# OBJECTS (MAKING THEM!)

# REMINDER: QUESTIONS DURING CLASS

You can ask us about anything at all durinng class when we are walking around. Even if it is not about the current activity, feel free to ask. Worst case we say "Ask us later/after class"

- Can ask about things covered 20 minutes earlier in the lecture
- Can ask questions about previous topics from previous lectures
- Homework questions are usually longer to answer, will probably tell you to talk about it after class

# CLASSES

So far we have talked about dataclasses, which make a simple (and convenient) way for us to define classes.
Dataclass mostly just creates the `__init__(self)` function (constructor).

```
# When we create an instance of a class like this, call __init__
my_ball = Ball(0.5, 0.25, 0.1, (10, 25, 216))
```

What does the code looks like that lets us write `Ball(...)`?

```python
@dataclass
class Ball:
  radius: int
  center_x: float
  center_y: float
  color: tuple[int, int, int]
```

```python
class Ball:
  def __init__(self, radius, center_x, center_y, color):
    self.radius = radius
    self.center_x = center_x
    self.center_y = center_y
    self.color = color
```

Two ways of doing the same thing: creating an object that stores attributes with these names.

What's the point of writing an __init__ method if it's just the same as writing a dataclass but with all this stupid pointless extra typing??

# WRITING OUR OWN __INIT__

What if we slightly modify our class `Ball` to randomly generate the colors?

```python
class Ball:
    def __init__(self, radius, center_x, center_y): # no more color
        self.radius = radius
        self.center_x = center_x
        self.center_y = center_y

        red = random.randint(0, 255)
        green = random.randint(0, 255)
        blue = random.randint(0, 255)
        self.color = (red, green, blue)
```

# WHAT ELSE...

Dataclasses assume that you are only writing an object for the purpose of storing data.

- e.g. a `Message` from the message board example

While you can technically extend a dataclass to add methods, we use standard class definitions in cases that we want to add several methods.

Here we have the `Square` class defined with the method `draw`

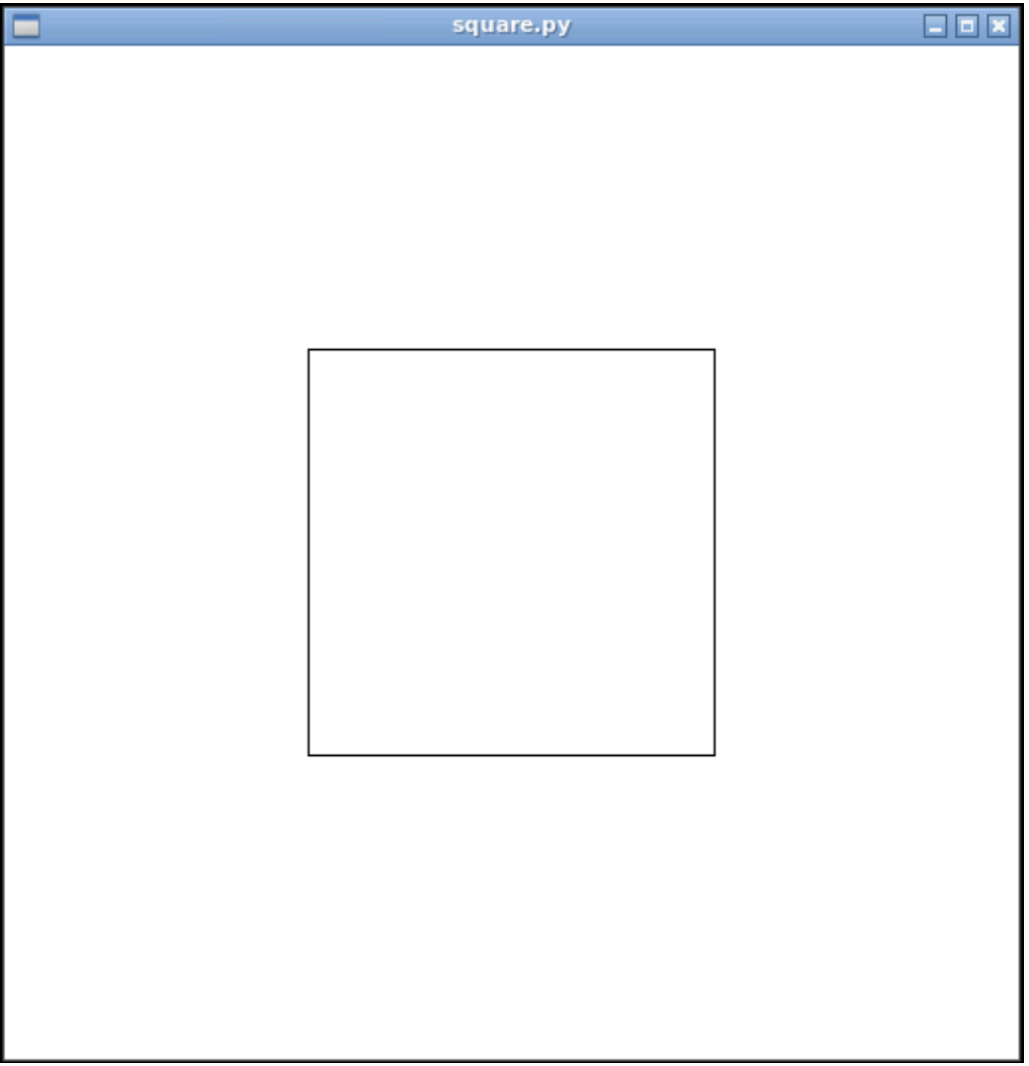
```
class Square:
    def __init__(self, half_length, center_x, center_y):
        self.half_length = half_length
        self.center_x = center_x
        self.center_y = center_y
        red = random.randint(0, 255)
        green = random.randint(0, 255)
        blue = random.randint(0, 255)
        self.color = (red, green, blue)

    def draw(self):
        penndraw.set_pen_color(self.color) # we use self for attrs.
        penndraw.filled_square(self.center_x, self.center_y,
                               self.half_length)
```

