**Download Your Daily Notes**

**Daily Notes - Classes**

 Classes in Python provide a way to bundle data and functionality together. Each class instance can have attributes and methods for maintaining and modifying its state. Python's class mechanism is a combination of features found in C++ and Modula-3, offering inheritance, method overriding, and dynamic behavior. Class members are typically public, and member functions are virtual. Classes themselves are objects, allowing for importing and renaming. Built-in types can serve as base classes, and operators can be redefined for class instances. It's important to understand that objects can have multiple names bound to them, which can affect the semantics, especially with mutable objects. However, this behavior is advantageous as it allows efficient passing of objects and enables modifications to be visible to the caller.

**Daily Notes - Python Scopes and Namespaces**

 Before diving into classes, it's important to understand Python's scope rules. Namespaces play a crucial role in class definitions and overall Python programming. A namespace is a mapping from names to objects, such as built-in names, global names in a module, local names in a function, and attributes of an object. Namespaces are created at different times and have different lifetimes.  
  
Scopes define the accessibility of namespaces in a Python program. There are three nested scopes accessible during execution: the local scope (innermost), enclosing function scopes, and the module's global scope. The outermost scope contains built-in names. By using the global statement, variables can be explicitly declared as part of the module's global scope, while the nonlocal statement indicates variables in an enclosing scope.  
  
Scopes are determined textually, meaning the global scope of a function is always the module's namespace, regardless of where or how the function is called. However, name resolution occurs dynamically at runtime, although the language is evolving towards static name resolution.  
  
Assignments and deletions of names are done in the innermost scope, unless the global or nonlocal statement is used to explicitly indicate the scope. Operations that introduce new names, like import statements and function definitions, bind the names in the local scope.  
  
An example is provided to demonstrate how different scopes and namespaces are referenced, along with the effects of global and nonlocal statements on variable binding.  
  
Understanding scopes and namespaces is crucial for advanced Python programming and provides insights into the behavior of classes and other programming constructs.

**Daily Notes - More on Classes**

 Classes in Python introduce new syntax, three object types (class objects, instance objects, and method objects), and new semantics. The simplest form of a class definition consists of statements enclosed within a class block. Class definitions, like function definitions, must be executed before they have any effect.  
  
Inside a class definition, function definitions are common, but other statements are also allowed. When a class definition is entered, a new namespace is created, and all assignments to local variables are made in this namespace. After the class definition is executed, a class object is created, which acts as a wrapper around the namespace contents. The class object is then bound to the class name.  
  
Class objects support attribute references and instantiation. Attribute references follow the standard syntax of obj.name, where valid attribute names are those present in the class's namespace at the time of class object creation. Class attributes can be assigned and accessed. Class instantiation is performed using function notation, where the class object behaves as a parameterless function that returns a new instance of the class.  
  
Instance objects can only perform attribute references. They have data attributes (also called instance variables) that come into existence when first assigned. Methods are the other type of instance attribute, and they are functions that belong to an object. Method names depend on the class and are defined by corresponding function attributes of the class.  
  
Method objects can be called immediately or stored for later use. When a method is called, the instance object is passed as the first argument (usually named self) automatically. The instance object is inserted before the other arguments when calling the method.  
  
Understanding classes enables the creation of objects with customized initial states and the utilization of attribute references, data attributes, and methods specific to instances of a class.

**Daily Notes - Class and Instance Variables**

 In object-oriented programming, instance variables and class variables serve different purposes. Instance variables hold data unique to each instance of a class, while class variables store attributes and methods shared by all instances of the class.  
  
In the given example, the class "Dog" has a class variable called "kind" which is shared by all instances of the class. Each instance also has an instance variable called "name" which stores a unique value for each instance.  
  
When creating instances of the class and accessing their variables, it is evident that "kind" is shared among all dogs, while "name" is specific to each dog.  
  
The example further highlights the potential issues when using mutable objects like lists as class variables. In the initial implementation, the "tricks" list is defined as a class variable, causing unexpected behavior. When adding tricks to different instances, all dogs end up sharing the same list, leading to undesired results.  
  
To address this, the correct design approach is to use an instance variable instead. By initializing "tricks" as an empty list within the instance's constructor, each dog will have its own separate list for storing tricks.  
  
Overall, understanding the distinction between instance variables and class variables is crucial for proper class design and avoiding unintended consequences when working with shared data.

