Classes

Take 2

Stanford | ENGINEERING
Computer Science

Frankie Cerkvenik, CS106A, 2023

Housekeeping

- Assignment 5 is due on **tonight**
- Assignment 6 (last assignment!) will be released **tonight** and is due **next Tuesday**
- Final exam info released!
- Rest of the quarter:
  - Lecture all this week
  - Next week:
    - Final lecture on Tuesday (AMA),
    - Thursday: no lecture, extra OH during lecture time
    - Friday, **final exam**

Stanford | ENGINEERING
Computer Science

Today

- **Recap Classes**
  - **What are they**
  - **How do we make them**
  - **How do we use them**

- Code demo: many bouncing balls
  - Exciting stuff

What are classes?

- They are our way of creating our own custom types
  - Bundle several variables together and give them special functions
- Classes you’ve used:

```
from graphics import Canvas

my_canvas = Canvas(200, 400, ‘example’)

from simpleimage import SimpleImage

my_image = SimpleImage(filename)
```

Stanford | ENGINEERING
Computer Science

General Form for Writing a Class

- Filename for class is usually ***<u>classname</u>*****.py**
  - Filename is usually lowercase version of class name in file

```
class ClassName:
  def __init__(self, var_val):
    # constructor sets up instance variables
    self.instance_var = var_val

  def method1(self):
    # methods do something with an instance
    # they always take in 'self'
  def method2(self, x):
    # methods can take in other params too!
```

Example of a class in Python

serviceline.py:

```
class ServiceLine:
  def __init__(self, name, avg_wait_time):
    self.name = name
    self.people_waiting = []
    self.wait_time = 10

  def add_person(self, name):
    self.people_waiting.append(name)

  def serve_next_person(self):
     self.people_waiting.pop()

 def get_wait_time(self):
     return self.wait_time * len(self.people_waiting)
```

Stanford | ENGINEERING
Computer Science

Example of a class in Python

serviceline.py:

```
class ServiceLine:
  def __init__(self, name, avg_wait_time_mins):
    self.name = name
    self.people_waiting = []
    self.wait_time = avg_wait_time_mins
```

- **This is the constructor** - it is the code that runs when someone makes a new "ServiceLine" type variable
- It takes in a **name** and an **average wait time** as **parameters** - self is a special, invisible-ish parameter that means "the variable I'm currently making"

```
# making ServiceLine variable runs the constructor
dmv_line = ServiceLine("DMV", 90)
deli_line =  ServiceLine("Deli", 15)
```

Example of a class in Python

- **These are “methods” -** things ServiceLine variables can do

```
dmv_line = ServiceLine(“DMV”, 90)
dmv_line.get_wait_time() # returns 0
# adds ‘frankie’ to internal list
dmv_line.add_person(“Frankie”)

dmv_line.get_wait_time() # returns 90
```

```
cl

def add_person(self, name):
  self.people_waiting.append(name)

def serve_next_person():
   self.people_waiting.pop(self)

def get_wait_time(self):
   return self.wait_time * len(self.people_waiting)
```

Stanford | ENGINEERING
Computer Science

Using a class in Python

- When you pass an object as a parameter, mutations in that function persist after function ends

```
from service_line import ServiceLine



def main():
  dmv_line = ServiceLine("DMV", 90)
  dmv_line.add_person("Frankie")
  print(dmv_line.wait_time)
  print(dmv_line.people_waiting)
```

Using a class in Python

- When you pass an object as a parameter, mutations in that function persist after function ends

```
from service_line import ServiceLine



def main():
  dmv_line = ServiceLine("DMV", 90)
  dmv_line.add_person("Frankie")
  print(dmv_line.wait_time) # prints 90
  print(dmv_line.people_waiting) # [‘Frankie’]

  dmv_line.serve_next_person()
  print(dmv_line.people_waiting) # []
```

Objects are Mutable

- When you pass an object as a parameter, mutations in that function persist after function ends

```
from service_line import ServiceLine

def big_rush(line, lots_of_people):
   for person in people:
    line.add_person(person)

def main():
  dmv_line = ServiceLine("DMV", 90)
  big_rush(dmv_line, ["frankie", "ecy", "chris"])
  print(dmv_line.people_waiting)
  # [‘frankie’, ‘ecy’, ‘chris’]
```

Remember: moves never persist!

