Welcome to Lecture 18: Intro to OOP

1) Open a Code Editor
    a) code.cs61a.org
    b) VS Code
2) Use Iclicker for attendance
3) Lecture 18 Guide: tinyurl.com/S24CS10L5

Topics

- **Class**: A blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class will have.
- **Object**: An instance of a class. It is created using the class blueprint and can have its own state (attributes) and behavior (methods).
- **Method**: A function defined within a class. It describes the behaviors of the objects created from the class.
- **Attribute**: A variable defined within a class. It describes the state or properties of the objects created from the class.

Announcements

- Project 4: Pyturis will be released on Thursday
- Midterm Retake on Friday, 1 to 4PM
  - Same logistics as Midterm
- Midterm Review Recording Available…

Review: Tuples

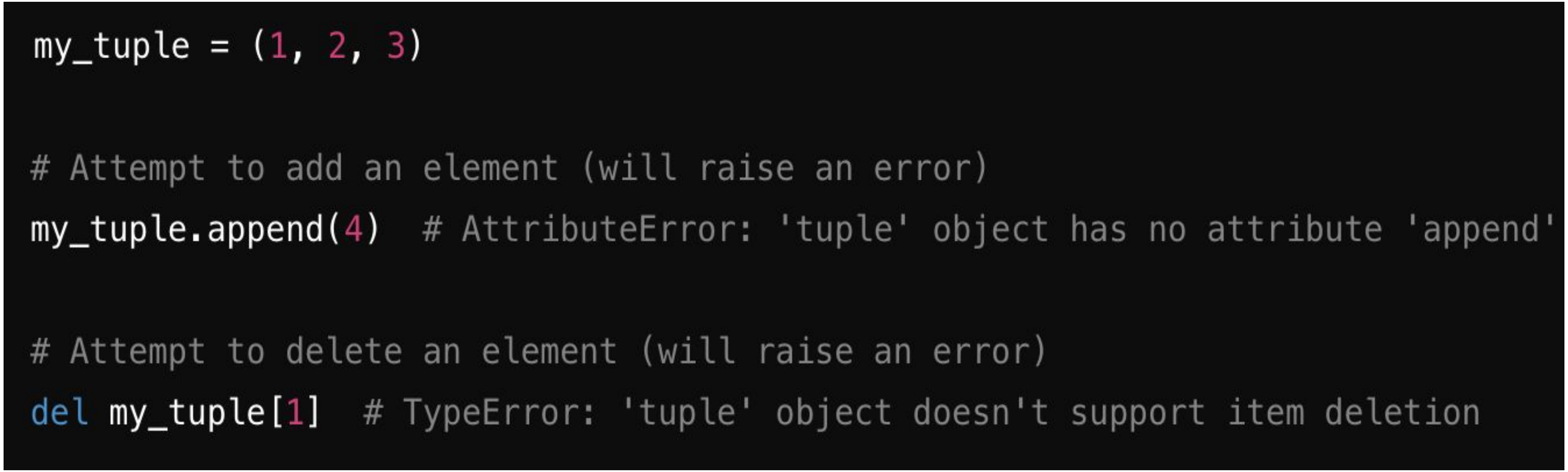
Tuples are similar to lists:

- You create them using comma separated lists inside parentheses rather than square brackets
- You can access values at specific indices with square brackets just like with lists, you just can’t change the values
- Tuples are immutable, lists are mutable

```
some_tuple = (1, 5, 10, 4, 7, 16, 2)

some_list = [1, 5, 10, 4, 7, 16, 2]
```

Tuples: Immutable

1. **You cannot add or delete elements**
   a. Once a tuple is created, you cannot add new elements to it or remove existing elements from it.
   b. The size and content of the tuple are fixed.



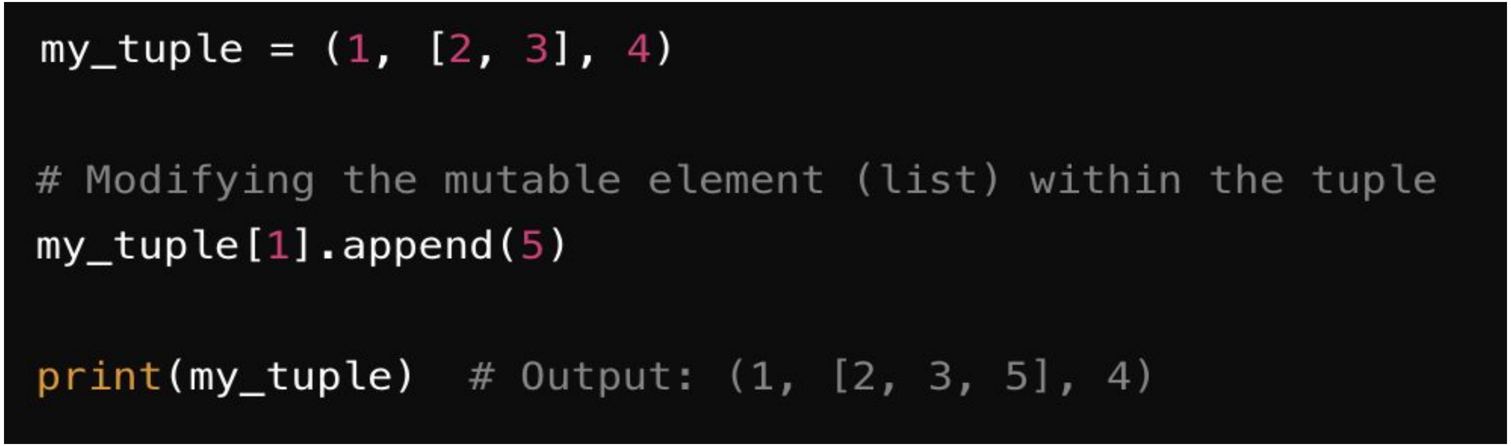
```python
my_tuple = (1, 2, 3)

# Attempt to add an element (will raise an error)
my_tuple.append(4)  # AttributeError: 'tuple' object has no attribute 'append'

# Attempt to delete an element (will raise an error)
del my_tuple[1]  # TypeError: 'tuple' object doesn't support item deletion
```

Tuples: Modifying Values within a Tuple

Modifying values

a. If the elements of the tuple are immutable types (like integers, strings, or another tuple), those values cannot be changed.

b. However, if the tuple contains mutable elements (like lists or dictionaries), those elements can be modified.



```
my_tuple = (1, [2, 3], 4)

# Modifying the mutable element (list) within the tuple
my_tuple[1].append(5)

print(my_tuple)  # Output: (1, [2, 3, 5], 4)
```

Review: Creating/Accessing Tuple Elements

```
my_tuple = (1, 'apple', 3.14, True, 'Python')
```

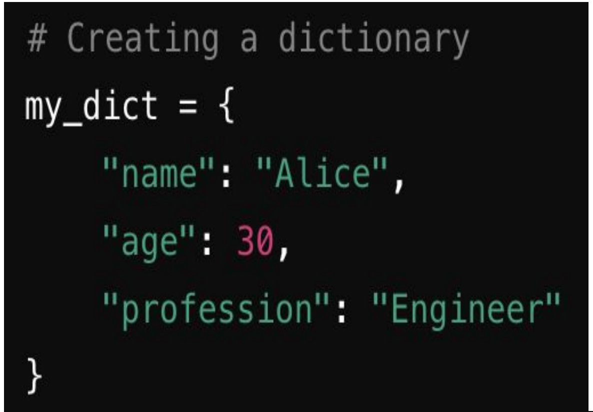
```
print(my_tuple[0]) # Output: 1
print(my_tuple[2]) # Output: 3.14
print(my_tuple[-1]) # Output: Python
```

Review: Tuple Operations

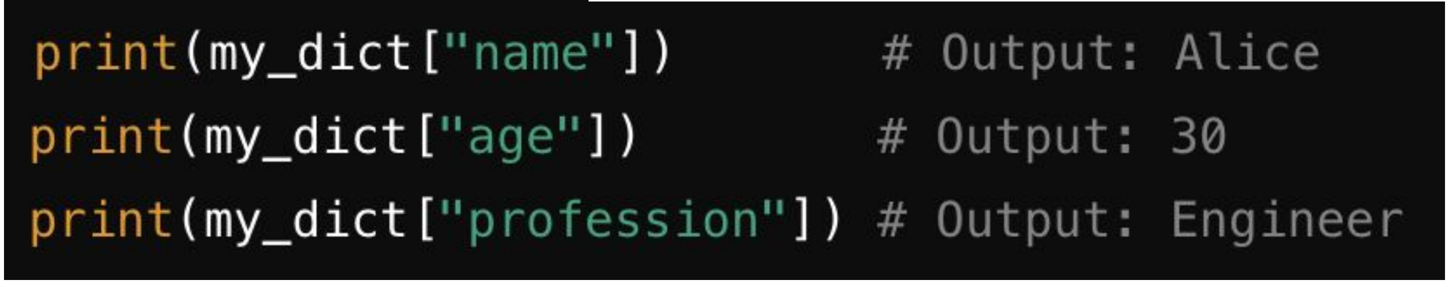
- Concatenate
- Slicing
- Unpacking a Tuple in Variables
- Iterating Over a Tuple

Review: Creating Dictionaries and Accessing Values

You can access values in a dictionary by using their corresponding keys.

