -9.8   # acceleration due to gravity, m/sec^2
BALL_RADIUS = 10  # radius of the ball in pixels

class Ball:
    def __init__(self, start_x, start_y, start_v_x, start_v_y, color='blue'):
        # Ball location, velocity, and color
        self.x = start_x
        self.y = start_y
        self.v_x = start_v_x
        self.v_y = start_v_y
        self.color = color

    def update_position(self, timestep=1):
        self.x = self.x + timestep * self.v_x
        self.y = self.y + timestep * self.v_y

    def update_velocity(self, timestep=1):
        self.v_y = self.v_y + timestep * EARTH_GRAVITY_ACCELERATION

    def animate_step(self, timestep=1):
        self.update_position(timestep)
        self.update_velocity(timestep)

    def draw(self):
        D.append(draw.Circle(self.x, self.y, BALL_RADIUS, fill=self.color))
```

```
ball1 = Ball(10.0, 15.0, 0.0, -5.0)
```

```
x:      10.0
y:      15.0
v_x:     0.0
v_y:    -5.0
color: blue
```

```
def __init__(self, start_x, start_y, start_v_x, start_v_y, color='blue'):
    # Ball location, velocity, and color
    self.x = start_x
    self.y = start_y
    self.v_x = start_v_x
    self.v_y = start_v_y
    self.color = color
```

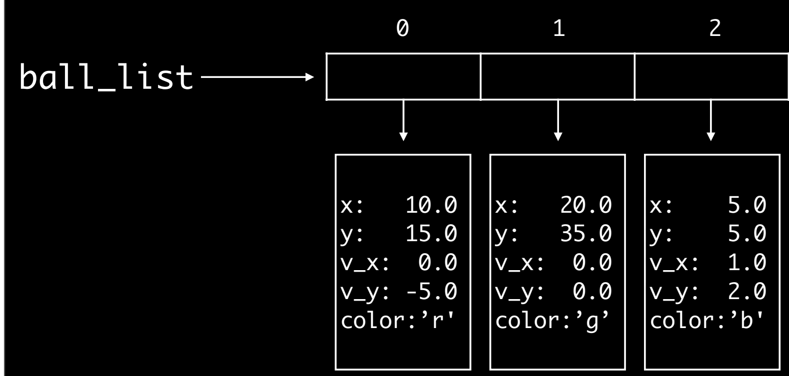
```
ball1 = Ball(10.0, 15.0, 0.0, -5.0)
ball1.update(0.1)
```

```
x:      10.0
y:      15.0
v_x:     0.0
v_y:    -5.0
color: blue
```

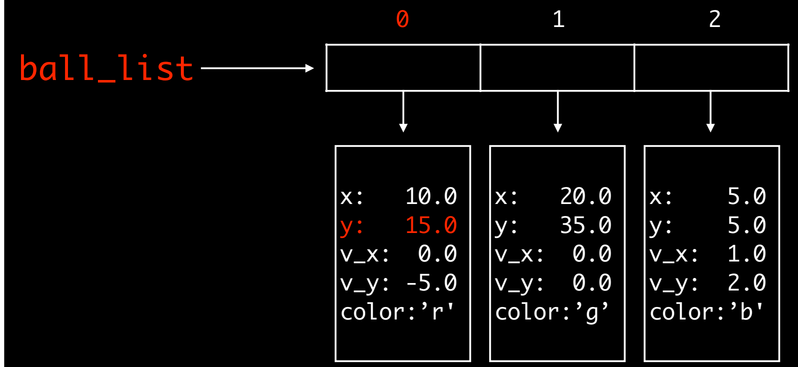
```
def update_position(self, timestep):
    self.x = self.x + timestep * self.v_x # ball1.x = ball1.x + ...
    self.y = self.y + timestep * self.v_y
```

[bouncingballs.ipynb]

Lists of Objects



Lists of Objects




```
ball_list[0].y
```

[bouncingball.ipynb]

```
b = Ball(0,0,1,-1)
print(b)

<__main__.Ball object at 0x113dea0d0>
```

```
def __str__(self):
    return str(self.x) + ", " + str(self.y)


b = Ball(0,0,1,-1)
print(b)
1, 2
```

```python
# BankAccount
class BankAccount:
    def __init__(self, initial):
        self.balance = initial

    def deposit(self, amount):
        self.balance = self.balance + amount

    def withdraw(self, amount):
        self.balance = self.balance - amount

    def overdrawn(self):
        return self.balance < 0

    def __str__(self):
        return "balance: " + str(self.balance)