- This isn’t special for classes - but good to remember!

```
from service_line import ServiceLine

def big_rush(line, lots_of_people):
   line = ServiceLine(“New Line”, 90)
   for person in people:
     line.add_person(person)

def main():
  dmv_line = ServiceLine("DMV", 90)
  big_rush(dmv_line, ["frankie", "ecy", "chris"])
  print(dmv_line.people_waiting) # prints []!
```

Questions?

Instance Variables

- Each Counter has its own count - trace!

```python
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

$$\begin{tikzcd}\textbf{counter1} \arrow[r] & \boxed{\texttt{????}}\end{tikzcd}$$

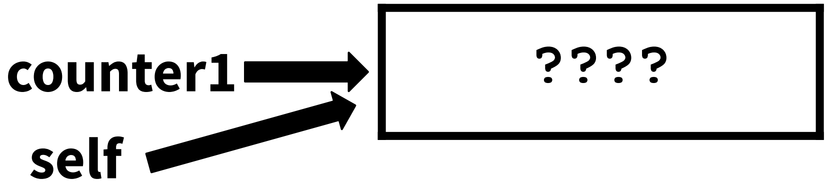
```python
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Stanford ENGINEERING
Computer Science

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
     self.count += 1
```



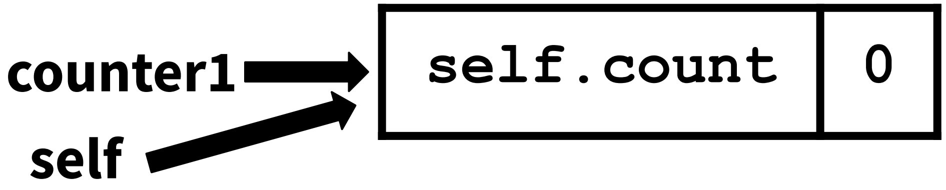
```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

```
from counter import Counter

def main():
  counter1 = Counter()
  counter2 = Counter()
  counter1.next()
  counter1.next()
  counter2.next()
```