# PREDICT WHAT HAPPENS!

`change_color()` method that we'd call to replace the color of a `Square` object with a new one, e.g. `my_square.change_color((155, 78, 300))`.

```
class Square:
  def __init__(self, half_length, center_x, center_y, color):
    self.half_length = half_length
    self.center_x = center_x
    self.center_y = center_y
    self.color = color
  def change_color(new_color):
    color = new_color
```

Can you spot the bug & predict what would happen if we tried to call `change_color`?

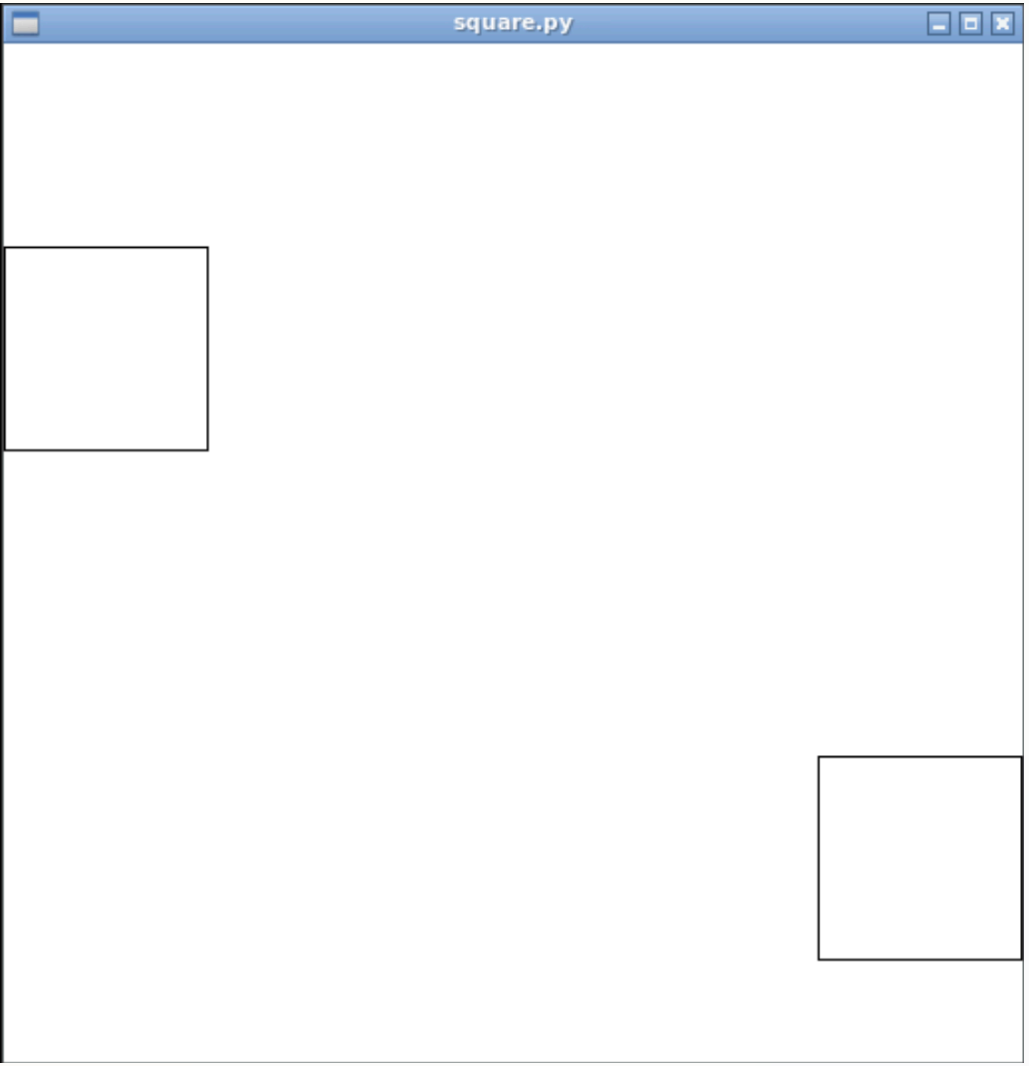



# REVIEW: USING METHODS

Methods are functions that belong to an object, so they are called (mostly) like any function

- Call by name and pass in arguments within parentheses
- Make sure to call the method *on the object* that you want to perform that behavior!

```
my_square = Square(0.5, 0.5, 0.2)
my_square.draw()
```




You might have several instances of a class in your program.

- A method called on an object should *modify/use just that object.*
- Other objects will be unchanged by another object's method call.

```
left_square = Square(0.1, 0.2, 0.1)
right_square = Square(0.9, 0.2, 0.1)
left_square.move_by(0, 0.5)
left_square.draw()
right_square.draw()
```

# PRACTICE: CONTAINS POINT

Write the method `contains_point` for the `Square` class that checks to see if a point is contained within the square.
Returns `True` if the point is in the square, or `False` if it is not in the square.

**C12**

```
def contains_point(self, other_x: float, other_y: float) --> bool:
    # TODO: probably use half_length, center_x and center_y
```

*Can you do this without writing an `if`?*

# BOUNCING BALL SIMULATION

Ingredients:

- `ball.py`, a class that defines how a 2D ball moves & bounces on a screen
- `bouncing_balls.py`:
    - contains a main method so that the simulation is runnable
    - creates a `list[Ball]` in which to store the objects to be simulated
    - defines a "physics" (animation) loop

# A BIT OF PHYSICS

To simulate an object's motion in 2D space over time, we need to keep track of its:

- position $(p_x, p_y)$
  - where the object is **right now**
- velocity/speed $(v_x, v_y)$
  - how much the object **should move** from where it is right now to where it will be next time we look
- acceleration $(a_x, a_y)$
  - how much the object's velocity should change from what it is right now to what it will be next time we look (always a constant)

# A BIT OF PHYSICS

Since our simulation is run using a loop, we do our calculations in *discrete steps*.

- We denote the step number using superscripts, so $p_x^t$ means "x position at step $t$"
- We'll assume a constant unit timestep, meaning that we don't have to account for the length of the timestep in our equations
  - *(ignore this point if the details of physical simulations are not interesting to you)*

# A BIT OF PHYSICS

| EQUATION | MEANING | CODE |
| --- | --- | --- |
| $p_x^{t+1} = p_x^t + v_x^t$ | x position in the next iteration is equal to the x position now plus the x speed now | `px = px + vx` |
| $p_y^{t+1} = p_y^t + v_x^t$ | y position in the next iteration is equal to the y position now plus the y speed now | `py = py + vy` |
| $v_x^{t+1} = v_x^t + a_x^t$ | x speed in the next iteration is equal to the x speed now plus the x acceleration | `vx = vx + ax` |
| $v_y^{t+1} = v_y^t + a_y^t$ | y speed in the next iteration is equal to the y speed now plus the y acceleration | `vy = vy + ay` |

Assume at time $t$ we have a Ball $(0.5, 0.5)$ with $v_x^t = 0.1, v_y^t = 0.2$ and $a_x^t = 0$, $a_y^t = -0.05$.

