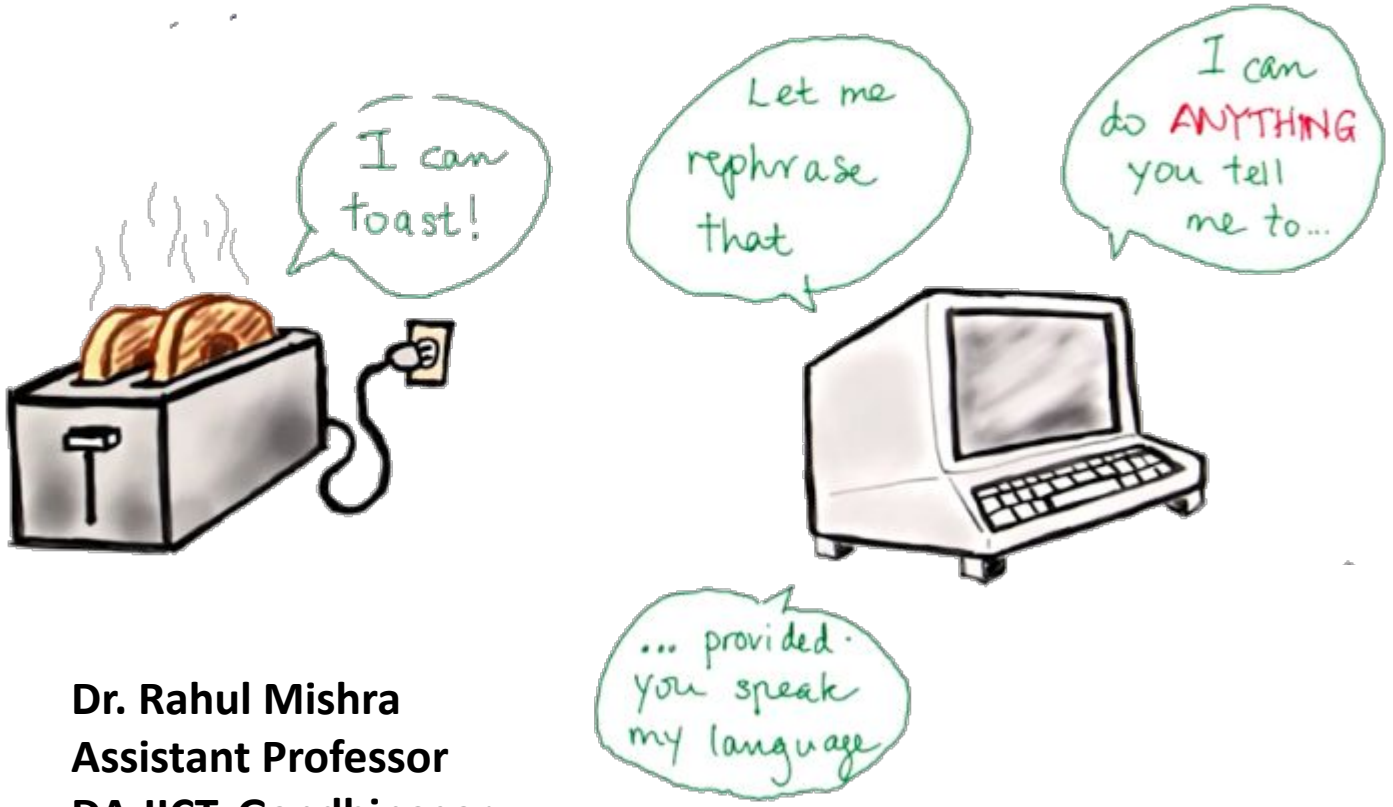
Programming Lab

Autumn Semester

Course code: PC503



Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 19

Classes

Python Scopes and Namespaces

- A **scope** is a textual region of a Python program where a namespace is directly accessible. "**Directly accessible**" here means that an unqualified reference to a name attempts to find the name in the namespace.

- Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- *the **innermost scope**, which is searched first, contains the local names*

- *the scopes of any enclosing functions, which are searched starting with the nearest **enclosing scope**, contain non-local, but also non-global names*

- *the **next-to-last scope** contains the current module's global names*

- *the **outermost scope** (searched last) is the namespace containing built-in names*

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Classes

- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a *new type of object*, allowing new *instances* of that type to be made.
- Each class instance can have *attributes attached to it to maintain its state*.
- Class instances can also have methods (defined by their class) for modifying their state.
- Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3.

Classes

- Python classes provide all the standard features of **Object Oriented Programming**: the class inheritance mechanism allows *multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name*.

- Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime and can be modified further after creation.

