Python Programming Language

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo | Machine Learning Researcher

spring 2019

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture

Inheritance
Function Overriding

Lecture Agenda

We will discuss in this lecture the following topics

1- Inheritance
2- Access Modifiers
3- Function Overriding
4- Multiple Inheritance
5- Composition Relationship
6- Aggregation Relationship

Let's STARTUP

Lecture Agenda

Section 1: Inheritance

Section 2: Access Modifiers

Section 3: Function Overriding

Section 4: Multiple Inheritance

Section 5: Composition Relationship

Section 6: Aggregation Relationship

Inheritance

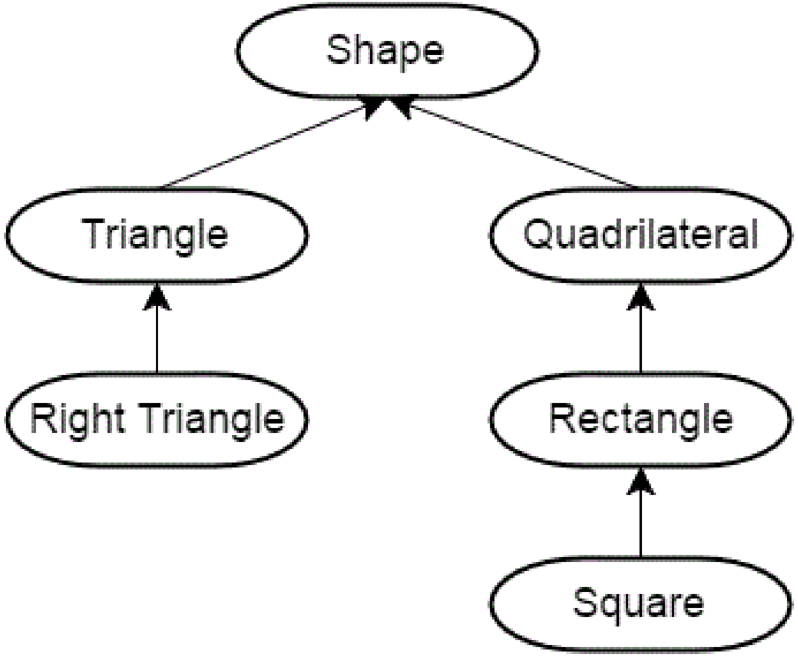
- Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more. In this article, you will learn to use inheritance in Python.

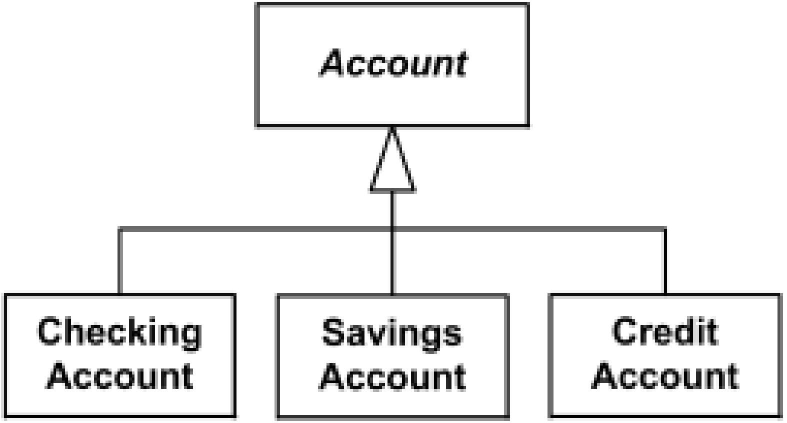
- Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

- One of the major advantages of Object Oriented Programming is re-use. Inheritance is one of the mechanisms to achieve the same. In inheritance, a class (usually called superclass) is inherited by another class (usually called subclass). The subclass adds some attributes to superclass.

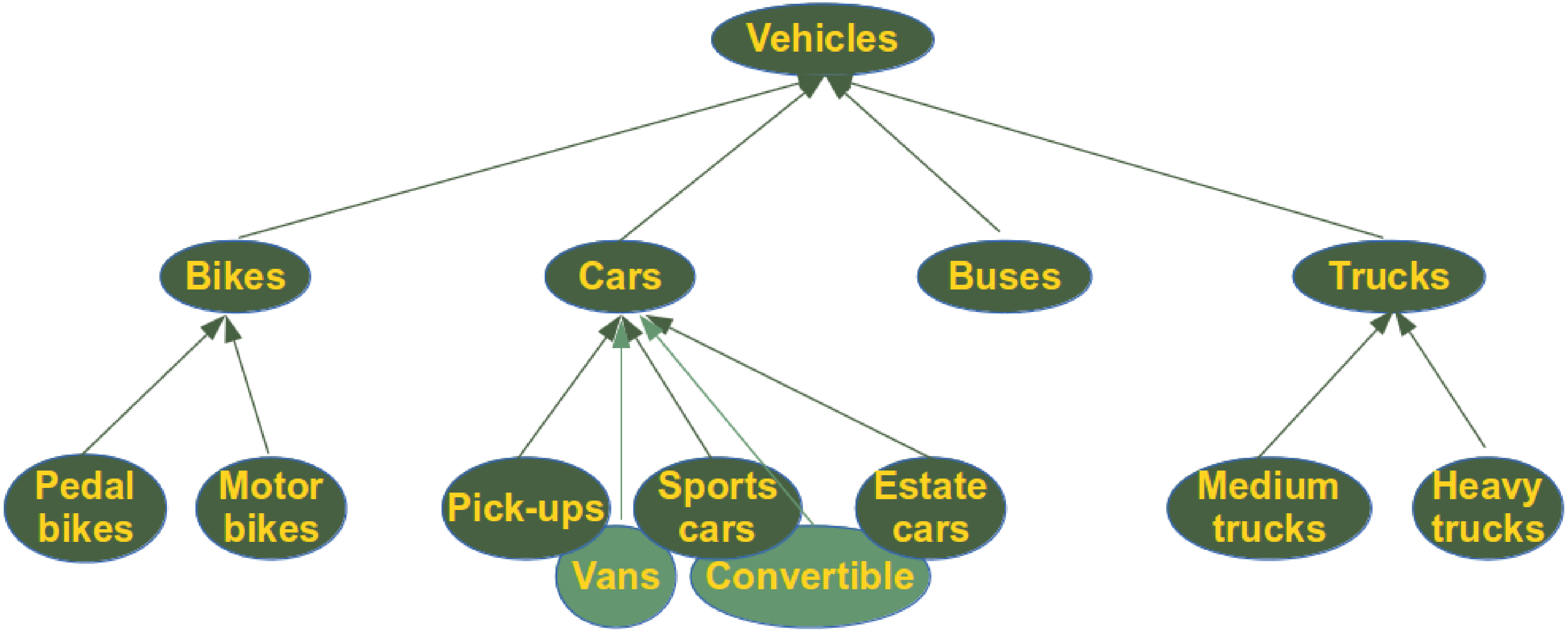
- We often come across different products that have a basic model and an advanced model with added features over and above basic model. A software modelling approach of OOP enables extending the capability of an existing class to build a new class, instead of building from scratch. In OOP terminology, this characteristic is called inheritance, the existing class is called base or parent class, while the new class is called child or sub class.