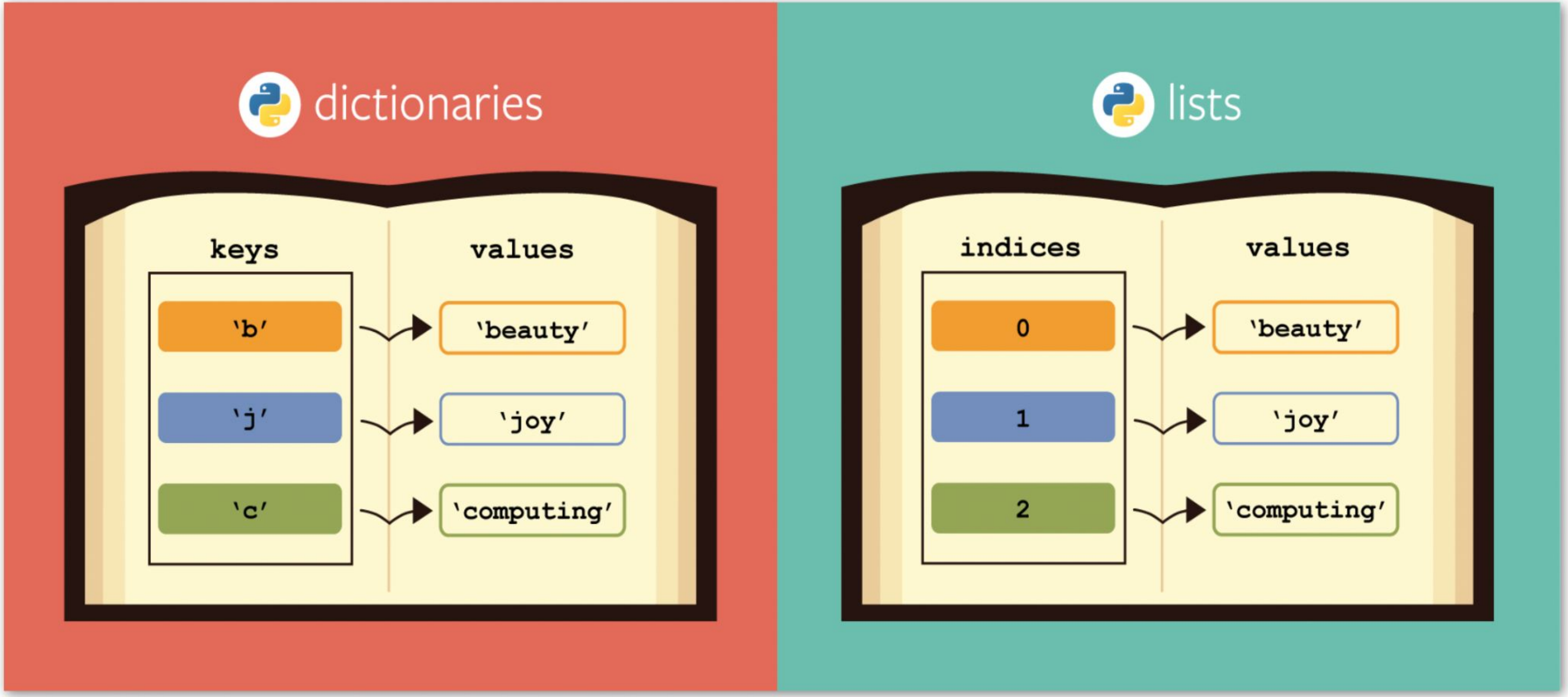


```
# Creating a dictionary
my_dict = {
    "name": "Alice",
    "age": 30,
    "profession": "Engineer"
}
```



```
print(my_dict["name"])       # Output: Alice
print(my_dict["age"])        # Output: 30
print(my_dict["profession"]) # Output: Engineer
```

Dictionaries vs Lists



Dictionary Methods

- **.keys()**: Returns a view object of all the keys in the dictionary
- **.values()**: Returns a view object of all the values in the dictionary
- .**items()**: Returns a view object of all the key-value pairs in the dictionary
- **.get()**: Returns the value for a specified key if the key is in the dictionary
- .**update()**: Updates the dictionary with elements from another dictionary object or from an iterable of key-value pairs

Iterating through a Dictionary

- Iterate through **keys**

```
for k in my_dictionary.keys():
for k in my_dictionary:
```

- Iterate through **values**

```
for v in my_dictionary.values():
```

- Iterate through **keys and values**

```
for k,v in my_dictionary.items():
```

- Check **if k is a key** in dictionary

```
k in my_dictionary.keys()
k in my_dictionary
```

- Check **if v is a value** in dictionary

```
v in
my_dictionary.values()
```

CS10 is not a course about Snap! Or Python…

What we’re learning is Computational Thinking and Creative Problem Solving

- Fundamentals of Computer Science
- Developing strategies to solve problems
  - Define: What is the problem asking?
  - Research/Reference: Where have I seen this before?
  - Apply the concepts in code
  - Feedback/Iterate: What worked? What needs to be changed?

Recall…Type( ) function

You can use the `type` **function to determine a variables’ type**

It gives you a little more than you need to know, but you can find the type in single quotes after the world `class`**.**

```python
greeting = "Hello,  world!"

print(type(greeting))
```

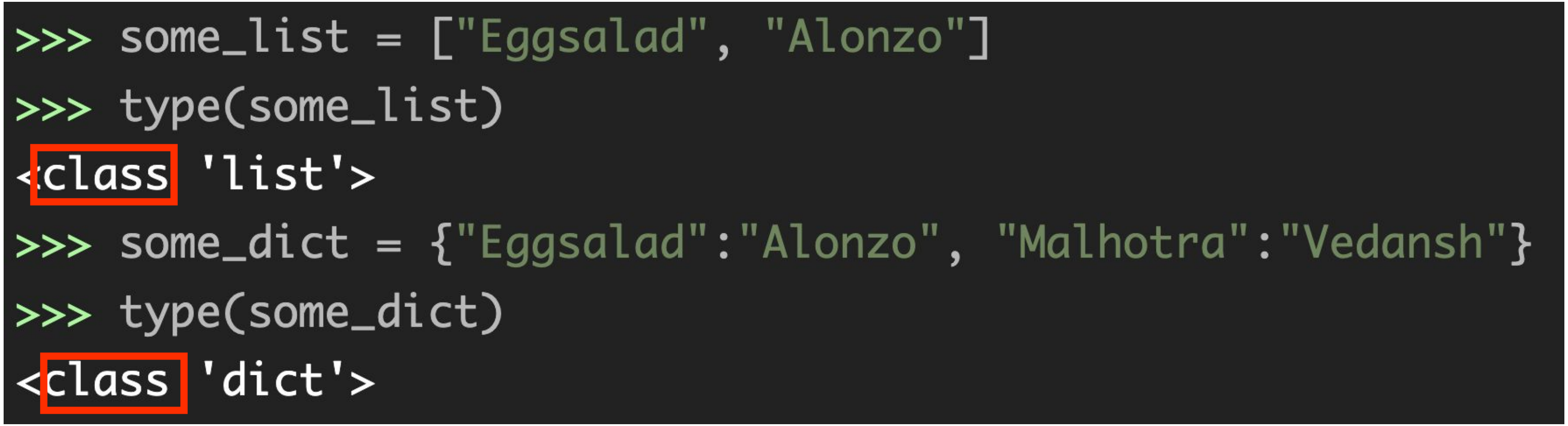
```
<class 'str'>
```

```
name = input("What is your name: ")

print(name)

print(type(name))
```

```
Name: 
```

```
Name: Lisa
Lisa
<class 'str'>
```

- Every object has a type, called its class.



```
>>> some_list = ["Eggsalad", "Alonzo"]
>>> type(some_list)
<class 'list'>
>>> some_dict = {"Eggsalad":"Alonzo", "Malhotra":"Vedansh"}
>>> type(some_dict)
<class 'dict'>
```

- Built-in classes, we'll make our own!

Enter: OBJECT-ORIENTED PROGRAMMING (OOP)

- OOP is a programming paradigm with its own vocabulary:
    - Class: A template for defining entities (called objects.)
    - Object: An entity defined by (an instance of) a particular class.
        - Every object has a type, called its class.
    - To create new types of data, we implement new classes.

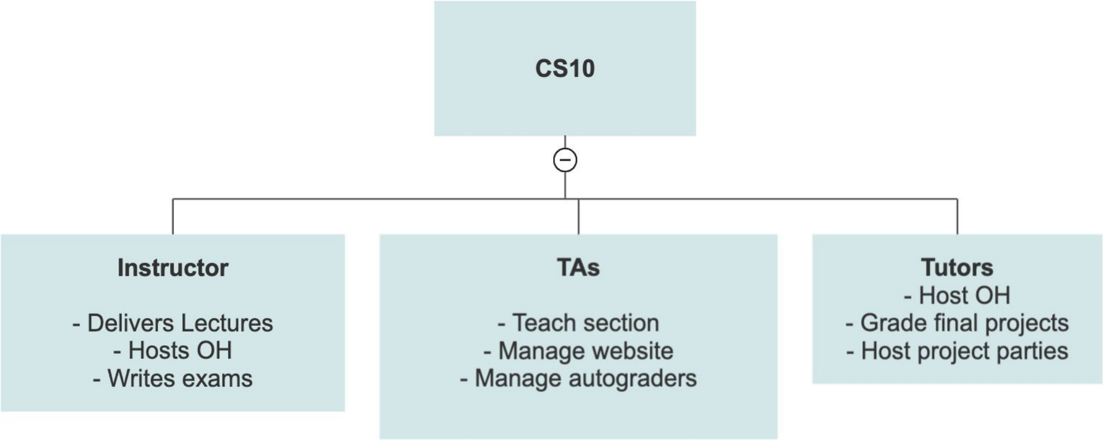
- Classes are an essential part organizing code in Object Oriented Programming (OOP)

OBJECT-ORIENTED PROGRAMMING (OOP)

- **Modular Programming:** Separating the functionality of a program into independent chunks (modules.)
- **What is it?**
  - It's a way of writing computer programs by breaking them into smaller, separate parts.
- **Why do it?**
  - It makes the program easier to understand, manage, and fix.
  - Each part (or module) can be worked on independently.

