Programming in Python

Dr Malu G
Kerala University of Digital Sciences, Innovation and Technology

DIGITAL UNIVERSITY KERALA
Curating a responsible digital world

What is a Program?

- A program is a group of logical, mathematical and sequential functions grouped together.
- When they are grouped, these functions perform a task.
- Each programming language focuses on different types of tasks as well as gives commands to the machine in different ways.

What Is a Class?

- In computer programming, a class contains a group of instructions that act as commands for the computer.
- The class is made up of variables, integers, decimals and other symbols.
- These are put together in certain orders to let the computer know what task to perform.

What Is a Function?

- Even if you are new to computer programming, you are familiar with functions. If you use an online music streaming program, you press the button to start or pause the play. Those are functions.
- When classes of programming languages are grouped together, they create functions.
- These functions allow you to perform certain tasks in a program.
- Some functions are small and control just one aspect of a piece of software or program.
- Other functions are big and ensure that the program itself runs.

What Is a Command?

- Commands are the methods to control certain aspects of the program or machine.
- Programming languages use classes and functions that control commands.
- The reason that programming is so important is that it directs a computer to complete these commands over and over again, so people do not have to do the task repeatedly.
- Instead, the software can do it automatically and accurately.

Object Oriented Programming Using Python

Index

Procedure Oriented Approach

- Procedure of functions to perform a task
- Main task is composed of several procedures and functions

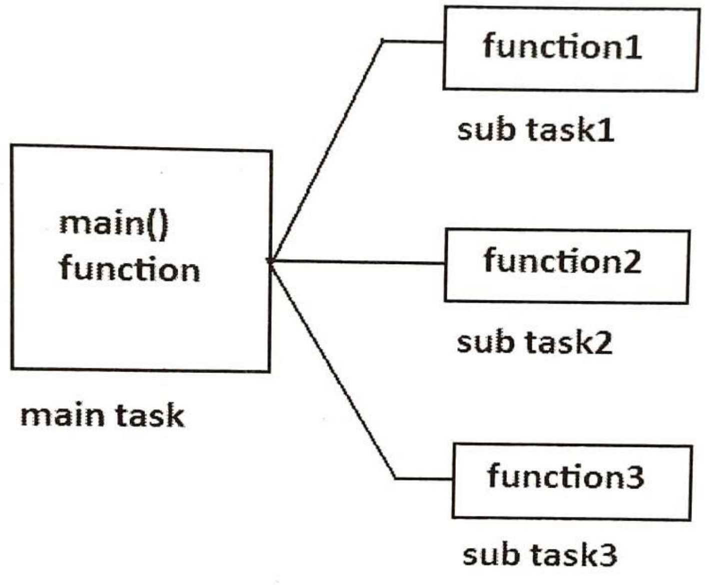


Figure 12.1: Procedure Oriented Approach

Object-Oriented Approach

- Class is a module which itself contains data and Methods(functions) to achieve a task.
- The main task is divided into several sub tasks, and these are represented as classes.
- Each class can perform several inter-related tasks for which several methods are written in a class.
- This approach is called object oriented approach.