Inheritance





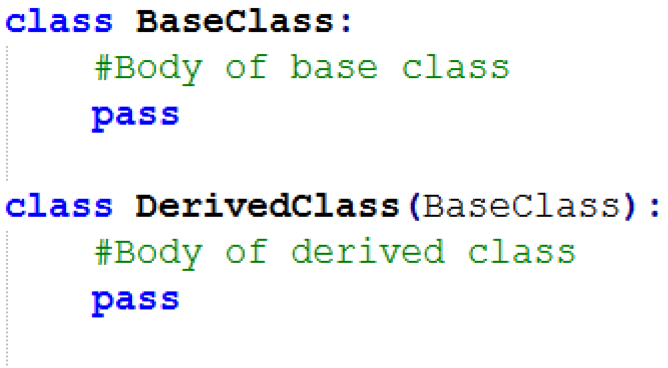
Inheritance



© Prepared by: Mohamed Ayman

Inheritance in Python

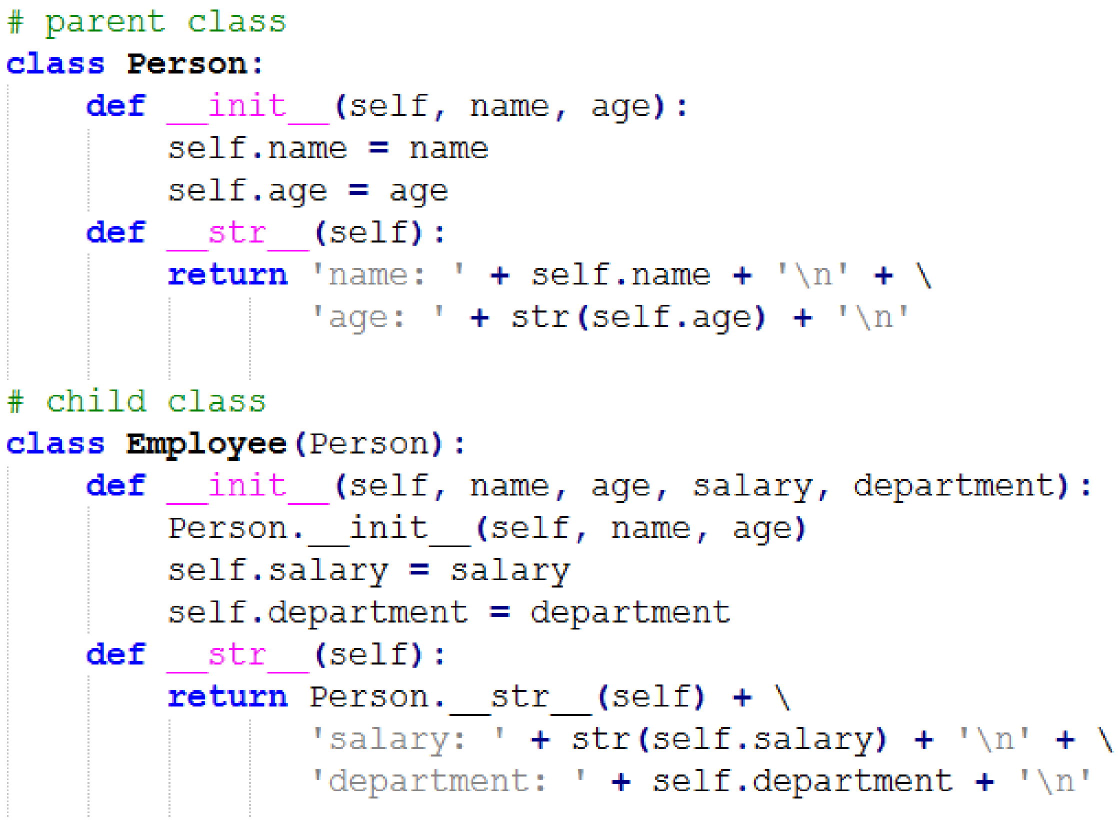
- One of the major advantages of Object Oriented Programming is re-use. Inheritance is one of the mechanisms to achieve the same. In inheritance, a class (usually called superclass) is inherited by another class (usually called subclass). The subclass adds some attributes to superclass.



```
class BaseClass:
    #Body of base class
    pass

class DerivedClass(BaseClass):
    #Body of derived class
    pass
```

- Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Inheritance in Python



```python
# parent class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return 'name: ' + self.name + '\n' + \
               'age: ' + str(self.age) + '\n'

# child class
class Employee(Person):
    def __init__(self, name, age, salary, department):
        Person.__init__(self, name, age)
        self.salary = salary
        self.department = department
    def __str__(self):
        return Person.__str__(self) + \
               'salary: ' + str(self.salary) + '\n' + \
               'department: ' + self.department + '\n'
```

Inheritance in Python

Example:

```
x = Person('Jack', 24)
print(x)

x = Employee('Robert', 26, 4000, 'IT')
print(x)
```

Output:

```
name: Jack
age: 24


name: Robert
age: 26
salary: 4000
department: IT
```

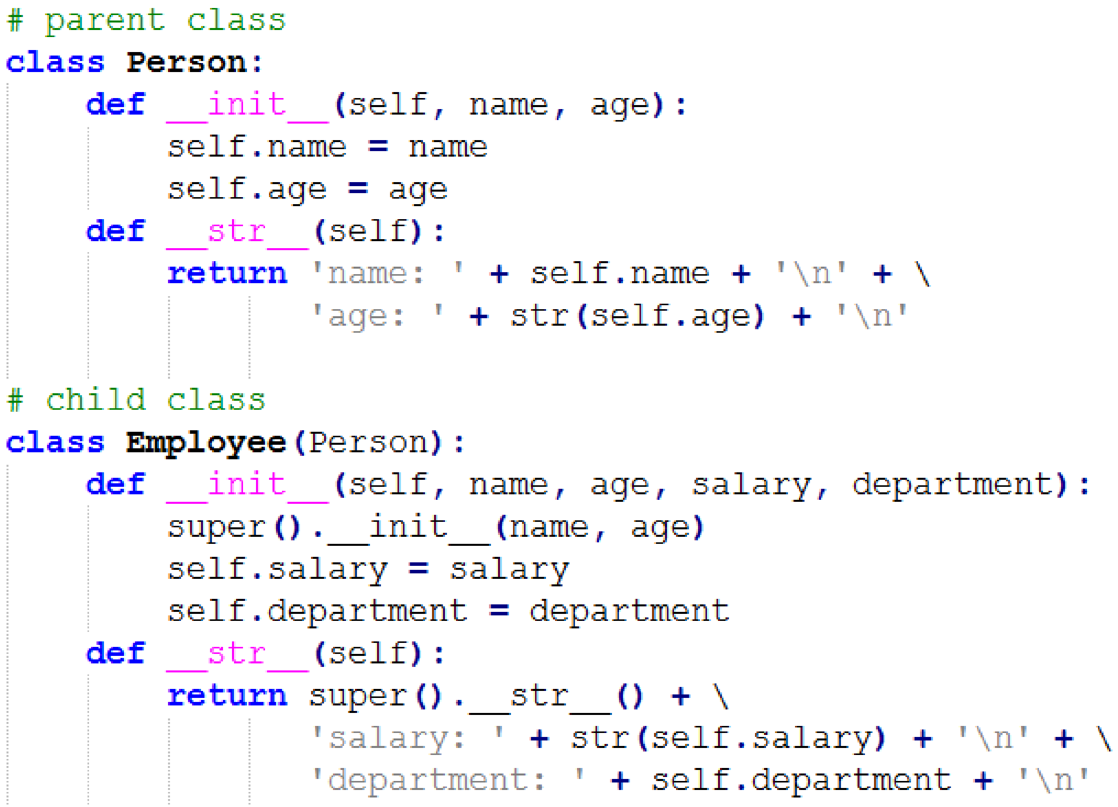
Super Keyword in Python

- While Python isn’t purely an object-oriented language, it’s flexible enough and powerful enough to allow you to build your applications using the object-oriented paradigm. One of the ways in which Python achieves this is by supporting inheritance, which it does with super().
- super() alone returns a temporary object of the superclass that then allows you to call that superclass’s methods.
- Why would you want to do any of this? While the possibilities are limited by your imagination, a common use case is building classes that extend the functionality of previously built classes.
- Calling the previously built methods with super() saves you from needing to rewrite those methods in your subclass, and allows you to swap out super classes with minimal code changes.

So what can super() do for you in single inheritance?

- Like in other object-oriented languages, it allows you to call methods of the superclass in your subclass. The primary use case of this is to extend the functionality of the inherited method.