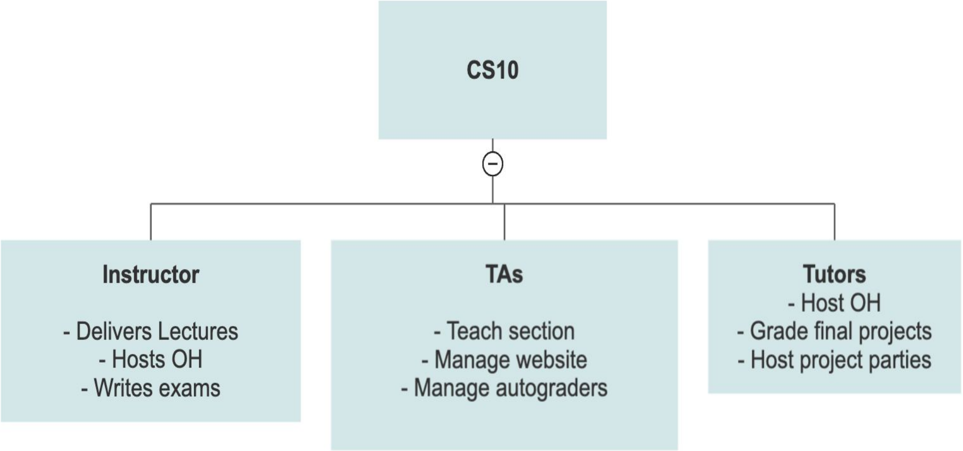
OBJECT-ORIENTED PROGRAMMING (OOP)

- Class → CS10 (Template)
- Object→ CS10 Summer 2024



OBJECT-ORIENTED PROGRAMMING (OOP)

- Modular Programming: Separating the functionality of a program into independent chunks (modules.)
- Example of a modular procedure:
  - Modules communicate
  - Abstraction barriers!



CS10

Instructor

- Delivers Lectures
- Hosts OH
- Writes exams

TAs

- Teach section
- Manage website
- Manage autograders

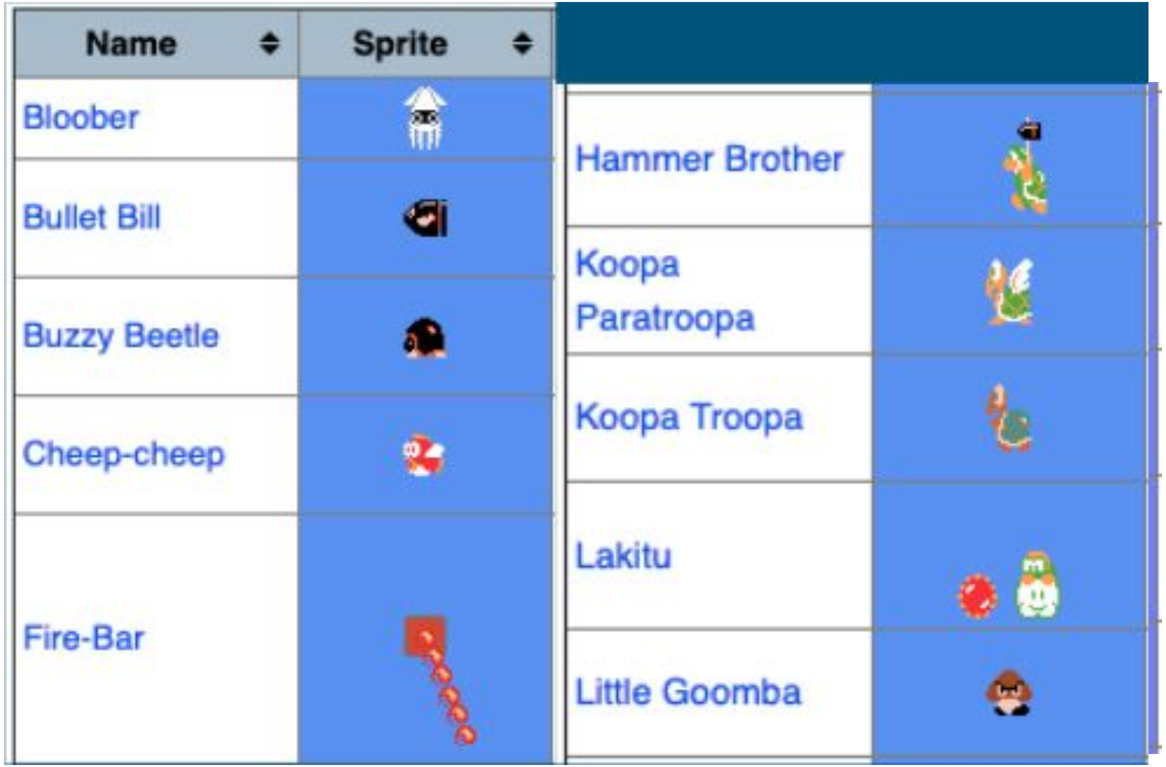
Tutors

- Host OH
- Grade final projects
- Host project parties

Defining a Class

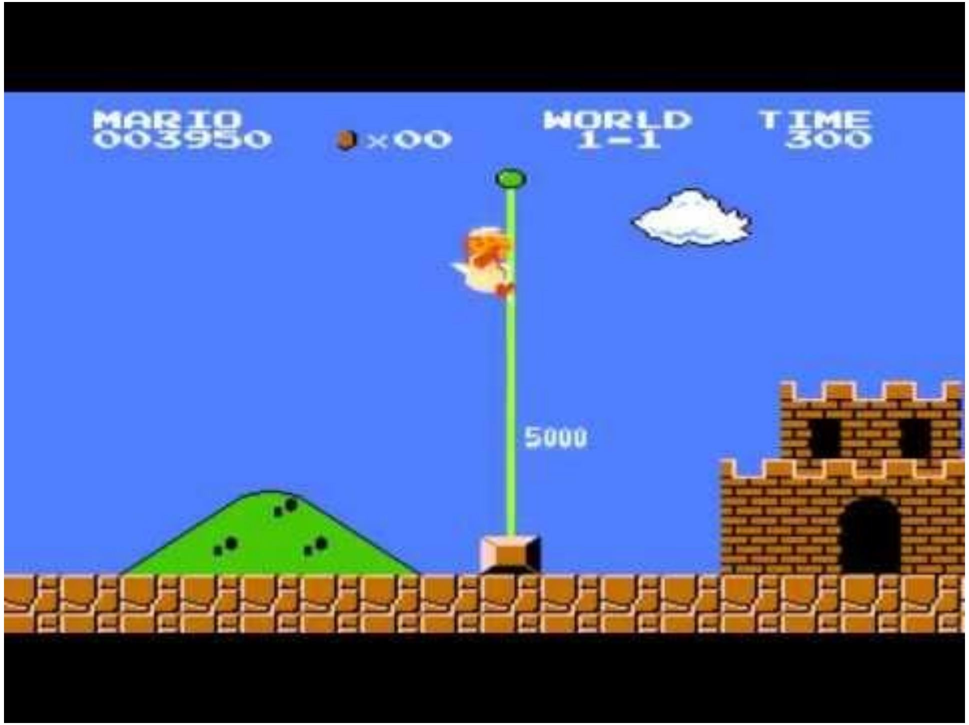
- A class is a blueprint or template for creating objects.
- It defines a set of attributes and methods that the objects created from the class will have

Super Mario Brother’s Villain Classes

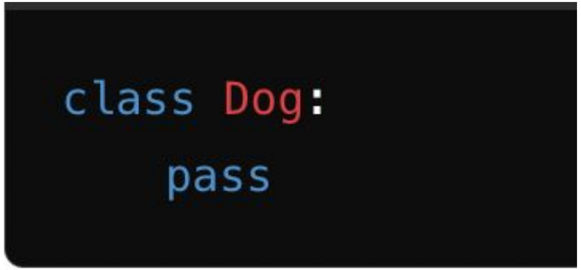


Name	Sprite
Bloober	
Bullet Bill	
Buzzy Beetle	
Cheep-cheep	
Fire-Bar	
Hammer Brother	
Koopa Paratroopa	
Koopa Troopa	
Lakitu	
Little Goomba	

Super Mario Brothers World 1-1



Defining a Class



```python
class Dog:
    pass
```

This code does the following:

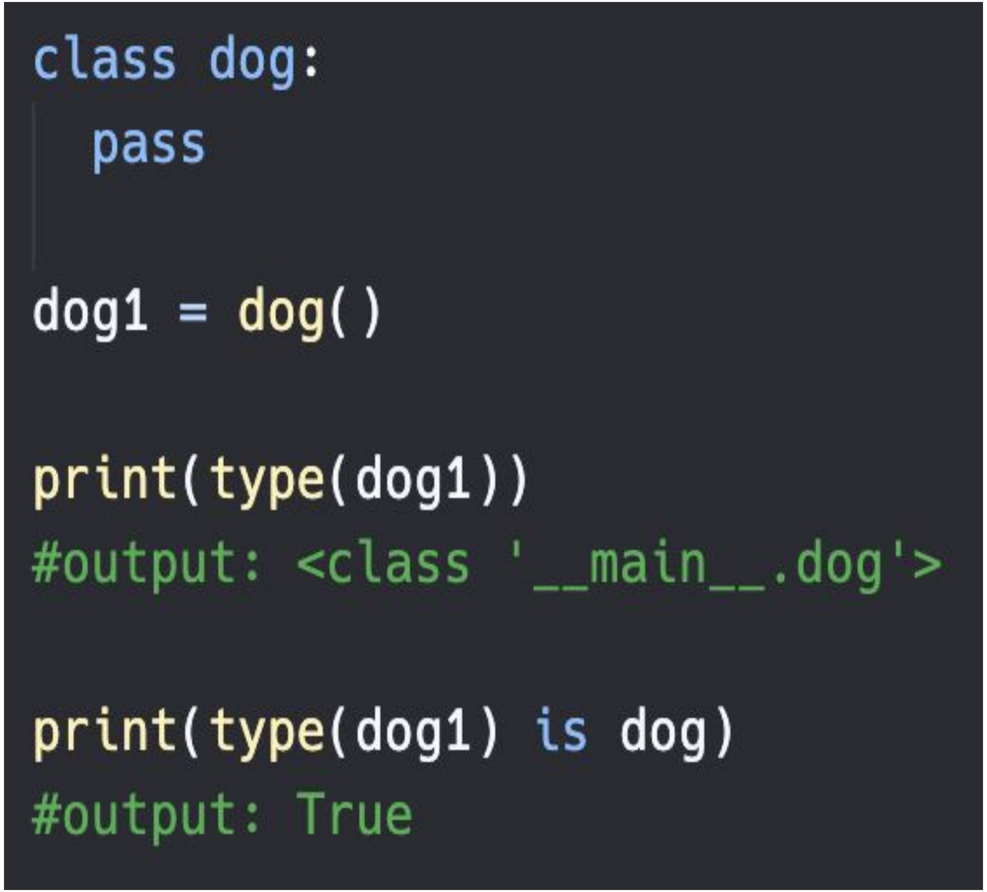
1. **Defines a Class Named Dog**: It tells Python that you are defining a new class called Dog.
2. **pass Statement**: The pass statement is a placeholder that does nothing.
   a. It is used to indicate an empty block of code.
   b. In this context, it means that the Dog class has no attributes or methods yet, but you are defining it as a class.

Defining a class

- Classes are created using class statements:

```
class <name>:
    <suite>
```

- dog1 is an object of the class Dog. Thus, the type of jack is Dog.



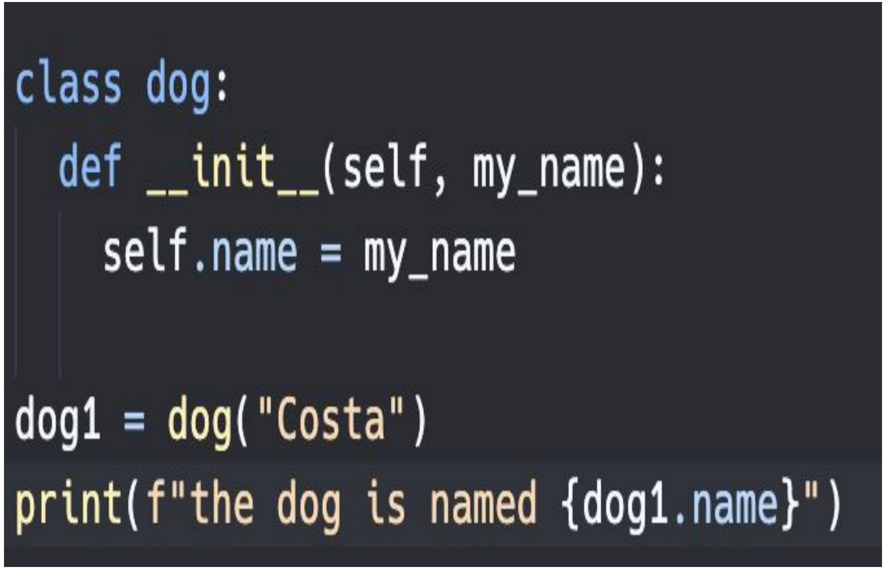
```
class dog:
    pass

dog1 = dog()

print(type(dog1))
#output: <class '__main__.dog'>

print(type(dog1) is dog)
#output: True
```

Constructors and Instance Attributes - Demo

- The “dunder init” (double-under) method is the constructor of the class `Dog`.

- When we call `dog1 = dog(“Costa”)`, the parameter `self` is bound to the newly created `dog` object.

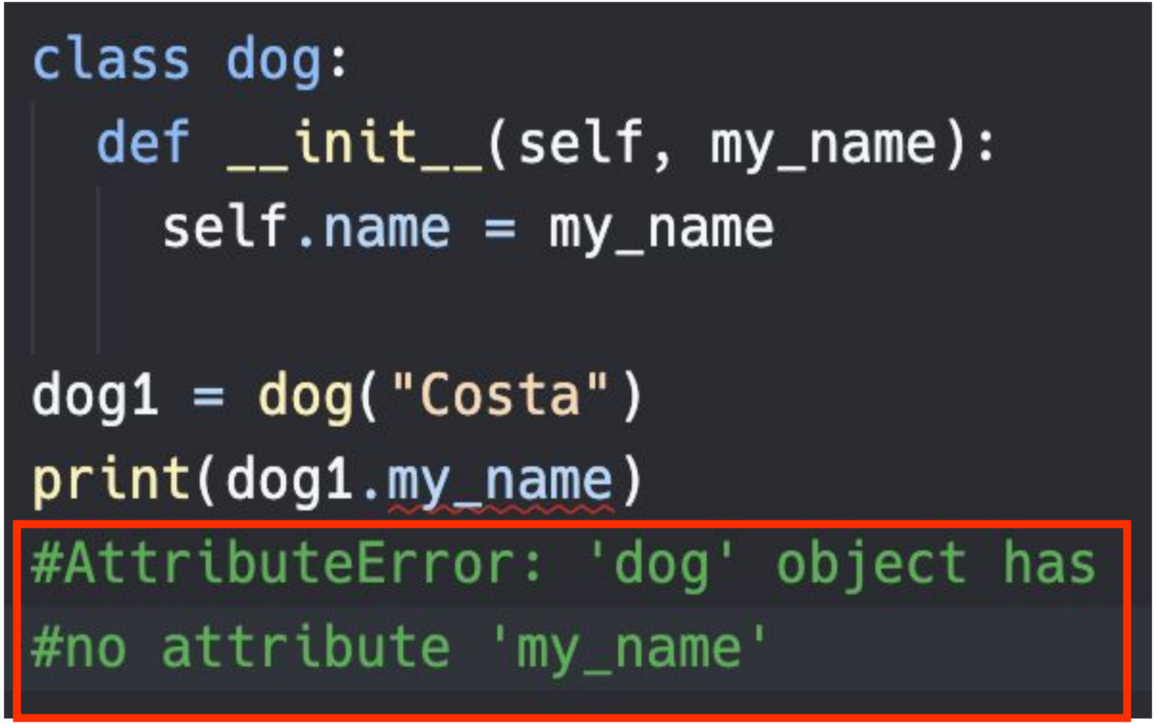
- The constructor binds the value `“Costa”` to the object’s `name` attribute.



```
class dog:
  def __init__(self, my_name):
    self.name = my_name

dog1 = dog("Costa")
print(f"the dog is named {dog1.name}")
```

QUESTION

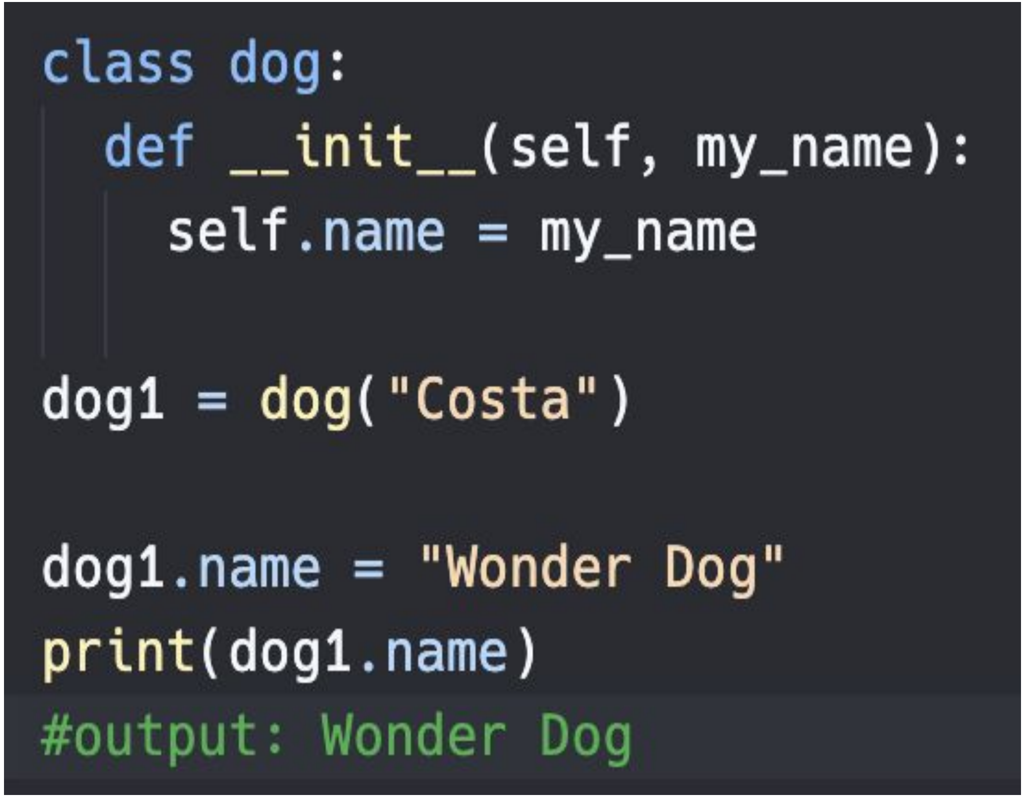
- Why does Python throw and error?



```
class dog:
  def __init__(self, my_name):
    self.name = my_name

dog1 = dog("Costa")
print(dog1.my_name)
#AttributeError: 'dog' object has
#no attribute 'my_name'
```