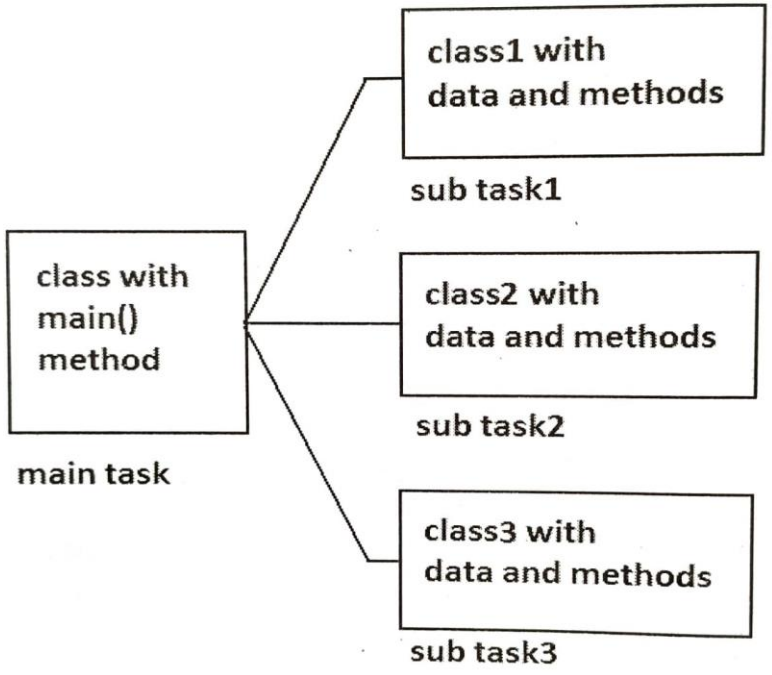

class with main() method
main task
class1 with data and methods
sub task1
class2 with data and methods
sub task2
class3 with data and methods
sub task3

Figure 12.2: Object Oriented Approach

Difference between Object-Oriented and Procedural Oriented Programming

Object-Oriented Programming (OOP)	**Procedural-Oriented Programming (Pop)**
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of Access modifiers 'public', private', protected'	Doesn't use Access modifiers
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

Problems in Procedure oriented Approach

- A new task every time requires developing the code from the scratch.
- Every task and subtask is required as a function and one function may depend on another function
- Error in the software needs examination of all the functions
- Updations to the software will also be difficult
- Lines of codes

Introduction to Object Oriented Programming in Python

- Object Oriented Programming is a way of computer programming using the idea of “objects” to represents data and methods.
- It is also, an approach used for creating neat and reusable code instead of a redundant one.

What are Classes and Objects?

- A class is a collection of objects or you can say it is a blueprint of objects defining the common attributes and behavior.
- Now the question arises, how do you do that?
- Class is defined under a “class” Keyword.

