
Department of Computer Science and Engineering
PES University, Bangalore, India

# Lecture Notes
# Python for Computational Problem Solving
# UE23CS151A

**Lecture #101**

***Inheritance***

**By,**
**Prof. Sindhu R Pai,**
**Anchor, PCPS - 2023**
**Assistant Professor**
**Dept. of CSE, PESU**
**&**
**Dr. Ramya C**
**Associate Professor**
**Dept. of CSE, PESU**

**Many Thanks to**
**Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)**
**Prof. Chitra G M(Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)**

# Introduction

**Obtaining or acquiring the properties of another class is known as Inheritance**. It deals with the ability of a class to inherit members of another class as part of its own definition. The inheriting class is called a **subclass** (also "derived class" or "child class"), and the class inherited from is called the **superclass** (also "base class" or "parent class"). Super classes may themselves inherit from other classes, resulting in a hierarchy of classes.

## Benefits/Advantages:

- It represents **real-world relationships** between the types.
- It provides the **reusability of a code.** Also, it allows to add more features to a class without modifying it.
- It offers a **simple, understandable model structure.**
- It is **transitive in nature**, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
- **Less development and maintenance expenses**

## Relationship between classes:

- **Is – a Relation:** One class is the parent of other class or classes. When this kind of specialization occurs, there are **three ways that parent and child can interact**.
  1. Action on child **imply** an action on the parent.
  2. Action on the child **overrides** the action on the parent.
  3. Action on the child **alters** the action on the parent.
- **Has – a Relation:** One class contains the object of other class as its attribute.

## "Is a" Relationship"

**Types:**

1. Single level inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

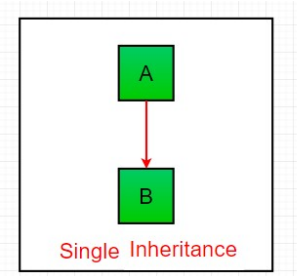
Let us try understanding these one by one.

**Single level inheritance:**

One class inherits the features of another class. In the diagram given below, B is the child class which inherits all the properties of class A.

**Syntax:**

```
class BaseClass1:
        #Body of base class
class DerivedClass(BaseClass1):
        #Body of derived - class
```




**Example_code_1: Action on the child imply an action on the parent.**

```
class A:
        def disp(self):
                print("in disp A")
class B(A):
        pass
a1=A()
a1.disp()
b1=B()
b1.disp()  #Action on the child
```

```
C:\Users\Dell>python classtest.py
in disp A
in disp A
```

**Example_code_2: Action on the child overrides the action on the parent.**

```
class A:
        def disp(self):
                print("in disp A")
class B(A):
```

```
        def disp(self):
                print("in disp B")
a1=A()
a1.disp()
b1=B()
b1.disp()  #Action on the child
```

```
C:\Users\Dell>python classtest.py
in disp A
in disp B
```

**Example_code_3: Action on the child alters the action on the parent.**

```
class A:
        def disp(self):
                print("in disp A")
class B(A):
        def disp(self):
                A.disp(self)
                print("in disp B")
a1=A()
a1.disp()
b1=B()
b1.disp()
```

```
C:\Users\Dell>python classtest.py
in disp A
in disp A
in disp B
```

## Usage of super():

A built-in function super() provides **a way to access methods and properties from a parent or superclass within a subclass**. It is commonly used in inheritance to call methods or access attributes from the parent class. A subclass can override methods or attributes from its superclass. However, there might be situations where you want to use the overridden method as well as the functionality of the parent method within the overridden method of the subclass. That's where **super()** becomes helpful.