DOT NOTATION

- We could also rename Costa using Dot notation



```
class dog:
  def __init__(self, my_name):
    self.name = my_name

dog1 = dog("Costa")

dog1.name = "Wonder Dog"
print(dog1.name)
#output: Wonder Dog
```

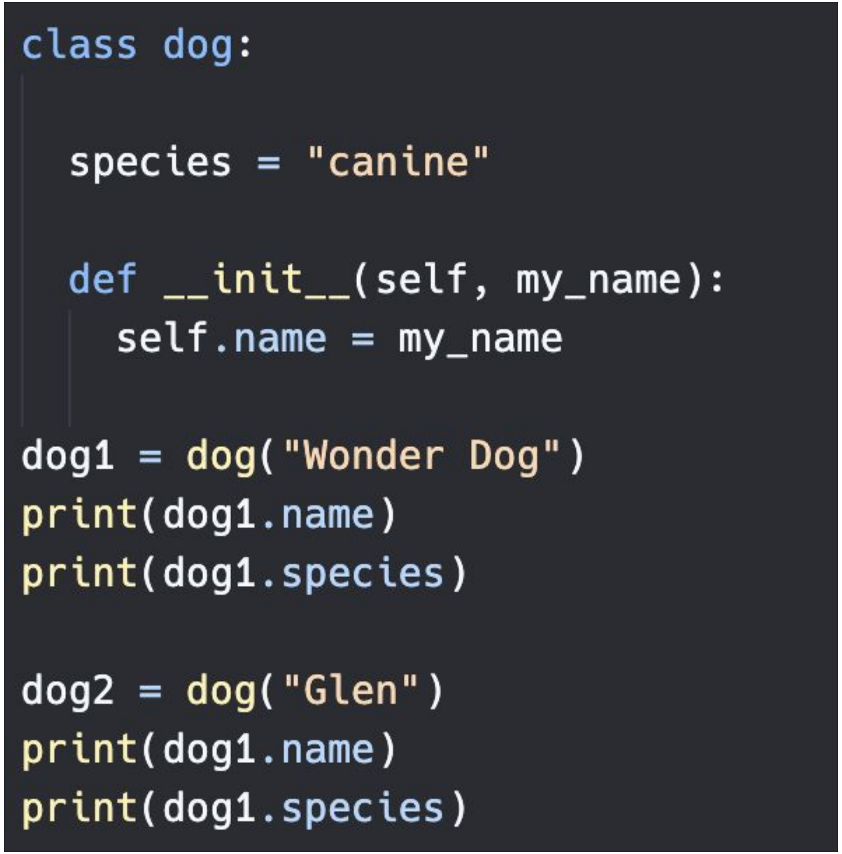
CLASS ATTRIBUTES

- Class Attributes: attributes whose values are shared across all objects of that class.
    - They typically represent properties of the class itself, and not necessarily those of a particular instance.

- Example: A class attribute for the `Dog` class might be species.

CLASS ATTRIBUTES

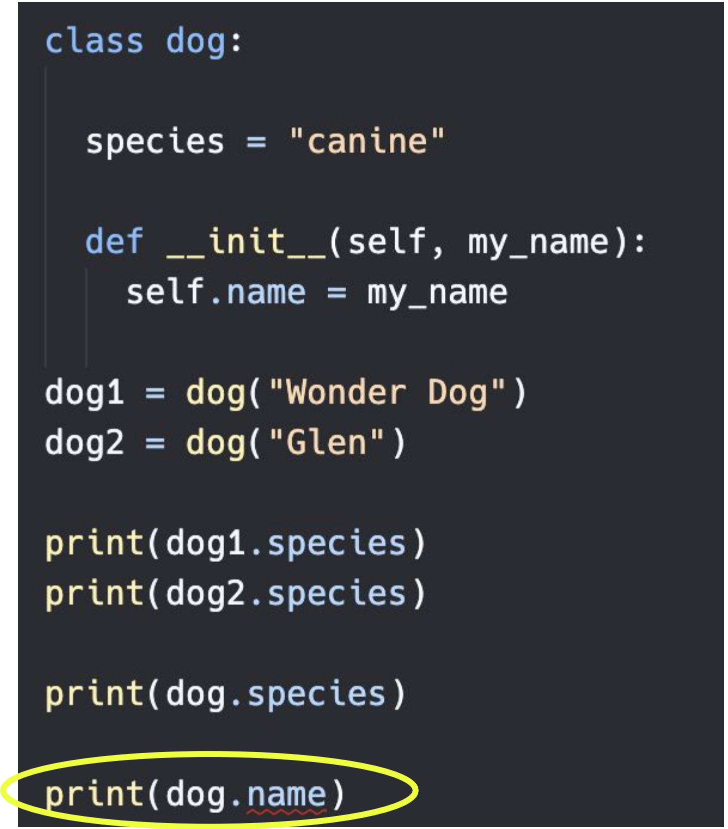
- Assigned in the suite of the class, outside any method definitions.



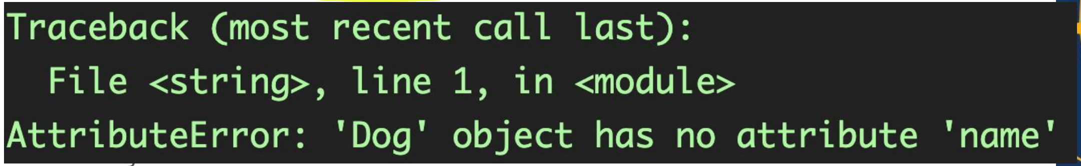
```
class dog:

  species = "canine"

  def __init__(self, my_name):
    self.name = my_name

dog1 = dog("Wonder Dog")
print(dog1.name)
print(dog1.species)

dog2 = dog("Glen")
print(dog1.name)
print(dog1.species)
```

CLASS ATTRIBUTES

- Be careful with instance vs class attributes!



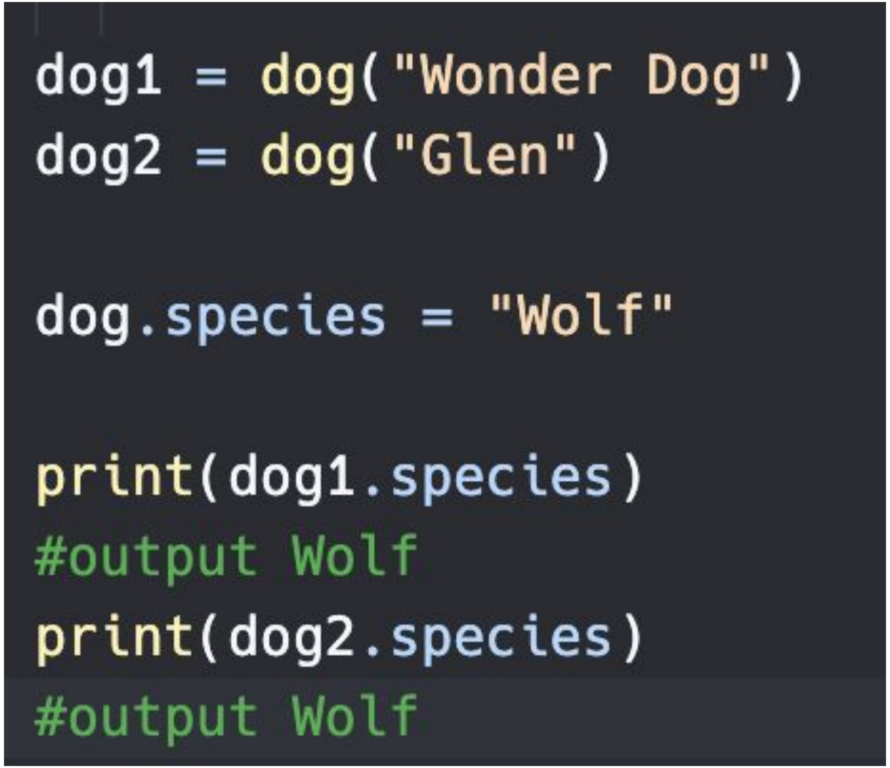
```
class dog:

  species = "canine"

  def __init__(self, my_name):
    self.name = my_name

dog1 = dog("Wonder Dog")
dog2 = dog("Glen")

print(dog1.species)
print(dog2.species)

print(dog.species)

print(dog.name)
```



```
Traceback (most recent call last):
  File <string>, line 1, in <module>
AttributeError: 'Dog' object has no attribute 'name'
```