Example:

```
class class1(): // class 1 is the name of the class
```

Example

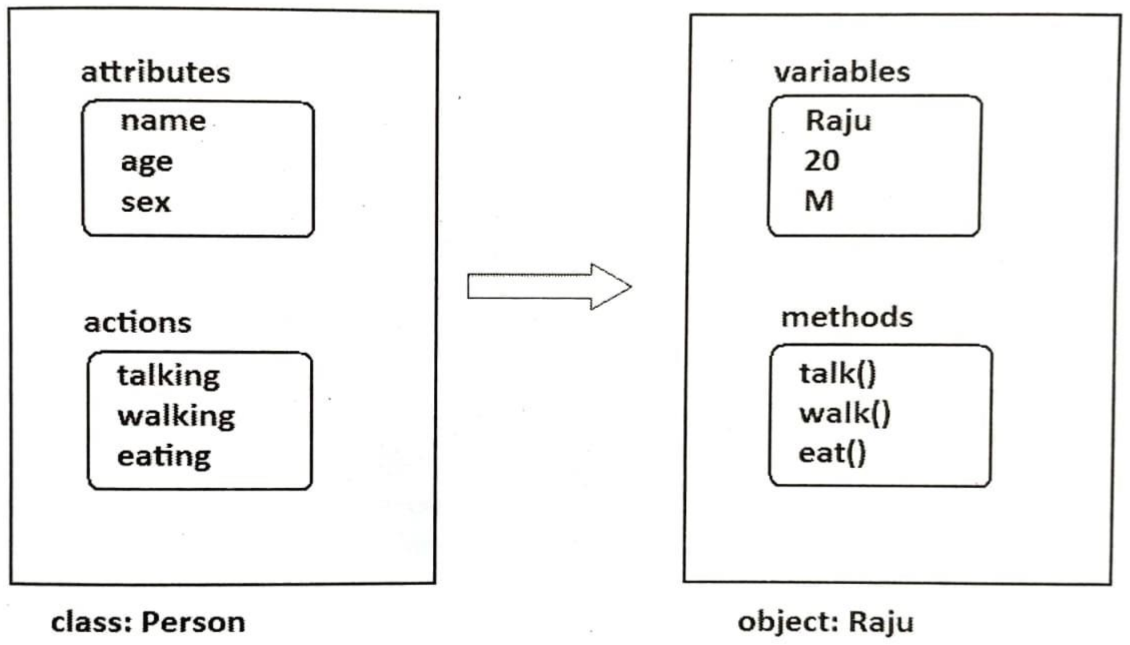


Figure 12.3: Person Class and Raju Object

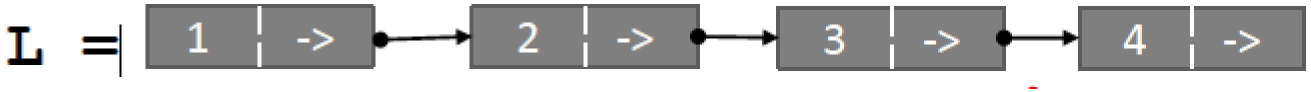
Creating an Object and Class in python:

Example:

```
# This is a class
Class Person:
# attributes means variables
Name=‘Raju’
Age=20
#actions means functions
def talk(cls);
      print (cls.name);
      print (cls.age);
```

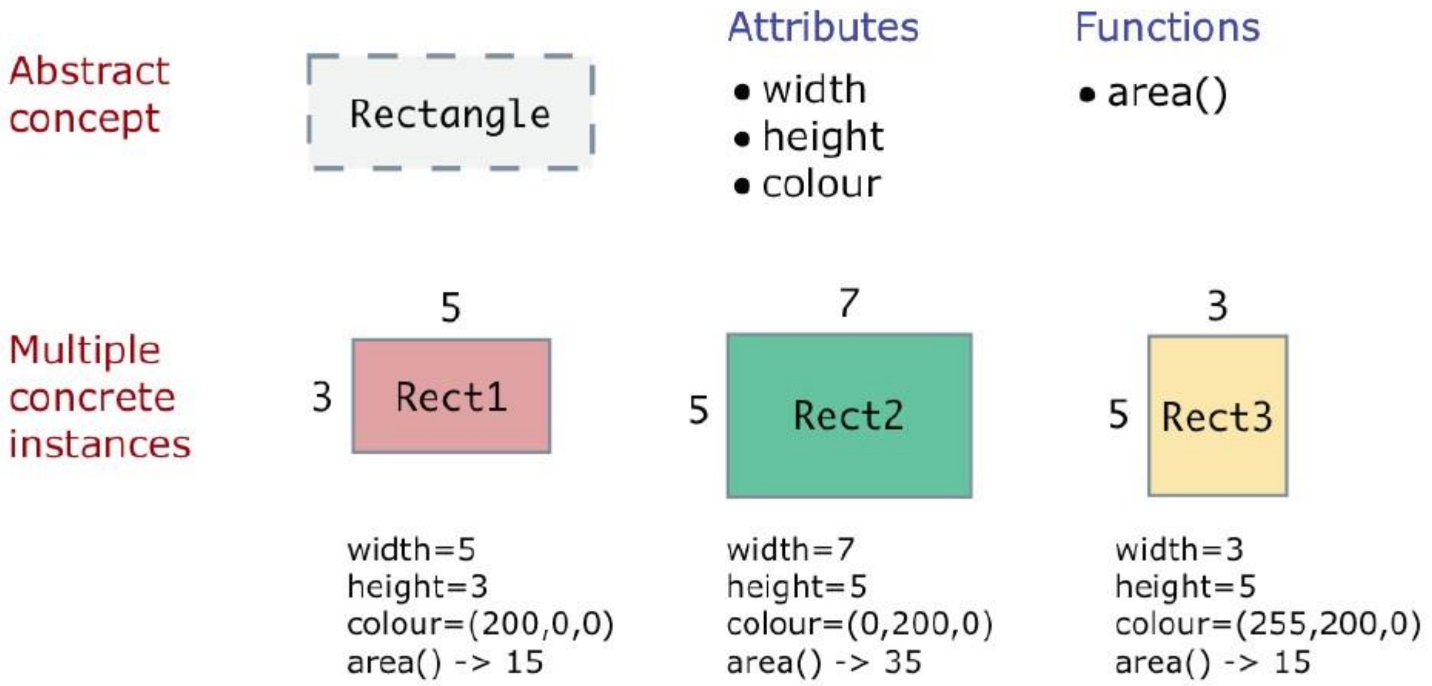
Example: List

- lists are internally represented as linked list
- [1,2,3,4]:



- manipulation of lists
  - L[i], L[i:j], +
  - len(), min(), max(), del(L[i])
  - L.append(), L.extend(), L.remove(), L.reverse()…

Classes


Abstract concept
Rectangle
Attributes
• width
• height
• colour
Functions
• area()
Multiple concrete instances
5
3
Rect1
7
5
Rect2
3
5
Rect3
width=5
height=3
colour=(200,0,0)
area() -> 15
width=7
height=5
colour=(0,200,0)
area() -> 35
width=3
height=5
colour=(255,200,0)
area() -> 15

Example

```
# A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Nikhil')
p.say_hi()
```

Output:

```
Hello, my name is Nikhil
```

Example

```
# A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

 # Creating different objects
p1 = Person('Nikhil')
p2 = Person('Abhinav')
p3 = Person('Anshul')

p1.say_hi()
p2.say_hi()
p3.say_hi()
```

Output:

```
Hello, my name is Nikhil
Hello, my name is Abhinav
Hello, my name is Anshul
```

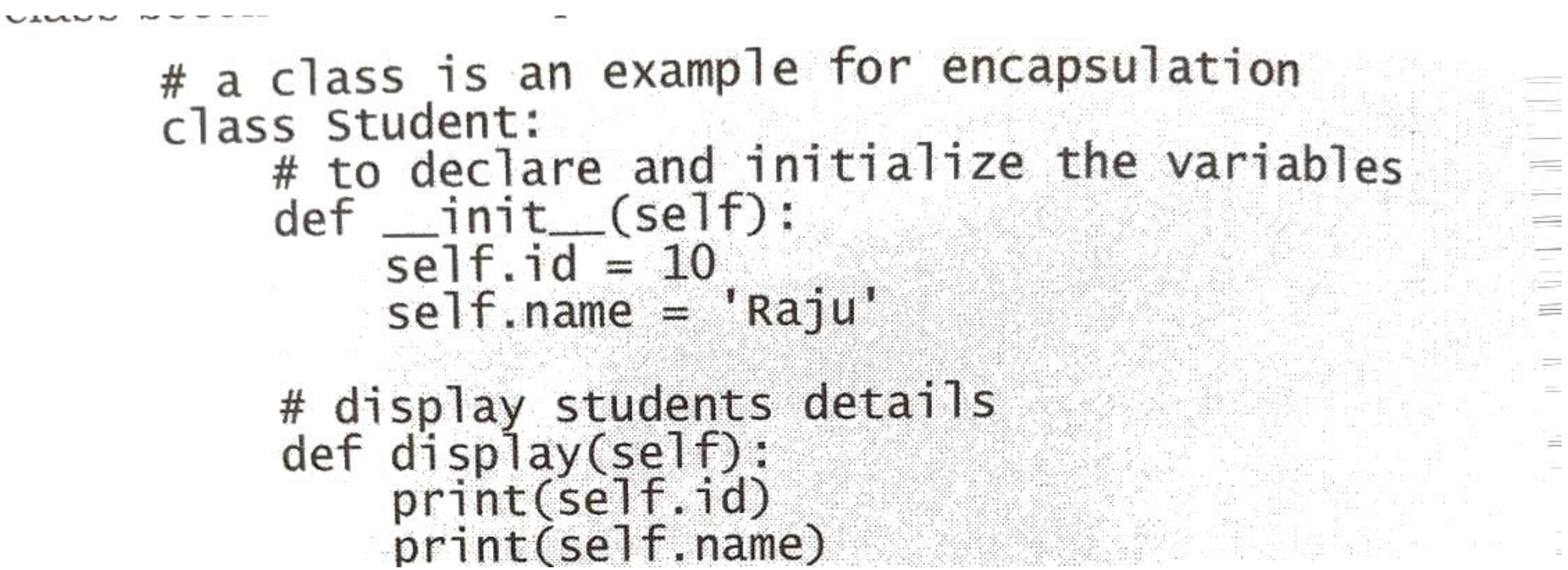
Object-Oriented Programming methodologies:

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Encapsulation

- Encapsulation is a mechanism where the data(variable) and the code (methods) that act on the data will bind together.
- Class is an example for encapsulation
- Variables and methods of the class are called ‘Members’ of the class.
- Methods are public by default in Python
- Encapsulation isolates the members of a class from the members of another class.