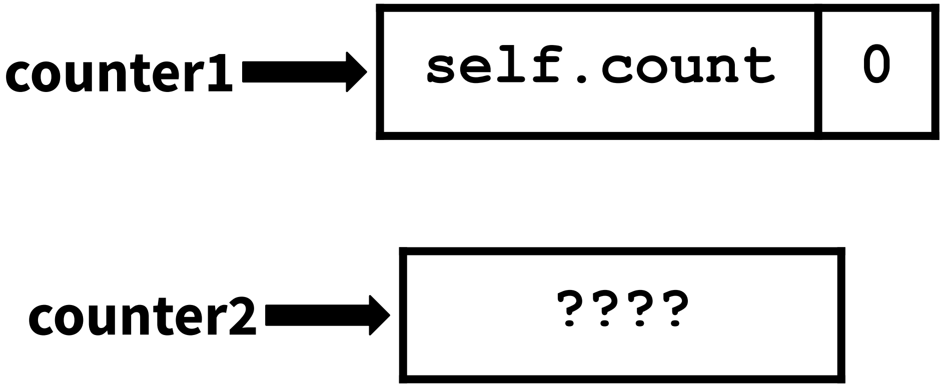


Stanford | ENGINEERING
Computer Science

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```



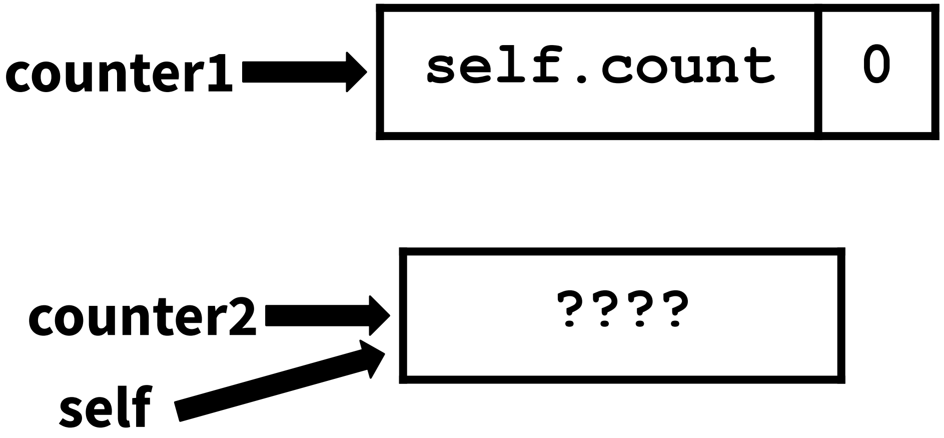
```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Stanford | ENGINEERING
Computer Science

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

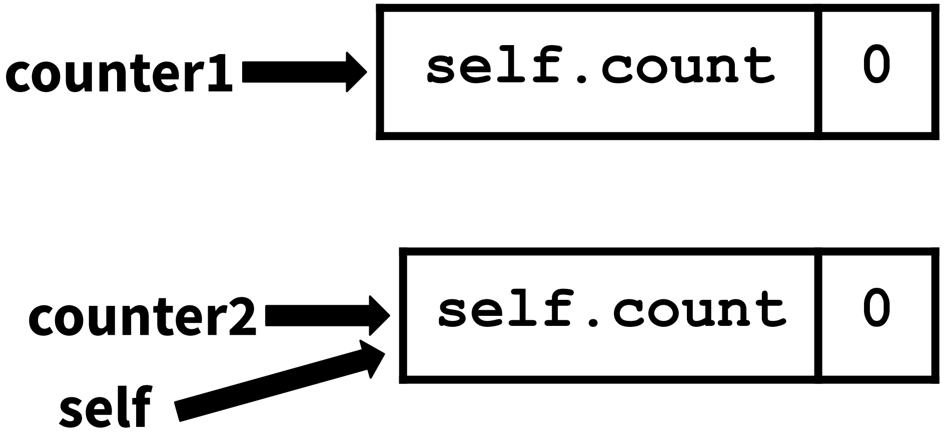


```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

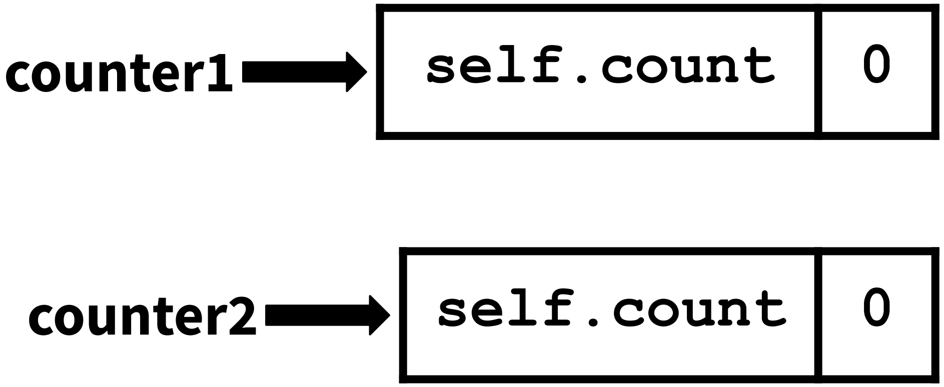


```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

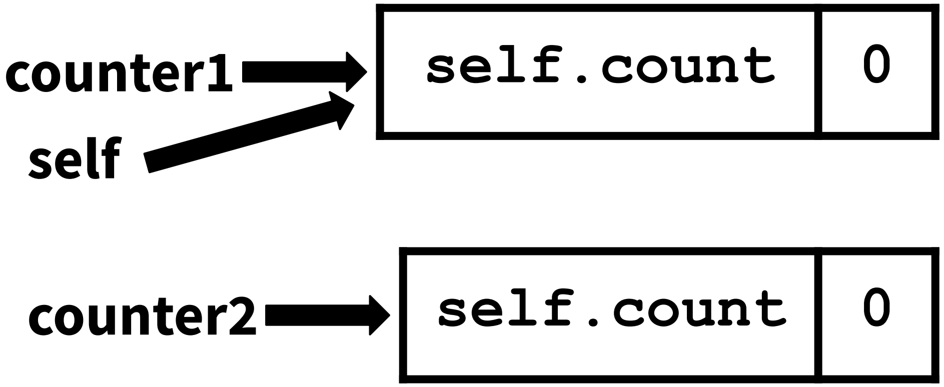


