Python Classes

Panchatcharam Mariappan

Associate Professor

Department of Mathematics and Statistics,
IIT Tirupati

OOPS CONCEPTS

- ✓ **Programming Paradigms**
  - Procedural
    - Modules, Data Structures, Procedures that Operate upon them
  - Objectural:
    - Objects which encapsulate data and behavior
    - Messages passed between objects
  - Functional
    - Functions, Closures, Recursion, lists, …

- ✓ **Python**
  - Procedural
    - Yes
  - Objectural:
    - Yes
  - Functional
    - Yes

- Python
  - ❑ allows programmer to choose the paradigm that best suits the problem
  - ❑ Mix of Paradigms
  - ❑ Switching paradigm if necessary

- ✓ **A data type**
  - ☞ **Stores Data**
  - ☞ **+ Operations defined to act on the data**
- ✓ **Tangible Entities (Physically exists in real world)**
  - **Person, Student, Locker, Air Ticket, etc**
- ✓ **Intangible Entities (Exists logically in real world)**
  - **Bank Account, Email, Reservation**
- ✓ **Interactions between objects define the system operations**

- ❖ **Take a Bank Details or Your Mobile Phone or PC**
  - ✓ **It is not necessary that everyone should know everything about your account**
  - ✓ **Manager/Administrator has a role**
  - ✓ **Cashier/User has a role**
- ✓ **Think: A piece of code as black box**
  - ✓ **Cannot See**
  - ✓ **Do not need to see**
  - ✓ **Do not want to see**
  - ✓ **High Coding details**

- ✓ **Attributes or Data Attributes**
  - ✓ **Characteristics or properties of an entity in a database table**
  - ✓ **A named piece of data or variable**
  - ✓ **Data members (class variables and instance variables)**

- ✓ **Example 1: Student has**
  - ☞ **Name**
  - ☞ **Roll Number**
  - ☞ **Marks**
  - ☞ **Branch/discipline**

- ❖ **Example 2: Circle has**
  - ☞ **Radius**
  - ☞ **Center**
- ❖ **Example 3: Rectangle has**
  - ☞ **Sides/Edges**
  - ☞ **Vertices**

- ✓ **Methods or Procedural Attributes**
  - ✓ **Attributes bound to functions/behavior/operators**

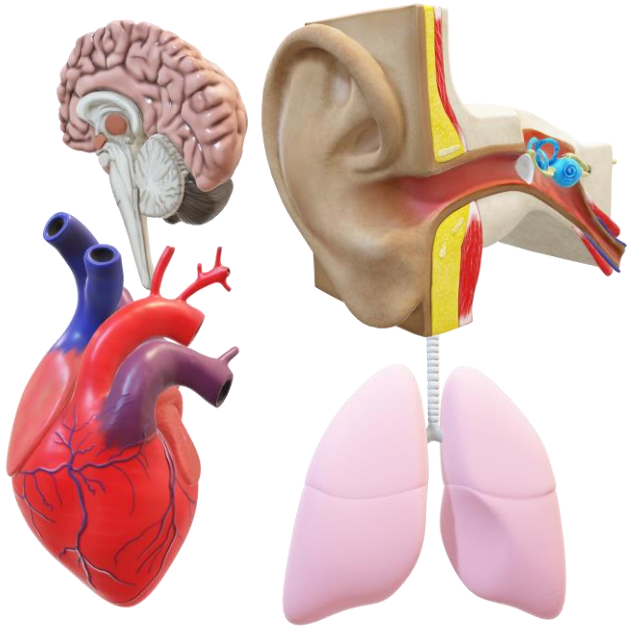
- ✓ **Example 1: Student has**
  - ☞ **Average Marks Calculation**
  - ☞ **Decide Grades**

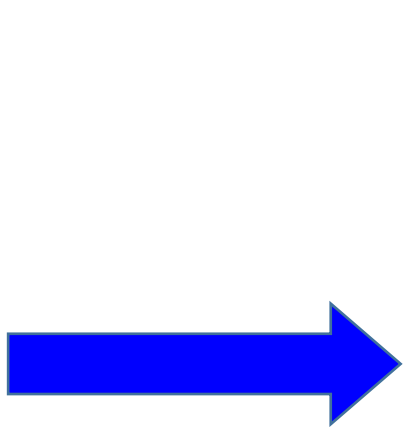
- ❖ **Example 2: Circle has**
  - ☞ **Area**
  - ☞ **Circumference**

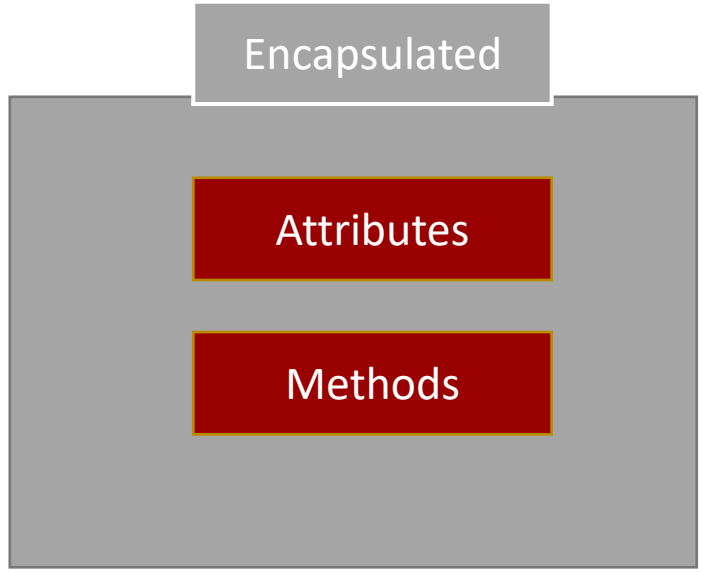
- ❖ **Example 3: Rectangle has**
  - ☞ **Area**
  - ☞ **Circumference**