**S7** What's the position after one time step (at time $t+1$)?
**S8** What's the x velocity after one time step, $v_x^{t+1}$?
**S9** What's the y velocity after one time step, $v_y^{t+1}$?

Recall:

- $p_x^{t+1} = p_x^t + v_x^t$
- $p_y^{t+1} = p_y^t + v_x^t$
- $v_x^{t+1} = v_x^t + a_x^t$
- $v_y^{t+1} = v_y^t + a_y^t$

# IMPLEMENTING BALL.PY

What behaviors does a `Ball` object need to exhibit as part of a simple physics simulation?

- Needs to be drawable so that we can see the simulation
- Needs to move & bounce pursuant to the previous equations

Methods:
`def draw(self)`, `def update(self)`

# IMPLEMENTING BALL.PY

What properties does a `Ball` object need to store in order to perform these operations?

- position, x and y
- velocity, x and y
- acceleration, x and y
  - we'll ignore x acceleration, and y acceleration is just gravity
- radius
  - used for drawing
  - used for deciding when to bounce

# THE SIMULATOR

The simulator will be responsible for initializing and keeping track of all of the balls in the simulation.

- How will we store all of the objects being simulated?
  - Create an `list[Ball]`
- How will we draw each of the objects being simulated?
  - Iterate through the list and call the `draw()` method on each of the `Ball` objects.
- How will we get each of the objects to move and bounce?
  - Iterate through the list and call the `update()` method on each of the `Ball` objects

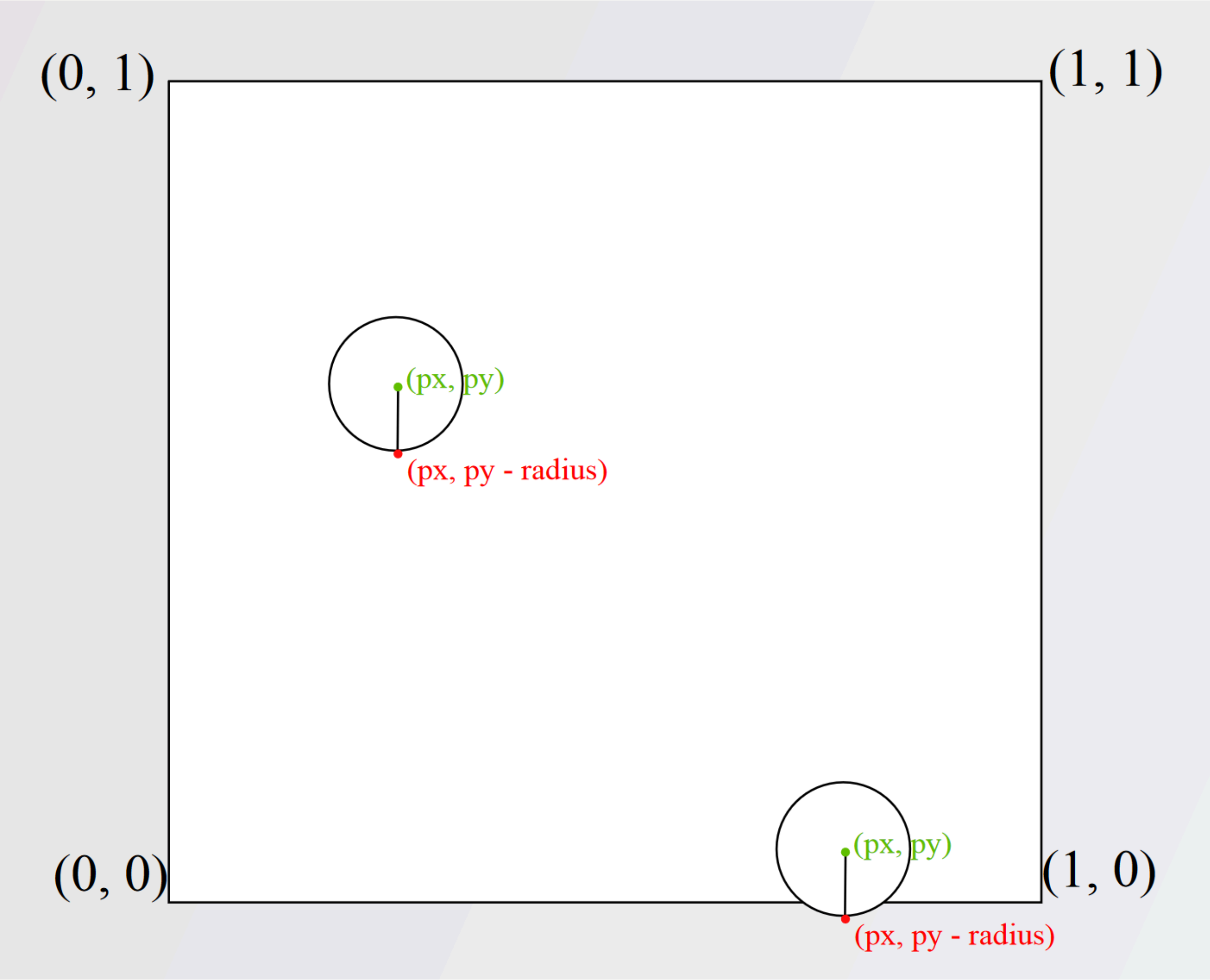
```python
import penndraw
import ball
def main():
    N = 10
    all_balls = []

    for _ in range(N):
        all_balls.append(ball.Ball())

    penndraw.set_canvas_size(600, 600)
    while True:
        penndraw.clear()
        for current_ball in all_balls:
            current_ball.draw()
            current_ball.update()
        penndraw.advance()
    penndraw.run()
```

```python
class Ball:
    def __init__(self):
        self.px = random.random()
        self.py = random.random()
        self.vx = -0.005 + (random.random() * 0.01) # [-0.005, 0.005]
        self.vy = -0.005 + (random.random() * 0.01)
        self.gravity = -0.0001
        self.radius = 0.02 + random.random() * 0.04 # [0.02, 0.06]

    def draw(self):
        penndraw.filled_circle(self.px, self.py, self.radius)

    def update(self): # No bouncing yet!
        self.px = self.px + self.vx
        self.py = self.py + self.vy
        self.vy = self.vy + self.gravity
```

