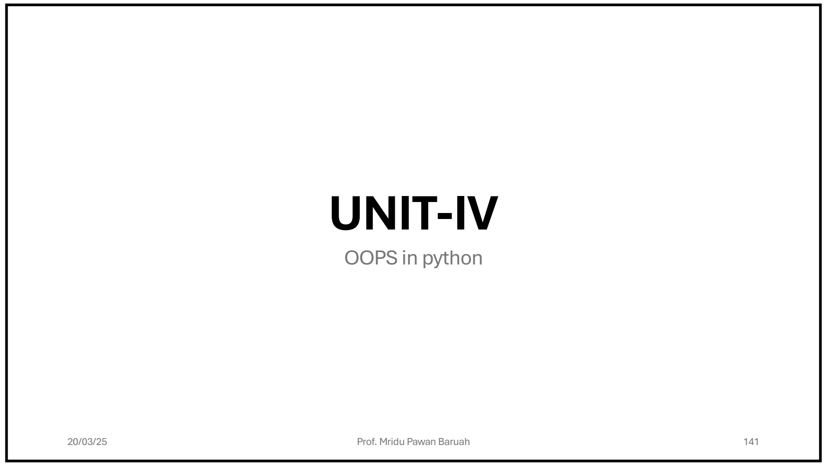
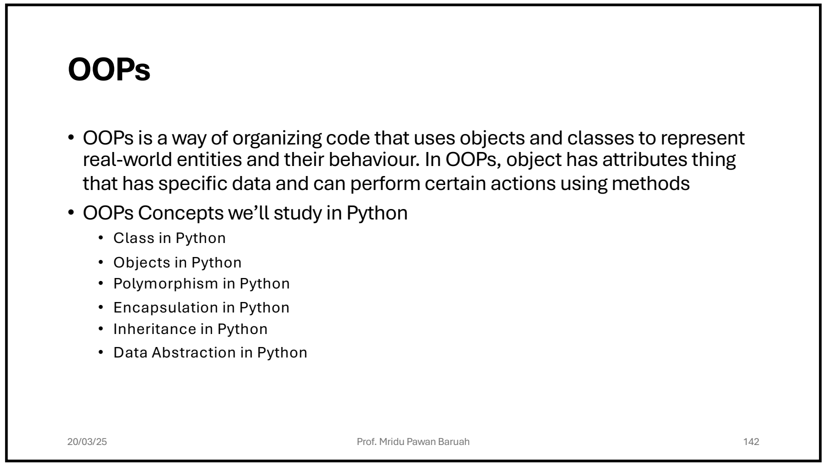


UNIT-IV

OOPS in python

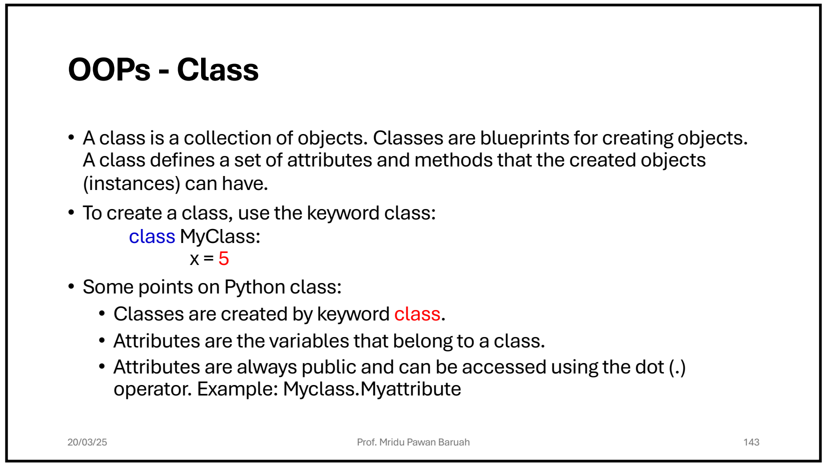
141



OOPs

- OOPs is a way of organizing code that uses objects and classes to represent real-world entities and their behaviour. In OOPs, object has attributes thing that has specific data and can perform certain actions using methods
- OOPs Concepts we'll study in Python
  - Class in Python
  - Objects in Python
  - Polymorphism in Python
  - Encapsulation in Python
  - Inheritance in Python
  - Data Abstraction in Python

142



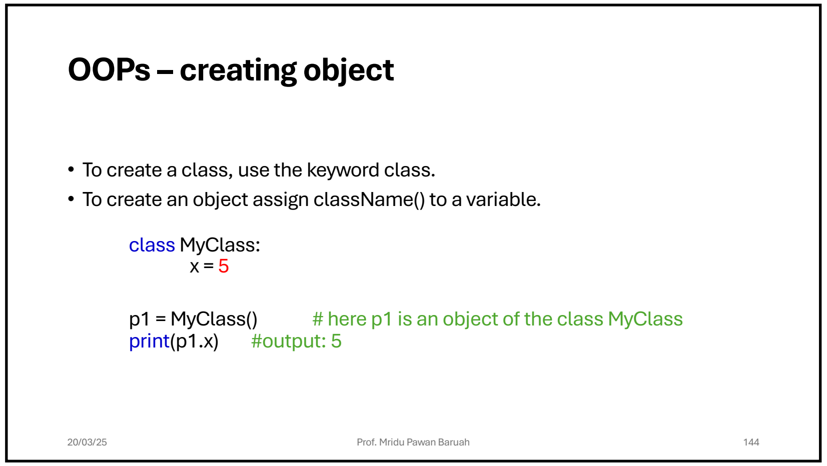
OOPs - Class

- A class is a collection of objects. Classes are blueprints for creating objects. A class defines a set of attributes and methods that the created objects (instances) can have.
- To create a class, use the keyword class:

```
class MyClass:
    x = 5
```

- Some points on Python class:
  - Classes are created by keyword class.
  - Attributes are the variables that belong to a class.
  - Attributes are always public and can be accessed using the dot (.) operator. Example: Myclass.Myattribute