- ✓ A process by which class components interact
  - Send data to another object
  - Request data from another object
  - Request object to perform some behaviour
- ✓ Implemented as methods (not called functions)
  - Functions are process that are object independent
  - Methods are dependent on the state of the object

- ✓ **Encapsulation implements the concept of abstraction**
  - ✵ **Details associated with object**
  - ✵ **End user could see the public interface, but implementation are hidden**







- ✓ **Attributes and methods are encapsulated within the logical boundary of the object entity**
  - ✵ **In procedural paradigms, data and functions are typically maintained as separate entities**
  - ✵ **In Objectural paradigms, each object has attributes (data) and methods (functions) that operates upon those attributes**

Glance at a Class

- ✓ **Classes**
  - **Bundling Data**
  - **+ Functionality**

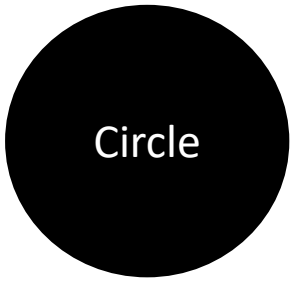
- ✓ **Classes**
  - **A collection of functions and attributes**
  - **Attached to a specific name to represent an abstract concept**

- ✓ **Classes**
  - **User-defined prototype for an object with attributes and methods**

- ✓ **A software item that contains variables and methods**
- ✓ **Object Oriented Design focuses on**
  - Encapsulation
    - dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:
    - the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:
    - the ability to create subclasses that contain specializations of their parents

- ✓ **View class/object as factories or templates**
- ✓ **An Individual Object of a certain class**
- ✓ **Each object instance takes all the properties of the class from which it was created**

Abstract Concept

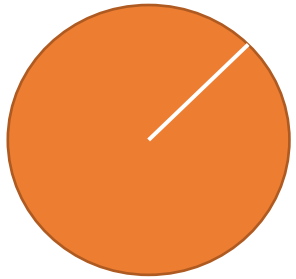


Attributes

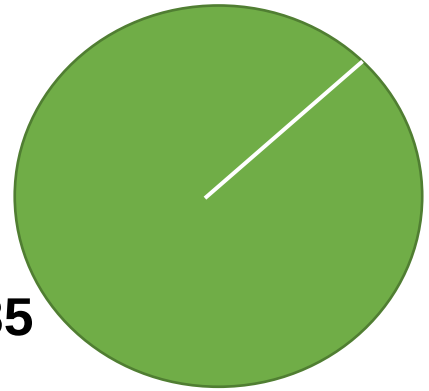
- ☞ **Radius**

Methods/Functions

- ☞ **Area()**
- ☞ **Circumference()**



Radius=3
Area()=28.27
Circumference()=18.85



Radius=6
Area()=113.1
Circumference()=37.7

Class Vs Instance of a Class

- ✓ class name is the type
  - ➢ `class circle(object)`
- ✓ Defined generically
  - ➢ Use `self` to refer to some instance while defining the class
  - ➢ `area=pi*self.r*self.r`
  - ➢ `self` is a parameter to methods in class definition
- ✓ Defines data and methods common across all instances
- ✓ Instance is one specific object
  - ➢ `mycirc=circle(2)`
- ✓ Data varies between instances
  - ➢ `mycirc1=circle(4)`
  - ➢ `mycirc2=circle(11)`
  - ➢ `mycirc1.r` and `mycirc2.r` are different
- ✓ Instance has the structure of the class

- **Object**
  - A unique instance of data structured defined by its class
  - Contains Data Members
    - Class Variables
    - Instance Variables
    - Methods

- **Class**
  - User-defined prototype for an object
  - Set of attributes to characterize any object of the class
  - Attributes
    - Data Members(Class Variables, Instance Variables)
    - Methods
    - Accessed via dot notation
- **Instance**
  - An individual object of a certain class
- **Instantiation**
  - Creation of an instance of a class

- **Class Variable**
  - Shared by all instance of a class
  - Defined within a class
  - Outside any of the class' methods
  - Not use as frequently as instance variable

- **Instance Variable**
  - Defined inside a method
  - Belongs to only to the current instance of the class

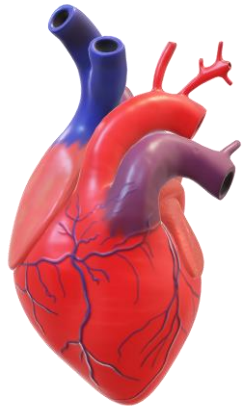
- **Data Member**
  - A class variable
  - Instance Variable
  - Holds data associated with a class and its objects