# test BankAccount
my_account = BankAccount(150)
my_account.deposit(200)
print( my_account )
```

[bankaccount.ipynb]

```python
# instance vs. class attributes

class BankAccount:
    interest = 0.02 # class attribute
    def __init__(self, initial):
        self.balance = initial

# test BankAccount
john = BankAccount(150)
jane = BankAccount(250)

john.interest
0.02
jane.interest
0.02
```


```python
# instance vs. class attributes

class BankAccount:
    interest = 0.02 # class attribute
    def __init__(self, initial):
        self.balance = initial

# test BankAccount
john = BankAccount(150)
jane = BankAccount(250)

john.interest
0.02
jane.interest
0.02
```


```python
BankAccount.interest = 0.01 # class attribute
```

```
# instance vs. class attributes

class BankAccount:
    interest = 0.02 # class attribute
    def __init__(self, initial):
        self.balance = initial

# test BankAccount
john = BankAccount(150)
jane = BankAccount(250)

john.interest
0.02
jane.interest
0.02

BankAccount.interest = 0.01 # class attribute
john.interest
0.01
jane.interest
0.01
```

```
# instance vs. class attributes

class BankAccount:
    interest = 0.02 # class attribute
    def __init__(self, initial):
        self.balance = initial

# test BankAccount
john = BankAccount(150)
jane = BankAccount(250)
jane.interest = 0.04 # instance attribute
```

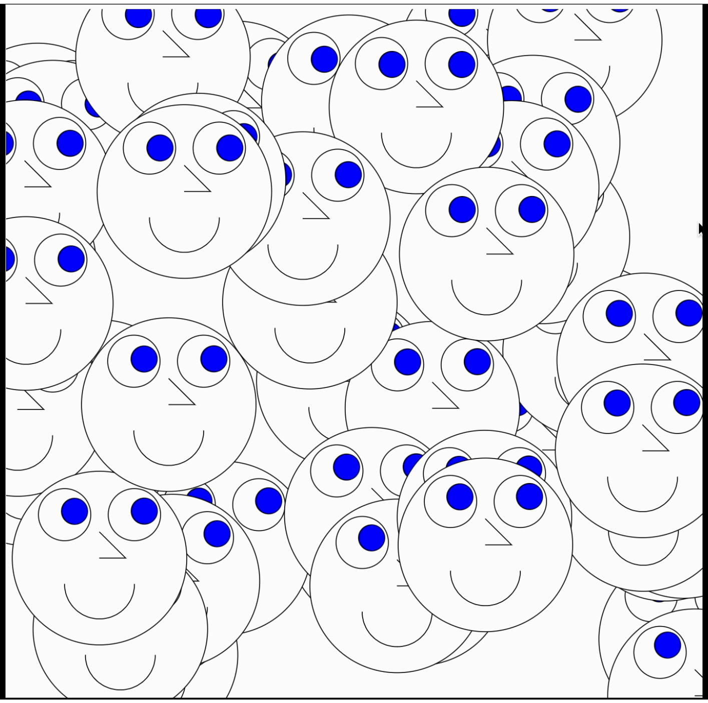
```
# instance vs. class attributes

class BankAccount:
    interest = 0.02 # class attribute
    def __init__(self, initial):
        self.balance = initial

# test BankAccount
john = BankAccount(150)
jane = BankAccount(250)
jane.interest = 0.04 # instance attribute

john.interest
0.02
jane.interest
0.04
```

```
# instance vs. class attributes

class BankAccount:
    interest = 0.02 # class attribute
    def __init__(self, initial):
        self.balance = initial

# test BankAccount
john = BankAccount(150)
jane = BankAccount(250)
jane.interest = 0.04 # instance attribute

john.interest
0.02
jane.interest
0.04

BankAccount.interest = 0.01 # class attribute
john.interest
0.01
jane.interest
0.04
```

```
# lists are objects (with different syntax)

s = []
s.append(1)
[1]

f = s.append
f(2)
s
[1, 2]
```

```
class Kangaroo:
    def __init__(self):
        self.pouch_contents = []

    def put_in_pouch(self,x):
        for item in self.pouch_contents:
            if item == x:
                print(x + " is already in pouch")
                return
         self.pouch_contents.append(x)

    def __str__(self):
        if( len(self.pouch_contents) == 0 ):
            return "The kangaroo's pouch is empty"
        else:
            return "The kangaroo's pouch contains: " + str(self.pouch_contents)
```

[kangaroo.ipynb]



[crowd.ipynb]