CLASS ATTRIBUTES

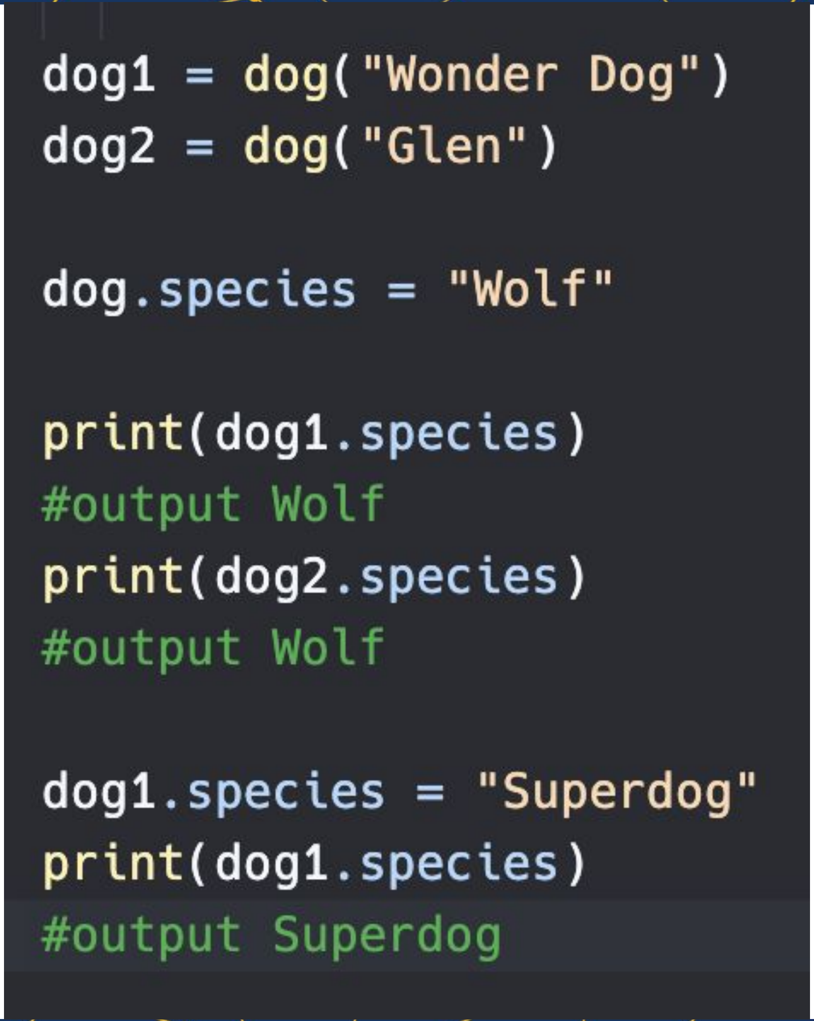
- Updating a class attribute.



```
dog1 = dog("Wonder Dog")
dog2 = dog("Glen")

dog.species = "Wolf"

print(dog1.species)
#output Wolf
print(dog2.species)
#output Wolf
```

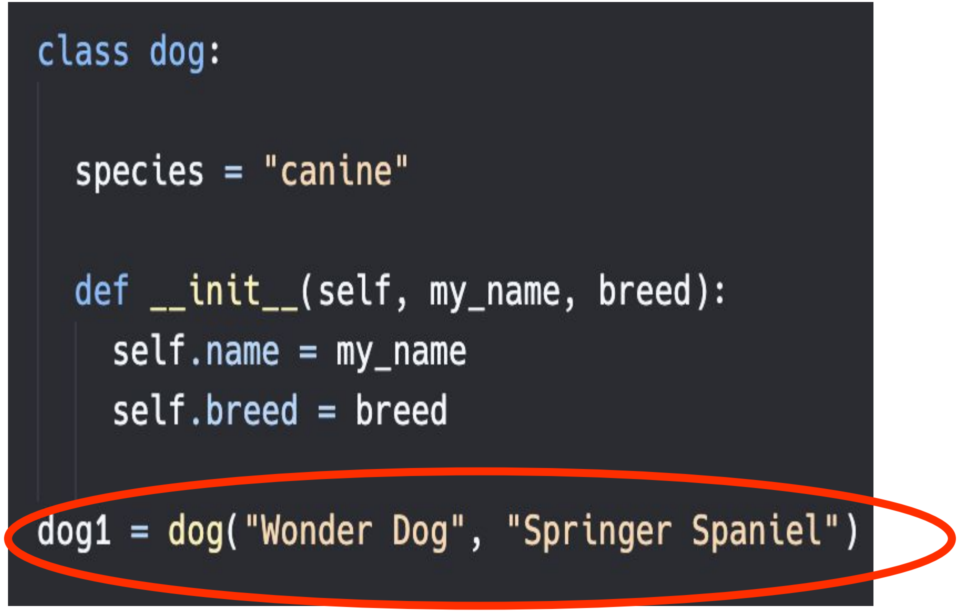
CLASS ATTRIBUTES

- Overriding a class attribute.



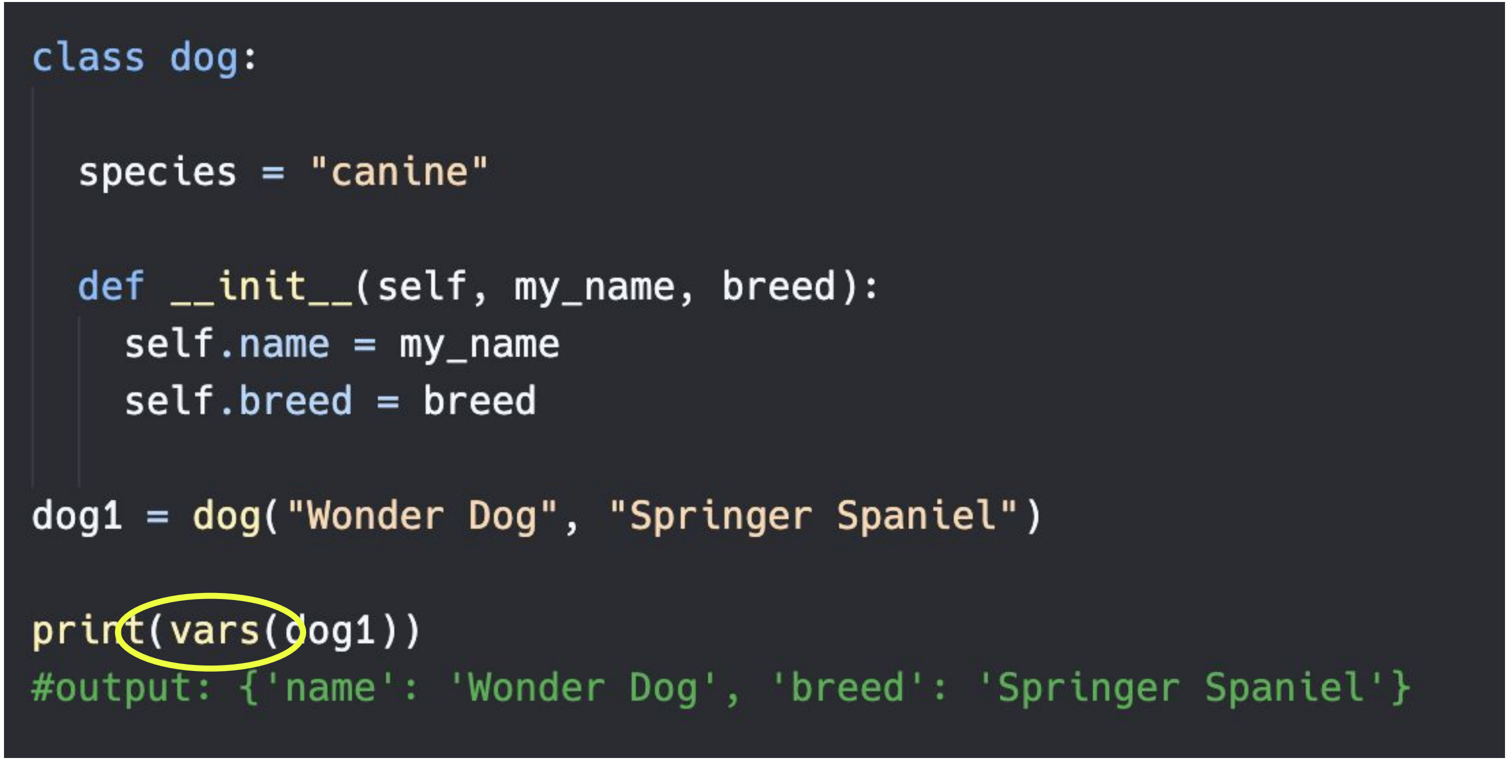
```
dog1 = dog("Wonder Dog")
dog2 = dog("Glen")

dog.species = "Wolf"

print(dog1.species)
#output Wolf
print(dog2.species)
#output Wolf

dog1.species = "Superdog"
print(dog1.species)
#output Superdog
```