# PROBLEM: NO BOUNCING!

Currently, the balls just drop off the sides or bottom of the screen. How can we get them to bounce?

- Check if the ball has gone past the left, right, or bottom of the screen
- Simulate a bounce by inverting the velocity for the next update step




# A BOUNCE

On the left, we have a sketch of the canvas with two balls.

- Which one should "bounce"?
- How can you formalize (write an equation for or just describe) what it means for a ball to bounce off of the bottom of the screen?

L13

# CHECKING A BOUNCE

A ball should bounce off the bottom of the screen when, at time step $t$:

- The ball is traveling downwards ($v_y^t < 0$)
- The bottom of the ball is at or below the bottom of the screen ($p_y^t - \text{radius} \leq 0$)

# MODELING A BOUNCE

What happens when an object bounces off of a surface?

- The object should change direction
- The object should lose a bit of momentum

The Bounce:

$$v_y = -0.9 * v_y$$

# A BETTER UPDATE()

```python
def update(self):
    self.px = self.px + self.vx
    self.py = self.py + self.vy
    self.vy = self.vy + self.gravity

    if (self.vy < 0 and self.py - self.radius <= 0):
        self.vy = -0.9 * self.vy
```

# COLLISION PRACTICE:

What if we want to make the balls bounce off of the walls?

A ball should bounce off the side of the screen when, at time step $t$:

- The ball is traveling in the direction of the wall
  - $v_x^t < 0$ for the left wall
  - $v_x^t > 0$ for the right wall
- The bottom of the ball is at or past the wall
  - $p_x^t - \text{radius} \leq 0$ for the left wall
  - $p_x^t - \text{radius} \geq 1$ for the right wall

What can we add to the update method to support bouncing off of the walls? **C16**

# A BEST UPDATE( )

```python
def update(self):
    self.px = self.px + self.vx
    self.py = self.py + self.vy
    self.vy = self.vy + self.gravity

    if (self.vy < 0 and self.py - self.radius <= 0):
        self.vy = -0.9 * self.vy

    if ((self.vx < 0 and self.px - self.radius <= 0) or
          (self.vx > 0 and self.px + self.radius >= 1)):
        self.vx = -0.9 * self.vx
```