```python
class Student:
    def __init__(self, name, exam_grade, height_in_cm):
        self.name = name
        self.grade = exam_grade
        self.height = height_in_cm
```

```python
class Student:
    def __init__(self, name, exam_grade, height_in_cm):
        self.name = name
        self.grade = exam_grade
        self.height = height_in_cm

    def __str__(self):
        return "(" + self.name + ", " + str(self.grade) \
               + ", " + str(self.height) + ")"
```

```python
# create a student
a = Student("Alice",92,160)
print(a)

(Alice, 92, 160)
```

```
# create a student
a = Student("Alice",92,160)

# access a student's information (don't do this)
print(a.height)
print(a.grade)
```

```
class Student:
    def __init__(self, name, exam_grade, height_in_cm):
        self.name = name
        self.grade = exam_grade
        self.height = height_in_cm

    def __str__(self):
        return "(" + self.name + ", " + str(self.grade) \
                + ", " + str(self.height) + ")"

    def getName(self):
        return self.name

    def getGrade(self):
        return self.grade

    def getHeight(self):
        return self.height
```

```
# create a student
a = Student("Alice",92,160)

# access a student's information
print( a.height )
print( a.grade )

# access a student's information
print( a.getHeight() )
print( a.getGrade() )
```

```
student_list = [Student("Alice", 92, 160), \
                Student("Bob", 42, 165), \
                Student("Chelsea", 76, 162)]
```

```
student_list = [Student("Alice", 92, 160), \
                Student("Bob", 42, 165), \
                Student("Chelsea", 76, 162)]


# print all students
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all students
for s in student_list:
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all students
for s in student_list:
    print(s) # calls __str__ of Student class
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all students
for s in student_list:
    print(s) # calls __str__ of Student class

# print all students that are failing
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all students
for s in student_list:
    print(s) # calls __str__ of Student class

# print all students that are failing
for s in student_list:
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all students
for s in student_list:
    print(s) # calls __str__ of Student class

# print all students that are failing
for s in student_list:
    if( s.getGrade() < 65 ):
        print(s)
```

```
class Student:
    ...

    def isFailing(self):


    ...
```

```
class Student:
    ...

    def isFailing(self):
        return self.grade < 65

    ...
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all students
for s in student_list:
    print(s) # calls __str__ of Student class

# print all students that are failing
for s in student_list):
    if( s.getGrade() < 65 ):
        print(s)

# print all students that are failing (better)
for s in student_list):
    if( s.isFailing() ):
        print(s)
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all exam scores in sorted order
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all exam scores in sorted order
student_list.sort()
```

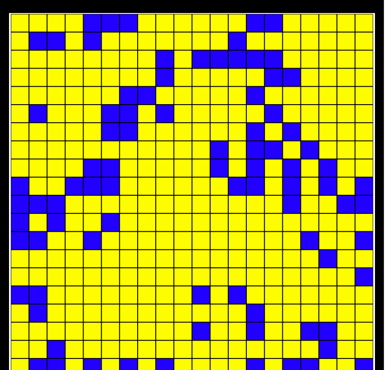
```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all exam scores in sorted order
student_list.sort()
```

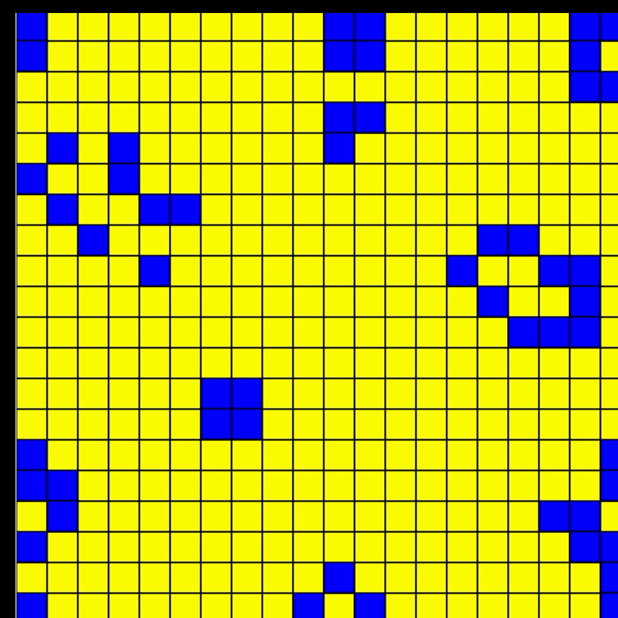
```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all exam scores in sorted order
student_list.sort(key=lambda s: s.grade)
```

