

# CS 260: Foundations of Data Science

Prof. Sara Mathieson

Spring 2025



**HAVERFORD**  
COLLEGE

# Admin

- **Roster** should hopefully be finalized soon
- **Lab 1** due Wednesday night
- Office hours: **4-5pm on Monday** (H110)

# Outline

- Object-oriented programming (OOP) in Python
- Reading in data in Python (file I/O)
- Numerical Python (numpy)

# Outline

- Object-oriented programming (OOP) in Python
- Reading in data in Python (file I/O)
- Numerical Python (numpy)

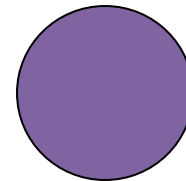
# Classes in Python represent the same idea as classes in Java

- Classes allow us to encapsulate common data structures and actions so we don't have to define them over and over again
- Example: say we have two classes: **Point** and **Circle**

# Classes in Python represent the same idea as classes in Java

- Classes allow us to encapsulate common data structures and actions so we don't have to define them over and over again
- Example: say we have two classes: **Point** and **Circle**
- We can create a new *instance* of a class using the *constructor*

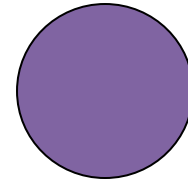
`dot = Circle(Point(x,y), r)`



# Classes in Python represent the same idea as classes in Java

- Classes allow us to encapsulate common data structures and actions so we don't have to define them over and over again
- Example: say we have two classes: **Point** and **Circle**
- We can create a new *instance* of a class using the *constructor*

```
dot = Circle(Point(x,y), r)
```



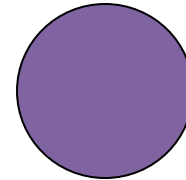
- We can access the instance's data using *methods*  

```
r = dot.get_radius()
```

# Classes in Python represent the same idea as classes in Java

- Classes allow us to encapsulate common data structures and actions so we don't have to define them over and over again
- Example: say we have two classes: **Point** and **Circle**
- We can create a new *instance* of a class using the *constructor*

```
dot = Circle(Point(x,y), r)
```

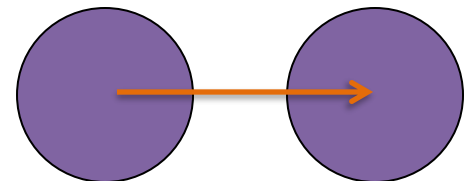


- We can access the instance's data using *methods*

```
r = dot.get_radius()
```

- We can use/modify class instances using *methods*

```
dot.move(dx,dy)
```





# Motivation for classes: LOLs

- List-of-lists let us keep track of things that should be “together”, but they get cumbersome to modify:

Type of pie



Number of slices



```
>>> pie_lst = ["apple",8], ["cherry",8], ["chocolate",8]]
>>>
>>> pie_lst[2][1] -= 1
>>>
>>> pie_lst
[['apple', 8], ['cherry', 8], ['chocolate', 7]]
```

# Motivation for classes: encapsulation and abstraction

- Encapsulated (student is represented as one thing, a list), but not abstract

```
kendre = ["Kendre", 2020, ["cs35", "act1", "relg43", "span1"]]  
name = kendre[0]  
year = kendre[1]
```

# Motivation for classes: encapsulation and abstraction

- Encapsulated (student is represented as one thing, a list), but not abstract

```
kendre = ["Kendre", 2020, ["cs35", "act1", "relg43", "span1"]]  
name = kendre[0]  
year = kendre[1]
```

- Neither encapsulated (data for one student is spread over multiple objects), nor abstract

```
name_lst = ["Kendre", "Rohan", "Ayaka", "Maleyah"]  
year_lst = [2020, 2021, 2020, 2021]  
name = name_lst[0]  
year = year_lst[0]
```

# Motivation for classes: encapsulation and abstraction

- Encapsulated (student is represented as one thing, a list), but not abstract

```
kendre = ["Kendre", 2020, ["cs35", "act1", "relg43", "span1"]]  
name = kendre[0]  
year = kendre[1]
```

- Neither encapsulated (data for one student is spread over multiple objects), nor abstract

```
name_lst = ["Kendre", "Rohan", "Ayaka", "Maleyah"]  
year_lst = [2020, 2021, 2020, 2021]  
name = name_lst[0]  
year = year_lst[0]
```

- Both abstract and encapsulated

Should be:  
get\_name()  
get\_year()

```
kendre = Student("Kendre", 2020)  
name = kendre.getName()  
year = kendre.getYear()
```

# Advantages of encapsulation/abstraction

- Interface (how you interact with something) is consistent even if the internal details change.
  - 1) If you change the engine in your car, you still drive it the same way – don't need to know how the engine works.
  - 2) In online shopping you have a “Cart”, which is an abstract concept and is roughly the same across sites. Probably represented as a list underneath but user doesn't need to know.