Creating an Object with multiple attributes



```python
class dog:


  species = "canine"

  def __init__(self, my_name, breed):
    self.name = my_name
    self.breed = breed

dog1 = dog("Wonder Dog", "Springer Spaniel")
```

vars() function to print all the attributes of an Object



```
class dog:

  species = "canine"

  def __init__(self, my_name, breed):
    self.name = my_name
    self.breed = breed

dog1 = dog("Wonder Dog", "Springer Spaniel")

print(vars(dog1))
#output: {'name': 'Wonder Dog', 'breed': 'Springer Spaniel'}
```

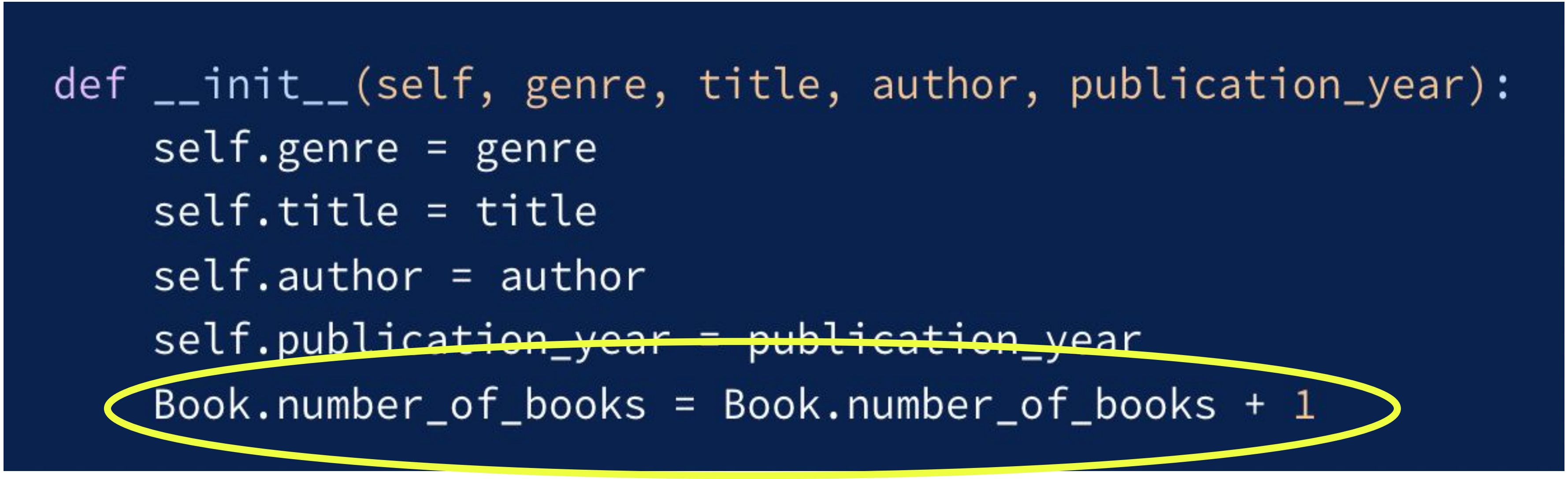
Task 1: Lets Make our own Class!

- Create a Class called “Book”
- In the Constructor, include 3 instance attributes
    - Title
    - Author
    - Publication Year
- Include 1 Class Attribute: Number of Books, set to 0

Question

How can we iterate on Number of Books each time a new Book Object is created?

Iterating Class attribute from the Constructor



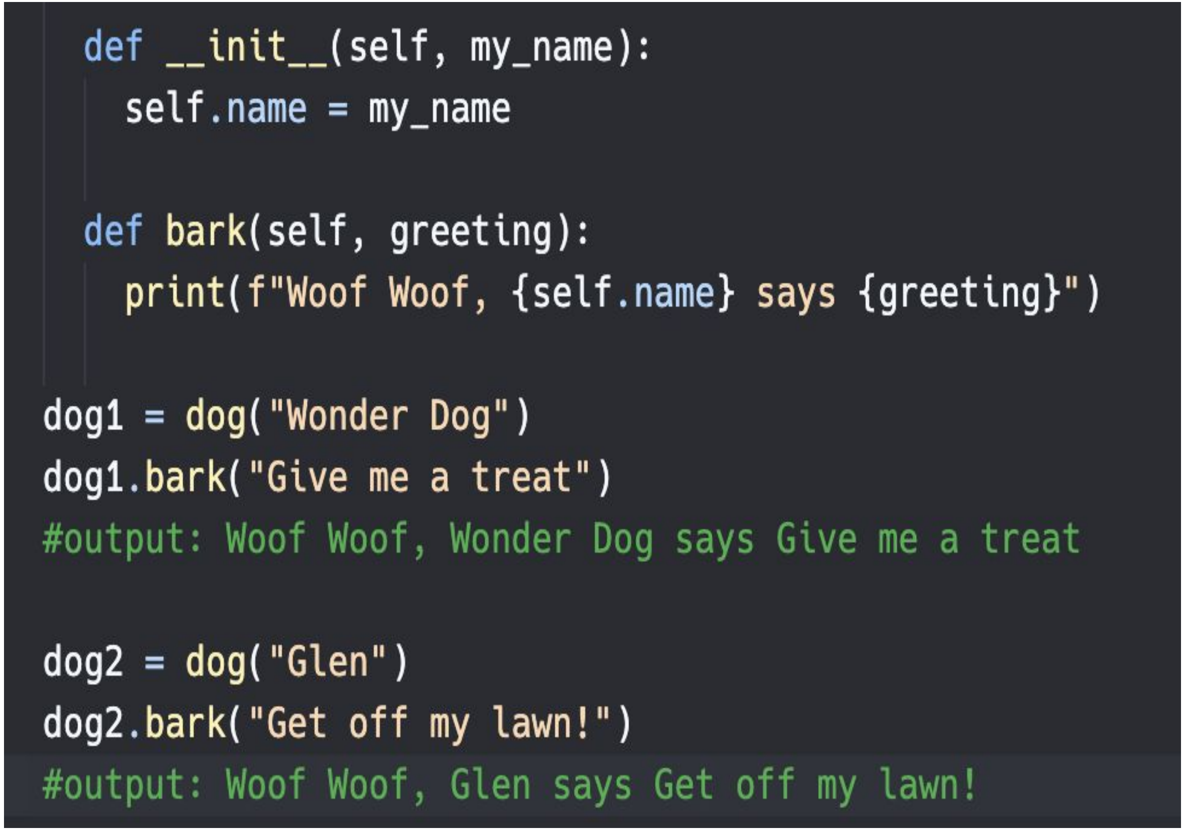
```
def __init__(self, genre, title, author, publication_year):
    self.genre = genre
    self.title = title
    self.author = author
    self.publication_year = publication_year
    Book.number_of_books = Book.number_of_books + 1
```

INSTANCE METHODS

- What can a Dog do? Woof Woof!

INSTANCE METHODS

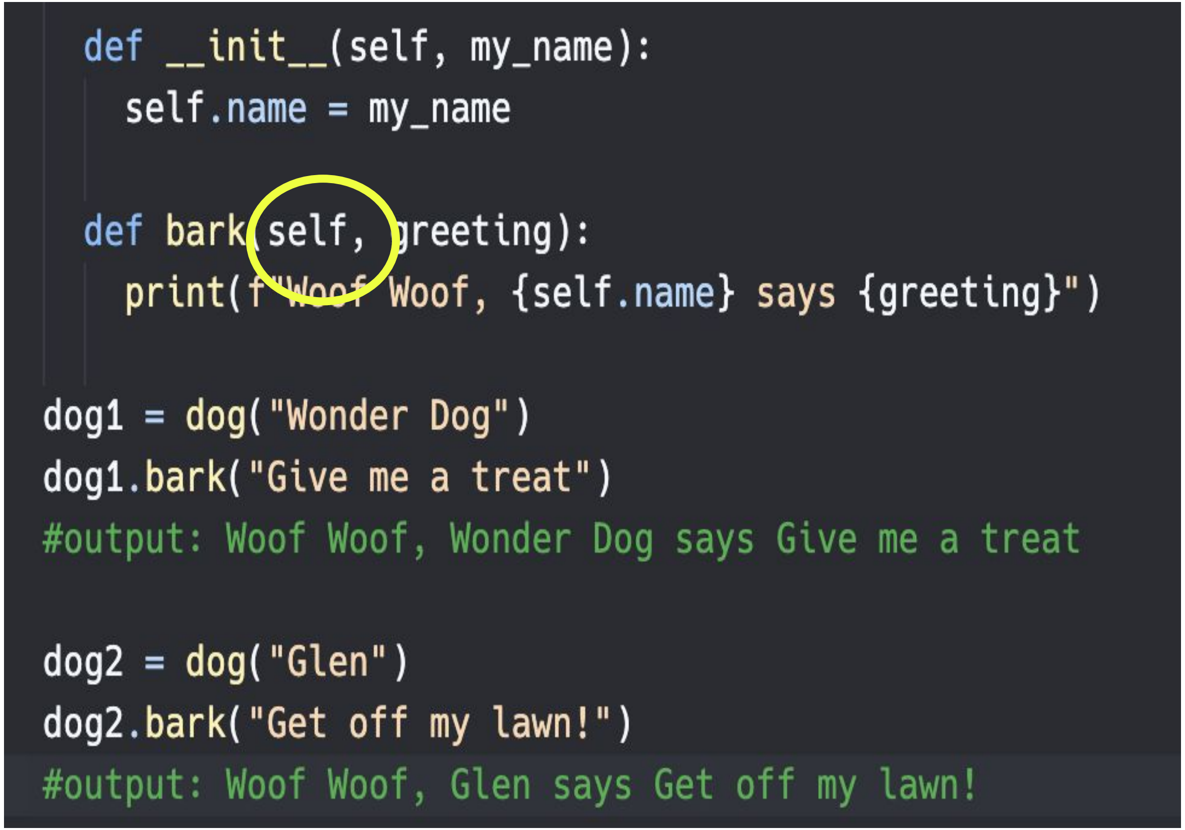
- Defined by a `def` statement in the suite of a `class` statement.



```
    def __init__(self, my_name):
        self.name = my_name

    def bark(self, greeting):
        print(f"Woof Woof, {self.name} says {greeting}")

dog1 = dog("Wonder Dog")
dog1.bark("Give me a treat")
#output: Woof Woof, Wonder Dog says Give me a treat

dog2 = dog("Glen")
dog2.bark("Get off my lawn!")
#output: Woof Woof, Glen says Get off my lawn!
```