**Example_code_4: Action on the child alters the action on the parent using super()**

```
class A:
        def disp(self):
                print("in disp A")
class B(A):
        def disp(self):
                super().disp()
                print("in disp B")
a1=A()
a1.disp()
b1=B()
b1.disp()
```

```
C:\Users\Dell>python classtest.py
in disp A
in disp A
in disp B
```

Note: There is only one parent available. So super() will refer to that class and calls disp()

**Example_code_5:** Program to demonstrate single level inheritance with constructors

```
#Base class
class Person:
        def __init__(self, name, id_no):
                self.name = name
                 self.id_no = id_no
        def display(self):
                print(self.name, self.id_no)
#child class
class Stud(Person):
        def Print(self):
                print("stud class(Base
class) called")
#client code
st1 = stud("Madan", 103)    # calls the constructor of parent implicitly
# Calling child class function
st1.Print()
# calling parent class function
st1.Display()
```

```
= RESTART: C:\Users\cramy\AppDat
y
Akash 1001
stud class(Base class) called
Madan 103
>>
```

**Example_code_6: Program to demonstrate the usage of parent constructors from the child**

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

 class Employee(Person):
   def __init__(self, name, idnumber, salary, desgn):
         self.salary = salary
         self.desgn = desgn
         Person.__init__(self, name, idnumber)
 emp = Employee('Riya', 802, 50000, "Admin")
 emp.display()
```

```
>>>
= RESTART: C:\Users\cram
y
Riya
802
>>>
```

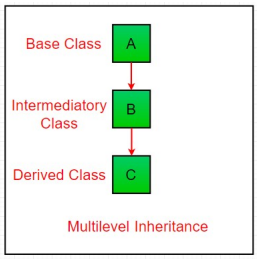
**Think! -> Can the bold line in the above example be replaced with super().__init__(name, idnumber) ?**

**Multi Level inheritance:**

One class say B inherits from the other class say A. Other class say C, inherits from class B forming a chain of classes. **At any point of time, direct parent is only of one type.**

**Syntax:**

```
class class1:
        <class-suite>
class class2(class1):
        <class suite>
class class3(class2):
        <class suite>
```




**Example_code_7: Program to understand how each subclass inherits attributes and methods from its parent classes, allowing for specialization and modification**

```
class Shape:
   def __init__(self, name):
      self.name = name
   def info(self):
      return self.name

class Polygon(Shape):
   def __init__(self, name, sides):
      super().__init__(name)
      self.sides = sides
   def info(self):
         return f"A {self.name} is a polygon with {self.sides} sides."

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

class Quadrilateral(Polygon):
   def __init__(self, name):
         super().__init__(name, 4)

triangle = Triangle("Triangle")
print(triangle.info())
quadrilateral = Quadrilateral("Quadrilateral")
print(quadrilateral.info())
```

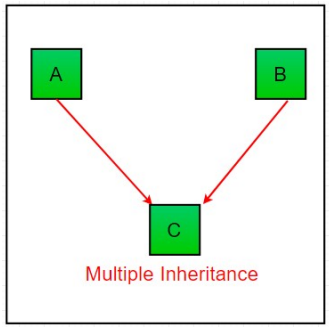
```
>>>
    = RESTART: C:\Users\cramy\AppData\Local\Pr
    y
    A Triangle is a polygon with 3 sides.
    A Quadrilateral is a polygon with 4 sides.
>>>
```

**Multiple inheritance:**

A class can have more than one super class and inherit the features from all parent classes.

**Syntax:**

```
class Base1:
   <class-suite>
class Base2:
   <class-suite>
. . .
class BaseN:
   <class-suite>
class Derived(Base1, Base2, ..... BaseN):
   <class-suite>
```




**Example_code_8:** Program to depict multiple inheritance

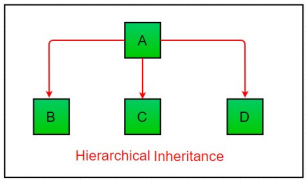
```
class Class2:
   def m(self):
      print("In Class2")
class Class3:
   def m(self):
      print("In Class3")
class Class4(Class2, Class3):
   pass

obj = Class4()
obj.m()
```

```
>>>
= RESTART: C:\User
y
In Class2
>>>
```

**If Class4 is declared as Class4(Class3, Class2), What is the output of code?**

**Hierarchical inheritance:** One class serves as super class for more than one sub classes.



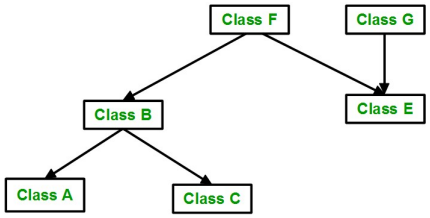
**Example_code_9:** Program to demonstrate Hierarchical inheritance.

