Computer Science

Summer Springboard

SUMMER SPRINGBOARD
Look Inward. Go Upward

Day 7: OOP

Pause: Discussion Time

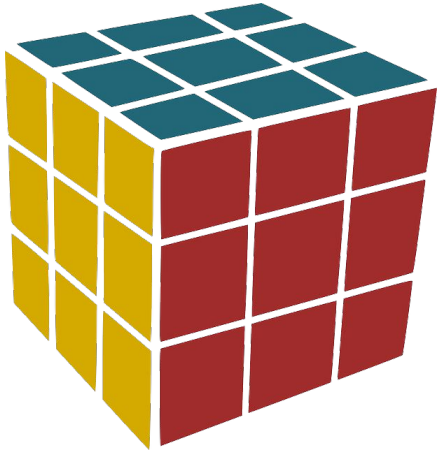
- Before getting into today's content, take some time to make sure you understand the following topics by discussing them in groups or as a **class** (no pun intended):
  - Variables and data types
  - Functions
  - Imports

Object-Oriented Programming

SUMMER
SPRINGBOARD
Look Inward. Go Upward.

Object-Oriented Programming

- Object-Oriented Programming (OOP) is a programming paradigm that uses “objects” to model real-world or abstract features, behaviors, and interactions
- OOP organizes software design around data and the methods that operate on these objects



Why OOP?

- OOP helps manage software complexity by mimicking how we perceive things in the real world
- Allows programmers to structure software in a way that is easier to understand, modify, and extend

Key Terminology

- **Class**: a blueprint or template from which objects are created. Classes define the attributes and methods necessary to create objects
- **Object**: an instance of a class. Objects have states (attributes / properties) and behaviors (methods / functions) defined by their class
- **Attribute**: a characteristic of an object. Attributes are data stored inside an object or class
- **Method**: a function defined inside a class that describes the behaviors of an object

Pillars of OOP

- There are a few important pillars of OOP that make the paradigm so powerful
- **Encapsulation**: bundling of data (attributes) and methods that operate on the data into a single unit (class) and controlling access to the properties of an object
- **Abstraction**: hiding the complex reality while exposing only the necessary parts; allows focusing on what an object does instead of how it does it

Pillars of OOP (Cont'd)

- These next two pillars will be covered tomorrow
- **Inheritance**: A mechanism for one class to inherit the attributes and methods of another class
- **Polymorphism**: The ability of different classes to respond to the same method call in different ways

Importance of OOP

- OOP is great for reusability; code can be reused through different mechanisms like inheritance, saving time and reducing errors
- OOP is also scalable; it is easier to maintain complex systems by building upon existing objects and classes
- Encapsulation provides a clear modular structure for programs, making OOP good for defining abstract datatypes where implementation details are hidden
- Abstraction and polymorphism increase the flexibility and efficiency of the code

Main Takeaway

- OOP is a powerful paradigm that structures software design around data, or objects, rather than functions and logic. By understanding and applying OOP principles, you can write more organized, efficient, and scalable code.

Creating a Class in Python

- Use the following code structure to create classes:

```
203  class ClassName:
204      def __init__(self, attribute1, attribute2):
205          self.attribute1 = attribute1
206          self.attribute2 = attribute2
207
208      def method1(self):
209          # action using self.attribute1
210          pass
```

__init__

- The **__init__** method is called automatically every time the class is being used to create a new object
  - Initializes the object's attributes with values
- You can have as many methods as you would like and they can take in whatever parameters you wish, but each method should have **self** as a parameter
  - **self** gives the method a way to access the attributes and other methods of the class instance
    - In other words, **self** is a way to refer to the object itself while you are doing things inside of the object

Creating an Object from a Class

- Once a class has been defined, you can create objects (**instance**s) of that class like so:

my_object = ClassName(“Value1”, “Value2”)

- See the next slide for a more concrete example

Creating an Object from a Class (Cont'd)