143



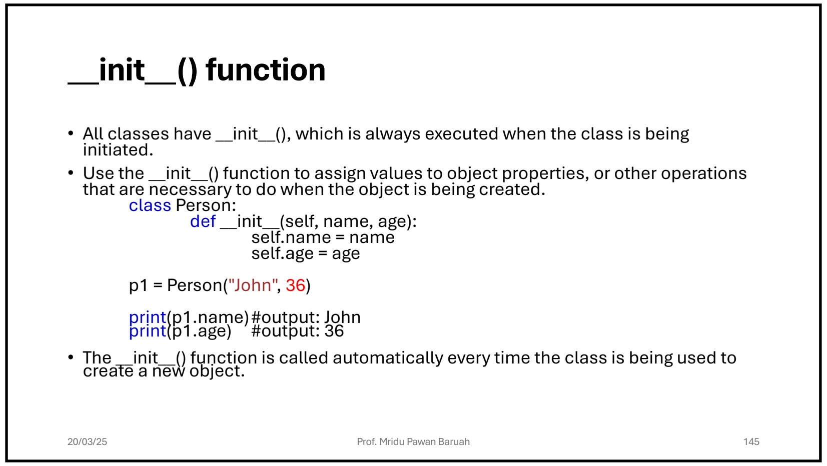
OOPs – creating object

- To create a class, use the keyword class.
- To create an object assign className() to a variable.

```
class MyClass:
    x = 5

p1 = MyClass()        # here p1 is an object of the class MyClass
print(p1.x)      #output: 5
```

144



__init__() function

- All classes have __init__(), which is always executed when the class is being initiated.
- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

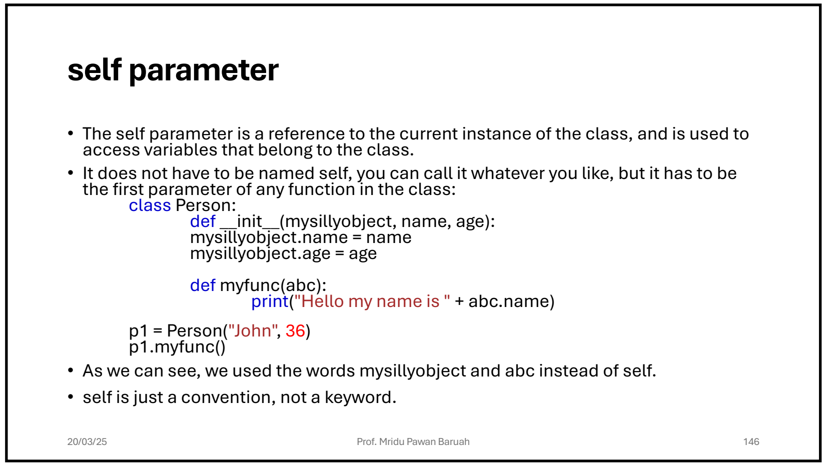
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1.name) #output: John
print(p1.age)   #output: 36
```

- The __init__() function is called automatically every time the class is being used to create a new object.

145



self parameter

- The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.
- It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class:

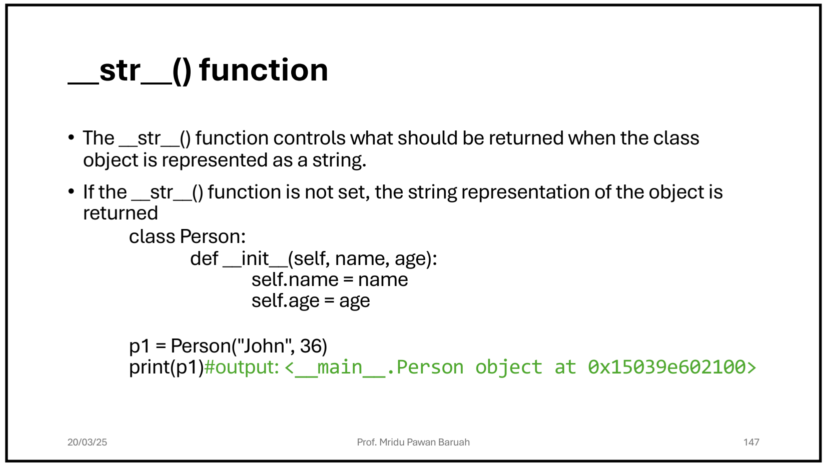
```
class Person:
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age

    def myfunc(abc):
        print("Hello my name is " + abc.name)

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

- As we can see, we used the words mysillyobject and abc instead of self.
- self is just a convention, not a keyword.

146



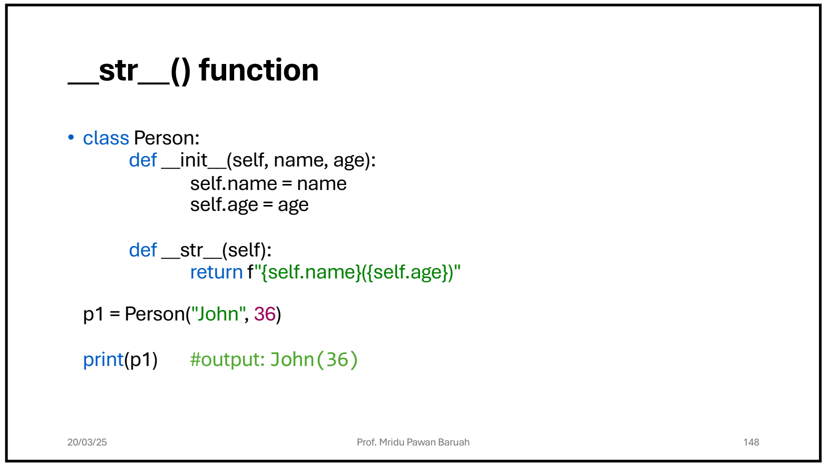
__str__() function

- The __str__() function controls what should be returned when the class object is represented as a string.
- If the __str__() function is not set, the string representation of the object is returned

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

p1 = Person("John", 36)
print(p1)#output: <__main__.Person object at 0x15039e602100>
```

147



__str__() function

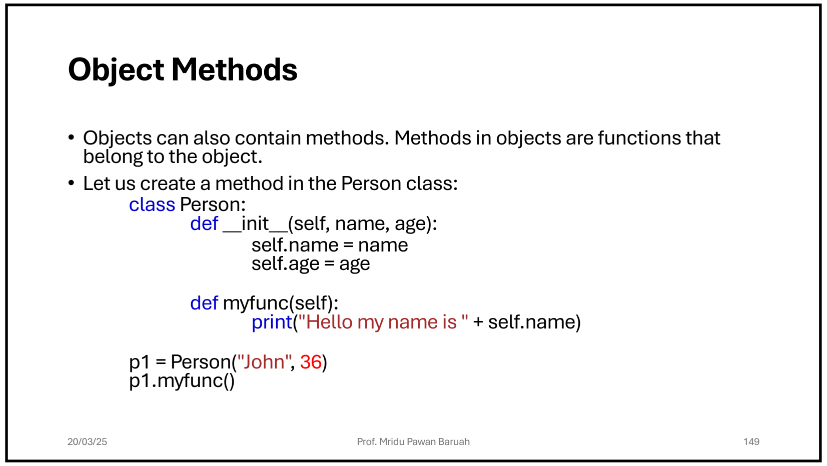
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)     #output: John(36)
```

148



Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.
- Let us create a method in the Person class:

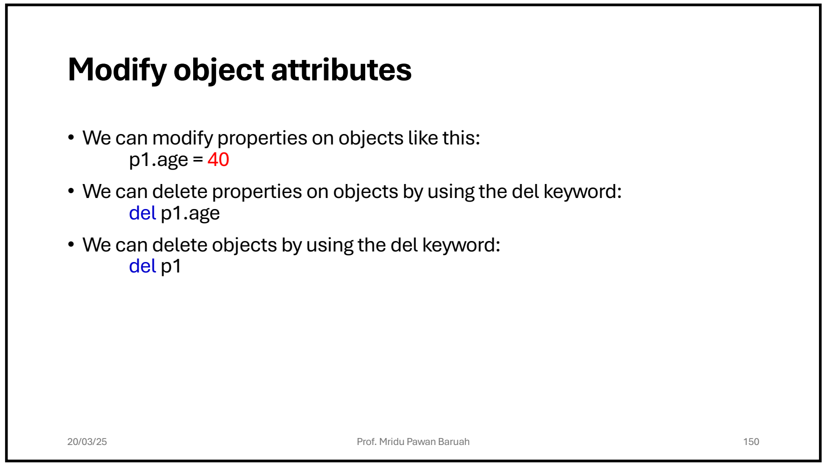
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

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

20/03/25 Prof. Mridu Pawan Baruah 149

149



Modify object attributes

- We can modify properties on objects like this:

```
p1.age = 40
```

- We can delete properties on objects by using the del keyword:

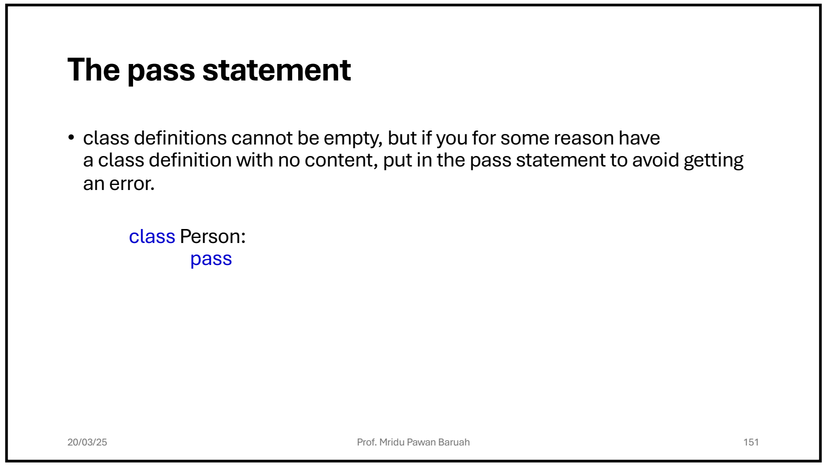
```
del p1.age
```

- We can delete objects by using the del keyword:

```
del p1
```

20/03/25 Prof. Mridu Pawan Baruah 150

150



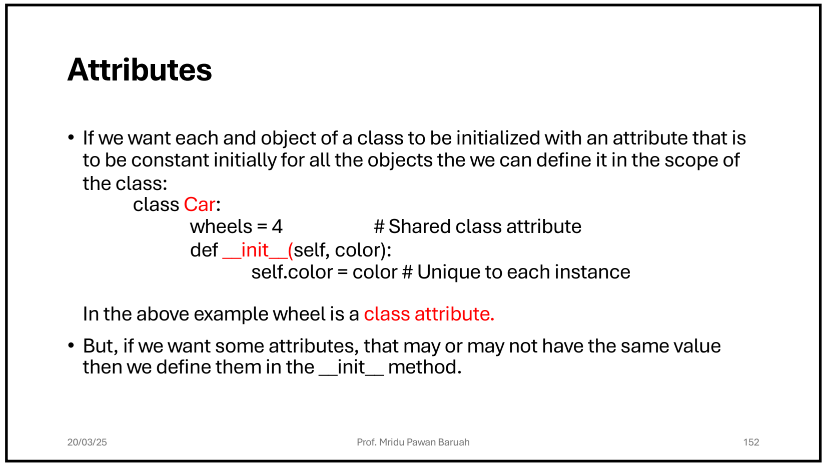
The pass statement

- class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

```
class Person:
    pass
```

20/03/25 Prof. Mridu Pawan Baruah 151

151



Attributes

- If we want each and object of a class to be initialized with an attribute that is to be constant initially for all the objects the we can define it in the scope of the class:

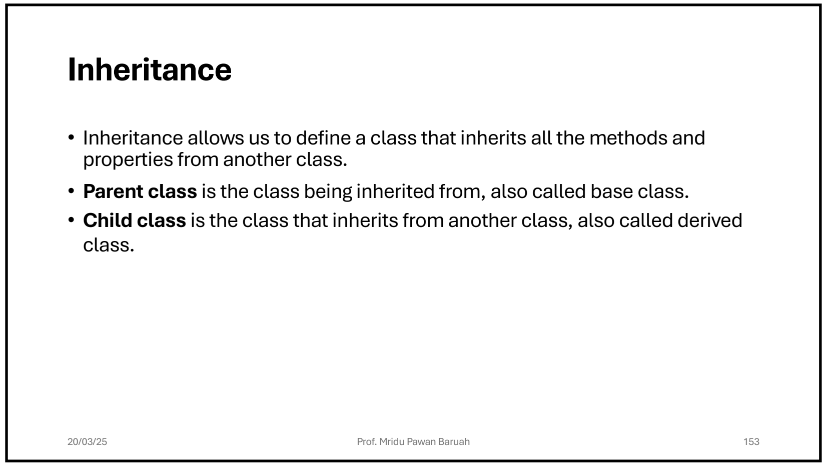
```
class Car:
    wheels = 4              # Shared class attribute
    def __init__(self, color):
        self.color = color # Unique to each instance
```

In the above example wheel is a class attribute.

- But, if we want some attributes, that may or may not have the same value then we define them in the __init__ method.