**Daily Notes - Random Remarks**

 Data attributes take precedence over method attributes with the same name. To avoid conflicts and potential bugs in large programs, it is advisable to follow naming conventions that minimize the chances of conflicts, such as capitalizing method names or prefixing data attribute names.  
  
Data attributes can be accessed by both methods and clients (users) of an object. Python does not enforce data hiding, and it relies on conventions for achieving it.  
  
Clients should use data attributes carefully to avoid interfering with the methods' invariants. Clients can add their own data attributes to an instance without affecting the validity of the methods, as long as naming conflicts are avoided.  
  
There is no shorthand for referencing data attributes or methods from within methods. This lack of shorthand actually enhances method readability by preventing confusion between local variables and instance variables.  
  
It is a convention to use "self" as the first argument of a method. While "self" has no special meaning in Python, following this convention improves code readability for other Python programmers.  
  
Any function object that is a class attribute becomes a method for instances of that class. The function definition can be enclosed in the class or assigned to a local variable within the class.  
  
Methods can call other methods using the "self" argument's method attributes.  
  
Methods can reference global names just like ordinary functions. The global scope associated with a method is the module containing its definition.  
  
Each value in Python is an object and has a class (or type) stored as "object.class".

**My own views on classes**

 Classes are a fundamental concept in object-oriented programming (OOP) that allow you to define your own data types. They serve as blueprints or templates for creating objects.  
  
An object is an instance of a class. It is a specific realization of the class, with its own unique set of data and behavior.  
  
Classes encapsulate data (in the form of attributes) and behavior (in the form of methods) that define the characteristics and actions of objects.  
  
Attributes are variables that store data associated with a class or an instance. They represent the state of an object. Class attributes are shared among all instances of a class, while instance attributes are unique to each instance.  
  
Methods are functions defined within a class that operate on objects created from the class. They define the behavior and actions that objects can perform. Methods have access to the attributes and other methods of the class.  
  
To create an instance of a class, you use the class name followed by parentheses, optionally passing any required arguments to the class's initializer method (often called \_\_init\_\_()).  
  
The initializer method (\_\_init\_\_()) is a special method in Python classes that is automatically called when creating a new instance. It is used to initialize the attributes of the instance.  
  
Instances can access attributes and invoke methods defined in the class using dot notation (instance\_name.attribute\_name or instance\_name.method\_name()).  
  
Classes can have class methods and static methods. Class methods are bound to the class and can access class attributes, while static methods are independent of the class and do not have access to instance or class attributes.  
  
Inheritance allows you to create a new class (derived or child class) based on an existing class (base or parent class), inheriting its attributes and methods. This promotes code reuse and supports the concept of hierarchical relationships between classes.  
  
Polymorphism allows objects of different classes to be treated as objects of a common base class. This allows for more flexible and generic coding, as different objects can respond differently to the same method invocation.  
  
Python provides several built-in classes and modules, such as lists, dictionaries, strings, and math, which can be used directly or as bases for creating custom classes.  
  
Overall, classes and instances form the core of object-oriented programming in Python, enabling the creation of reusable, modular, and organized code structures. They allow for the modeling of real-world entities, abstraction of data and behavior, and support key OOP principles such as encapsulation, inheritance, and polymorphism.

**Daily Notes - Python Classes**

 Classes are a fundamental concept in object-oriented programming (OOP) that allow you to define your own data types. They serve as blueprints or templates for creating objects.  
  
An object is an instance of a class. It is a specific realization of the class, with its own unique set of data and behavior.  
  
Classes encapsulate data (in the form of attributes) and behavior (in the form of methods) that define the characteristics and actions of objects.  
  
Attributes are variables that store data associated with a class or an instance. They represent the state of an object. Class attributes are shared among all instances of a class, while instance attributes are unique to each instance.  
  
Methods are functions defined within a class that operate on objects created from the class. They define the behavior and actions that objects can perform. Methods have access to the attributes and other methods of the class.  
  
To create an instance of a class, you use the class name followed by parentheses, optionally passing any required arguments to the class's initializer method (often called \_\_init\_\_()).  
  
The initializer method (\_\_init\_\_()) is a special method in Python classes that is automatically called when creating a new instance. It is used to initialize the attributes of the instance.  
  
Instances can access attributes and invoke methods defined in the class using dot notation (instance\_name.attribute\_name or instance\_name.method\_name()).  
  
Classes can have class methods and static methods. Class methods are bound to the class and can access class attributes, while static methods are independent of the class and do not have access to instance or class attributes.  
  