```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")
# Derived class1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")
# Derived class2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

```
>>>
    = RESTART: C:\Users\cramy\AppData\
    y
    This function is in parent class.
    This function is in child 1.
    This function is in parent class.
    This function is in child 2.
>>>
```

**Hybrid inheritance**

A mix of two or more above types of inheritance. Also known as **Diamond shaped inheritance.**




## Few points to know in detail!

- **Who is the parent of all classes or types?**
- **Is there any way to list the parent/s of classes?**
- **If there are multiple objects involved in a code, can we check the type of object?**
- **If we use the type function, will it take of is-a relationship? If no, what is the other method? If yes, demo it.**
- **If multiple classes are involved, can we check "Is class B a child of class A"?**

Let us try to understand the above points in detail.

**Consider this user defined type,**

```
class Person:
        pass
class Student(Person):
        pass
class UGStudent(Student):
        pass
class PGStudent(Student):
        pass

#Creating 4 objects
p1 = Person()
s1 = Student()
u_s1 = UGStudent()
p_s1 = PGStudent()
```

```
#Printing the type of each object.
print(type(p1))
print(type(s1))
print(type(u_s1))
print(type(p_s1))
```

```
C:\Users\Dell>python classtest.py
<class '__main__.Person'>
<class '__main__.Student'>
<class '__main__.UGStudent'>
<class '__main__.PGStudent'>
```

type() returns the type of the object. It doesn't return the details about its parent object. If the object of child is created, the object of parent is created by default. How to get the details of parent types for a specific child type?

**__bases__** : An attribute on each class that can be used to obtain a list of classes the given class inherits. This is a **tuple.**

```
print(Person.__bases__)
print(Student.__bases__)
print(UGStudent.__bases__)
print(PGStudent.__bases__)
```

```
C:\Users\Dell>python classtest.py
(<class 'object'>,)
(<class '__main__.Person'>,)
(<class '__main__.Student'>,)
(<class '__main__.Student'>,)
```

Notice that object is the parent of all the classes. The type object is considered as the root of all types. This __bases__ attribute provides the details of immediate parent only. Is there any function which returns all the parent types for a given specific class? Use __mro__ in 3.10. OR use mro() from inspect module in other versions. This is also a **tuple.**

```
print(Person.__mro__)
print(Student.__mro__)
print(UGStudent.__mro__)
print(PGStudent.__mro__)
```

```
C:\Users\Dell>python classtest.py
(<class '__main__.Person'>, <class 'object'>)
(<class '__main__.Student'>, <class '__main__.Person'>, <class 'object'>)
(<class '__main__.UGStudent'>, <class '__main__.Student'>, <class '__main__.Person'>, <class 'object'>)

(<class '__main__.PGStudent'>, <class '__main__.Student'>, <class '__main__.Person'>, <class 'object'>)
```

Next, let us see in detail about isinstance() and issubclass().

**isinstance():**

**syntax: isinstance(instance, type)**

Returns True if the specified instance is of the specified type, otherwise returns False. If the type parameter is a tuple, this function will return True if the instance is one of the types in the tuple.

```
>>> isinstance(5, int)
True
>>> isinstance(5.7, int)
False
>>> isinstance("python", str)
True
```

```
>>> isinstance(["python",6], (int,str, float, list, tuple, set))
True
>>> isinstance(("python"), (int,str, float, list, tuple, set))
True
>>> isinstance(("python"), (int,tuple, float, list))
False
```