20/03/25 Prof. Mridu Pawan Baruah 152

152

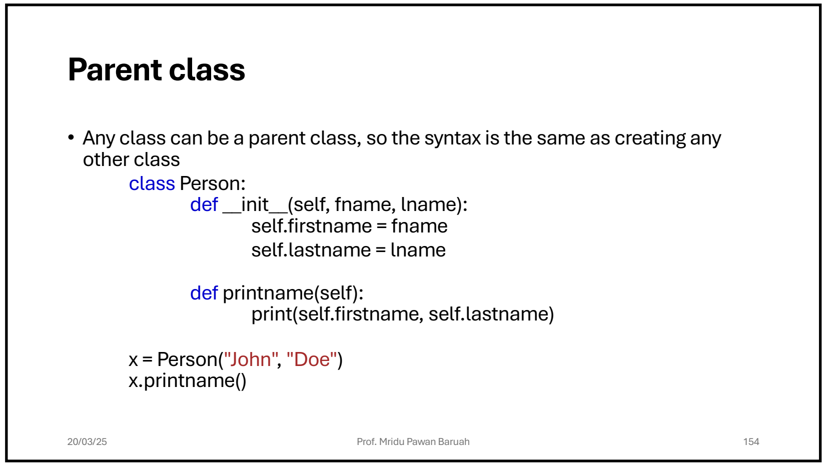


Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

20/03/25 Prof. Mridu Pawan Baruah 153

153



Parent class

- Any class can be a parent class, so the syntax is the same as creating any other class

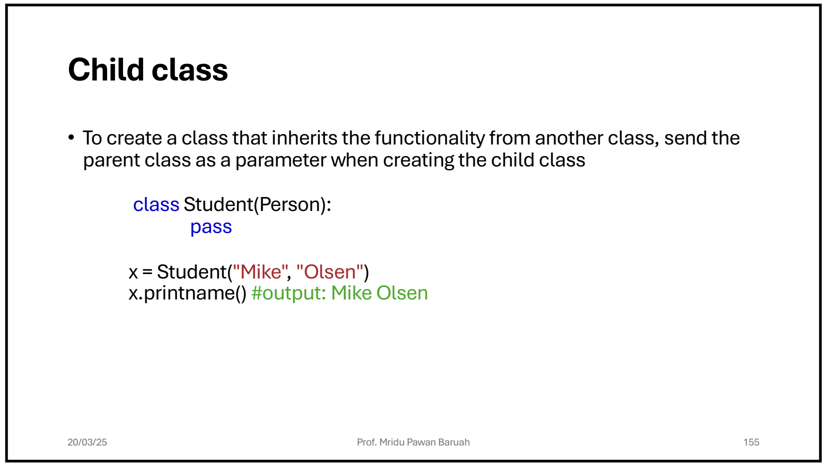
```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

x = Person("John", "Doe")
x.printname()
```

20/03/25 Prof. Mridu Pawan Baruah 154

154



Child class

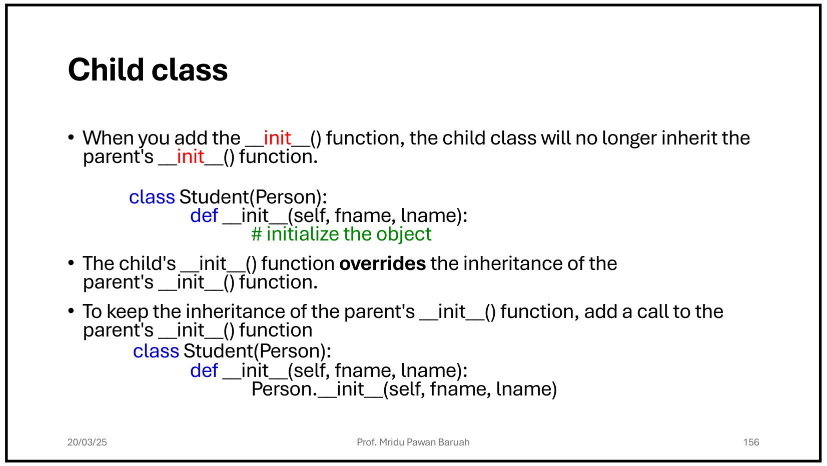
- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class

```
class Student(Person):
    pass

x = Student("Mike", "Olsen")
x.printname() #output: Mike Olsen
```

20/03/25 Prof. Mridu Pawan Baruah 155

155



Child class

- When you add the __init__() function, the child class will no longer inherit the parent's __init__() function.

```
class Student(Person):
    def __init__(self, fname, lname):
        # initialize the object
```

- The child's __init__() function **overrides** the inheritance of the parent's __init__() function.
- To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

20/03/25 Prof. Mridu Pawan Baruah 156

156

super() function

- Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
```

- By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

157

Adding an attribute

- Add an attribute called graduationyear to the Student class

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2019
```

- In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
print(x.graduationyear)          #output: 2019
```

158

Types of inheritance

- Single inheritance: A child class inherits from one parent class.
- Multiple inheritance: A child class inherits from more than one parent class.
- Multilevel inheritance: A class is derived from a class which is also derived from another class.
- Hierarchical inheritance: Multiple classes inherit from a single parent class.
- Hybrid inheritance: A combination of more than one type of inheritance.

159

Types of Inheritance

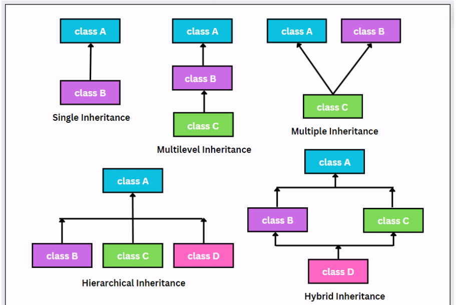
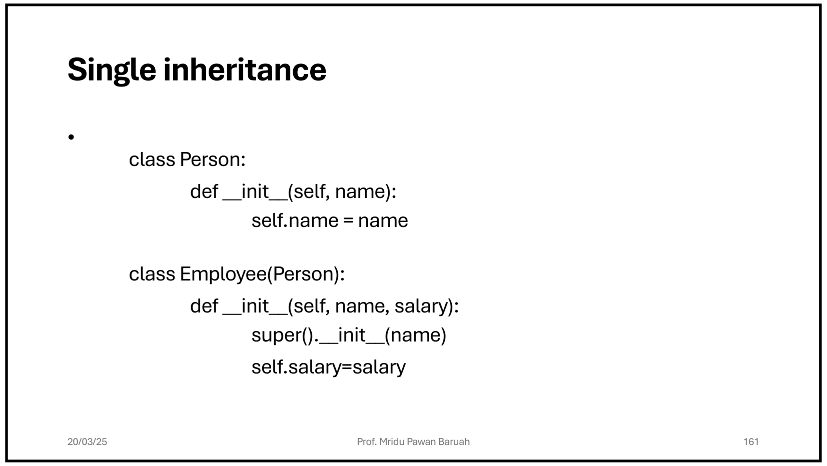


Fig: Types of inheritance in Python

160



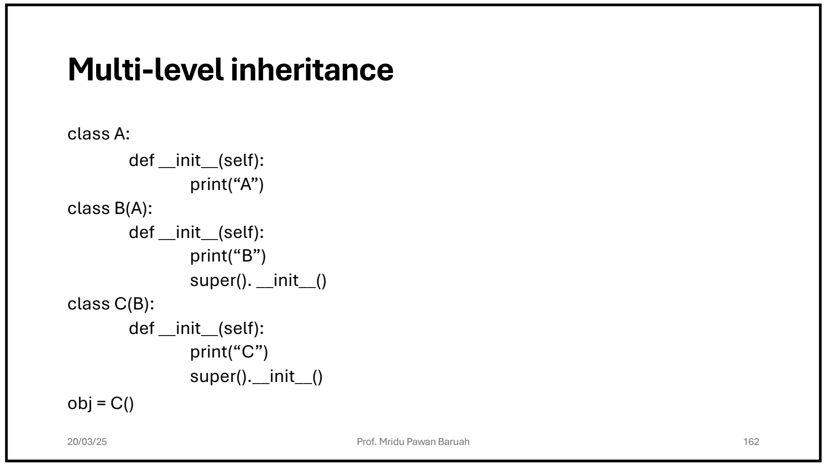
Single inheritance

•

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

class Employee(Person):
        def __init__(self, name, salary):
                super().__init__(name)
                self.salary=salary
```