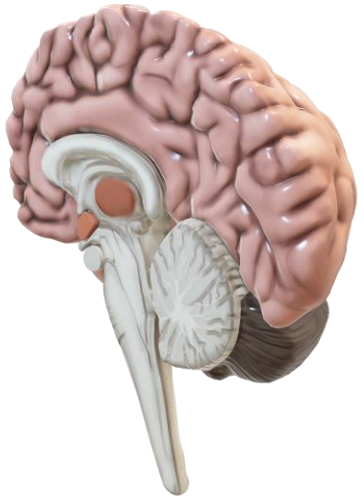
- **Method**
  - A special kind of function that is defined in a class definition

- **Function Overloading**
  - ☞ **Assignment of more than one behaviour to a particular function**
- **Operator Overloading**
  - ☞ **Assignment of more than one function to a particular operator**
- **Inheritance**
  - ☞ **Transfer of the characteristic of a class to other classes that are derived from it**

- **Overriding**
  - When inheriting from a class, we can alter behaviour of the parent class by overriding function
  - Declaring functions in the subclass with the same name
  - More precedence over parent class
- **Polymorphism**
  - Two objects of different classes
  - Supports same set of functions
  - Attributes can be treated identically
  - Implementation are different, but appears to be same

- Do you know or care how a smartphone or TV or washing machine or any electrical appliances or your own body??
  - ✕ No. As long as you are the user of the appliances and the appliance functions well




Why Use OOP and Classes of Objects

- **Group different object of the same type**
  - **Classes and objects are more like the real world**
  - **Mimic the real world**
  - **Minimize the semantic gap by modelling the real world**
- **Semantic Gap:**
  - **Difference Between the real world and the representation in a computer**
- **Allow you to define an interface to some object and its operations**
  - **Use it without knowing the internals**
- **Modularize the program into multiple objects that work together, each has its own purpose**

PYTHON CLASSES

- **Type()**
  - **It returns the data type of the argument passed to it**
- **Python Class:**
  - **Is a template for a data type**
  - **It can be defined using the `class` keyword**

```
class name:
    "documentation"
    statements
```

```
class name(base1, base2, ...):
    ...
```

Most, *statements* are method definitions:

```
    def name(self, arg1, arg2, ...):
        ...
```

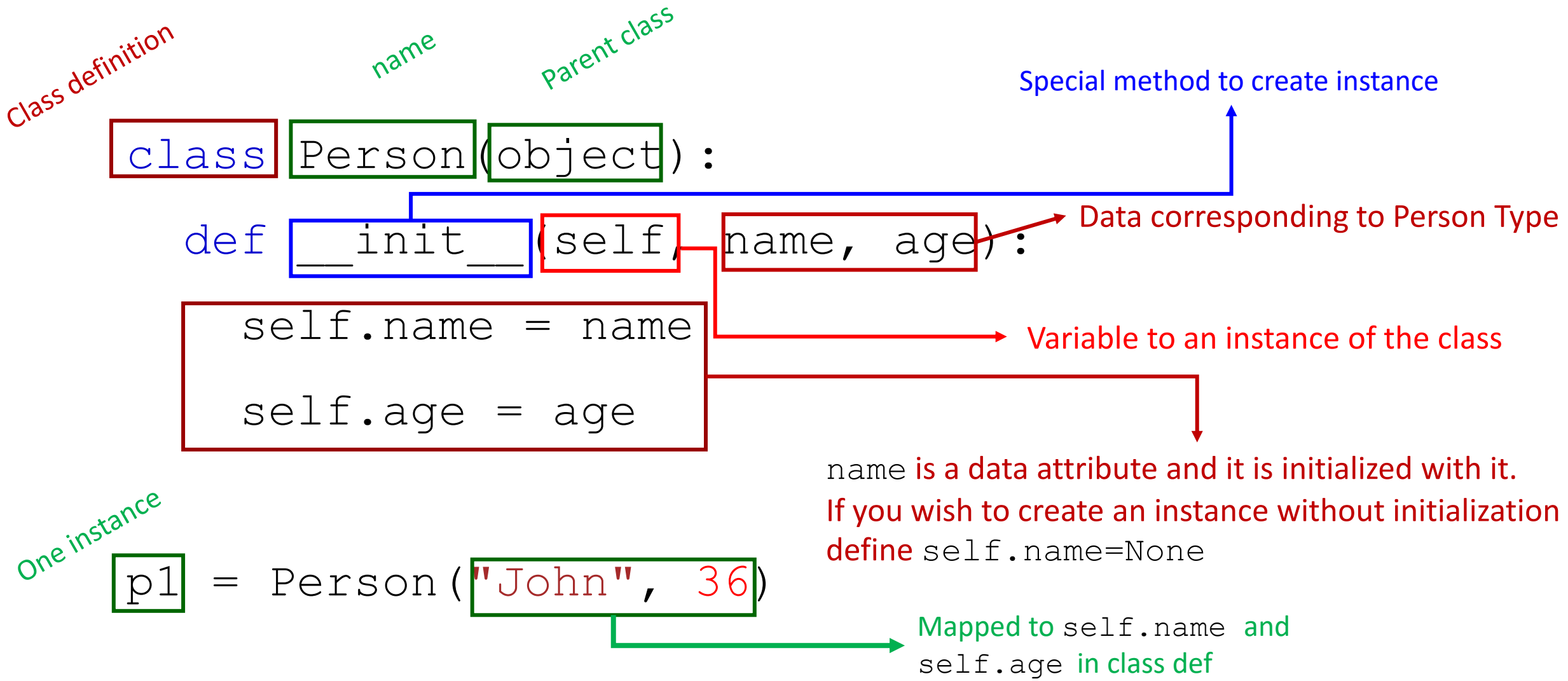
May also be *class variable* assignments

```
class Person(object):
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

How to define a Class



Class Variable vs Instance Variable