Encapsulation in Python



```
# a class is an example for encapsulation
class Student:
    # to declare and initialize the variables
    def __init__(self):
        self.id = 10
        self.name = 'Raju'

    # display students details
    def display(self):
        print(self.id)
        print(self.name)
```

Creating an Object in Python

- The class object could be used to access different attributes
- It can also be used to create new object instances (instantiation) of that class.

  The procedure to create an object is similar to a function call.

  ```
  harry = Person()
  ```

  This will create a new object instance named harry. We can access the attributes of objects using the object name prefix.
- Attributes may be data or method. Methods of an object are corresponding functions of that class.
- This means to say, since Person.greet is a function object (attribute of class), Person.greet will be a method object.

Creating an Object in Python

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello’)

# create a new object of Person class
harry = Person()
```

```
# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(harry.greet)

# Calling object's greet() method
# Output: Hello
harry.greet()
```

Constructors in Python

- Class functions that begin with double underscore __ are called special functions as they have special meaning.
- Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated.
- This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Constructors in Python

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')

# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)

# Call get_data() method
# Output: 2+3j
num1.get_data()
```

Constructors in Python

```
# Create another ComplexNumber object
# and create a new attribute 'attr'
num2 = ComplexNumber(5)
num2.attr = 10

# Output: (5, 0, 10)
print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
print(num1.attr)
```

Deleting Attributes and Objects

```
>>> num1 = ComplexNumber(2,3)
>>> del num1.imag
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.get_data
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'get_data'
```

Delete object

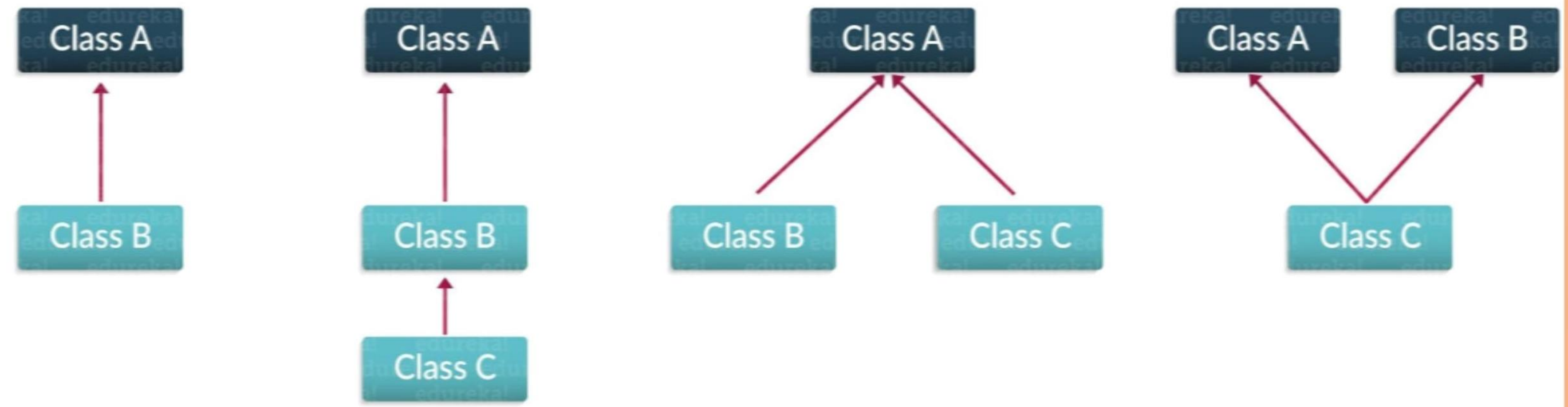
We can even delete the object itself, using the del statement.