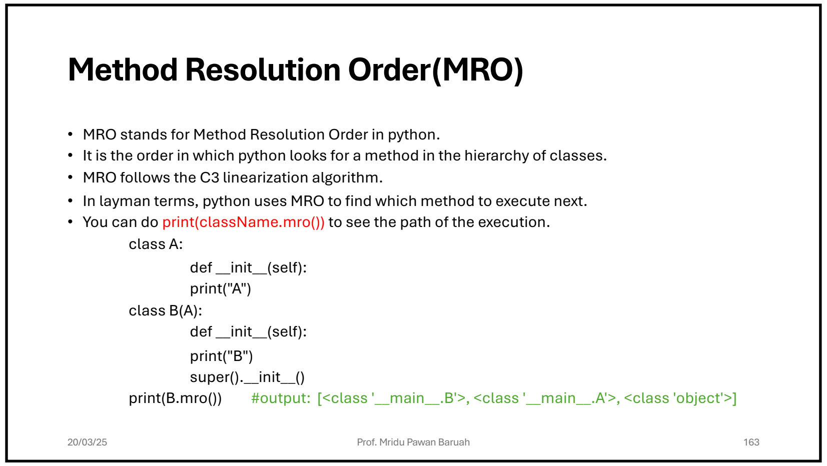
161



Multi-level inheritance

```
class A:
        def __init__(self):
                print(“A”)
class B(A):
        def __init__(self):
                print(“B”)
                super(). __init__()
class C(B):
        def __init__(self):
                print(“C”)
                super().__init__()
obj = C()
```

162

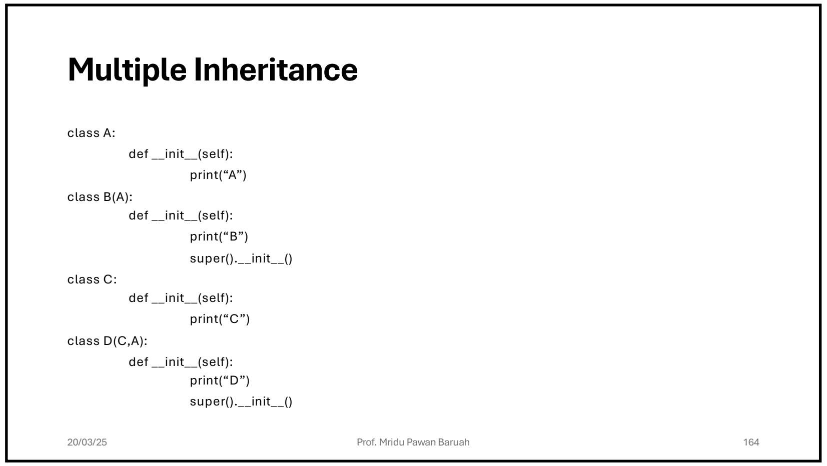


Method Resolution Order(MRO)

- MRO stands for Method Resolution Order in python.
- It is the order in which python looks for a method in the hierarchy of classes.
- MRO follows the C3 linearization algorithm.
- In layman terms, python uses MRO to find which method to execute next.
- You can do print(className.mro()) to see the path of the execution.

```
class A:
        def __init__(self):
        print("A")
class B(A):
        def __init__(self):
        print("B")
        super().__init__()
print(B.mro())     #output:  [<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

163



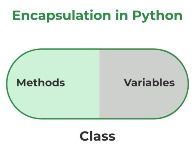
Multiple Inheritance

```
class A:
        def __init__(self):
                print(“A”)
class B(A):
        def __init__(self):
                print(“B”)
                super().__init__()
class C:
        def __init__(self):
                print(“C”)
class D(C,A):
        def __init__(self):
                print(“D”)
                super().__init__()
```

164

Encapsulation

- In Python, encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, typically a class.
- It also restricts direct access to some components, which helps protect the integrity of the data and ensures proper usage.
- Encapsulation is the process of hiding the internal state of an object and requiring all interactions to be performed through an object's methods.



165

Encapsulation

- **Data Hiding**: The variables (attributes) are kept private or protected, meaning they are not accessible directly from outside the class. Instead, they can only be accessed or modified through the methods.
- **Access through Methods**: Methods act as the interface through which external code interacts with the data stored in the variables. For instance, getters and setters are common methods used to retrieve and update the value of a private variable.
- **Control and Security**: By encapsulating the variables and only allowing their manipulation via methods, the class can enforce rules on how the variables are accessed or modified, thus maintaining control and security over the data.

166

Protected and Private members

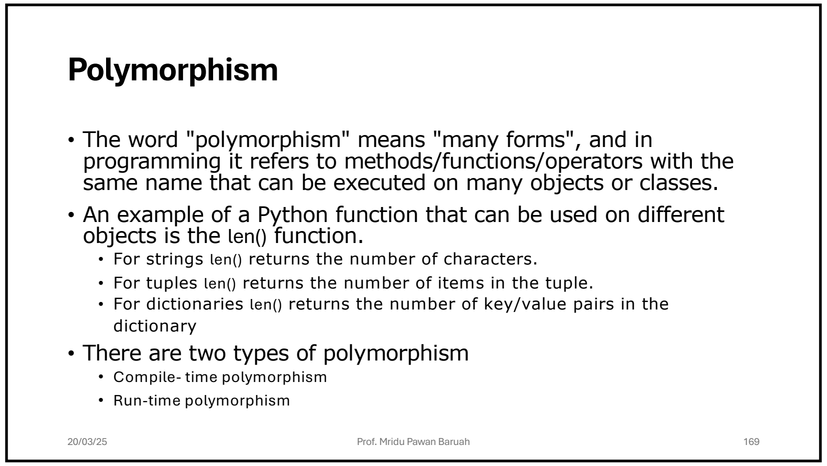
- These are marked by a single leading underscore "_"(e.g., self._attribute). While Python does not enforce any real access restrictions, this convention indicates that the attribute is intended for internal use only (i.e., within the class and its subclasses). It's a signal to developers that they should avoid accessing or modifying such attributes directly from outside the class.
- Private members are identified with a double underscore **(__)** and cannot be accessed directly from outside the class.

167

Example of a Protected Member