- In C++ terminology, normally class members (including the data members) are public (except see below Private Variables), and all member functions are virtual.

- Unlike C++, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting, etc.) can be redefined for class instances.

Classes :-

* Object-oriented programming is one of the most effective approaches to write software.
* In oops we can write classes that represent real-world things and situations, and we can create objects based on these classes.
* When we write a class, we define the general behavior that a whole category of objects can have.
* Notably, when we create individual objects from the class, each object is automatically equipped with the general behavior.
* Then we can give each object whatever unique traits we desire.
* Making an object from a class is called instantiation and we work with instances of a class.

Preface:

> Write classes and create instances
> Specify the kind of information that can be stored in instances.
> Define actions that can be taken with these instances.
> classes that extend the functionality of existing classes (Inheritance)

✡ Creating and Using a Class :—

Let us start with a simple class, Student, that represent some features of student.

↳ What do we know about most student :— name, age, class of study (cos)

↳ We also know two common behavior :— ① sitting in lecture hall / lab
② wondering outside

* This class student will tell Python how to make one object representing a student.
* After our class is written, we will use it to make individual instances, each of which represents one specific student.

student.py

<class Dog>

```
class Student:
    """A simple attempt to model a student"""

    def __init__(self, name, age, cos):
        """Initialize name and age and other attributes"""
        self.name = name
        self.age = age
        self.cos = cos

    def sitting(self):
        """Simulate a student sitting in respond to the command"""
        print(f"{self.name} is now sitting in lab/lecture.")
```

```
def roll(self):
    """Simulate rolling over in response to a command"""
    print(f"{self.name} rolling over!")
```

---

① The __init__() Method

* A function that's part of class is a method.
* Everything we learned about functions applies to method as well; the only practical difference from now is the way we will call methods
* The ___init()___ method is a special method that Python runs automatically whenever we create a new instance based on the Student class.
* Two leading underscores and two trailing underscores, a convention that helps to prevent Python's default method names from conflicting with your method names.
* __init() or init__() results in error. (that may be difficult to identify).

* We define the __init()__ method to have ~~three~~ four parameter, self, name, age, and cos.

* The self parameter is required in the method definition, and it must come first before other parameters.

* It must be included in the definition because when Python call this method later (to create instance of student), this method call will automatically passes the self argument.

* Every method call associated with an instance automatically pass the self, which is a reference to the instance itself; it gives the individual instances access to the attributes and methods in this class.

Imp:- • When we make an instance according to student, Python will call the __init()__ method in the class.

• We will pass student() to name, an age, and cos (↗ It is always applicable) as arguments; self is passed automatically, so we don't need to pass it.

* The three variables defined in the init method have the prefix self.
* Any variable prefixed with self is available to every method in the class and we be also be able to access these variables through any instance created from the class.

* For example, the line self.name = name takes the value associated with the parameter name and assigns it to the variable name, which is then attached to the instance being created.

* The Student class has two other method defined: sit() and roll()
  - These methods do not need additional information to run, we just define them to have one, we just define them to have one parameter, self.
  - The instance we create later will have access to these methods.
  - At these point they only perform simple operation.
  - Example:- If this class was written to control a robot, these methods would direct movements that cause a robotic dog to sit and roll over.

Making an Instance from a class :-

→ s1 = Student ('Rohit', 29, 'PhD')
print(f "My student's name is {s1.name}.")
print(f "My student's is {s1.age} years old")
print(f "My student's is studying {s1.cos}")

↳ The init() method creates an instance representing this particular student and set name, age, and cos using the value provided.

* **Accessing Attributes:**

s1.name ⟶ print(s1.name) => Rohit
↳ Dot notation is used in Python

=> This is the same attribute referred to as self.name in the class student.

Calling method :

```
S1.sit()
S1.roll()
```

- When Python reads S1.sit(), it looks for the method sit() in the class Student and runs that code.
- Python interprets the line S1.roll() in the same way.

Output :- Rohit is now sitting in lab/lecture
Rohit is rolling over

* This syntax is quite useful when attributes and methods have been given appropriately descriptive names like name, age, cos, sit(), and roll()

Creating Multiple Instances :-

Let us create a second student called S2

```
Class Student:
    :
    :
    .

S1 = Student ('Rohit', 29, 'PhD')
S2 = Student ('Mayank', 25, 'MTech')
```

```
print(f"My student's name is {S1.name}.")
print(f"My student's age is {S1.age}")
S1.sit()
print(f"Your student's name is {S2.name}")
print(f"Your student's age is {S2.age}")
S2.sit()
S2.roll()
```

Imp: Even if we need/use the same name and age of the second student, Python would still create a separate instance from the student.