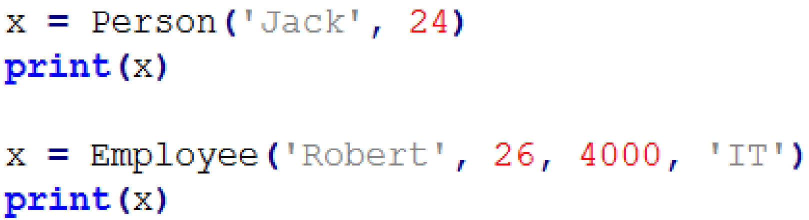
Super Keyword in Python



```python
# parent class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return 'name: ' + self.name + '\n' + \
               'age: ' + str(self.age) + '\n'

# child class
class Employee(Person):
    def __init__(self, name, age, salary, department):
        super().__init__(name, age)
        self.salary = salary
        self.department = department
    def __str__(self):
        return super().__str__() + \
               'salary: ' + str(self.salary) + '\n' + \
               'department: ' + self.department + '\n'
```

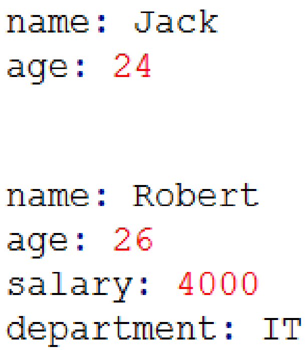
Super Keyword in Python

Example:



```
x = Person('Jack', 24)
print(x)

x = Employee('Robert', 26, 4000, 'IT')
print(x)
```

Output:



```
name: Jack
age: 24


name: Robert
age: 26
salary: 4000
department: IT
```

Lecture Agenda

✓ Section 1: Inheritance

Section 2: Access Modifiers

Section 3: Function Overriding

Section 4: Multiple Inheritance

Section 5: Composition Relationship

Section 6: Aggregation Relationship

Access Modifiers

- The access modifiers in Python are used to modify the default scope of variables. There are three types of access modifiers in Python: public, private, and protected.

- Variables with the public access modifiers can be accessed anywhere inside or outside the class, the private variables can only be accessed inside the class, while protected variables can be accessed within the same package.

- To create a private variable, you need to prefix double underscores with the name of the variable. To create a protected variable, you need to prefix a single underscore with the variable name. For public variables, you do not have to add any prefixes at all.

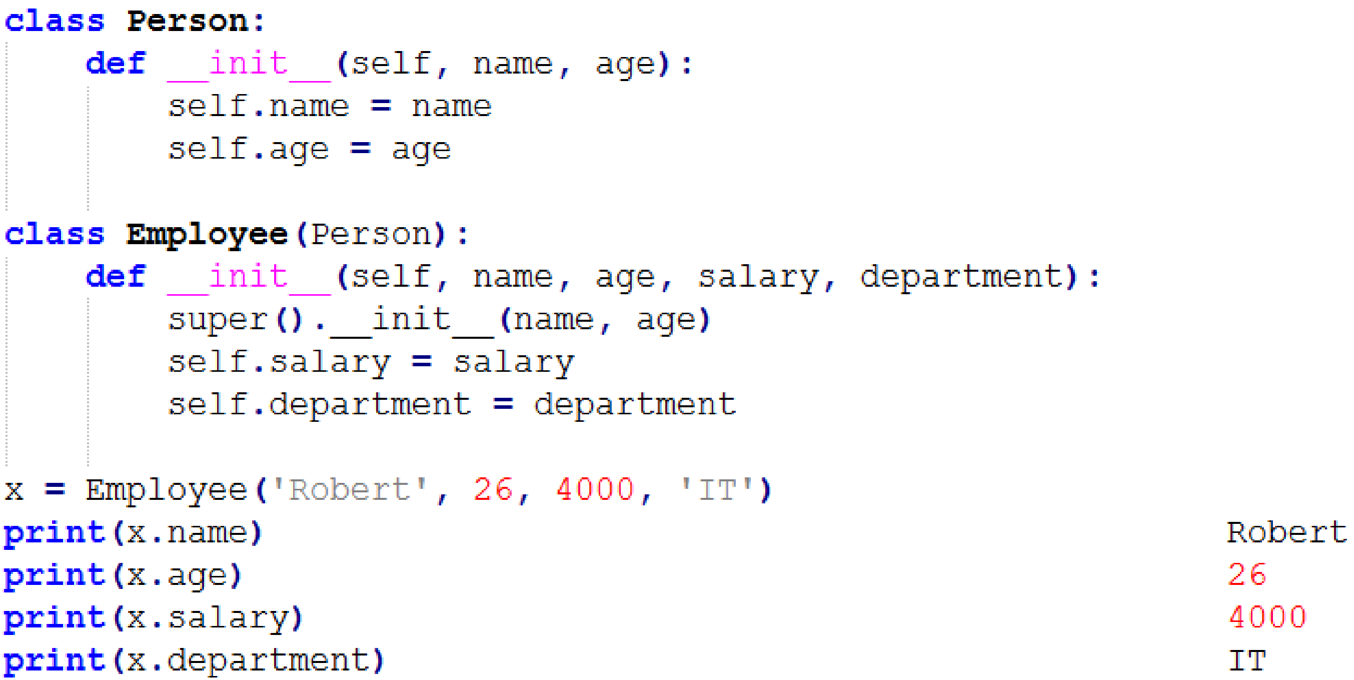
- There are 3 types of access modifiers for a class in Python. These access modifiers define how the members of the class can be accessed. Of course, any member of a class is accessible inside any member function of that same class. Moving ahead to the type of access modifiers, they are:

Access Modifiers

- Private members of a class are denied access from the environment outside the class. They can be handled only from within the class.
- Public members (generally methods declared in a class) are accessible from outside the class. The object of the same class is required to invoke a public method. This arrangement of private instance variables and public methods ensures the principle of data encapsulation.
- Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.
- Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with single or double underscore to emulate the behavior of protected and private access specifies.

Access Modifiers in Parent Class

- All members in a Python class are public by default. Any member can be accessed from outside the class environment.

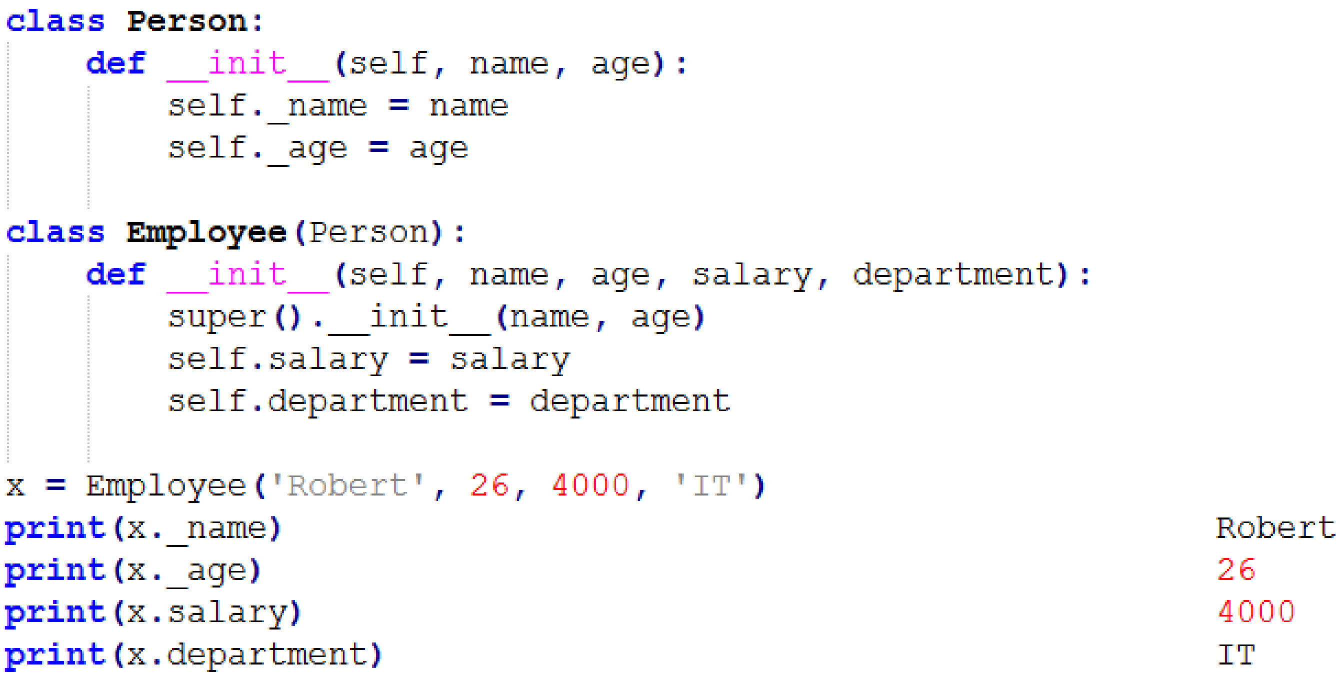


```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

x = Employee('Robert', 26, 4000, 'IT')
print(x.name)
print(x.age)
print(x.salary)
print(x.department)
```

```
Robert
26
4000
IT
```

Access Modifiers in Parent Class

- Python’s convention to make an instance variable protected is to add a prefix _(single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class.