INSTANCE METHODS

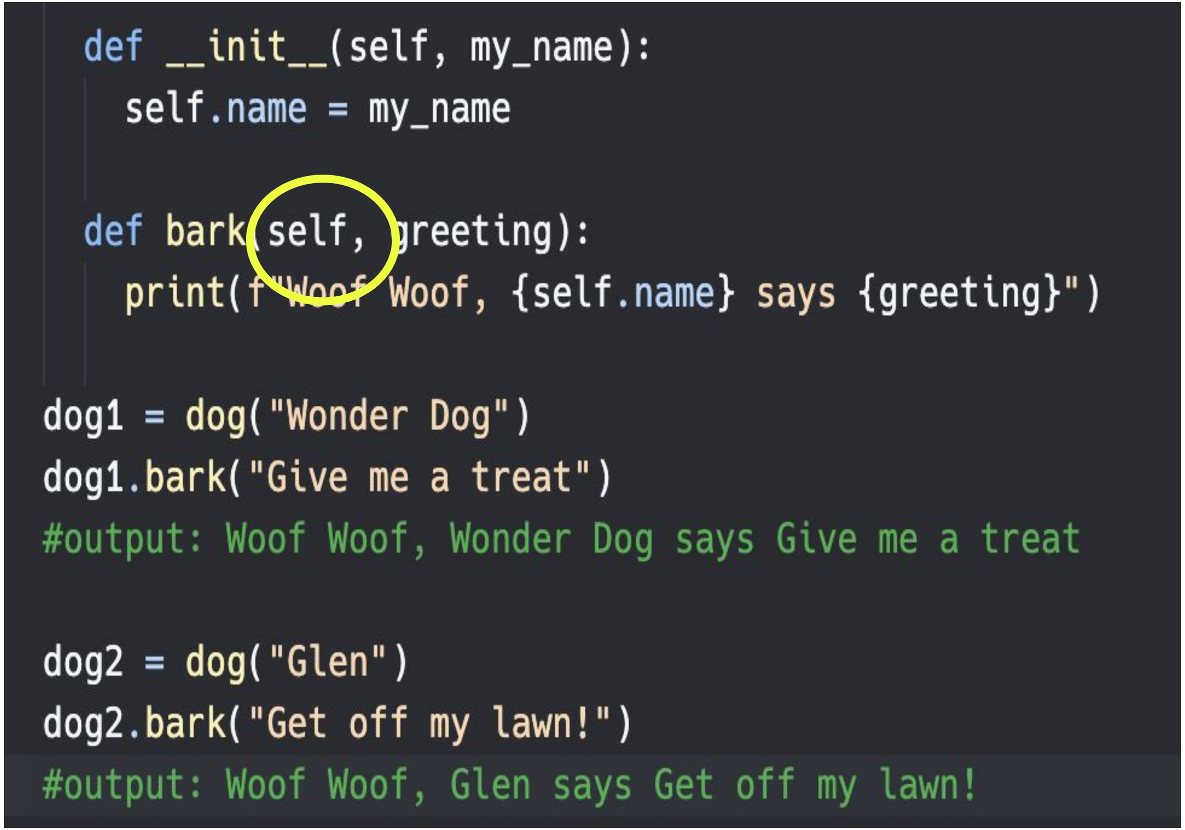
- Include a special first parameter `self`,
- implicitly bound to the object on which the method is invoked, thanks to dot notation.



```
    def __init__(self, my_name):
        self.name = my_name

    def bark(self, greeting):
        print(f"Woof Woof, {self.name} says {greeting}")

dog1 = dog("Wonder Dog")
dog1.bark("Give me a treat")
#output: Woof Woof, Wonder Dog says Give me a treat

dog2 = dog("Glen")
dog2.bark("Get off my lawn!")
#output: Woof Woof, Glen says Get off my lawn!
```

INSTANCE METHODS

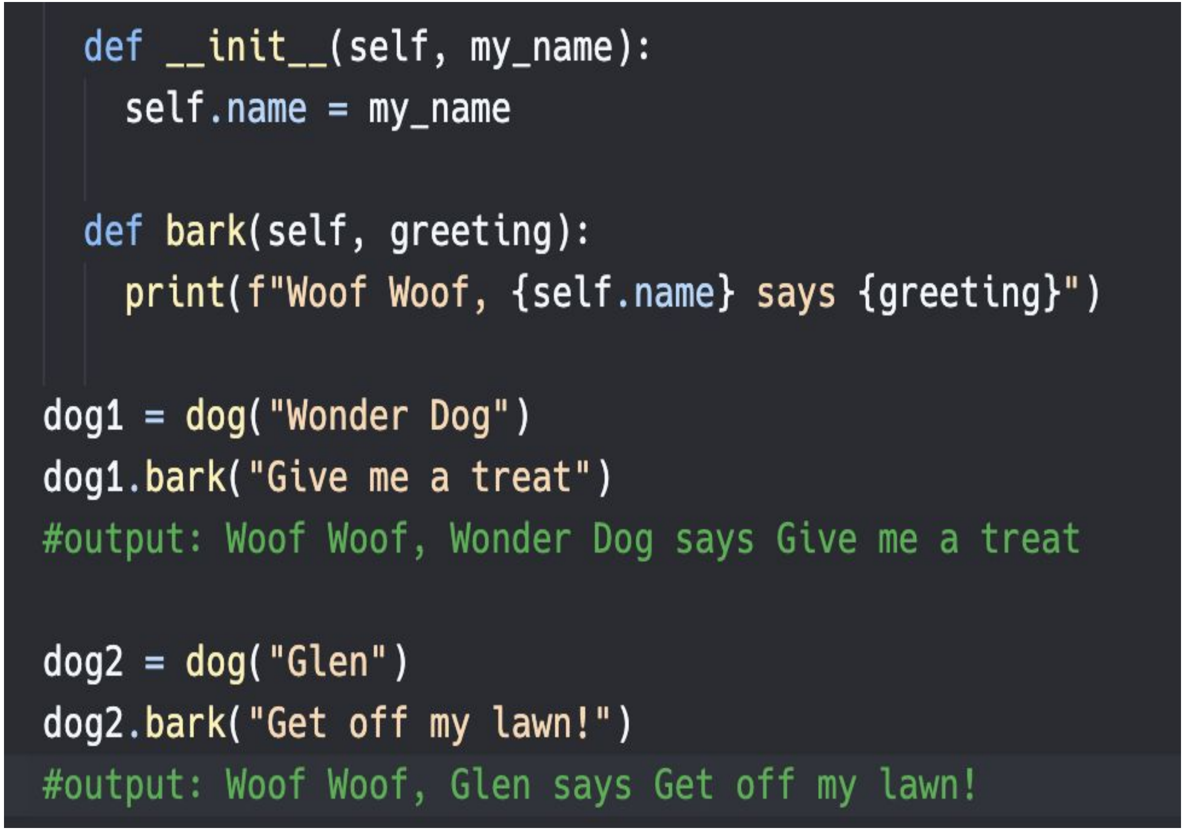
- In an instance method, we have access to the object’s attributes via the parameter `self`.



```
    def __init__(self, my_name):
        self.name = my_name

    def bark(self, greeting):
        print(f"Woof Woof, {self.name} says {greeting}")

dog1 = dog("Wonder Dog")
dog1.bark("Give me a treat")
#output: Woof Woof, Wonder Dog says Give me a treat

dog2 = dog("Glen")
dog2.bark("Get off my lawn!")
#output: Woof Woof, Glen says Get off my lawn!
```

INSTANCE METHODS

- Formatted strings / f-strings.



```
    def __init__(self, my_name):
        self.name = my_name

    def bark(self, greeting):
        print(f"Woof Woof, {self.name} says {greeting}")

dog1 = dog("Wonder Dog")
dog1.bark("Give me a treat")
#output: Woof Woof, Wonder Dog says Give me a treat

dog2 = dog("Glen")
dog2.bark("Get off my lawn!")
#output: Woof Woof, Glen says Get off my lawn!
```

Task 2: Calculate Age

- Create an Instance Method that will return that age of a Book object

Lab 16 Part I

- Repr method
- Takes attributes of Object, returns a string representation of the objects attributes in a string that, when passed to the `eval()` function, would recreate the object exactly