We can make any number of instance for a class.

Output :->

```
My student's name is Rohit
My student's age is 29
Rohit is now sitting in lab/lecture
Your student's name is Mayank
Your student's age is 25
Mayank is now sitting in lab/lecture
Mayank is rolling over
```

* Assignment → Try it yourself

① Restaurant :- Make a class called Restaurant. The __init__() method for Restaurant should store two attributes : a restaurant_name and a cuisine_type.

Make a method called describe_restaurant() that prints these two pieces of information, and a method called open_restaurant() that prints a message indicating the restaurant is open

↳ make instance of the class and print all

② For above consider three restaurants

③ Create user ~~all~~ class with your own methods and variable. Perform instance creation and Calling.

* Working with Classes and Instances

The Car Class :-

→ car.py

```
class Car:
    """A simple attempt to represent a car"""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
```

⇒ `my_new_car = Car('Audi', 'a6', 2023)`
`print(my_new_car.get_descriptive_name())` ⟶ 2023 Audi a6

Setting a Default Value for an Attribute

> When an instance is created, attributes can be defined without being passed in as parameters.
> These attributes can be defined in the `__init()__` method, where they are assigned a default value.
> Let's add an attribute called `odometer_reading` that always starts with a value of 0.
> We will also add a method `read_odometer()` that helps us read each car's odometer:

```
class Car:

    def __init()__(self, make, model, year):
        """Initialize attributes to describe a car"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
```

```
def read_odometer(self):
    """Print a statement showing the car's mileage"""
    print(f"This car has {self.odometer_reading} miles on it")

my_new_car = Car('Audi', 'A6', 2023)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

* Since our car starts with a mileage of 0

```
2023 Audi A6
This car has 0 miles on it
```

→ Note many cars are sold with exactly 0 miles on the document (odometer), so we need a way to change the value of this attribute.

Modifying Attribute Value:

We can change an attribute's value in three ways:

(I) We can change the value directly through an instance
(II) Set the value through a method
(III) Increment the value (add a certain amount to it) through a method.

(I) → Modifying an Attribute's Value Directly

```
Class Car:
    --snip--

my_new_car = Car('Audi', 'A6', 2023)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

update the value

```
my_new_car.odometer_reading = 23
```

⇒ 2023 Audi A6
⇒ This car has 23 miles on it.

2. Modifying an Attribute's Value Through a Method

Instead of accessing the attribute directly, we pass the new value to a method that handles the updating internally.

```
class Car:
    def update_odometer(self, mileage):
        """ set the odometer reading to the given value."""
        self.odometer_reading = mileage
```

```
my_new_car = Car('Audi', 'A6', 2023)
print(my_new_car.get_descriptive_name())
my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

```
2023 Audi A6
This car has 23 miles on it.
```

> We can extend the method update_odometer() to do additional work every time the odometer reading is modified.

> Let's add a little logic to make sure no one tries to roll back the odometer reading:

```
class Car:
    --snip--
    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
```

* Now, update_odometer() checks that the new reading makes sense before modifying the attribute.

3. Incrementing an Attribute's Value Through a Method

→ Sometimes we will want to increment an attribute's value by a certain amount rather than set an entirely new value.

→ Suppose, we buy a used car and put 100 miles on it between the time we buy it and the time we register it.

```
class Car:
    --snip--

    def update_odometer(self, mileage)
        --snip--

    def increment_odometer(self, miles)
        """Add the given amount to the odometer reading"""
        self.odometer_reading += miles
```

⇒
```
my_used_car = Car('Than', 'zxi', 2022)
print(my_used_car.get_descriptive_name())
```

```
my_used_car.update_odometer(23500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

Output:-

```
2022 Tata ZXi
This car has 23500 miles on it.
This car has 23600 miles on it.
```

You can easily modify this method to reject negative increments so no one uses this function to roll back an odometer.

Try it yourself:-

Add new method in restaurant class

↳ Number served.

Inheritance :

→ We don't always have to start from scratch when writing a class.

→ If the class you are writing is a specialized version of another class you wrote
→ you can use inheritance.

→ When one class inherits from another, it takes on the attributes and methods of the first class.

→ The original class is called the parent class, and the new class is the child class.

The __init__() Method for a Child Class

When we are writing a new class based on an existing class, we will often want to call the __init__() method from the parent class.

electric_car.py

Main / Parent class / Primary class

Indent ← class Car:

```
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
```

```
    def increment_odometer(self, miles):
        self.odometer_reading += miles

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles"""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
        super().__init__(make, model, year)

my_tesla = ElectricCar('tesla', 'models', 2023)
print(my_tesla.get_descriptive_name())
```

Output → 2023 Tesla Models

Note: ① When you create a child class, the parent class must be part of the current file and must appear before the child class in the file.