```
from counter import Counter

def main():
    counter1 = Counter()
    counter2 = Counter()
    counter1.next()
    counter1.next()
    counter2.next()
```

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

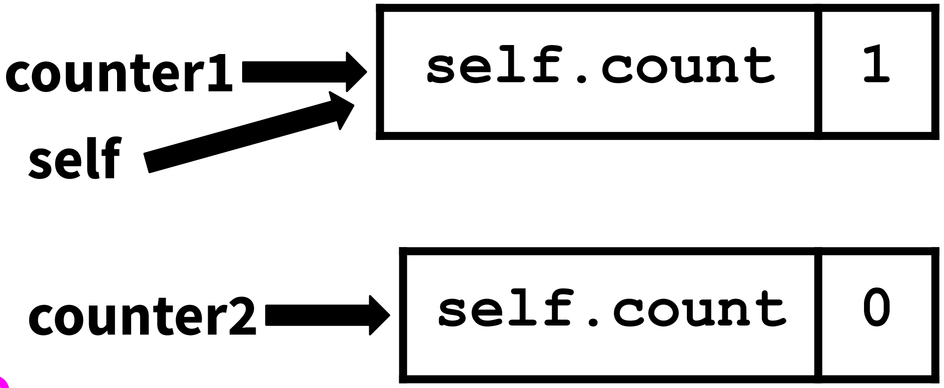


```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```



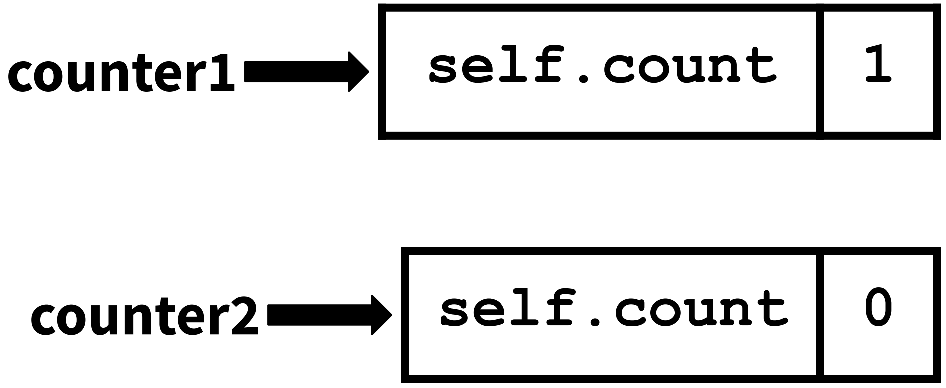
```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

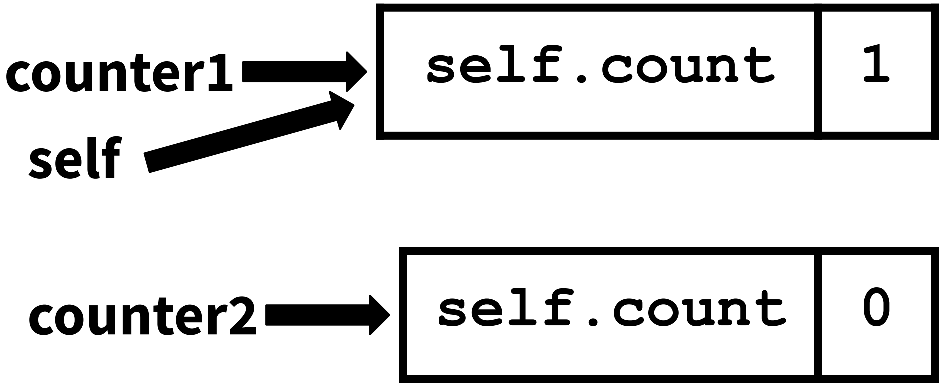
```
from counter import Counter