```
class Person(object):
  address ="Tirupati" #class variable Shared
  def __init__(self, name, age):
    self.name = name #instance variable
(unique to each instance)
    self.age = age
p1 = Person("John", 36)
p2 = Person("Navier", 45)
```

```
print(p1.address) #Tirupati (Class Variable)
print(p1.name) #John (instance variable)
print(p2.name) #Navier (instance variable)
print(p1.age)
print(p2.address) #Tirupati (Class Variable)
Person.address = "Hyderabad" #Change in
Class variable
print(p1.address)#Hyderabad (Class Variable)
print(p2.address)#Hyderabad (Class Variable)
```

```
# define the Vehicle class
class Vehicle:
    def __init__(self,name,kind,color,value):
        self.name=name
        self.kind=kind
        self.color=color
        self.value=value
    def description(self): # Method
        desc_str = "%s is a %s %s worth $%.2f."
% (self.name, self.color, self.kind, self.value)
        return desc_str
car1=Vehicle("Ford",'car','red',50000)
car2=Vehicle("BMW",'car','black',50000)
# test code
print(car1.description())
print(car2.description())
```

Getters and Setters Methods

```
class Person(object):
    def __init__(self, name=None, age=None):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def set_age(self,age):
        self.age=age
    def set_name(self,name):
        self.name=name
    def print_person(self):
        print(f"{self.name} is {self.age} years old")
```

getter

setter

```
p1 = Person("John", 36)
p1.print_person()
p2=Person()
p2.set_age(44)
p2.set_name("Ramya")
p2.print_person()
print(p2.get_age())
print(p2.get_name())
```

- **Getters and setters must be used outside class to access data attributes**

Instance, Dot Notation and Data Hiding

- **Instantiation creates an instance of an object**

```
p1 = Person("John", 36)
```

- **Dot notation used to access attributes (data and procedural)**
- **It is better to use getters and setters to access data attributes**
- **Outside the class use getters and setters**
  - **Use p1.get_name() instead of p1.name**
  - **Easy to maintain, debug and document**

```
p2.print_person()
print(p2.get_age())
print(p2.get_name())
print(p2.name,p2.age)
```

Empty Class

```
class EmptyClass:
  "Thi is an empty Class"
  pass
A=EmptyClass()
```

```
class Book:
  "Thi is an empty Book"
  pass
Rudin=Book()
```

- It has no attributes or methods
- Use pass to avoid syntax errors
- Purpose
  - Placeholder for future development
  - For dynamic attributes
  - Base class
  - For performance test
  - Object Tagging

```
# define the Vehicle class
class Vehicle:
    def __init__(self,name,kind,color,value):
        self.name=name
        self.kind=kind
        self.color=color
        self.value=value
    def description(self): # Method
        desc_str = "%s is a %s %s worth $%.2f." % (self.name,
self.color, self.kind, self.value)
        return desc_str
```

```
if __name__=="__main__":
  print(dir())
  print(dir(Vehicle))
  car1=Vehicle("Ford",'car','red',50000)
  car2=Vehicle("BMW",'car','black',50000)
  print(car1.name) # Ford
  print(car2.name) # BMW
```

- `__name__=="__main__"` **ensures code only runs when executed directly**
- **Prevents unintended execution when importing a script**
- **Used for unit testing**

DIFFERENT METHODS

```
# define the Vehicle class
class Vehicle:
    def __init__(self,name,kind,color,value):
        self.name=name
        self.kind=kind
        self.color=color
        self.value=value
car1=Vehicle("Ford",'car','red',50000)
car2=Vehicle("BMW",'car','black',50000)
print(car1.name) # Ford
print(car2.name) # BMW
```

- **Initialize a newly created object.**
- **It is called every time when the class is instantiated**

dir() Method

```
# define the Vehicle class
class Vehicle:
    def __init__(self,name,kind,color,value):
        self.name=name
        self.kind=kind
        self.color=color
        self.value=value
    def description(self): # Method
        desc_str = "%s is a %s %s worth $%.2f." % (self.name,
self.color, self.kind, self.value)
        return desc_str
```

```
print(dir())
print(dir(Vehicle))
car1=Vehicle("Ford",'car','red',50000)
car2=Vehicle("BMW",'car','black',50000)
print(car1.name) # Ford
print(car2.name) # BMW
```

- **dir() returns a list of all attributes in the current scope**
- **dir(objectname) returns all valid object attributes**

```
class ObjectName:
    @classmethod
    def some_class_method(cls, *args, **kwargs):
        # Method Implementation
        pass
```

- **Used to define a method that is bound to the class, not the instance of the class**

```python
class Institute:
    Institute_name = "IIT Tirupati"

    @classmethod
    def change_insti(cls, new_insti):
        cls.Institute_name = new_insti  # Changes class variable

Institute.change_insti("IIT Madras")
print(Institute.Institute_name)  # ✅ IIT Madras
```