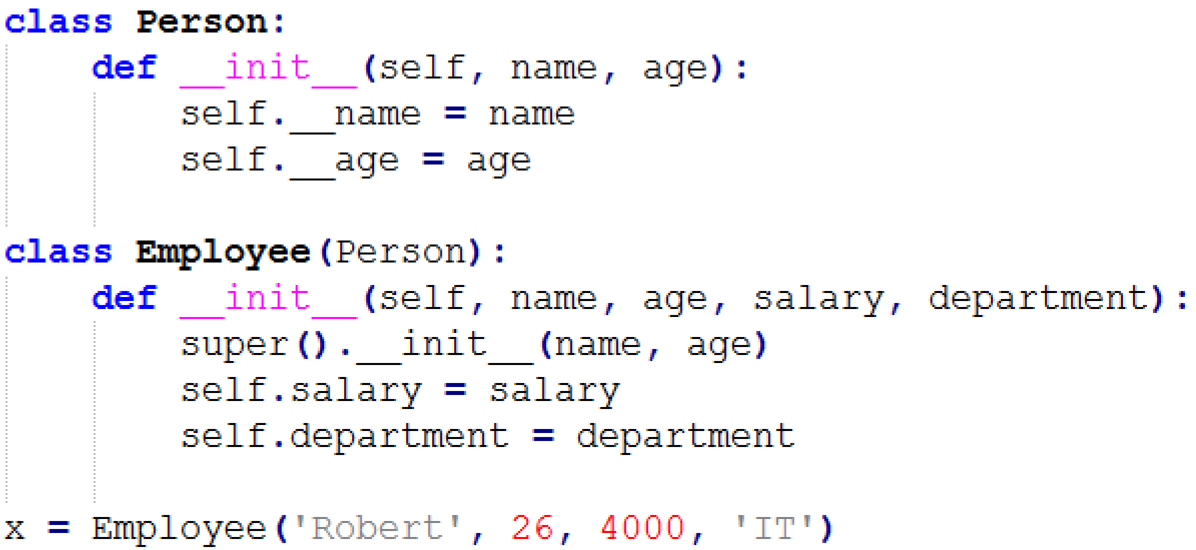


```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

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

x = Employee('Robert', 26, 4000, 'IT')
print(x._name)
print(x._age)
print(x.salary)
print(x.department)
```

```
Robert
26
4000
IT
```

Access Modifiers in Parent Class

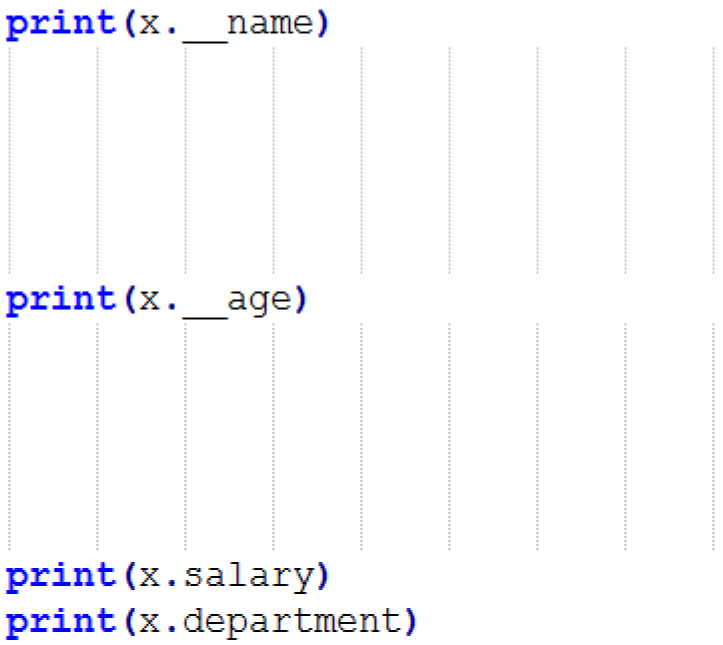
- Similarly, a double underscore prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:



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

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

x = Employee('Robert', 26, 4000, 'IT')
```

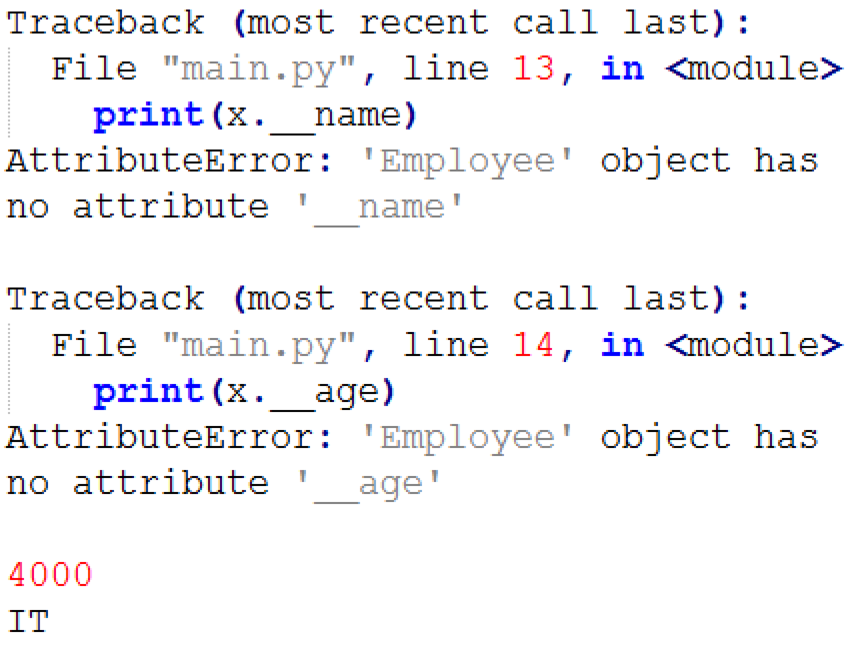
Access Modifiers in Parent Class

Example:



```
print(x.__name)



print(x.__age)



print(x.salary)
print(x.department)
```

Output:



```
Traceback (most recent call last):
  File "main.py", line 13, in <module>
    print(x.__name)
AttributeError: 'Employee' object has
no attribute '__name'

Traceback (most recent call last):
  File "main.py", line 14, in <module>
    print(x.__age)
AttributeError: 'Employee' object has
no attribute '__age'