```
student_list = [Student("Alice", 92, 160),
                Student("Bob", 42, 165),
                Student("Chelsea", 76, 162)]


# print all exam scores in sorted order
student_list.sort(key=lambda s: s.grade)

for s in student_list):
    print(s)

(Bob, 42, 165)
(Chelsea, 76, 162)
(Alice, 92, 160)
```

The Game of Life

- The game simulates a bunch of (biological) *cells* that live in a *colony*.
- The colony is a two-dimensional grid; each cell is a square in the grid.
- Each cell is either alive or dead.
- Living cells are blue, and dead cells are yellow.





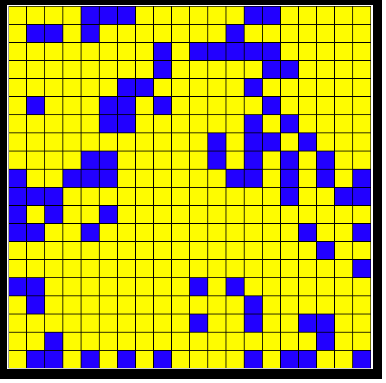
The Game of Life

Time moves in steps, called *generations*. In each new generation, cells might be born, others survive, and some might die.

The Game of Life

Time moves in steps, called *generations*. In each new generation, cells might be born, others survive, and some might die.

A cell has eight neighbors. The number of living neighbors that a cell has in one generation determines its fate in the next generation:



The Game of Life

Time moves in steps, called *generations*. In each new generation, cells might be born, others survive, and some might die.

A cell has eight neighbors. The number of living neighbors that a cell has in one generation determines its fate in the next generation:

- If the cell is alive and has 0 or 1 living neighbors, it dies of exposure and is dead in the next generation.