```
class ObjectName:
    @staticethod
    def some_class_method(*args, **kwargs):
        # Method Implementation
        pass
```

- It does not receive any implicit argument
- Bound to the class, but not the object of the class
- It can’t access or modify the class state

```
class MathFunction:
  @staticmethod
  def factorial(n):
    if n==0:
      return 1
    else:
      return n*MathFunction.factorial(n-1)
print(MathFunction.factorial(5))
```

- **If you define a function in a module and don't want it to be associated with an instance of a class or module, you can use the `@staticmethod` decorator to declare that function as static.**
- **It does not access self**

Class vs Static Method

Class Method	Static Method
Takes cls as the first parameter	No specific parameters required
Can access or modify the class state	Can't access or modify the class state
It knows about the class state	It does not know anything about the class state
It must have a class parameter	It takes some parameters and work upon those parameters
@classmethod	@staticmethod

```
class Rectangle:
  def print_test(self):
    print("Parent Class: Opposite Sides Are Equal")
class Square(Rectangle):
  def print_test(self):
    print("Child Class: All Sides are Equal")
    # Calls the parent's version of print_test()
    super().print_test()
```

```
A = Square()
A.print_test()
```

- **super() allows a subclass to invoke its parent's version of an overriden method**

```
class Rectangle:
  def print_test(self):
    print("Parent Class: Opposite Sides Are Equal")
class Square(Rectangle):
  def print_test(self):
    print("Child Class: All Sides are Equal")
    # Calls the parent's version of print_test()
    super().print_test()
```

```
A = Square()
A.print_test()
```

- **super() allows a subclass to invoke its parent's version of an overriden method**

issubclass() Method

```
class Rectangle:
  def print_test(self):
    print("Parent Class: Opposite Sides Are Equal")
class Square(Rectangle):
  def print_test(self):
    print("Child Class: All Sides are Equal")
    # Calls the parent's version of print_test()
print(issubclass(Square,Rectangle))
print(isinstance(Square(),Rectangle))
```

```
True
True
```

- **built-in function checks if the first argument is a subclass/instance of the second argument.**

```
class Rectangle:
  def print_test(self):
    print("Parent Class: Opposite Sides Are Equal")
class Square(Rectangle):
  def print_test(self):
    print("Child Class: All Sides are Equal")
```

```
A = Square()
A.print_test()
B = Rectangle()
B.print_test()
```

- **Two classes with same `print_test()`**

Can we hide data? Python says "no"

Python is not good at data hiding

- ~~You can access data from outside class definition~~

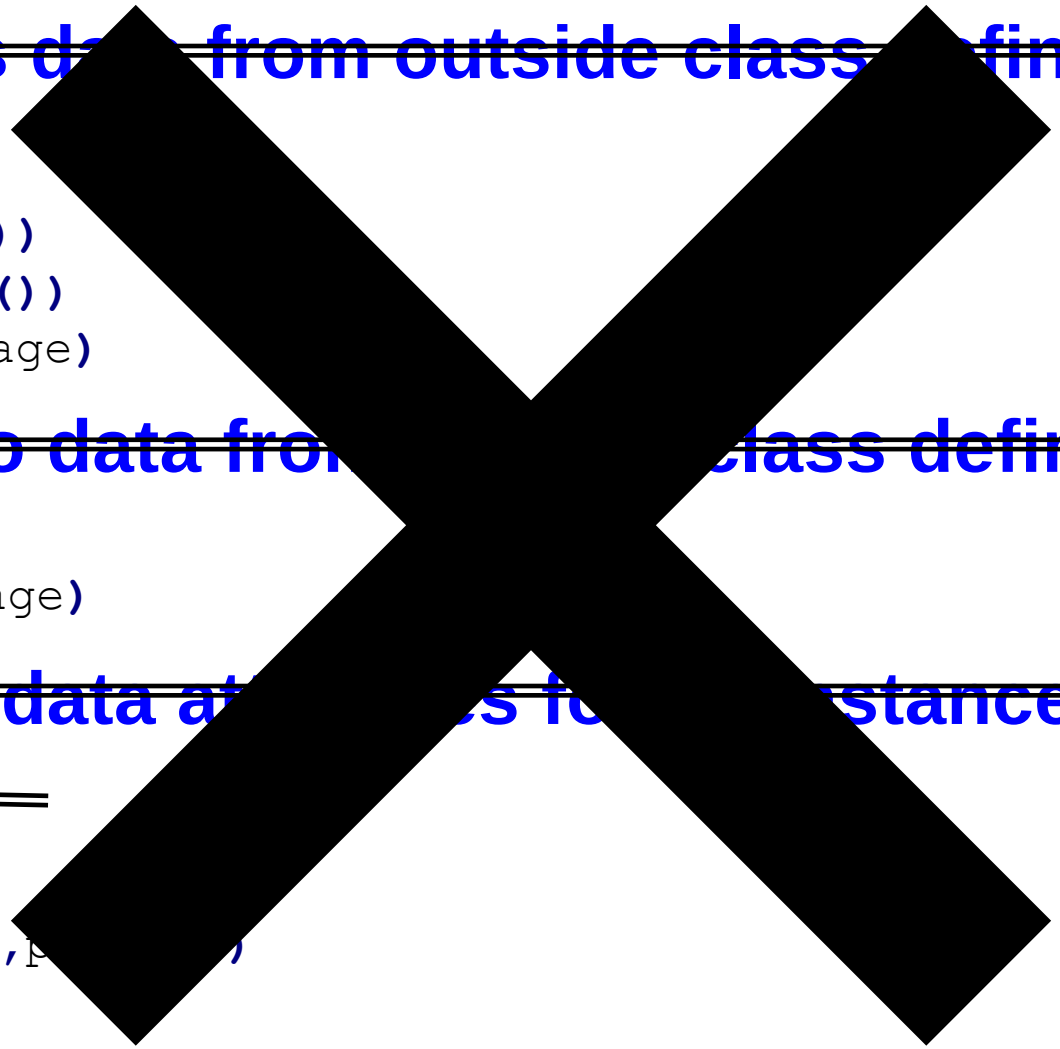
```
p2.print_person()
print(p2.get_age())
print(p2.get_name())
print(p2.name,p2.age)
```

- ~~You can write to data from outside class definition~~