def main():
    counter1 = Counter()
    counter2 = Counter()
    counter1.next()
    counter1.next()
    counter2.next()
```



Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```

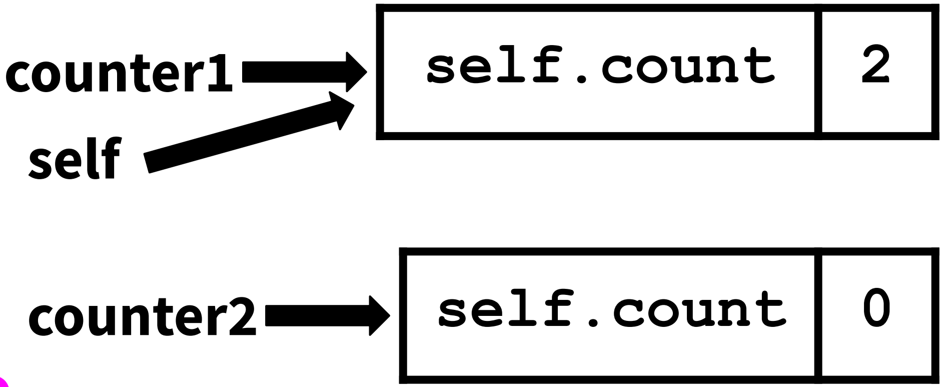


```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```



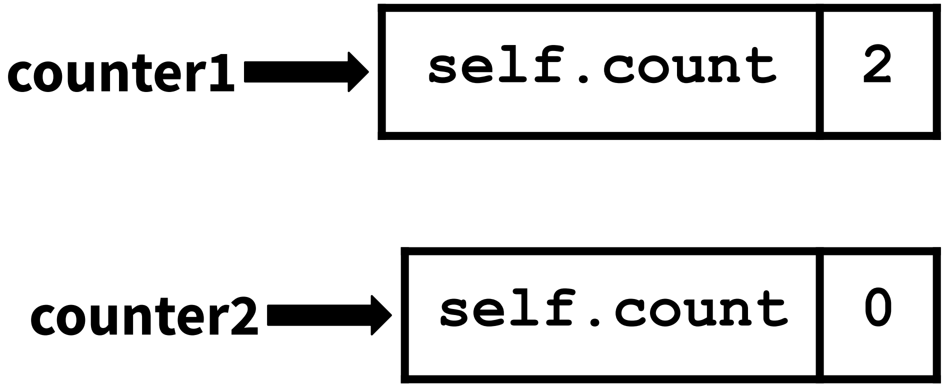
```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Stanford | ENGINEERING
Computer Science

Instance Variables

- Each Counter has its own count - trace!

```python
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```



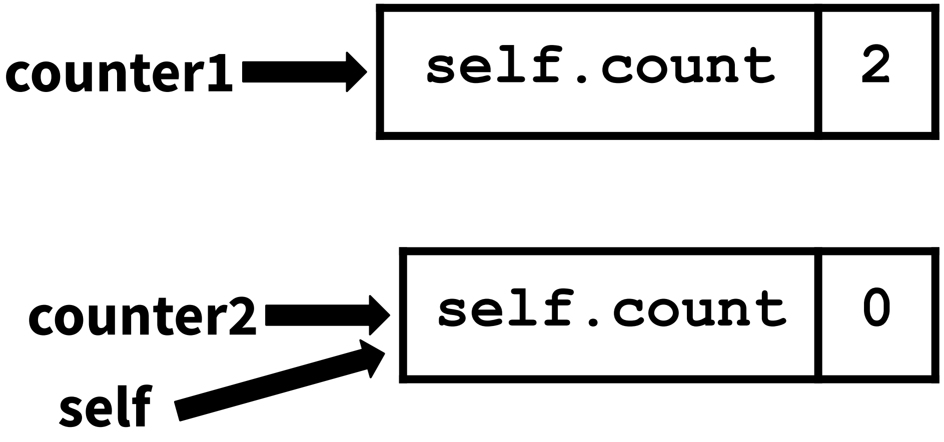
```python
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Stanford | ENGINEERING
Computer Science

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
    self.count += 1
```