**The type 'object' is the parent of all classes.**

```
>>> isinstance(("python"), object)
True
>>> isinstance(["python", 6], object)
True
>>>
```

For the above Person, Student example code

```
print(isinstance(p1, object))
print(isinstance(s1, Person))
print(isinstance(u_s1, Person))
print(isinstance(p_s1, Person))
print(isinstance(p_s1, Student))
print(isinstance(p_s1, object))
print(isinstance(p_s1, float))
print(isinstance(s1, UGStudent))
```

```
C:\Users\Dell>python classtest.py
True
True
True
True
True
True
False
False
```

**Think about it -> What is the type(object) ?**

**issubclass()**

**syntax: issubclass(sub, sup)**

It returns True if the first class is the subclass of the second class, and False otherwise.

```
>>> issubclass(int, float)
False
>>> issubclass(int, object)
True
>>>
```

```
print(issubclass(Person, Student))
print(issubclass(Student, Person))
print(issubclass(UGStudent, Student))
print(issubclass(UGStudent, Person))
print(issubclass(PGStudent, Student))
print(issubclass(PGStudent, Person))
print(issubclass(PGStudent, UGStudent))
```

```
C:\Users\Dell>python classtest.py
False
True
True
True
True
True
False
```

## "Has-a" Relationship":

The requirement is to store the details of few persons and sort it in ascending order based on the age. Also sort it based on the sum of integers in date, month and year.

Person details include Name, age and date of birth. Name will be a string, age is a number and what about date of birth? Think!

Can we create a type called MyDate which contains date, month and year? Then can we build relation between these two classes but not parent –child relation.

Consider the type,

```
class MyDate:
   def __init__(self, d,m,y):
      self.dd = d
      self.mm = m
      self.yy = y

class Person:
    def __init__(self, n,a,d,m,y):
      self.name = n
      self.age = a
      self.dob = MyDate(d,m,y)  #Person has date of birth
```

```
#creating three objects of Person
p1 = Person("sindhu", 56, 2,4,1999)
p2 = Person("indhu", 53, 2,8,1919)
p3 = Person("bindhu", 51, 2,1,2000)
print(p1)
print(p2)
print(p3)
```

```
C:\Users\Dell>python classtest.py
<__main__.Person object at 0x0000022678994BD0>
<__main__.Person object at 0x0000022678994C50>
<__main__.Person object at 0x0000022678994CD0>
```

If we print the inbuilt object, we get the value.

```
>>> a = 5
>>>a
5
```

But if we print the user defined object, we get the output as above. Can we do something to get the value of a user defined object? We have been using the function str on different types like int. When we invoke this function, a special function __str__ of that class will be called. We can also provide such a function in our type to convert an object of our type to a string. **In the print context, __str__ function gets called automatically. Hence, Override __str__() in the user defined type. This function must always return a string.**

Let us add the definition of __str__ to our type Person.

```
class Person:
    def __init__(self, n,a,d,m,y):
      self.name = n
      self.age = a
      self.dob = MyDate(d,m,y)  #Person has date of birth
    def __str__(self):
      return self.name+" "+str(self.age)+" "+str(self.dob)
```

```
C:\Users\Dell>python classtest.py
sindhu 56 <__main__.MyDate object at 0x00000287BB914ED0>
indhu 53 <__main__.MyDate object at 0x00000287BB914F50>
bindhu 51 <__main__.MyDate object at 0x00000287BB914FD0>
```

But here, we did not add __str__ to our another type MyDate. Hence the output is as above.

Let us add __str__ in MyDate type too.

```
class MyDate:
   def __init__(self, d,m,y):
      self.dd = d
      self.mm = m
      self.yy = y
   def __str__(self):
      return str(self.dd)+" "+str(self.mm)+" "+str(self.yy)
```

```
C:\Users\Dell>python classtest.py
sindhu 56 2 4 1999
indhu 53 2 8 1919
bindhu 51 2 1 2000
```

Now let us see how to sort these objects as per our initial requirement.