```
p2.name="Karan"
print(p2.name,p2.age)
```

- ~~You can create data attributes for instance from outside class definition~~

```
p2.city="Tirupati"
print(p2.name,p2.age,p
```



Hackaround for Data Hiding

- Python attributes and methods are public by default
  - Public attributes means any class or function can see and change the attribute
  - Public method means any other class or function call the method
  - Refer Previous slide
- There is a hackaround to make it private
  - Add __ (two underscores) to the beginning of the name
  - `self.__name=name`
  - `self.__age=age`
  - `def __functionname():`

Hackaround for Data Hiding

```
class Shape(object):
    def __init__(self):
        self.color=(0,0,0)
    def get_color(self):
        return self.color
    def set_color(self,color):
        self.color=color
    def print_color(self):
        print(self.color)
```

```
import math
class Circle(Shape):
    def __init__(self,r=None):
        Shape.__init__(self)
        self.radius=r
    def get_radius(self):
        return self.radius
    def set_radius(self,r):
        self.radius=r
    def area(self):
        return
math.pi*self.radius*self.radius
    def circumference(self):
        return 2*math.pi*self.radius
```

```
C=Circle()
C.radius=5
print(C.area())
```

Hackaround for Data Hiding

```
class Shape(object):
    def __init__(self):
        self.color=(0,0,0)
    def get_color(self):
        return self.color
    def set_color(self,color):
        self.color=color
    def print_color(self):
        print(self.color)
```

```
C=Circle()
C.__radius=7
print(C.area())
```

```
C=Circle()
C.set_radius(5)
print(C.area())
```

```
import math
class Circle(Shape):
    def __init__(self,r=None):
        Shape.__init__(self)
        self.__radius=r
    def get_radius(self):
        return self.__radius
    def set_radius(self,r):
        self.__radius=r
    def area(self):
        return
math.pi*self.__radius*self.__radius
    def __circumference(self):
        return 2*math.pi*self.__radius
```

Hackaround of the Hackaround for Data Hiding

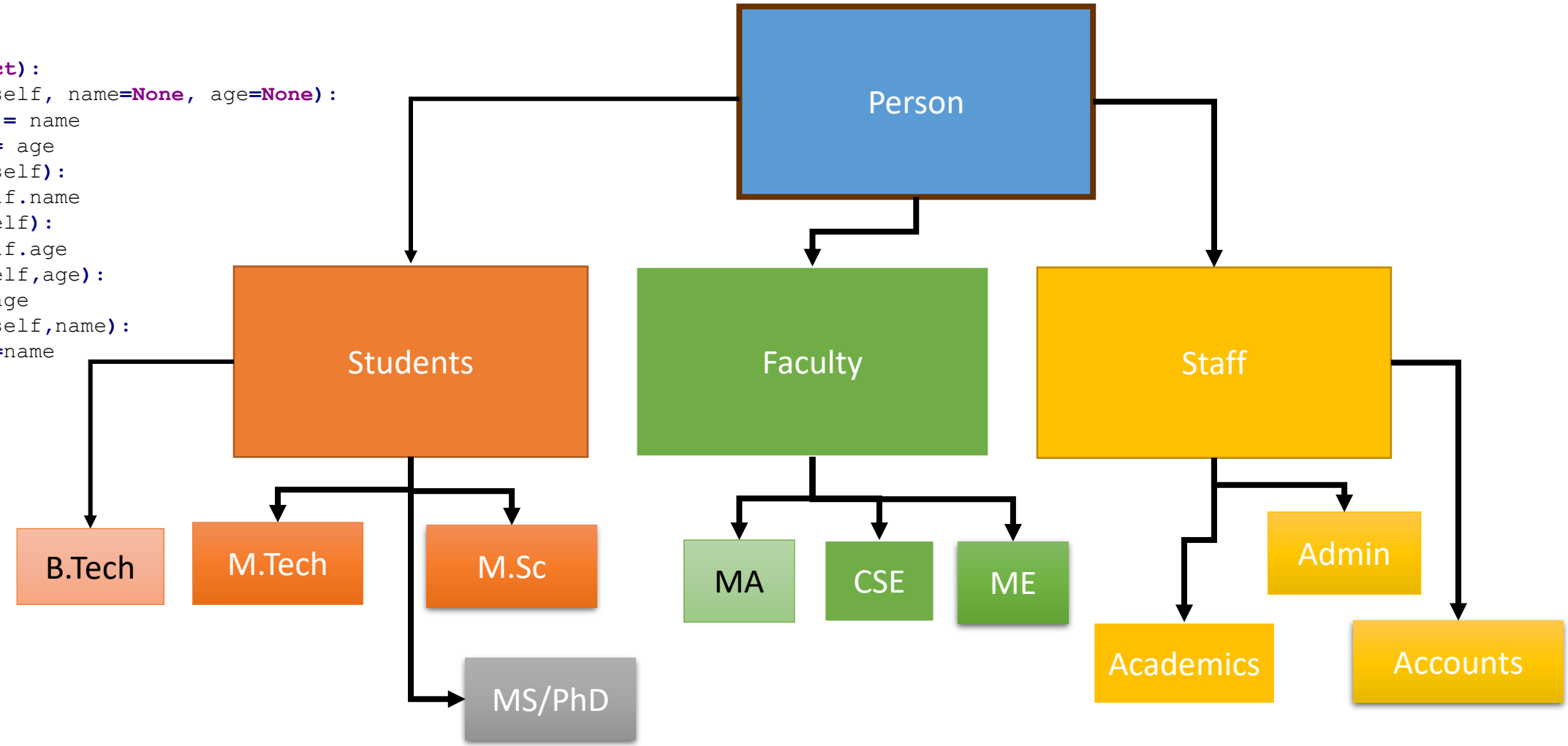
```
class Shape(object):
    def __init__(self):
        self.color=(0,0,0)
    def get_color(self):
        return self.color
    def set_color(self,color):
        self.color=color
    def print_color(self):
        print(self.color)
```