Inheritance allows you to create a new class (derived or child class) based on an existing class (base or parent class), inheriting its attributes and methods. This promotes code reuse and supports the concept of hierarchical relationships between classes.  
  
Polymorphism allows objects of different classes to be treated as objects of a common base class. This allows for more flexible and generic coding, as different objects can respond differently to the same method invocation.  
  
Python provides several built-in classes and modules, such as lists, dictionaries, strings, and math, which can be used directly or as bases for creating custom classes.  
  
Overall, classes and instances form the core of object-oriented programming in Python, enabling the creation of reusable, modular, and organized code structures. They allow for the modeling of real-world entities, abstraction of data and behavior, and support key OOP principles such as encapsulation, inheritance, and polymorphism.

**Daily Notes - Activity 1 - Classes**

 1. Classes in Python are a fundamental concept of object-oriented programming (OOP), allowing for the creation of reusable and modular code structures. Here's a more detailed exploration of classes in Python:  
  
Class Definition:  
  
A class is defined using the class keyword followed by the class name, typically using CamelCase naming convention.  
The class body is indented, and it can contain attributes and methods.  
Class Attributes:  
  
Class attributes are defined within the class body but outside any methods.  
They are shared among all instances of the class.  
Class attributes can be accessed using the class name or any instance of the class.  
Instance Attributes:  
  
Instance attributes are specific to each instance of a class.  
They are defined within the class's \_\_init\_\_() method, which is called when creating a new instance.  
Instance attributes are typically initialized with values specific to each instance.  
Methods:  
  
Methods are functions defined within a class and operate on instances of that class.  
They are defined using the def keyword and have the first parameter conventionally named self, which refers to the instance on which the method is called.  
Methods can access instance attributes and other methods using self.attribute\_name or self.method\_name().  
Constructor (Initializer):  
  
The \_\_init\_\_() method is a special method called the constructor or initializer.  
It is automatically called when creating a new instance of the class.  
The constructor is used to initialize the instance attributes with specific values.  
Inheritance:  
  
Inheritance allows the creation of a new class (derived class) based on an existing class (base class).  
The derived class inherits attributes and methods from the base class and can override or extend them.  
In Python, a class can inherit from one or more base classes, using parentheses after the class name.  
Encapsulation:  
  
Encapsulation refers to the bundling of data (attributes) and methods within a class.  
It provides a way to control access to the attributes and methods, ensuring data integrity and hiding implementation details.  
Python uses naming conventions (such as prefixing with an underscore) to indicate private attributes or methods.  
Polymorphism:  
  
Polymorphism allows objects of different classes to be treated as objects of a common base class.  
It enables the use of a common interface for objects that may have different implementations.  
Polymorphism is achieved through method overriding and method overloading.  
Class Variables and Instance Variables:  
  
Class variables are shared among all instances of a class and are defined outside any methods.  
Instance variables are unique to each instance of a class and are defined within the \_\_init\_\_() method.  
Classes in Python provide a powerful way to organize code, promote reusability, and implement complex systems. They facilitate the creation of objects that can interact with each other and model real-world entities. By utilizing classes, you can write cleaner, more maintainable, and modular code.  
  
2. Here's an example code that demonstrates the creation of a Person class in Python:  
  
python  
Copy code  
class Person:  
def \_\_init\_\_(self, name, age):  
self.name = name  
self.age = age  
  
def say\_hello(self):  
print(f"Hello, my name is {self.name} and I am {self.age} years old.")  
  
# Creating instances of the Person class  
person1 = Person("Alice", 25)  
person2 = Person("Bob", 30)  
  
# Accessing attributes and calling methods of the instances  
print(person1.name) # Output: Alice  
print(person2.age) # Output: 30  
person1.say\_hello() # Output: Hello, my name is Alice and I am 25 years old.  
person2.say\_hello() # Output: Hello, my name is Bob and I am 30 years old.  
In this example, the Person class represents a person with a name and an age. The \_\_init\_\_() method is the constructor that initializes the name and age attributes of each instance. The say\_hello() method prints a greeting message including the person's name and age.  
  
We then create two instances of the Person class (person1 and person2) with different names and ages. We can access the attributes (name and age) of each instance using dot notation (instance.attribute). Additionally, we can call the say\_hello() method on each instance to print a personalized greeting.  
  
This example showcases the basic structure of a class, the usage of the constructor, instance attributes, and instance methods.