# “Pie” class example

```
class Pie: # class names should be capitalized

    # must use init for the constructor
    def __init__(self, flavor):
        """Constructor for the Pie class."""
        # in the constructor, define the data (i.e. self.data)
        # data are called: attributes or instance variables
        self.flavor = flavor
        self.slices = 8

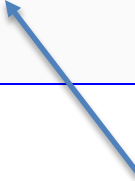
    def get_slices(self):
        """Return the number of slices left (int)."""
        return self.slices

    def get_flavor(self):
        """Return the flavor of the pie (str)."""
        return self.flavor
```

# “Pie” class example

```
def serve(self):
    """If there is at least one slice left, reduce the number of slices."""
    if self.slices > 0:
        print("Here is a slice of %s pie!" % self.flavor)
        self.slices -= 1
    else:
        print("Sorry, there is no more %s pie!" % self.flavor)

def __str__(self):
    """Return a string representation of a pie."""
    s = "%s pie has %i slices left!" % (self.flavor, self.slices)
    return s
```



better to use: f“{self.flavor} pie has {self.slices} left!”

# "Pie" class example

```
def main():  
    pie1 = Pie("apple")  
    print(pie1) # __str__ is automatically called when we call print(..)  
    for i in range(12):  
        pie1.serve()  
    print(pie1.get_slices())  
    print(pie1.get_flavor())  
    print(pie1)  
  
    pie2 = Pie("pumpkin")  
    print(pie2)  
    pie2.serve()  
    print(pie2)
```

```
apple pie has 8 slices left!  
Here is a slice of apple pie!  
Here is a slice of apple pie!  
Here is a slice of apple pie!  
Here is a slice of apple pie!  
Here is a slice of apple pie!  
Here is a slice of apple pie!  
Here is a slice of apple pie!  
Here is a slice of apple pie!  
Sorry, there is no more apple pie!  
Sorry, there is no more apple pie!  
Sorry, there is no more apple pie!  
Sorry, there is no more apple pie!
```

```
0
```

```
apple  
apple pie has 0 slices left!  
pumpkin pie has 8 slices left!  
Here is a slice of pumpkin pie!  
pumpkin pie has 7 slices left!
```



# Best Practices workflow demo

- Running code on the command line
- Classes demo

# TwitterUser class similar to demo

```
class TwitterUser: # only time camel case is okay!

    # constructor
    def __init__(self, name, curr_following, curr_followers):
        self.name = name
        self.following = curr_following
        self.followers = curr_followers

    def add_follower(self): # always have to use self!
        self.followers += 1
        # TODO we could make this better by creating a list of followers who
        # are themselves instances of TwitterUser

    def follow(self):
        self.following += 1

    def __str__(self):
        # must return a string, not print a string!
        return "name: %s\nnum following: %i\nnum followers: %i" % (self.name, \
            self.following, self.followers)
```

# Handout 2

- Complete on your own

# Recap Die class

- Defining the Constructor: builds an instance of the class (self), and initializes all instance variables (self.xxx)

```
class Die:

    def __init__(self, num_sides):
        """Construct a new die with the given number of sides."""
        self.sides = num_sides
        self.value = 1 # default starting value
```

# Recap Die class

- Defining the Constructor: builds an instance of the class (self), and initializes all instance variables (self.xxx)

```
class Die:

    def __init__(self, num_sides):
        """Construct a new die with the given number of sides."""
        self.sides = num_sides
        self.value = 1 # default starting value
```

- Using the Constructor: assign the new object to a variable, making the “self” placeholder a concrete instance

```
def main():
    # create 8-sided dice
    die1 = Die(8)
    die2 = Die(8)
```

# Recap Die class

- Defining Methods: always use “self” as the first argument (placeholder for the instance). Getters are a type of method that return instance variables or their derivatives.

```
def get_value(self):  
    """Getter for the die's current value."""  
    return self.value  
  
def roll(self):  
    """Choose a new random value for the die, i.e. roll it."""  
    self.value = random.randrange(1,self.sides+1)
```

# Recap Die class

- Defining Methods: always use “self” as the first argument (placeholder for the instance). Getters are a type of method that return instance variables or their derivatives.

```
def get_value(self):  
    """Getter for the die's current value."""  
    return self.value  
  
def roll(self):  
    """Choose a new random value for the die, i.e. roll it."""  
    self.value = random.randrange(1, self.sides+1)
```

- Using Methods: instance.method(...), don't use self

```
# roll both until we get the same value  
same = False  
while not same:  
    die1.roll()  
    die2.roll()  
    print(die1)  
    print(die2)  
    print()  
    # check if the values are the same  
    same = (die1.get_value() == die2.get_value())
```

# Recap Die class

- Defining the `__str__` method: no `print(..)` statements!  
Build and return a single string. (no arguments besides `self`)

```
def __str__(self):  
    """String representation of the die (with current value)."""  
    return f"{self.sides}-sided die, current value: {self.value}"
```



# Recap Die class

- Defining the `__str__` method: no `print(..)` statements! Build and return a single string. (no arguments besides `self`)

```
def __str__(self):  
    """String representation of the die (with current value)."""  
    return f"{self.sides}-sided die, current value: {self.value}"
```

- Using the `__str__` method: simply call `print(instance)`!

```
print(die1)  
print(die2)
```