```
class Base1:
        def __init__(self):
                self.p = "CSE"          # public attribute
                self.__q = "-FY"        # protected attribute
        def __str__(self):
                print(self.__q)
obj_1 = Base1()
print(obj_1.p)              # accessing the public attribute doesn't throw error
print(obj_1._q)             # accessing the private attribute throws an error
obj_1.__str__()             # printing the protected attribute
```

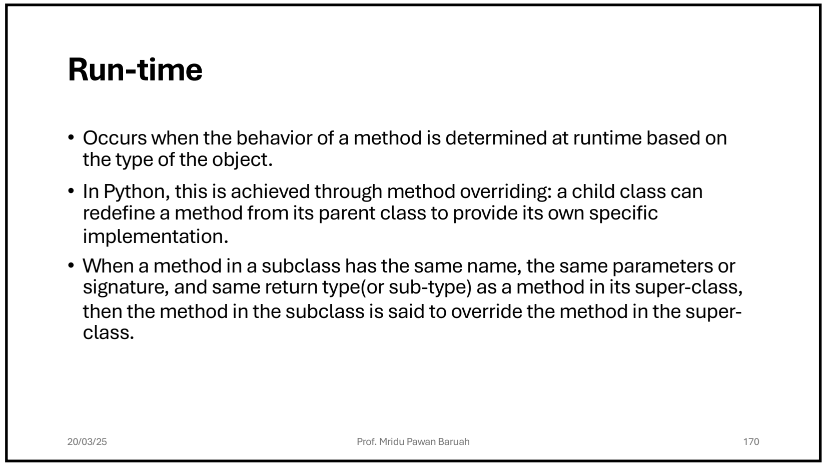
168



Polymorphism

- The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.
- An example of a Python function that can be used on different objects is the len() function.
  - For strings len() returns the number of characters.
  - For tuples len() returns the number of items in the tuple.
  - For dictionaries len() returns the number of key/value pairs in the dictionary
- There are two types of polymorphism
  - Compile- time polymorphism
  - Run-time polymorphism

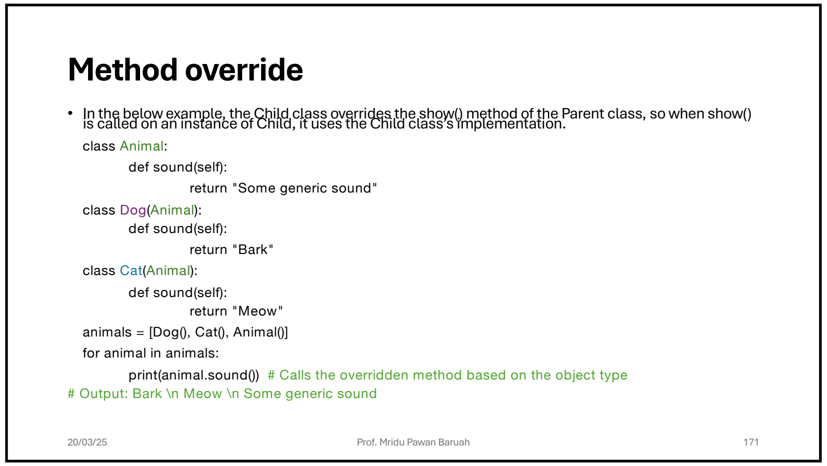
169



Run-time

- Occurs when the behavior of a method is determined at runtime based on the type of the object.
- In Python, this is achieved through method overriding: a child class can redefine a method from its parent class to provide its own specific implementation.
- When a method in a subclass has the same name, the same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

170

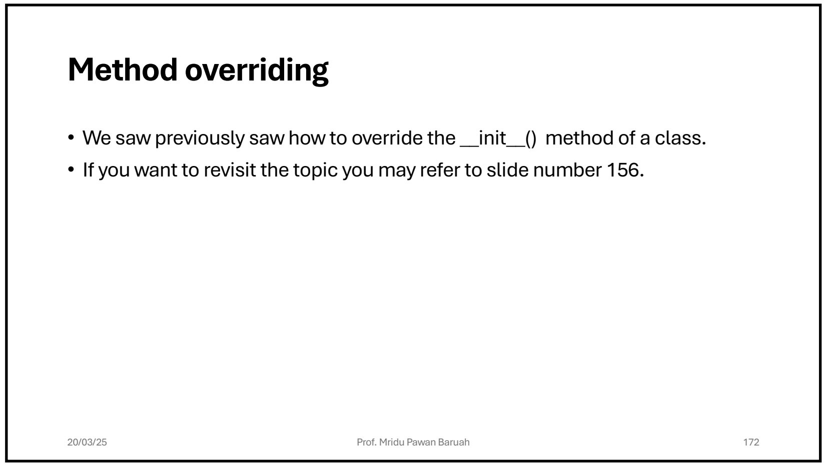


Method override

- In the below example, the Child class overrides the show() method of the Parent class, so when show() is called on an instance of Child, it uses the Child class's implementation.