4000
IT
```

Lecture Agenda

- ✓ Section 1: Inheritance
- ✓ Section 2: Access Modifiers
- Section 3: Function Overriding
- Section 4: Multiple Inheritance
- Section 5: Composition Relationship
- Section 6: Aggregation Relationship

Function Overriding

- Method overriding is a concept of object oriented programming that allows us to change the implementation of a function in the child class that is defined in the parent class. It is the ability of a child class to change the implementation of any method which is already provided by one of its parent class(ancestors).

- Following conditions must be met for overriding a function:

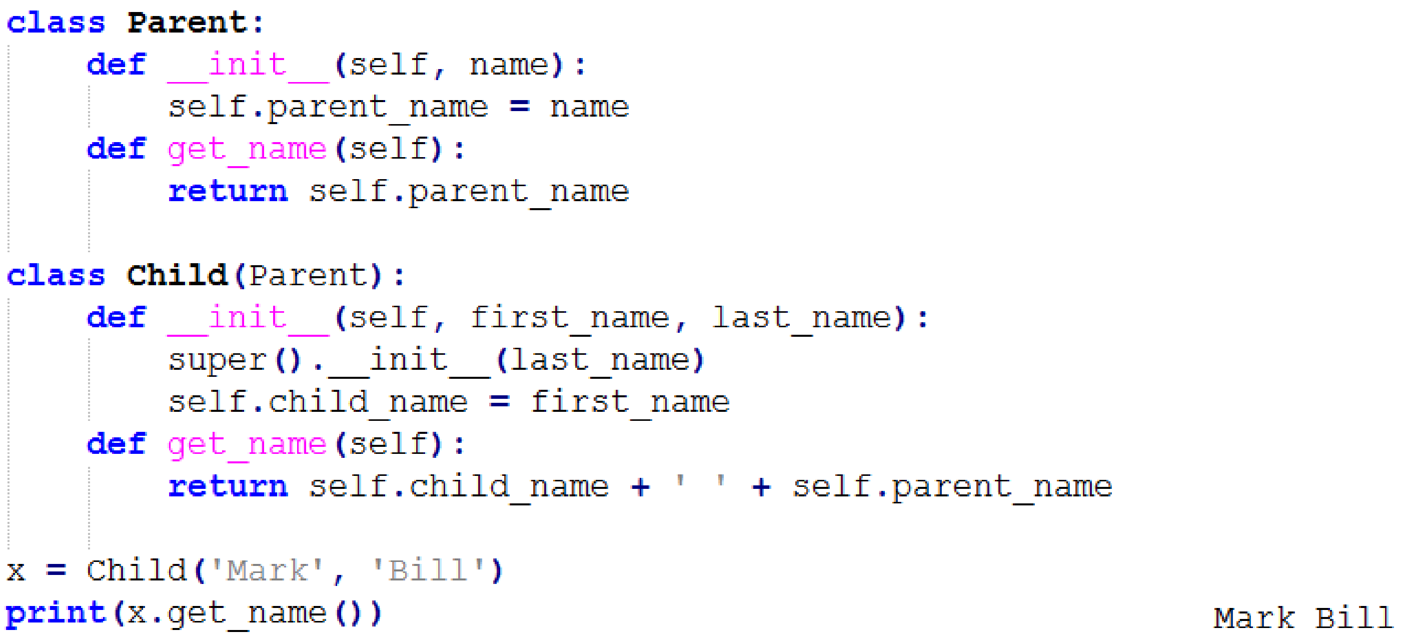
  - Inheritance should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.

  - The function that is redefined in the child class should have the same signature as in the parent class i.e. same number of parameters.

- As we have already learned about the concept of Inheritance, we know that when a child class inherits a parent class it also get access to it public and protected(access modifiers in python) variables and methods.

Function Overriding in Python

- In Python method overriding occurs by simply defining in the child class a method with the same name of a method in the parent class. When you define a method in the object you make this latter able to satisfy that method call, so the implementations of its ancestors do not come in play.

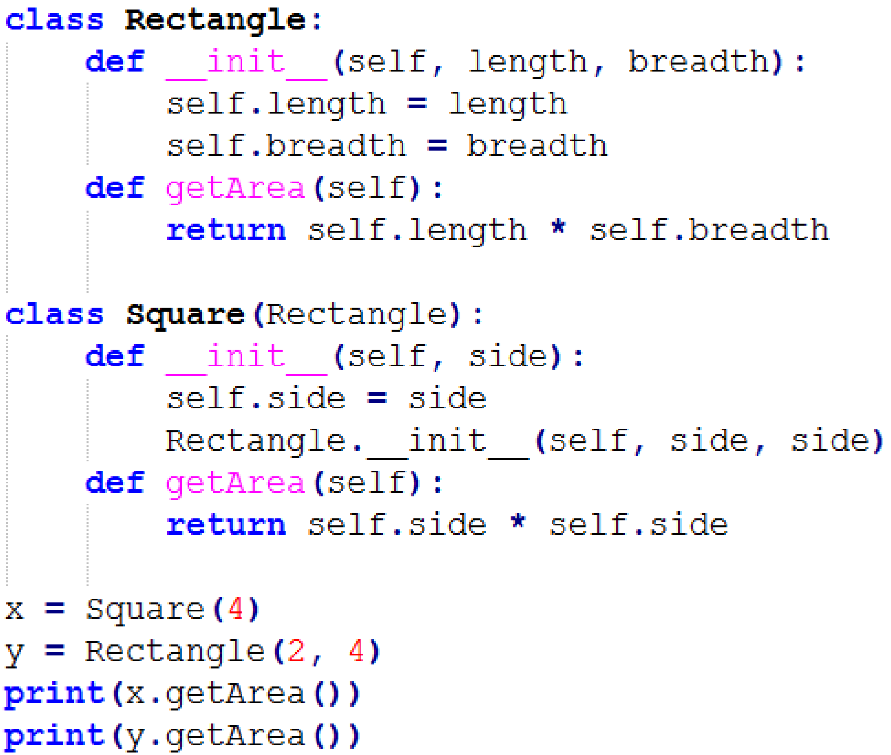


```
class Parent:
    def __init__(self, name):
        self.parent_name = name
    def get_name(self):
        return self.parent_name

class Child(Parent):
    def __init__(self, first_name, last_name):
        super().__init__(last_name)
        self.child_name = first_name
    def get_name(self):
        return self.child_name + ' ' + self.parent_name

x = Child('Mark', 'Bill')
print(x.get_name())
```

```
Mark Bill
```

Function Overriding in Python

- Override means having two methods with the same name but doing different tasks. It means that one of the methods overrides the other. If there is any method in the superclass and a method with the same name in a subclass, then by executing the method, the method of the corresponding class will be executed.



```
class Rectangle:
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth
    def getArea(self):
        return self.length * self.breadth

class Square(Rectangle):
    def __init__(self, side):
        self.side = side
        Rectangle.__init__(self, side, side)
    def getArea(self):
        return self.side * self.side

x = Square(4)
y = Rectangle(2, 4)
print(x.getArea())
print(y.getArea())
```

```
16
8
```

Function Overriding

- In the previous example, notice that ___init ___() method was defined in both classes, Employee as well Person. When this happens, the method in the derived class overrides that in the base class. This is to say, ___init ___() in Employee gets preference over the same in Person.

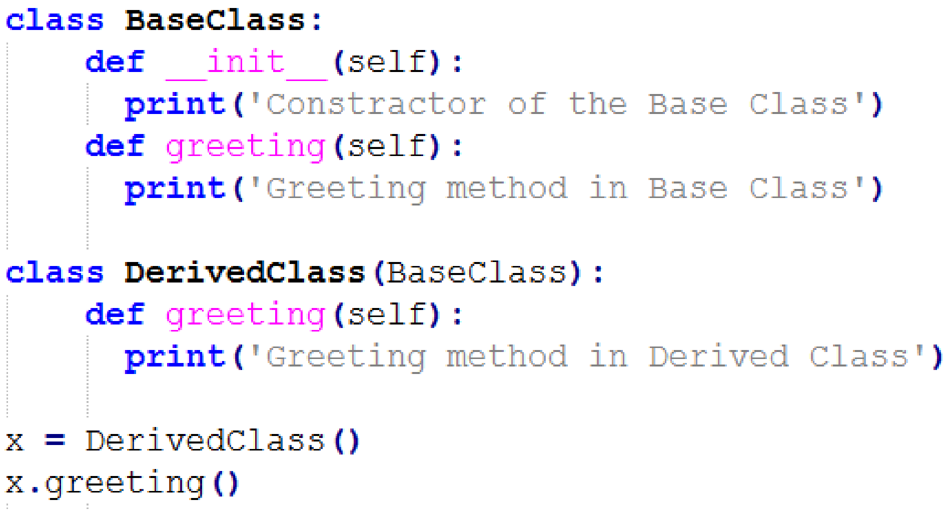
- Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Person. ___init ___() from ___init ___() in Employee).

- We see how resources of the base class are reused while constructing the inherited class. However, the inherited class can have its own instance attributes and methods.

- Methods of the parent class are available for use in the inherited class. However, if needed, we can modify the functionality of any base class method. For that purpose, the inherited class contains a new definition of a method (with the same name and the signature already present in the base class). Naturally, the object of a new class will have access to both methods, but the one from its own class will have precedence when invoked. This is called method overriding.

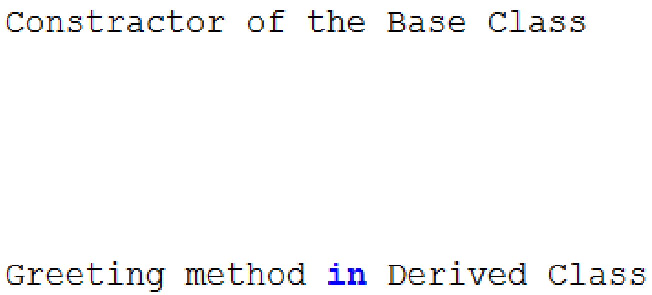
Function Overriding

Example:



```
class BaseClass:
    def __init__(self):
      print('Constractor of the Base Class')
    def greeting(self):
      print('Greeting method in Base Class')

class DerivedClass(BaseClass):
    def greeting(self):
      print('Greeting method in Derived Class')

x = DerivedClass()
x.greeting()
```

Output:



```
Constractor of the Base Class

Greeting method in Derived Class
```

Lecture Agenda

- ✓ Section 1: Inheritance
- ✓ Section 2: Access Modifiers
- ✓ Section 3: Function Overriding
- **Section 4: Multiple Inheritance**
- Section 5: Composition Relationship
- Section 6: Aggregation Relationship

Multiple Inheritance

- A class can be derived from more than one base classes in Python. 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.

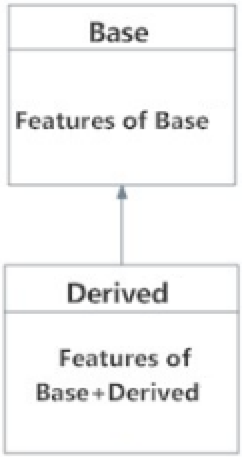
- In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

- Multiple Inheritance means that you're inheriting the property of multiple classes into one. In case you have two classes, say A and B, and you want to create a new class which inherits the properties of both A and B.

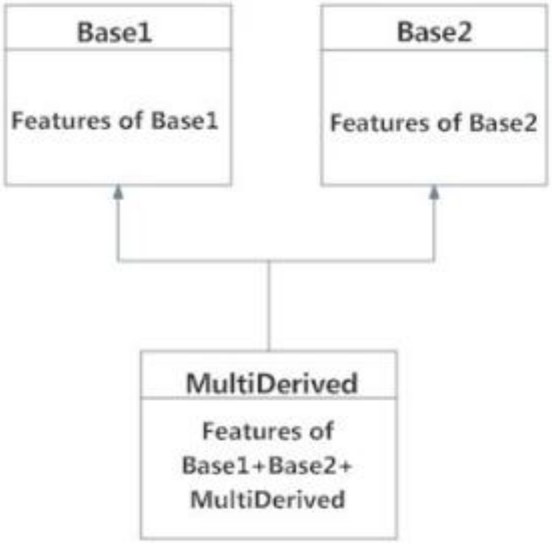
- In multilevel inheritance, we inherit the classes at multiple separate levels. We have three classes A, B and C, where A is the super class, B is its sub(child) class and C is the sub class of B.

Types of Inheritance

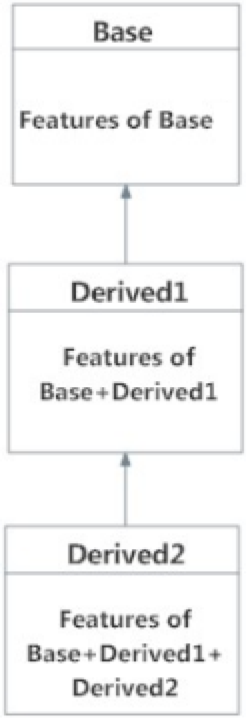
- Single inheritance



- Multiple inheritance

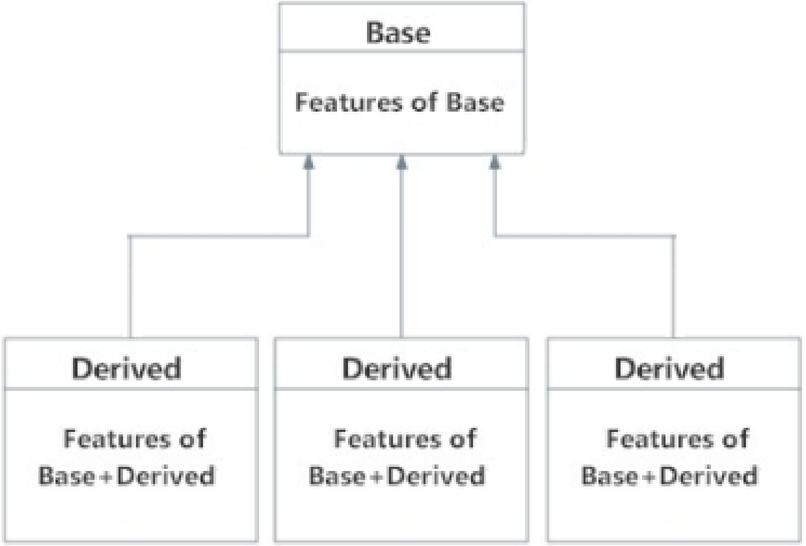


- Multilevel inheritance

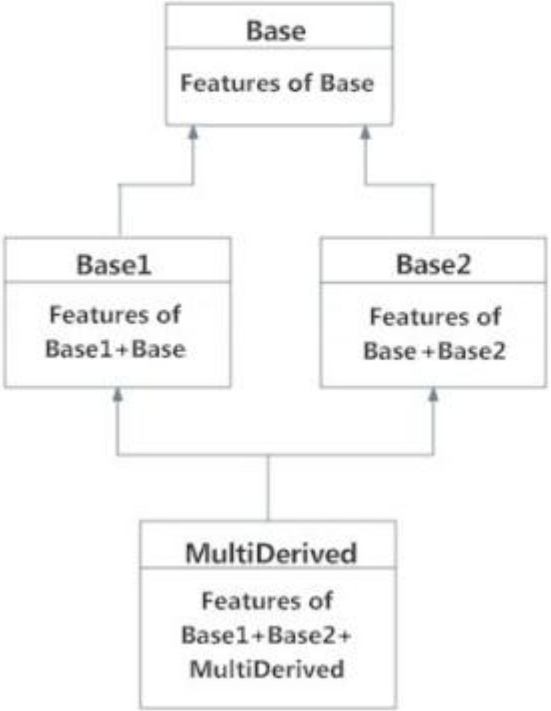

Base
Features of Base
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Types of Inheritance

- Hierarchical Inheritance

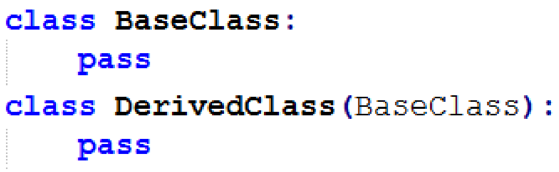


- Hybrid Inheritance



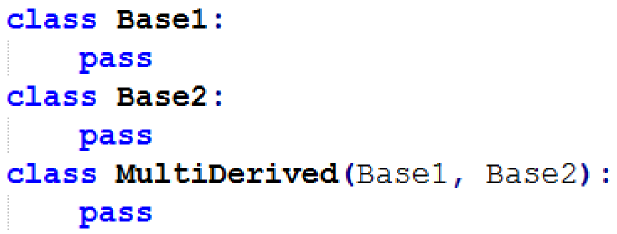
Types of Inheritance

- Single inheritance: When a child class inherits from only one parent class, it is called as single inheritance.



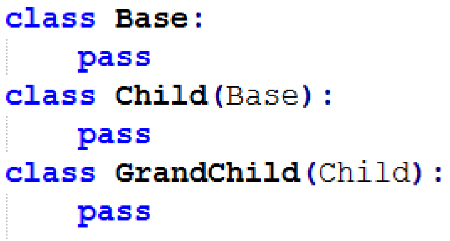
```
class BaseClass:
    pass
class DerivedClass(BaseClass):
    pass
```

- Multiple inheritance: When a child class inherits from multiple parent classes, it is called as multiple inheritance.



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

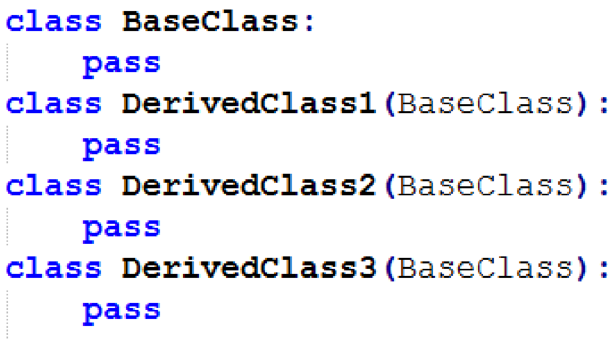
- Multilevel inheritance: When we have child and grand child relationship.