Note: (2) We define the child class, ElectricCar. The name of the parent class must be included in parenthesis in the definition of the child class.

(3) The __init__ method takes in the information required to make a car instance.

Important:- The `super()` function is a special function that allows you to call a method from the parent class.

- It tells Python to call the `__init__()` method from Car, which gives our ElectricCar instance all the attributes defined in that method.

→ The name super comes from a convention of calling the parent class a superclass and the child class a subclass.

→ Aside from `__init__()`, there are no attributes or methods yet that are particular to our electric car.

Defining Attributes and Methods for the child class :-

```
class Car:
    --snip--
class ElectricCar(Car):
    "Represent aspects of a car, specific to electric vehicles"

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car"""
        super().__init__(make, model, year)
        self.battery_size = 200
    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-KWh battery.")
my_tesla = ElectricCar('tesla', 'model s', 2023)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

⇒ Output:
2023 Tesla Model S
This car has a 75-KWh battery

» There's no limit to how much you can specialize the ElectricCar class.

» You can add as many attributes and methods as you need to model an electric car to whatever degree of accuracy you need.

Overriding Methods from the Parent class :-

» We can (override) any method from the parent class that doesn't fit what you are trying to model with the child class.

» We define a method in the child class with the same name as the method we want to override in the parent class.

» Python will disregard the parent class method and only pay attention to the method you define in the child class.

Let the class Car had a method called fill_gas_tank().

This method is meaningless for an all-electric vehicle, so we might want to override this method.

```
class ElectricCar(Car):
    --snip--
    def fill_gas_tank(self):
        """Electric cars don't have gas tanks."""
        print("This car doesn't need a gas tank!")
```

⇒ If someone tries to call fill_gas_tank() with an electric car, Python will ignore the method fill_gas_tank() in Car and run this code instead.

Instances as Attributes :-

* We have a growing list of attributes and methods and that your files are becoming lengthy.
* In these situations, you might recognize that part of one class can be written as a seperate class.
* We can break our large class into smaller classes that work together.

```
class Car:
    --Snip--
class Battery:
    """A Simple attempt to model a battery for an electric car"""
    def __init__(self, battery_size=75):
        """Initialize the battery's attributes"""
        self.battery_size = battery_size
    def describe_battery(self):
        """Print a statement describing the battery size"""
        print(f"This car has a {self.battery_size}-kWh battery")
```

```
class ElectricCar(Car)
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()

my_tesla = ElectricCar('tesla', 'models', 2023)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

output :-
```
2023 Tesla Models
This car has a 75-kWh battery.
```

Let's add another method to Battery that reports the range of the car based on the battery size:

```
class Car:
    --snip--
class Battery:
    --snip--

    def get_range(self):
        """Print a statement about the range this battery provides"""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")
class ElectricCar(Car):
    --snip--
```

```
my_tesla = ElectricCar('tesla', 'model s', 2023)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

Output
```
2023 Tesla Model S
This car has a 75-KWh battery
This car can go about 260 miles on a fully charge.
```

Try it Yourself →

i) Ice Cream stand

ii) Admin

iii) Privileges

iv) Battery Upgrade

Importing a Single class :

Similar as importing modules as discussed previously.

car.py →

```
class Car:
    "A simple attempt to represent a car"
    def __init__(self, make, model, year):
        --snip--
    def get_descriptive_name(self):
        --snip--
        return long_name.title()
    def read_odometer(self):
        --snip--
    def update_odometer(self, mileage):
        --snip--
    def increment_odometer(self, miles):
        --snip--
```

my-car.py

```
from car import Car
my_new_car = Car('Audi', 'A6', 2023)
print(my_new_car.get_descriptive_name())
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

⇒ 2023 Audi A6.
This car has 23 miles on it.

Storing Multiple Classes in a Module:

```
class Car:
    --snip--
class Battery:
    --snip--
class ElectricCar(Car):
    --snip--
```

> Importing Multiple classes from a Module :-

```
from car import Car, ElectricCar
```

> Importing an Entire Module :-

```
import car
my_beetle = car.Car('volkswagen', 'beetle', 2020)
print(my_beetle.get_descriptive_name())
my_tesla = car.ElectricCar('tesla', 'roadster', 2021)
print(my_tesla.get_descriptive_name())
```

> Importing All classes from a Module :-

```
from module_name import *
```

Using Aliases

```
from electric_car import ElectricCar as EC
```

↳ `my_tesla = EC('tesla', 'roadster', 2023)`

Try it yourself :-

> Dice

> Lottery

> Lottery Analysis.