Week 8 - Object-Orientated Programming

tuple - Used to store multiple item in a single variable. Similar to list, however they are immutable (cannot be changed after being created)

return x, y → TUPLE
return [x, y] → list

Classes - Blueprints / Templates for creating objects. Defines the properties (Attributes) and behaviors (methods) that the objects created from the class will have.

Objects - An instance of a class. A specific realization of the blueprint defined by the class.
example - If Car is a class, then a specific car like "Ford Mustang" is an object of the Car class.

Uses - Classes allow you to group data (attributes) and functions (methods) that operate on the data into a single unit. This makes code cleaner and easier to maintain

→ Encapsulation: Group data
→ Reusability: Can be reused to create multiple objects
→ Inheritance: Can inherit attributes and methods from other class reducing redundancy & promoting code reuse
→ Abstraction: Can hide complex logic behind simple interfaces; Making code easier to use and understand
→ Modularity: Helps organize code into logical sections. Easier readability and debugging
→ Scalability: Better code organization, reusability, & maintainability as project grows

#Real World Use Cases (next page)

Real-World Use Cases of Classes + Objects

↳ Game Development: Classes can represent Players, Enemies, or other entities

↳ Web Development: Classes can represent Users, Products, Orders, etc

↳ Simulations: Useful for modeling real world systems, like traffic, or physics simulations

↳ Data Management: Useful for structure + management of complex datasets

Getters + Setters - Methods used to access (get) and update (set) the values of an objects attributes (properties).

→ Uses: Used to enforce encapsulation, restricting direct access to an objects internal state.

Getter Example

```
# Getter for name
@property
def name(self):
    return self._name
```

* Prevent Function Collision
↳ _ for protected variables

Setter Example

```
# Setter for name_
@name.setter
def name(self, value):
    if len(value) > 2:
        self._name = value
    else:
        print("Name must be longer than 2 characters")
```

Use validation checking

Decorators

@property - Makes method behave like a regular attribute (getter)

@<property_name>.setter - Defines the setter for that property

Decorator

@Classmethod - Defines a method as a class method

↳ A class method - belongs to the class itself rather than an instance of the class. It takes the class (cls) as its first parameter instead of the instance (self). This allows it to operate on class attributes or call other class methods

Inheritance

Super() - Used to give access to methods and properties of a parent class. It allows you to call a method from the parent class inside a subclass, enabling you to extend or modify the behavior of inherited methods.

example.

```
class Parent:
    def greet(self):
        print("Hello from Parent")

Class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from Child!")

obj = Child()
obj.greet()
```

Output:

Hello from Parent

Hello from Child!

Override Operator

__add__ - Used to define the behavior of the + operator for user defined objects. By overiding __add__ in a class, you can customize how to instances of that class are added together

example.

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + other.value

num1 = Number(10)
num2 = Number(20)
print(num1 + num2) #Output: 30
```

More Advanced Example: Adding 2 Points

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
    def __repr__(self):
        return f"Point({self.x}, {self.y})"
p1 = Point(2, 3)
p2 = Point(4, 5)
result = p1 + p2
print(result)
```

#Output: Point(6, 8)