The Game of Life

Time moves in steps, called *generations*. In each new generation, cells might be born, others survive, and some might die.

A cell has eight neighbors. The number of living neighbors that a cell has in one generation determines its fate in the next generation:

- If the cell is alive and has 0 or 1 living neighbors, it dies of exposure and is dead in the next generation.
- If the cell is alive and has 4 or more living neighbors, it dies of overcrowding and is dead in the next generation.

The Game of Life

Time moves in steps, called *generations*. In each new generation, cells might be born, others survive, and some might die.

A cell has eight neighbors. The number of living neighbors that a cell has in one generation determines its fate in the next generation:

- If the cell is alive and has 0 or 1 living neighbors, it dies of exposure and is dead in the next generation.
- If the cell is alive and has 4 or more living neighbors, it dies of overcrowding and is dead in the next generation.
- If the cell is dead and has exactly 3 living neighbors, it is born and is alive in the next generation.

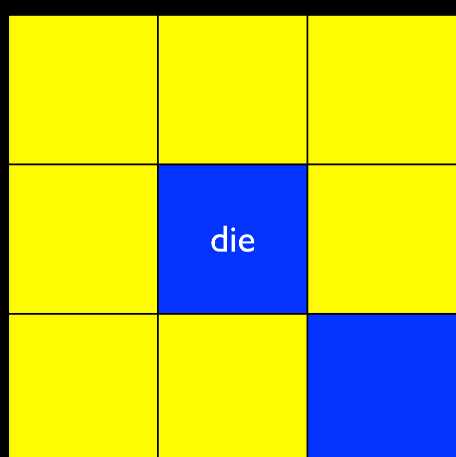
The Game of Life

Time moves in steps, called *generations*. In each new generation, cells might be born, others survive, and some might die.

A cell has eight neighbors. The number of living neighbors that a cell has in one generation determines its fate in the next generation:

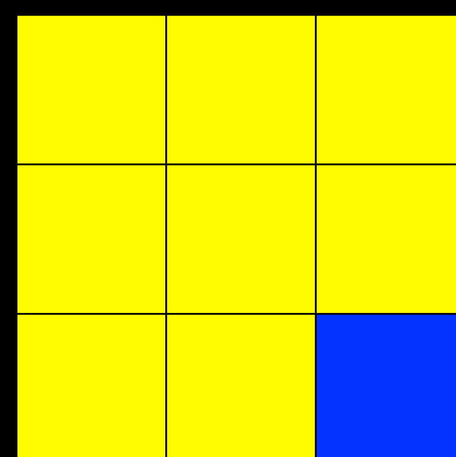
- If the cell is alive and has 0 or 1 living neighbors, it dies of exposure and is dead in the next generation.
- If the cell is alive and has 4 or more living neighbors, it dies of overcrowding and is dead in the next generation.
- If the cell is dead and has exactly 3 living neighbors, it is born and is alive in the next generation.
- Otherwise, the cell stays the same in the next generation as it is in the current generation:

The Game of Life

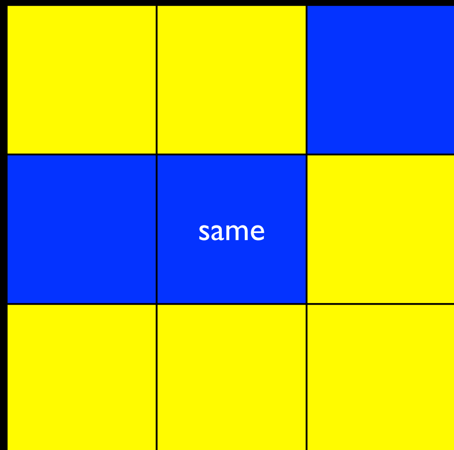


if(alive & living_neighbor == 0 or 1) then die

The Game of Life

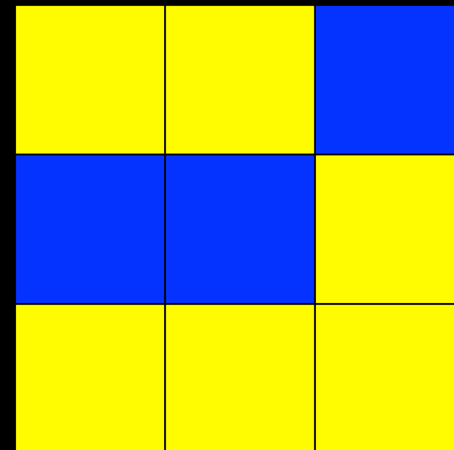


The Game of Life

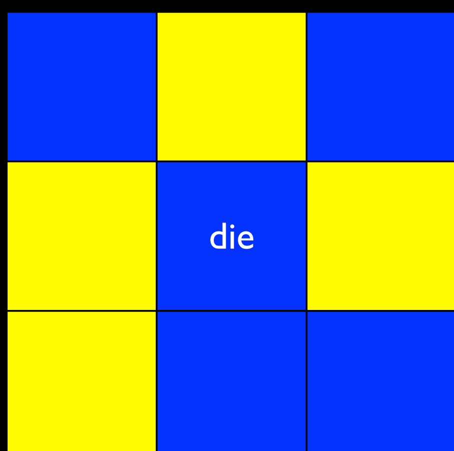


if(alive & living_neighbor == 2 or 3) then do_nothing

The Game of Life

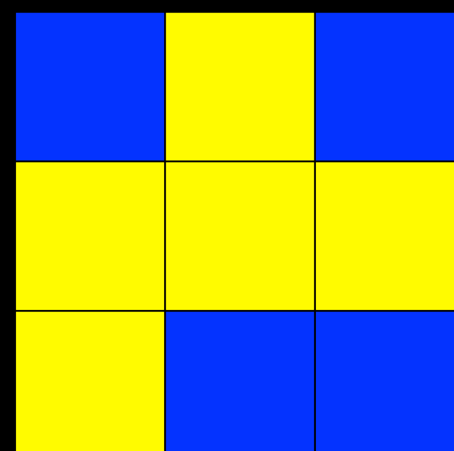


The Game of Life

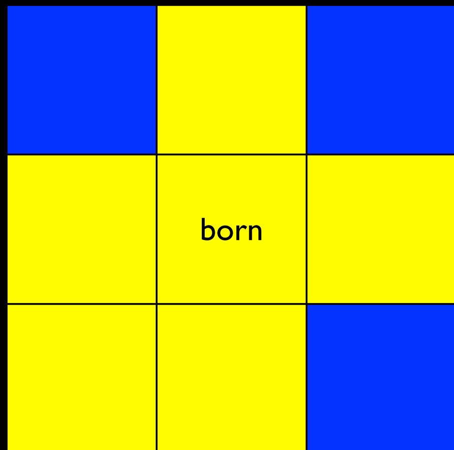


if(alive & living_neighbor >= 4) then die

The Game of Life

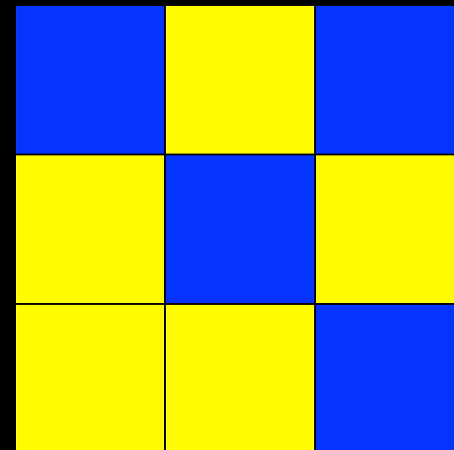


The Game of Life

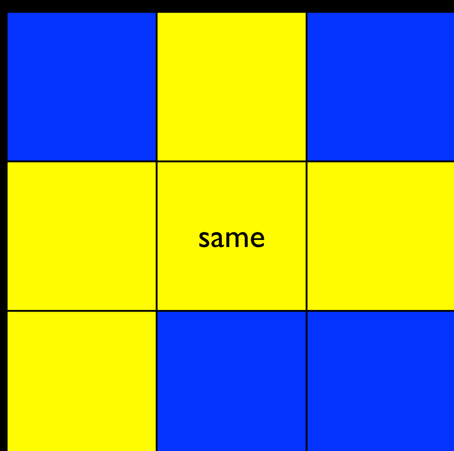


if(dead & living_neighbor == 3) then born

The Game of Life

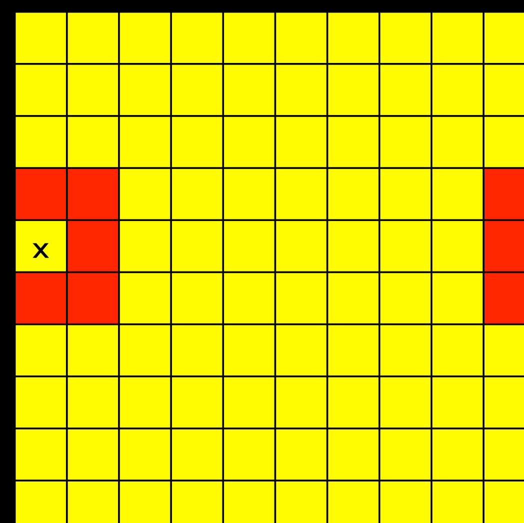


The Game of Life

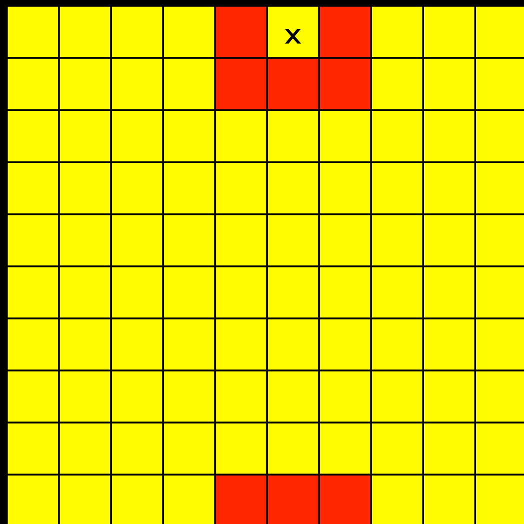


if(dead & living_neighbor != 3) then do_nothing

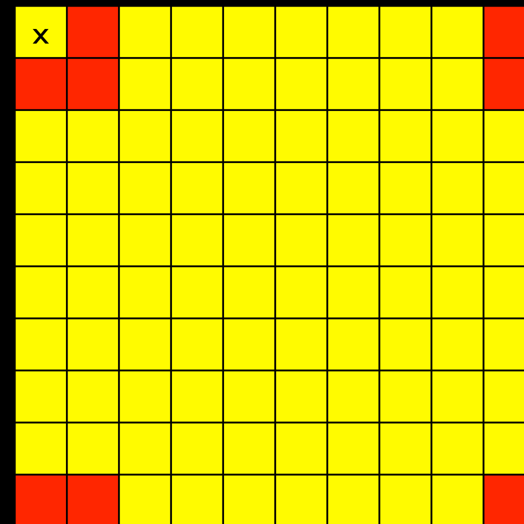
The Game of Life



The Game of Life



The Game of Life



[life.ipynb]