# Outline

- Object-oriented programming (OOP) in Python
- Reading in data in Python (file I/O)
- Numerical Python (numpy)

```
# open(..) returns a file object (called an TextIOWrapper but think: file)  
c_file = open("colleges.txt", 'r') # 'r' for read, 'w' for write
```

```
enroll_lst = []
```

```
# one way to read a file: loop through each line of the file colleges.txt
```

```
for line in c_file:  
    # split breaks up the line on spaces, it is a method t  
    tokens = line.split()  
  
    # extract information from specific tokens  
    name = tokens[0]  
    enroll = int(tokens[1])  
    enroll_lst.append(enroll)
```

```
# always remember to close your files!
```

```
c_file.close()
```

```
Amherst 1792  
Bates 1792  
Bowdoin 1806  
BrynMawr 1709  
Colby 1815  
Davidson 1950  
HarveyMudd 735  
Haverford 1290  
Middlebury 2526  
Pomona 1663  
Reed 1411  
Smith 2600  
Swarthmore 1620  
Vassar 2450  
Wellesley 2474  
Williams 2099
```

# Demo of reading in data

# File reading methods

```
import csv
import numpy as np

# 1) read line by line
fb_file = open("data/facebook_users.csv", 'r') # 'r' for read mode
for line in fb_file:
    tokens = line.split(",") # split on comma
    year = int(tokens[0])
    num_users = int(tokens[1])
    print(year, num_users)
fb_file.close()

# 2) csv reader
with open("data/facebook_users.csv", 'r') as fb_file:
    csv_reader = csv.reader(fb_file)
    for line in csv_reader:
        print(line)

# 3) load into numpy array
data = np.loadtxt("data/facebook_users.csv", dtype=int, delimiter=",")
print(data)
```

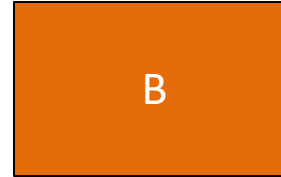
# Outline

- Object-oriented programming (OOP) in Python
- Reading in data in Python (file I/O)
- Numerical Python (numpy)

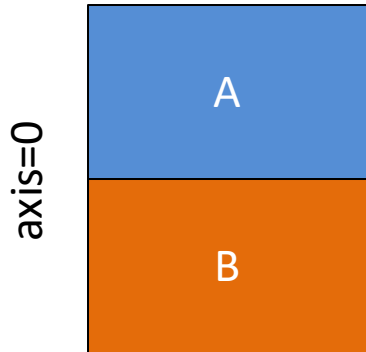
# Numpy

- Numerical Python
- Designed for fast computation on arrays
- Implemented in C underneath
- **pip3 install numpy** (on the terminal) OR  
**python3 -m pip install numpy**

# Numpy concatenation

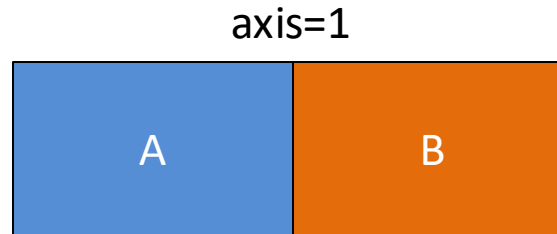


`np.concatenate((A,B), axis=0)`



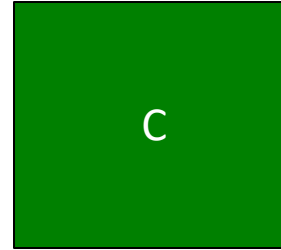
*must match along axis 1*

`np.concatenate((A,B), axis=1)`

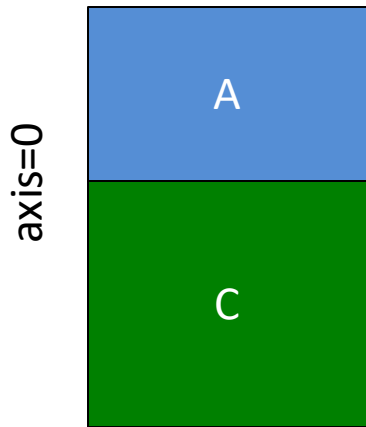


*must match  
along axis 0*

# Numpy concatenation

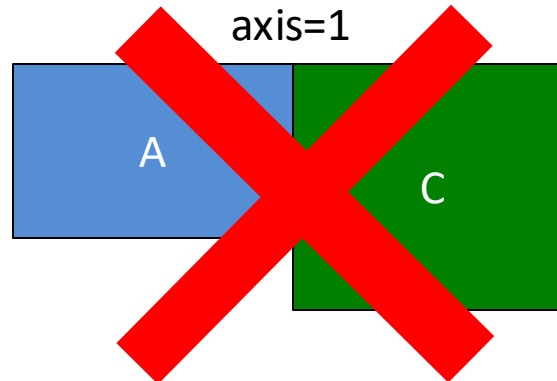


`np.concatenate((A,C), axis=0)`



*must match along axis 1*

`np.concatenate((A,C), axis=1)`



*Error: must  
match along  
axis 0!*