```
class Base:
    pass
class Child(Base):
    pass
class GrandChild(Child):
    pass
```

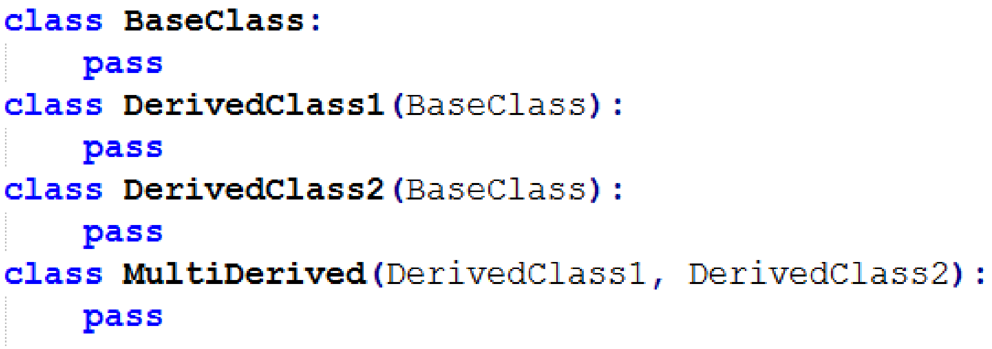
Types of Inheritance

- Hierarchical Inheritance: When one class is inherited by many sub classes.



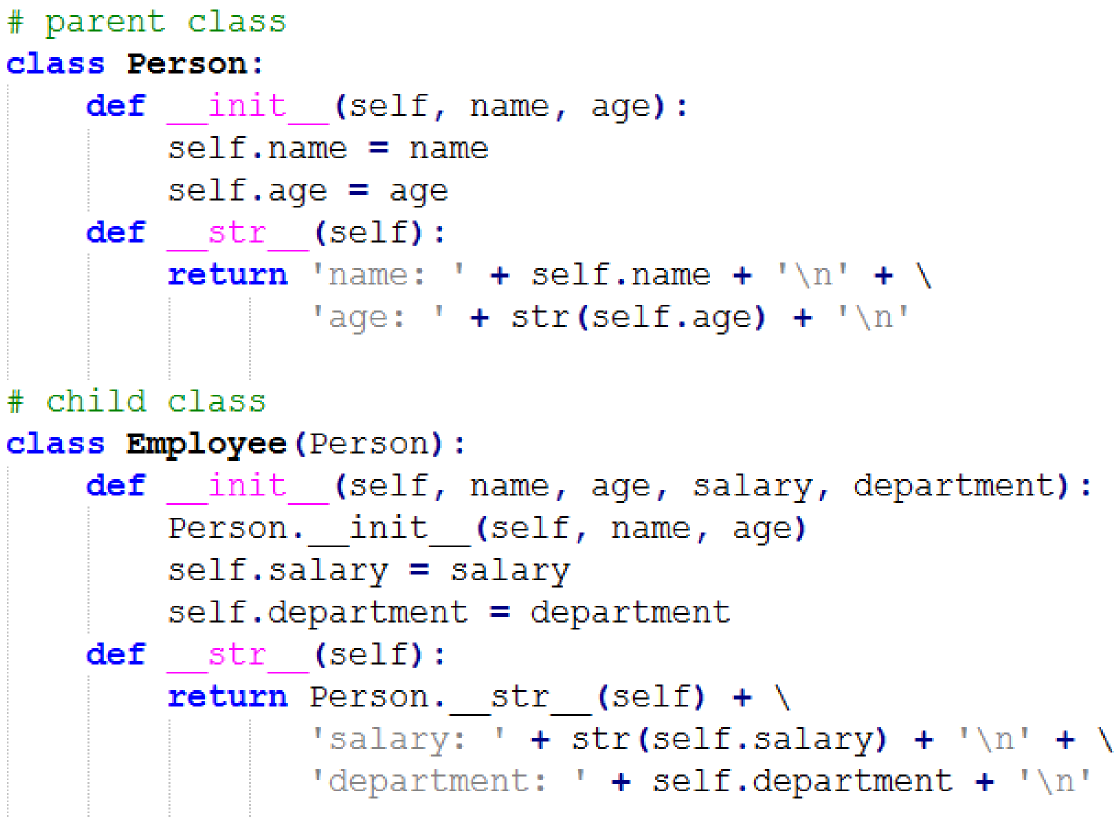
```
class BaseClass:
    pass
class DerivedClass1(BaseClass):
    pass
class DerivedClass2(BaseClass):
    pass
class DerivedClass3(BaseClass):
    pass
```

- Hybrid Inheritance: it is a combination of Single and Multiple inheritance.



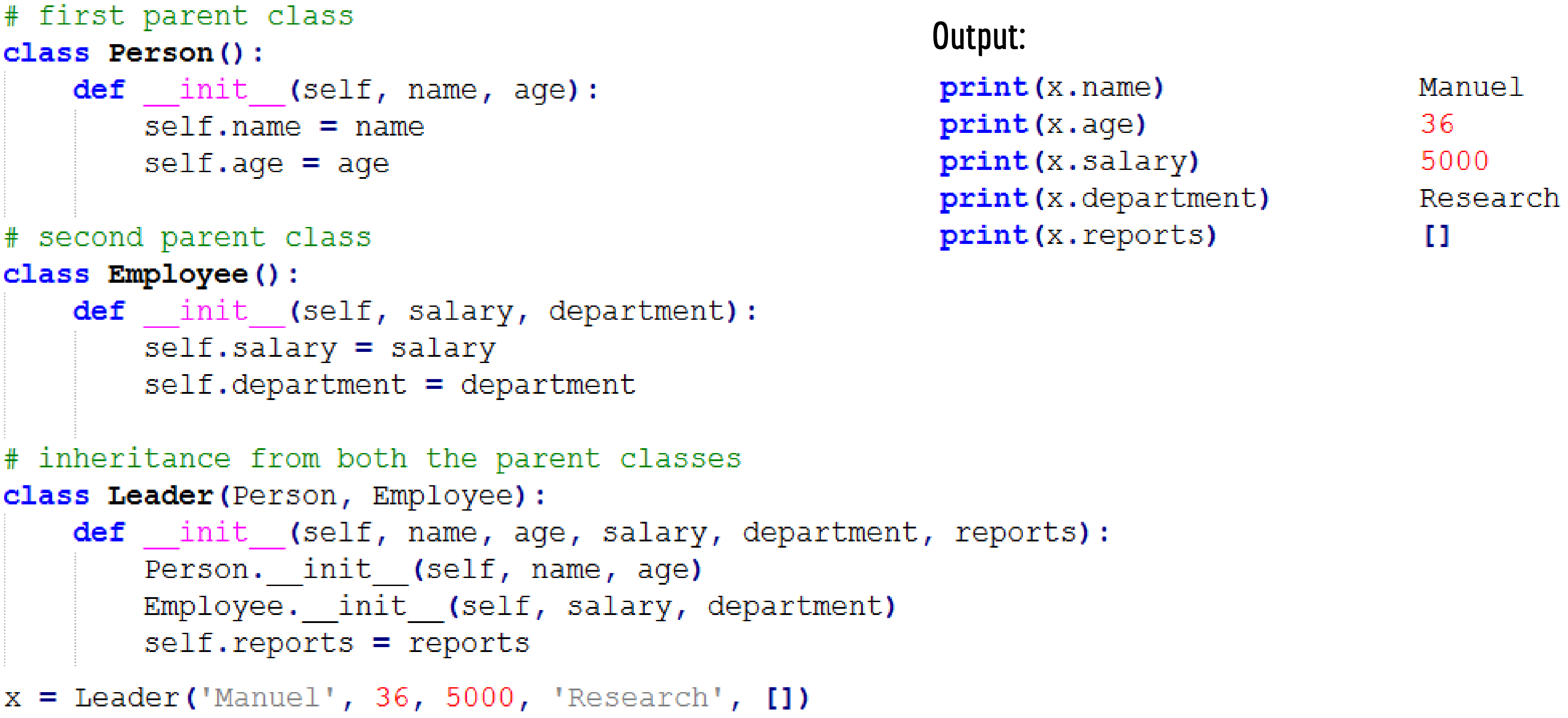
```
class BaseClass:
    pass
class DerivedClass1(BaseClass):
    pass
class DerivedClass2(BaseClass):
    pass
class MultiDerived(DerivedClass1, DerivedClass2):
    pass
```

Single inheritance



```
# parent class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return 'name: ' + self.name + '\n' + \
               'age: ' + str(self.age) + '\n'