- Here is an example of a Dog class and an **instance** of that class named **my_dog**:

```
212  class Dog:
213      def __init__(self, name, age):
214          self.name = name
215          self.age = age
216
217      def bark(self):
218          return f"{self.name} says woof!"
219
220  my_dog = Dog("Doug", 4)
221  print(my_dog.bark())  # Output: Doug says woof!
```

Creating an Object from a Class (Cont'd)

- In this example, we have a class named **Dog**, meaning that we are allowed to make **Dog** type objects in our code
- We use this class by defining an instance of **Dog** which we called **my_dog**, which has name **Doug** and age **4**
- We then see an example of using a method by calling **Dog**'s **bark()** method, which prints a statement with the dog's **name** attribute

Using Object Attributes and Functions

- Attributes are defined within the class's **__init__** method, allowing for different instances of the same class to hold different values
- Use the following syntax: **object.attribute** to access the attributes
    - For example, if we have an instance of a **Car** class named **my_car** with attributes **make** and **model**, we can say **print(my_car.make)** to print the **make** of the **Car**.

Using Object Attributes and Functions (Cont'd)

- Methods, similar to attributes, are accessed using a period, but they are callable, meaning you can execute the method's code on the object
    - For example, if we have an instance of a **Dog** class named **my_dog** and there is a **bark()** method, we can call this method by using **my_dog.bark()**

Encapsulation

- **Encapsulation** is the bundling of data (**attributes**) and methods into a single unit called a class
- Involved restricting access to some of the object's components
- Protects the object's integrity by preventing - or, in Python's case, discouraging - external code from directly accessing its internal state

“Private” Members in Python

- Unlike some other languages, Python does not have a strict way to create **private** methods
- If you have an attribute or method that should only be used internally within a class, prepend a single leading underscore to the attribute / method name
- If you have an attribute or method that should not be used externally and also should not be used by subclasses, prepend two leading underscores to the attribute / method name
    - Python will use “name mangling” to make it difficult for subclasses to accidentally use the same name

Abstraction

- This concept is certainly a bit *abstract*, but it is very useful, as it reduces complexity and isolates the impact of changes made to the code
- **Abstraction** involves hiding the complex implementation details of a system and exposing only the necessary parts of it
- In Python, abstraction is done through abstract classes and methods (example on next page)
    - Abstract classes cannot be instantiated on their own; they require subclasses (covered tomorrow)

Abstraction (Cont'd)

- Below is an example of an abstraction (note use of import and @ statements)

```
223  from abc import ABC, abstractmethod
224
225  class Shape(ABC):
226      @abstractmethod
227      def area(self):
228          pass
229
230      @abstractmethod
231      def perimeter(self):
232          pass
```

- Note that it does not make sense to find the area or perimeter "Shape" on its own - in real life, you need a tangible shape to be able to find its area / perimeter; you can't just find the perimeter of "Shape."
  - However, we can grasp what a "Rectangle" or a "Circle" is and what their areas / perimeters are; "Rectangle" is a Shape and "Circle" is a Shape
    - More on this later with inheritance / subclasses

Predict the Output...

- Predict the output of the following code snippets

```
234  class CoffeeMachine:
235      def __init__(self, brand, model):
236          self.brand = brand
237          self.model = model
238          self.__water_level = 0
239
240      def add_water(self, amount):
241          self.__water_level += amount
242          if self.__water_level > 100:
243              self.__water_level = 100
244
245      def brew(self):
246          if self.__water_level >= 20:
247              self.__water_level -= 20
248              return "Coffee Ready!"
249          else:
250              return "Please add more water!"
251
252  my_coffee_machine = CoffeeMachine("JavaBeans", "Xpresso")
253  my_coffee_machine.add_water(50)
254  print(my_coffee_machine.brew())
```

Snippet 1: What will be printed when we run this snippet?

A) **Coffee Ready!**
B) **Please add more water!**
C) **AttributeError**
D) **None of the above**

SUMMER SPRINGBOARD
Look Inward. Go Upward

Predict the Output... (Cont'd)

```
256  class Rectangle:
257      def __init__(self, width, height):
258          self.width = width
259          self.height = height
260
261      def area(self):
262          return self.width * self.height
263
264  class Square:
265      def __init__(self, side_length):
266          self.side_length = side_length
267
268      def area(self):
269          return self.side_length * self.side_length
270
271  my_square = Square(4)
272  print(my_square.area())
```

Snippet 2: What will be printed when we run this snippet?