```
import math
class Circle(Shape):
    def __init__(self,r=None):
        Shape.__init__(self)
        self.__radius=r
    def get_radius(self):
        return self.__radius
    def set_radius(self,r):
        self.__radius=r
    def area(self):
        return
math.pi*self.__radius*self.__radius
    def __circumference(self):
        return 2*math.pi*self.__radius
```

```
C=Circle()
C._Circle__radius=7
print(C.area())
```

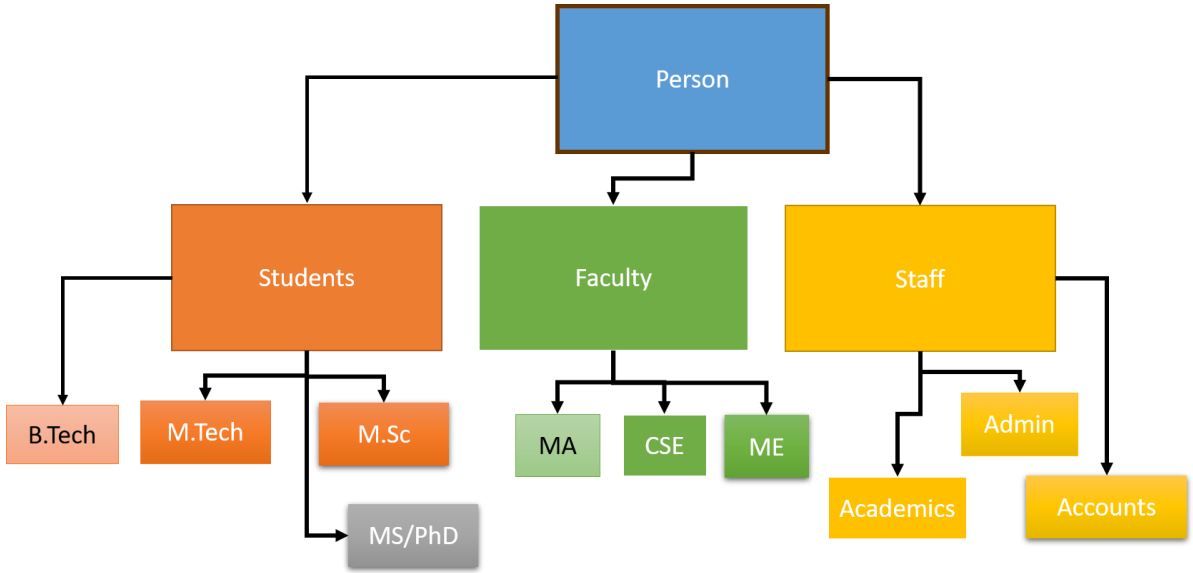
Hierarchies and Inheritance