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

Inheritance:

- Ever heard of this dialogue from relatives “you look exactly like your father/mother” the reason behind this is called ‘inheritance’.
- From the Programming aspect, It generally means “inheriting or transfer of characteristics from parent to child class without any modification”.
- The new class is called the derived/child class and the one from which it is derived is called a parent/base class.

Types Of Inheritance



Single Inheritance Multilevel Inheritance Hierarchical Inheritance Multiple Inheritance

Single Inheritance:

- Single level inheritance enables a derived class to inherit characteristics from a single parent class

Python Inheritance

- Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more
- Inheritance is a powerful feature in object oriented programming.
- It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Python Inheritance Syntax

```
class BaseClass:
        Body of base class
class DerivedClass(BaseClass):
        Body of derived class
```

- Derived class inherits features from the base class where new features can be added to it.
- This results in re-usability of code.

Example of Inheritance in Python

- To demonstrate the use of inheritance, let us take an example.
- A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

Example of Inheritance in Python

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides n and magnitude of each side as a list called sides.

Example of Inheritance in Python

- We don't need to define them again (code reusability). Triangle can be defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

Example of Inheritance in Python

- However, class Triangle has a new method findArea() to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

We can see that even though we did not define methods like inputSides() or dispSides() for class Triangle separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Example: Single

```
class employee1()://This is a parent class
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

class childemployee(employee1)://This is a child class
    def __init__(self, name, age, salary,id):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
emp1 = employee1('harshit',22,1000)
print(emp1.age)
```

Output: 22

Method Overriding in Python

- In the above example, notice that __init__() method was defined in both classes, Triangle as well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, __init__() in Triangle gets preference over the __init__ in Polygon.

- Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Polygon.__init__() from __init__() in Triangle).

- A better option would be to use the built-in function super(). So, super().__init__(3) is equivalent to Polygon.__init__(self,3) and is preferred.

Method Overriding in Python

- Two built-in functions isinstance() and issubclass() are used to check inheritances.
- The function isinstance() returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```

Method Overriding in Python

- Similarly, issubclass() is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

Python Multiple Inheritance

- A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.

- In multiple inheritance, the features of all the base classes are inherited into the derived class.

- The syntax for multiple inheritance is similar to single inheritance.

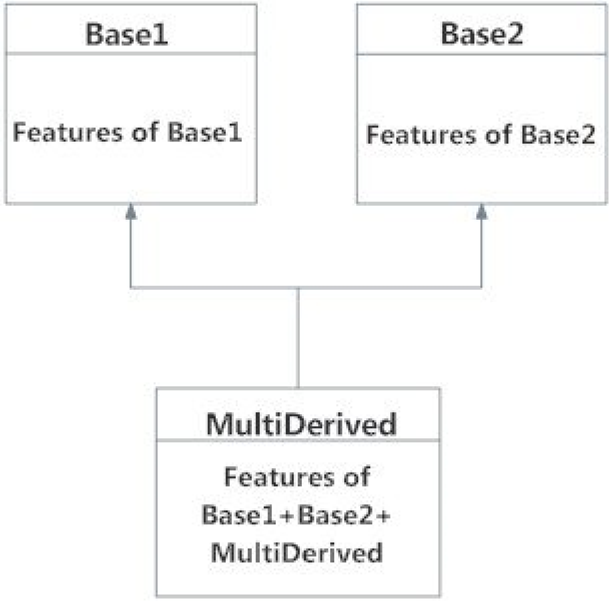
```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

Python Multiple Inheritance

Here, the MultiDerived class is derived from Base1 and Base2 classes.



Multiple Inheritance in Python

Python Multilevel Inheritance

- We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python.
- In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.
- An example with corresponding visualization is given below.

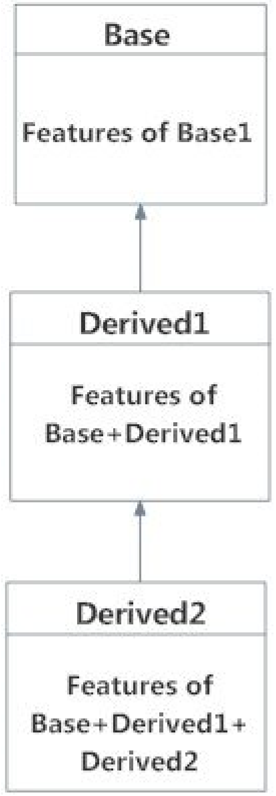
```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

Python Multilevel Inheritance

- Here, the Derived1 class is derived from the Base class, and the Derived2 class is derived from the Derived1 class.