Create a list containing these objects. Sort it directly. Either using sorted() or using sort() from list type. Will it sort? – No.

```
p_objects = [p1, p2, p3]
print(sorted(p_objects))
```

```
C:\Users\Dell>python classtest.py
Traceback (most recent call last):
  File "C:\Users\Dell\classtest.py", line 24, in <module>
    print(sorted(p_objects))
          ^^^^^^^^^^^^^^^^^
TypeError: '<' not supported between instances of 'Person' and 'Person'
```

How to compare Person objects, it is not defined. Hence the Error.

First requirement is sort p_objects list of persons based on the month. Add a method to extract month in Person type.

```
class Person:
    def __init__(self, n,a,d,m,y):
      self.name = n
      self.age = a
      self.dob = MyDate(d,m,y)  #Person has date of birth
    def __str__(self):
      return self.name+" "+str(self.age)+" "+str(self.dob)
    def get_month(self):
      return self.dob.mm
#client code
p_objects = [p1, p2, p3]
for obj in sorted(p_objects, key = Person.get_month):
   print(obj)
```

```
C:\Users\Dell>python classtest.py
bindhu 51 2 1 2000
sindhu 56 2 4 1999
indhu 53 2 8 1919
```

This works fine. But Rather than using self.dob.mm, can we call some function from MyDate type? Let us add a method getmonth in MyDate type.

```
class MyDate:
   def __init__(self, d,m,y):
      self.dd = d
      self.mm = m
      self.yy = y
   def __str__(self):
      return str(self.dd)+" "+str(self.mm)+
" "+str(self.yy)
   def getmonth(self):
      return self.mm
```

```
class Person:
    def __init__(self, n,a,d,m,y):
      self.name = n
      self.age = a
      self.dob = MyDate(d,m,y)
    def __str__(self):
      return self.name+" "+str(self.age)+"
"+str(self.dob)
    def get_month(self):
      return self.dob.getmonth()
```

**Note: In the output, there is no change.** Functions in Person type in turn invoke the functions of MyDate class through the attribute dob – this is called **delegation or forwarding.**

Could you try sorting this data in descending order? Only client code to be changed to below.

```
p_objects = [p1, p2, p3]
for obj in sorted(p_objects, key = Person.get_month, reverse = True):
    print(obj)
```

```
C:\Users\Dell>python classtest.py
indhu 53 2 8 1919
sindhu 56 2 4 1999
bindhu 51 2 1 2000
```

**Try this**! -> Now it is your job to start the code for requirement 2. Sorting the person objects based on the sum of date, month and year.

**Example_code_10: The type MyEvent uses the MyDate type. An Event occurs on a date. An event is not a date. Print the event details.**

```
class MyDate:
        def __init__(self, dd, mm, yy):
                self.dd = dd
                self.mm = mm
                self.yy = yy
        def __str__(self):
                return str(self.dd) + "-" + str(self.mm) + "-" + str(self.yy)
        def key(self):
                return self.yy * 365 + self.mm * 30 + self.dd

class MyEvent:
        def __init__(self, dd, mm, yy, detail):
                self.date = MyDate(dd, mm, yy)
                self.detail = detail
        def __str__(self):
                return str(self.date) + " => " + self.detail
        def key(self):
                return self.date.key()
e= MyEvent(15, 8, 1947, "Independence Day")
print(e)
```

```
>>>
    = RESTART: C:/Users/cramy/AppDa
position.py
15-8-1947 => Independence Day
>>>
```

**Few points to think!**

- **Given an object, can you find the class to which it belongs to?**

  **Use __class__.name__ on the object.**
- **There is a function repr very similar to str. What is the difference?**
- **Is super and init, a keyword?**
- **If there is is-a relation between two classes, and there are many objects created for parent and child classes. If we call del on child object, will be destructor of parent also called? If yes, answer this-> If explicit destructor call is not there on the child object, what is the sequence in which destructors will be called? Is there any sequence actually?**

**-END-**