```
class Person(object):
    def __init__(self, name=None, age=None):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def set_age(self,age):
        self.age=age
    def set_name(self,name):
        self.name=name
```



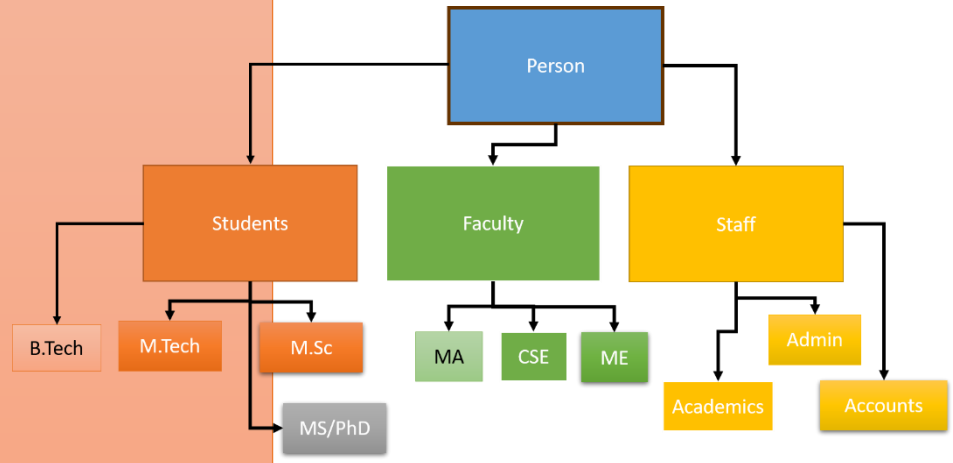
Instance, Dot Notation and Data Hiding

- Parent class (Superclass)
- Child Class (subclass)
  - Inherits all data and procedural attributes of parent class
- You can add more data
- You can add more functions
- You can override methods

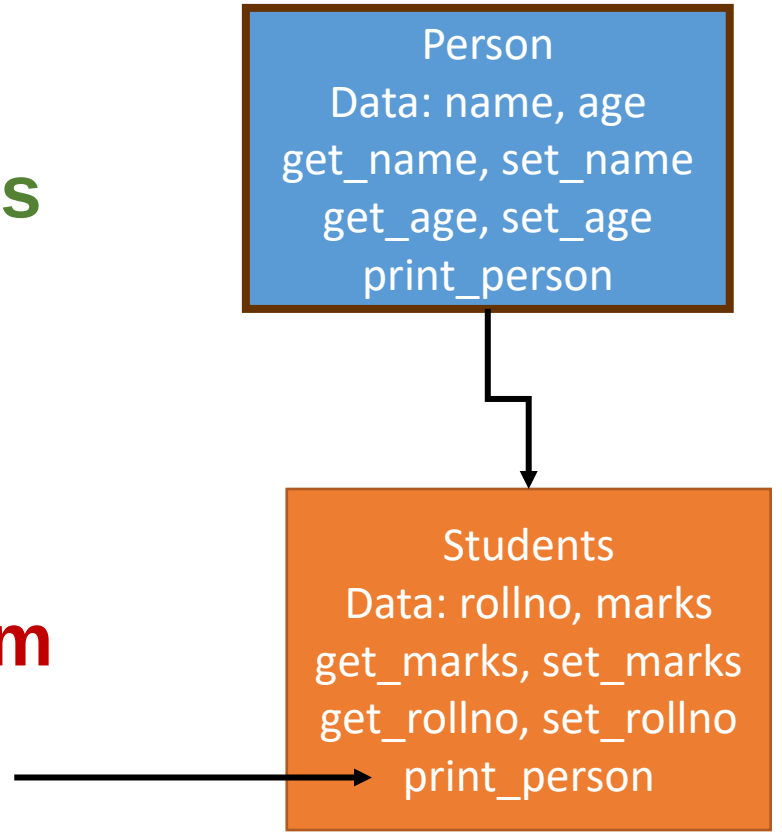


```
class Student(Person):
    def __init__(self, rollno=None, marks=None):
        self.rollno = rollno
        self.marks = marks
    def get_rollno(self):
        return self.rollno
    def get_marks(self):
        return self.marks
    def set_marks(self,marks):
        self.marks=marks
    def set_rollno(self,rollno):
        self.rollno=rollno
```

New Functionalities Added



- **Subclass and superclass can have methods with same name (`print_person`)**
- **First look for the method in current class**
- **If not found (`get_name`), go to the parent class, if not grandparent, and so on**
- **It stops after finding the least top level (from backward) and uses the method name**



```
class Student(Person):
    def __init__(self, rollno=None, marks=None):
        self.rollno = rollno
        self.marks = marks
    def get_rollno(self):
        return self.rollno
```

Constructor

- ✓ When an instance of a class is created the class constructor function is automatically called
- ✓ Constructor is always named `__init__()`
- ✓ A piece of code to initialize a new instance by setting data attributes (mostly)

OPERATOR OVERLOADING

Operator Overloading

```
class Complex:
  def __init__(self,x,y) -> None:
     self.real=x
     self.imaginary=y
  def print(self):
    if(self.imaginary<0):
      print(str(self.real)+str(self.imaginary)+"i")
    else:
      print(str(self.real)+"+"+str(self.imaginary)+"i")
  def __add__(self,other):
    return Complex(self.real+other.real,self.imaginary+other.imaginary)
  def __sub__(self,other):
    return Complex(self.real-other.real,self.imaginary-other.imaginary)
```

```
A=Complex(2,3)
A.print()
B=Complex(4,5)
B.print()
C=A+B
C.print()
D=A-B
D.print()
```

Operator Overloading

Operator	Method	Example
+	__add__	A+B
-	__sub__	A-B
*	__mul__	A*B
/	__truediv__	A/B
//	__floordiv__	A//B
%	__mod__	A%B
**	__pow__	A**B
==	__eq__	A==B
!=	__ne__	A!=B
<	__lt__	A<B
<=	__le__	A<=B
>	__gt__	A>B
>=	__ge__	A>=B

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __gt__(self, other):
        return self.age > other.age  # Compare by age

people = [Person("Alice", 30), Person("Bob", 25), Person("Charlie", 35)]
people.sort()  # ✅ Uses __gt__ to sort

for p in people:
    print(p.name, p.age)
```

End of Python Classes