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Multilevel Inheritance in Python

Method Resolution Order in Python

- Every class in Python is derived from the object class. It is the most base type in Python.
- So technically, all other classes, either built-in or user-defined, are derived classes and all objects are instances of the object class.
- In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice.

Method Resolution Order in Python

- So, in the above example of MultiDerived class the search order is [MultiDerived, Base1, Base2, object]. This order is also called linearization of MultiDerived class and the set of rules used to find this order is called Method Resolution Order (MRO).

- MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents. In case of multiple parents, the order is the same as tuples of base classes.

- MRO of a class can be viewed as the __mro__ attribute or the mro() method. The former returns a tuple while the latter returns a list.

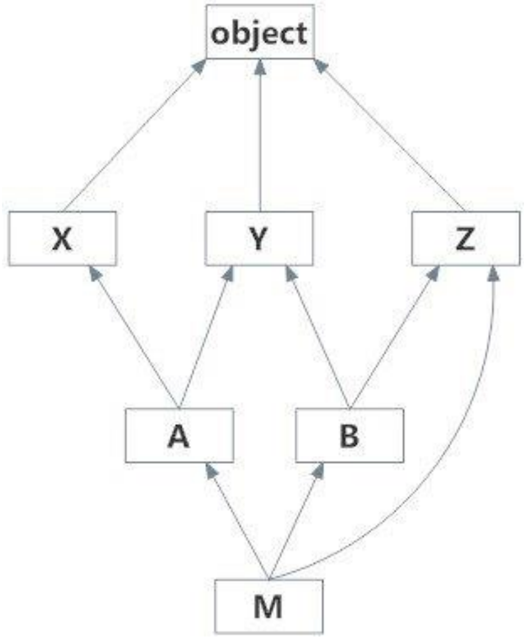
Method Resolution Order in Python

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>]
```

Method Resolution Order in Python

- Here is a little more complex multiple inheritance example and its visualization along with the MRO.



Method Resolution Order in Python

```
# Demonstration of MRO

class X:
    pass

class Y:
    pass

class Z:
    pass

class A(X, Y):
    pass

class B(Y, Z):
    pass

class M(B, A, Z):
    pass

print(M.mro())
```

Output

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.X'>, <class
'__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```

Python Operator Overloading

- You can change the meaning of an operator in Python depending upon the operands used.
- Python operators work for built-in classes.
- But the same operator behaves differently with different types.
  - For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.
- This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.
- So what happens when we use them with objects of a user-defined class?
- Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

Python Operator Overloading

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output

```
Traceback (most recent call last):
  File "<string>", line 9, in <module>
    print(p1+p2)
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Python Special Functions

- Class functions that begin with double underscore __ are called special functions in Python.
- These functions are not the typical functions that we define for a class. The __init__() function we defined above is one of them. It gets called every time we create a new object of that class.
- There are numerous other special functions in Python.
- Using special functions, we can make our class compatible with built-in functions.

Python Special Functions

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

Suppose we want the print() function to print the coordinates of the Point object instead of what we got. We can define a __str__() method in our class that controls how the object gets printed.