```
class Animal:
    def sound(self):
        return "Some generic sound"
class Dog(Animal):
    def sound(self):
        return "Bark"
class Cat(Animal):
    def sound(self):
        return "Meow"
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.sound())  # Calls the overridden method based on the object type
# Output: Bark \n Meow \n Some generic sound
```

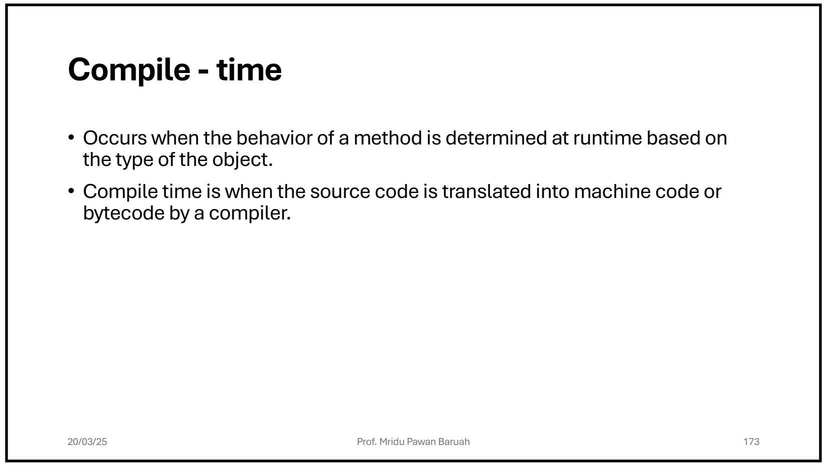
171



Method overriding

- We saw previously saw how to override the __init__() method of a class.
- If you want to revisit the topic you may refer to slide number 156.

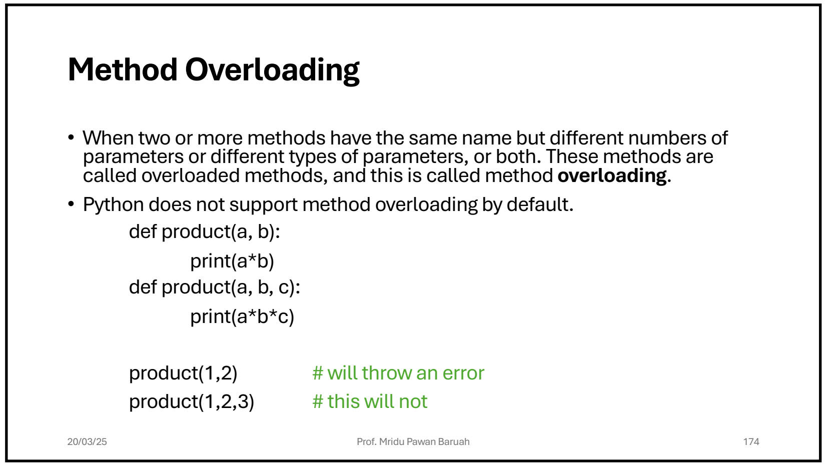
172



Compile - time

- Occurs when the behavior of a method is determined at runtime based on the type of the object.
- Compile time is when the source code is translated into machine code or bytecode by a compiler.

173



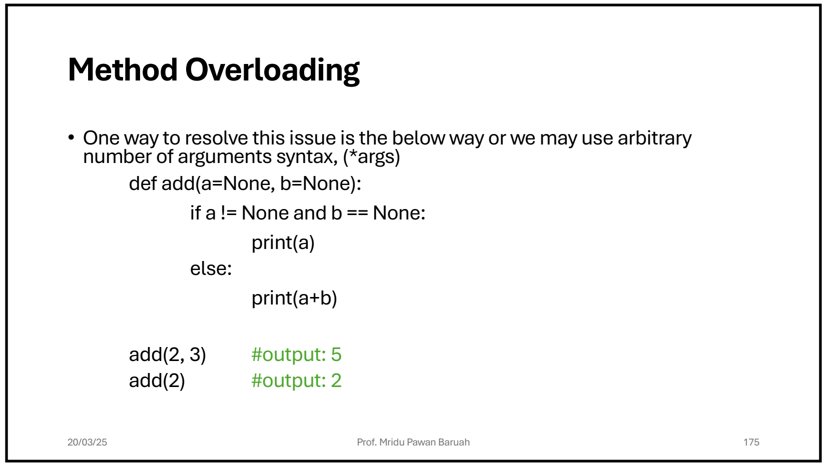
Method Overloading

- When two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods, and this is called method **overloading**.
- Python does not support method overloading by default.

```
def product(a, b):
    print(a*b)
def product(a, b, c):
    print(a*b*c)

product(1,2)        # will throw an error
product(1,2,3)      # this will not
```

174



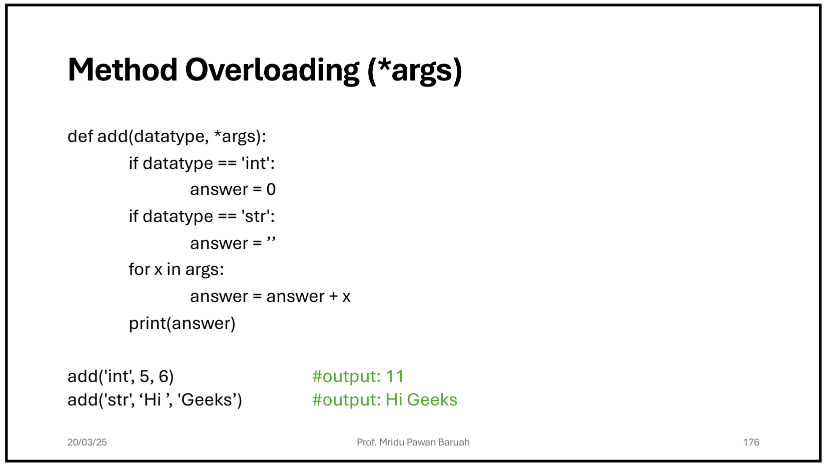
Method Overloading

- One way to resolve this issue is the below way or we may use arbitrary number of arguments syntax, (*args)

```
def add(a=None, b=None):
    if a != None and b == None:
        print(a)
    else:
        print(a+b)

add(2, 3)   #output: 5
add(2)      #output: 2
```

175

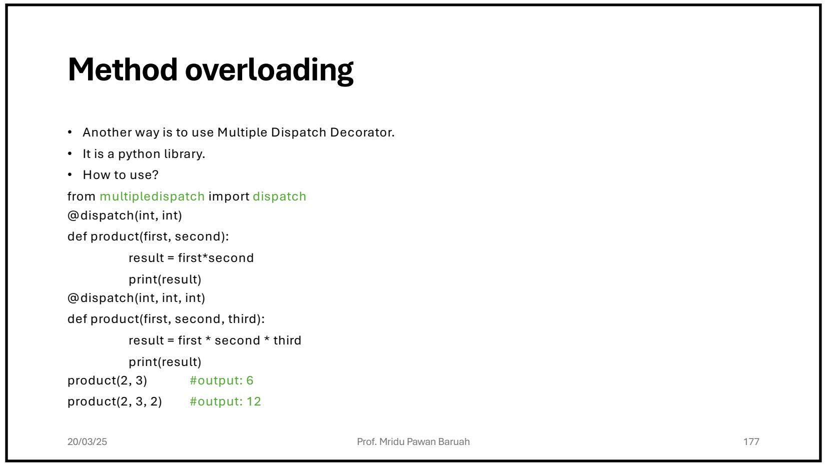


Method Overloading (*args)