A) **8**
B) **16**
C) **4**
D) **12**

SUMMER SPRINGBOARD
Look Inward. Go Upward

Predict the Output... (Cont'd)

```
275  class BankAccount:
276      def __init__(self, initial_balance):
277          self.balance = initial_balance
278
279      def deposit(self, amount):
280          self.balance += amount
281
282      def withdraw(self, amount):
283          if amount <= self.balance:
284              self.balance -= amount
285          else:
286              print("Insufficient funds")
287
288  account = BankAccount(100)
289  account.deposit(50)
290  account.withdraw(120)
291  account.deposit(30)
292  print(account.balance)
```

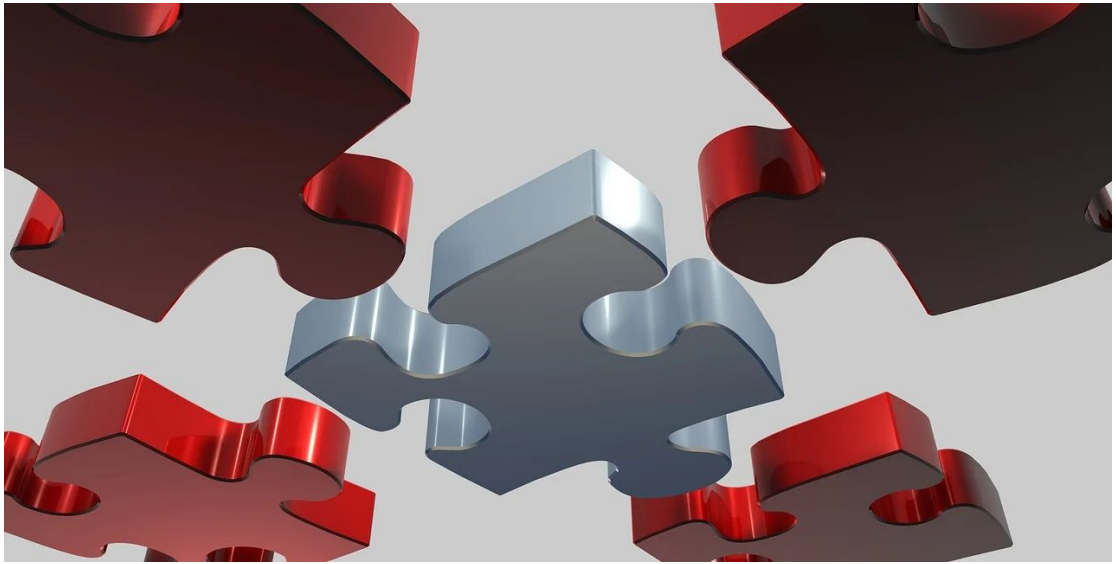
Snippet 3: What will be printed when we run this snippet?

A) **60**
B) **100**
C) **160**
D) **80**

SUMMER SPRINGBOARD
Look Inward. Go Upward

Practice Problems

- Now it’s time to work through today’s practice problems! Complete all of the problems in the Google Colab notebook.



SUMMER SPRINGBOARD
Look Inward. Go Upward

One Last Thing

- Before we move on, take a few minutes as partners or groups to discuss the following questions:
  - What makes an object unique in OOP?
  - How do constructors (**__init__** methods) contribute to the functionality of an object?
  - Discuss how creating objects from classes allows for more modular and maintainable code compared to solely using functions and global variables.

Final Project Development Time

SUMMER
SPRINGBOARD
Look Inward. Go Upward.

Final Project

- The final project may seem daunting (it is certainly a lot of work!), so let's take some time to work on it
- Use this time to make meaningful progress on your project and ask your professor any important questions you may have
- You could also use this time to try to implement objects into your project