Let's look at how we can achieve this:

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
        def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```

Python Special Functions

Now let's try the print() function again.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0}, {1})".format(self.x, self.y)

p1 = Point(2, 3)
print(p1)
```

Output

```
(2, 3)
```

Overloading the + Operator

- To overload the + operator, we will need to implement __add__() function in the class.
- With great power comes great responsibility.
- We can do whatever we like, inside this function.
- But it is more sensible to return a Point object of the coordinate sum.

Overloading the + Operator

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

Now let's try the addition operation again:

Overloading the + Operator

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)

p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)
```

Output

```
(3,5)
```

Overloading the + Operator

- What actually happens is that, when you use p1 + p2, Python calls p1.__add__(p2) which in turn is Point.__add__(p1,p2). After this, the addition operation is carried out the way we specified.

- Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Overloading the + Operator

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading Comparison Operators

- Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

- Suppose we wanted to implement the less than symbol < symbol in our Point class.

- Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

Overloading Comparison Operators

```
# overloading the less than operator
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)
```

```
# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)
```

Output

```
True
False
False
```

Overloading Comparison Operators

- Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	**Expression**	**Internally**
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Polymorphism in Python

- The literal meaning of polymorphism is the condition of occurrence in different forms.

- Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

Example 1: Polymorphism in addition operator

Similarly, for string data types, + operator is used to perform concatenation.

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

- As a result, the above program outputs Python Programming.
- Here, we can see that a single operator + has been used to carry out different operations for distinct data types.
- This is one of the most simple occurrences of polymorphism in Python.

Function Polymorphism in Python

- There are some functions in Python which are compatible to run with multiple data types.
- One such function is the len() function. It can run with many data types in Python. Let's look at some example use cases of the function.
- Example 2: Polymorphic len() function

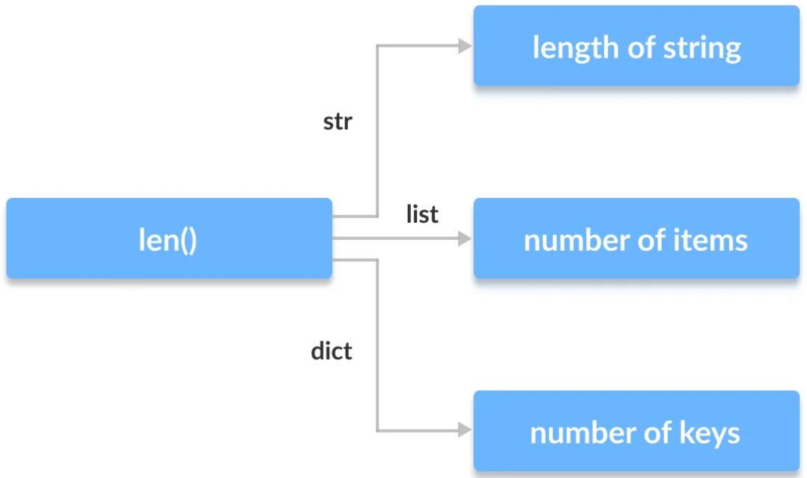
```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

  - Output

```
9
3
2
```

Function Polymorphism in Python

- Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the len() function.
- However, we can see that it returns specific information about specific data type



Polymorphism in len() function in Python

Class Polymorphism in Python

- We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

- We can then later generalize calling these methods by disregarding the object we are working with.

Polymorphism in Class Methods

- Here, we have created two classes Cat and Dog. They share a similar structure and have the same method names info() and make_sound().

- However, notice that we have not created a common superclass or linked the classes together in any way.

- Even then, we can pack these two different objects into a tuple and iterate through it using a common animal variable.

- It is possible due to polymorphism.

Polymorphism in Class Methods

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

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age}
years old.")

    def make_sound(self):
        print("Meow")

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

    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age}
years old.")
```

```
    def make_sound(self):
        print("Bark")


cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Polymorphism in Class Methods

- Here, we have created two classes Cat and Dog. They share a similar structure and have the same method names info() and make_sound().

- However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through it using a common animal variable. It is possible due to polymorphism.

Abstraction:

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that "what function does" but they don't know "how it does."

Why Abstraction is Important?

- In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity.
- It also enhances the application efficiency. Next, we will learn how we can achieve abstraction using the Python program.

Abstraction:

Suppose you booked a movie ticket from bookmyshow using net banking or any other process.
You don’t know the procedure of how the pin is generated or how the verification is done.

This is called ‘abstraction’ from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user.