```
def add(datatype, *args):
    if datatype == 'int':
        answer = 0
    if datatype == 'str':
        answer = ’’
    for x in args:
        answer = answer + x
    print(answer)

add('int', 5, 6)              #output: 11
add('str', ‘Hi ’, 'Geeks’)    #output: Hi Geeks
```

176



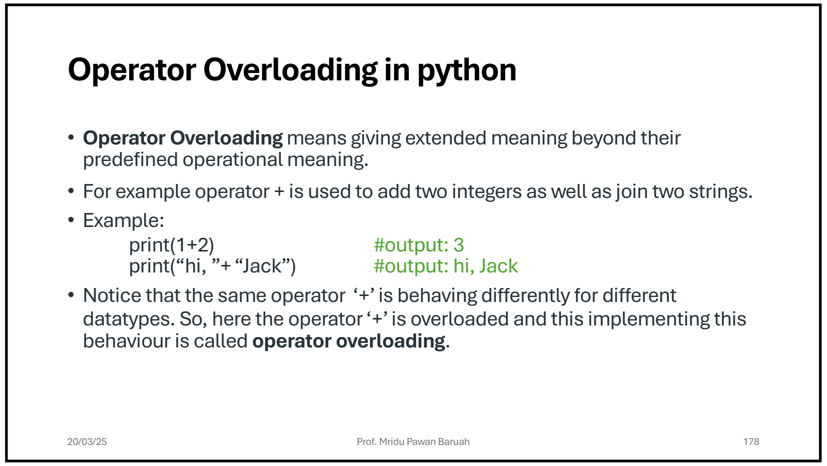
Method overloading

- Another way is to use Multiple Dispatch Decorator.
- It is a python library.
- How to use?

```
from multipledispatch import dispatch
@dispatch(int, int)
def product(first, second):
            result = first*second
            print(result)
@dispatch(int, int, int)
def product(first, second, third):
            result = first * second * third
            print(result)
product(2, 3)        #output: 6
product(2, 3, 2)     #output: 12
```

20/03/25 Prof. Mridu Pawan Baruah 177

177



Operator Overloading in python

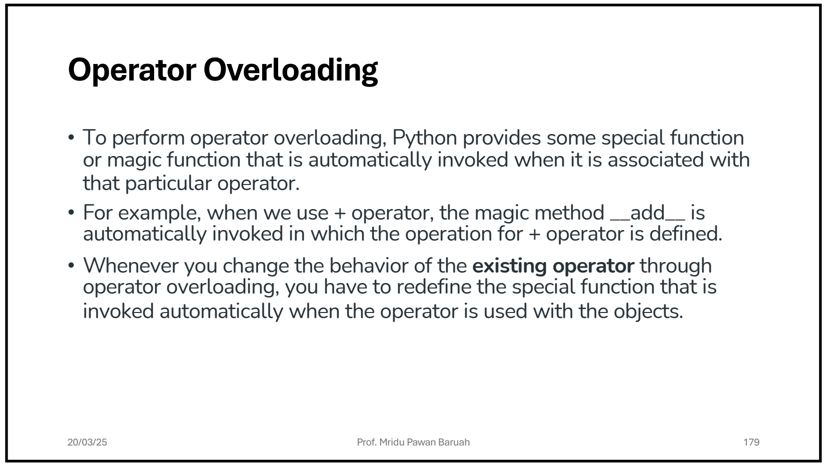
- **Operator Overloading** means giving extended meaning beyond their predefined operational meaning.
- For example operator + is used to add two integers as well as join two strings.
- Example:

```
        print(1+2)                    #output: 3
        print(“hi, ”+ “Jack”)         #output: hi, Jack
```

- Notice that the same operator ‘+’ is behaving differently for different datatypes. So, here the operator ‘+’ is overloaded and this implementing this behaviour is called **operator overloading**.

20/03/25 Prof. Mridu Pawan Baruah 178

178

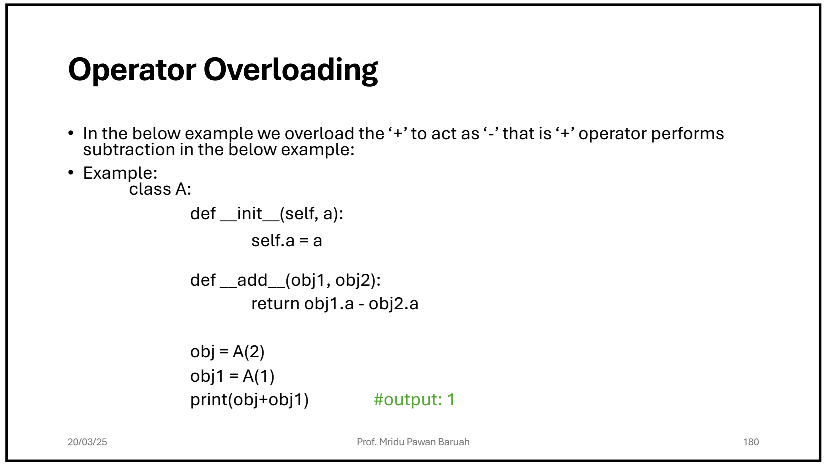


Operator Overloading

- To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator.
- For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.
- Whenever you change the behavior of the **existing operator** through operator overloading, you have to redefine the special function that is invoked automatically when the operator is used with the objects.

20/03/25 Prof. Mridu Pawan Baruah 179

179



Operator Overloading

- In the below example we overload the ‘+’ to act as ‘-’ that is ‘+’ operator performs subtraction in the below example:
- Example:

```
        class A:
                def __init__(self, a):
                        self.a = a

                def __add__(obj1, obj2):
                        return obj1.a - obj2.a

                obj = A(2)
                obj1 = A(1)
                print(obj+obj1)          #output: 1
```

20/03/25 Prof. Mridu Pawan Baruah 180

180

Operator Overloading

- When we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined. Thereby changing this magic method's code, we can give extra meaning to the + operator.

Operator	Magic Method
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

181

Operator	Magic Method
-=	__isub__(self, other)
+=	__iadd__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
%=	__imod__(self, other)
**=	__ipow__(self, other)
>>=	__irshift__(self, other)
<<=	__ilshift__(self, other)
&=	__iand__(self, other)
|=	__ior__(self, other)
^=	__ixor__(self, other)

Operator	Magic Method
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

182

TODO

- Create a class Greeter with a method greet(name) that prints a greeting for the provided name.

  example: If the user enters "Jack" you're to print "Namaskar Jack!" using a class method.
- Develop a class Calculator with methods to add and multiply two numbers.

  Name the class as "Calculator", it should implement two functions multiplication and addition.

183

TODO

- Given a number x, determine whether the given number is Armstrong number or not.
  An Armstrong number (also known as a narcissistic number) is a number that is equal to the sum of its digits each raised to the power of the number of digits in the number.

  e.g.: x =153
  1^3+5^3+3^3 = 153, hence 153 is an Armstrong number.

- Implement a Python program to calculate the square root of a number using the "*math"* library and display the current working directory using the "os" library.

184