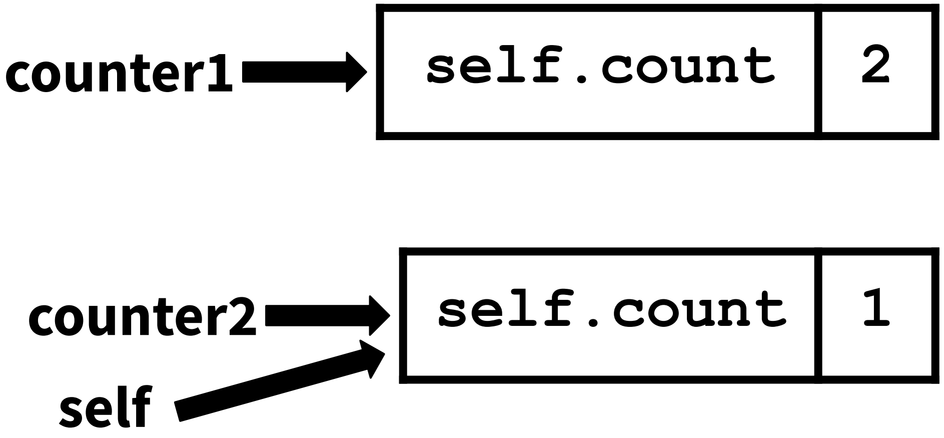
```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Stanford | ENGINEERING
Computer Science

Instance Variables

- Each Counter has its own count - trace!

```
class Counter:
  def __init__(self):
      self.count = 0

  def next(self):
      self.count += 1
```



```
from counter import Counter

def main():
   counter1 = Counter()
   counter2 = Counter()
   counter1.next()
   counter1.next()
   counter2.next()
```

Stanford | ENGINEERING
Computer Science

Animal class

- Let's make a class to represent an animal
- Animals should at the very least know their name (like "Dog" "Cat" "Lion" etc)
- What other information (aka instance variables should an Animal know about itself?)
  - Name
  - Sound
  - ? (if time - design your own!)
- What should an animal be able to do?
  - Speak (print its sound)
  - ? (if time - design your own!)

To Pycharm!

animal.py and zoo.py

We are ready!

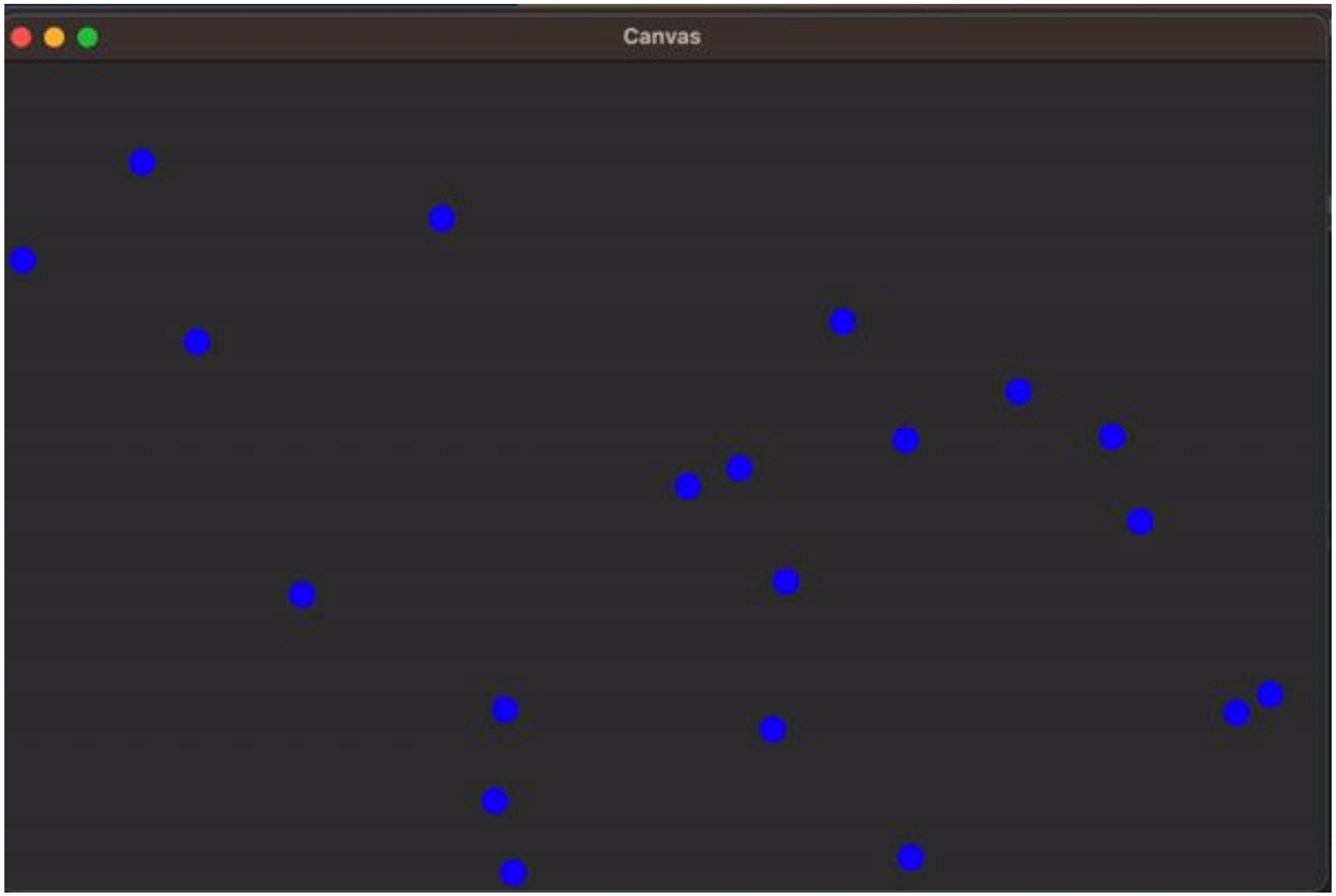
Lets make some BouncyBalls

Bouncy Balls

- Write a program that takes in as command line arguments a **number of balls and a color**
- Create that number of circles of that color on a canvas with random positions
- Make each ball bounce around with random and independent trajectories
  - We will use change_x and change_y of -1/1

Goal

(Don’t worry about my canvas being in dark mode)



Stanford | ENGINEERING
Computer Science

Step1: BouncyBall class variables/constructor

- Make a class that represents one bouncing ball in our goal program
- What pieces of information does a bouncy ball need to track?
  - The canvas it is a part of
  - The actual oval object on the canvas
  - ?

Stanford | ENGINEERING
Computer Science

To Pycharm!

Step2: BouncyBall class methods

- Make a class that represents one bouncing ball in our goal program
- What pieces of information does a bouncy ball need to track?
  - The canvas it is a part of
  - The actual oval object on the canvas
  - Its own velocity (change_y and change_x)
- What does a bouncy ball need to be able to do?
  - Move, and bounce when it should

To Pycharm!

Step3: Use the BouncyBall class

- Write a function (which you should call from main in animate_bouncyballs.py) that:
  - Creates all the bouncy balls we need
  - Returns a list of all the bouncy balls we made
- Test by calling it from main and looking at the canvas

Step4: Animate!

- Introduce an animation loop that causes all of the bouncy balls to bounce around the screen~
- Run and admire your creation

Stanford | ENGINEERING
Computer Science

Recap

- We can define our own custom types 🧠🧠🧠🧠

- To do so, we define a **Class** with a constructor, instance variables and methods
    - Instance variables: What does a class instance **know** about itself?
    - Methods: What can a class instance **do?**

- Then we can use as many variables of that type as we want! And they are all independent!