# child class
class Employee(Person):
    def __init__(self, name, age, salary, department):
        Person.__init__(self, name, age)
        self.salary = salary
        self.department = department
    def __str__(self):
        return Person.__str__(self) + \
               'salary: ' + str(self.salary) + '\n' + \
               'department: ' + self.department + '\n'
```

Multiple inheritance



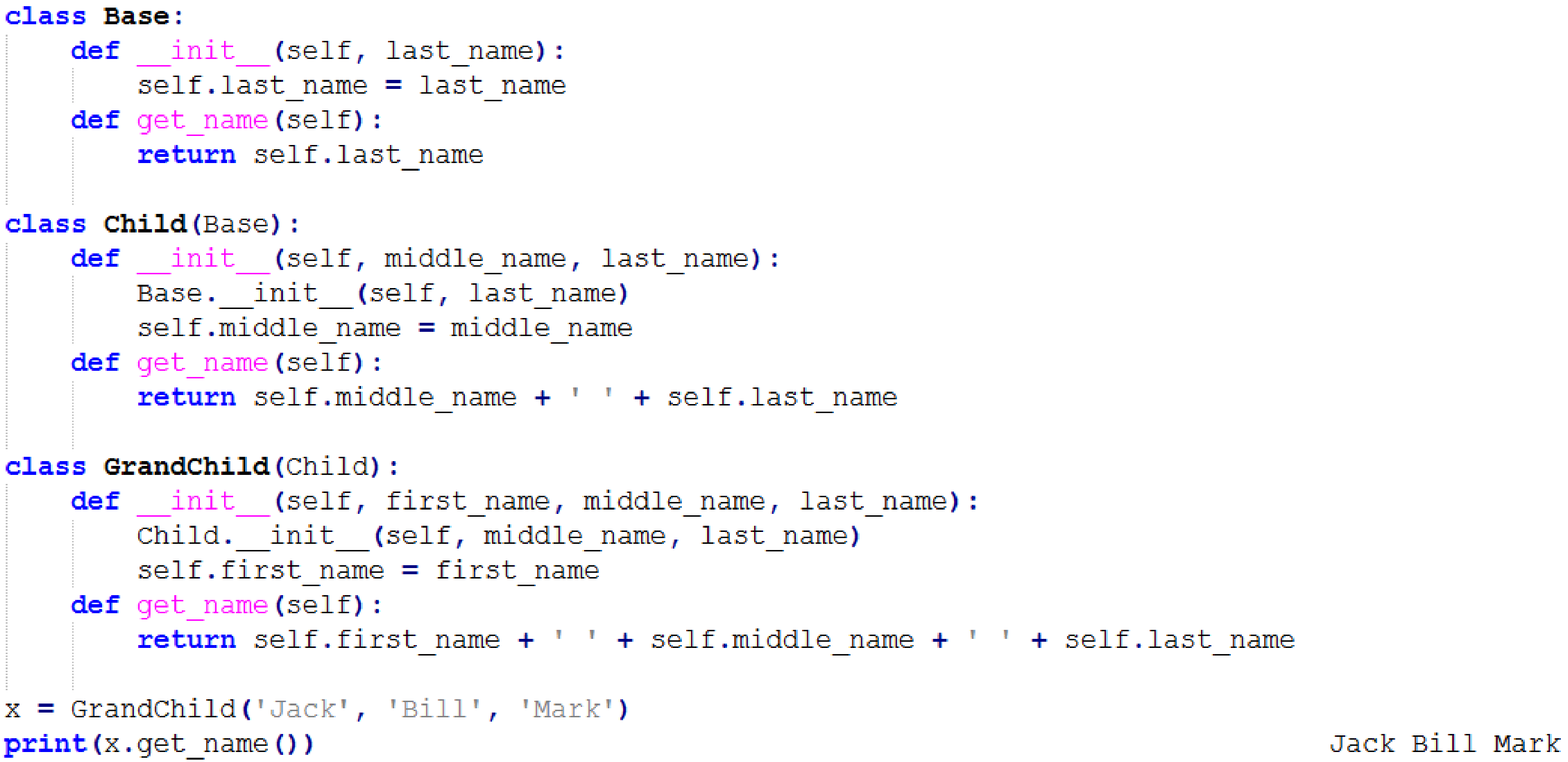
```python
# first parent class
class Person():
    def __init__(self, name, age):
        self.name = name
        self.age = age

# second parent class
class Employee():
    def __init__(self, salary, department):
        self.salary = salary
        self.department = department

# inheritance from both the parent classes
class Leader(Person, Employee):
    def __init__(self, name, age, salary, department, reports):
        Person.__init__(self, name, age)
        Employee.__init__(self, salary, department)
        self.reports = reports

x = Leader('Manuel', 36, 5000, 'Research', [])
```

Output:

```
print(x.name)                 Manuel
print(x.age)                  36
print(x.salary)               5000
print(x.department)           Research
print(x.reports)              []
```

© Prepared by: Mohamed Ayman

Multilevel inheritance



```python
class Base:
    def __init__(self, last_name):
        self.last_name = last_name
    def get_name(self):
        return self.last_name

class Child(Base):
    def __init__(self, middle_name, last_name):
        Base.__init__(self, last_name)
        self.middle_name = middle_name
    def get_name(self):
        return self.middle_name + ' ' + self.last_name

class GrandChild(Child):
    def __init__(self, first_name, middle_name, last_name):
        Child.__init__(self, middle_name, last_name)
        self.first_name = first_name
    def get_name(self):
        return self.first_name + ' ' + self.middle_name + ' ' + self.last_name

x = GrandChild('Jack', 'Bill', 'Mark')
print(x.get_name())
```

```
Jack Bill Mark
```

Super Keyword in Python

- Python super function provides us the facility to refer to the parent class explicitly. It is basically useful where we have to call superclass functions. It returns the proxy object that allows us to refer parent class by 'super'.

- Python Super function provides us the flexibility to do single level or multilevel inheritances and makes our work easier and comfortable. Keep one thing in mind that while referring the superclass from subclass, there is no need of writing the name of superclass explicitly.

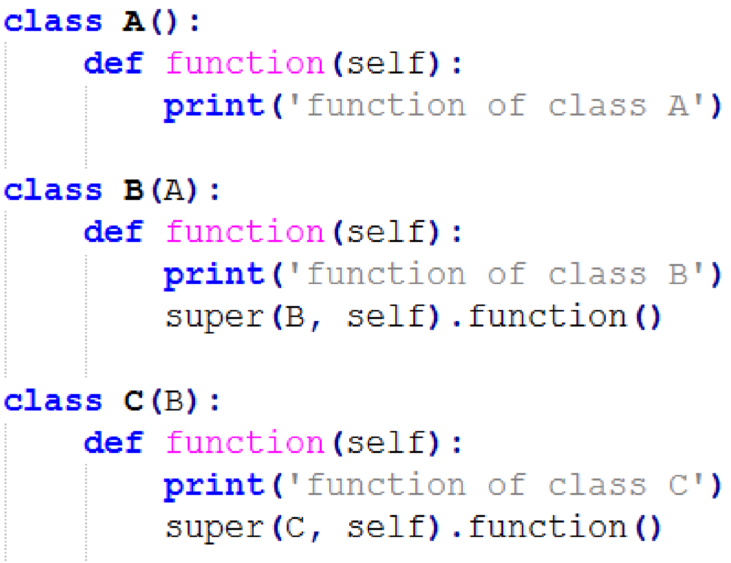
- As we have studied that the Python super() function allows us to refer the superclass implicitly. But in multi-level inheritances, the question arises that there are so many classes so which class did the super() function will refer?

- Well, the super() function has a property that it always refers the immediate superclass. Also, super() function is not only referring the ___init___() but it can also call the other functions of the superclass when it needs.

© Prepared by: Mohamed Ayman

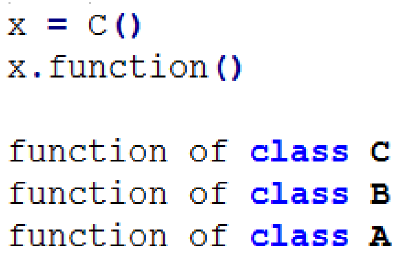
Super Keyword in Python

- The super() method helps us in overriding methods in new style classes. Its syntax is as follows: super(class_name, instance-of-class).overridden_method_name()
- Let us assume there are 3 classes A, B, and C. All 3 of them have a common function called ‘function’. Here comes the work of super().



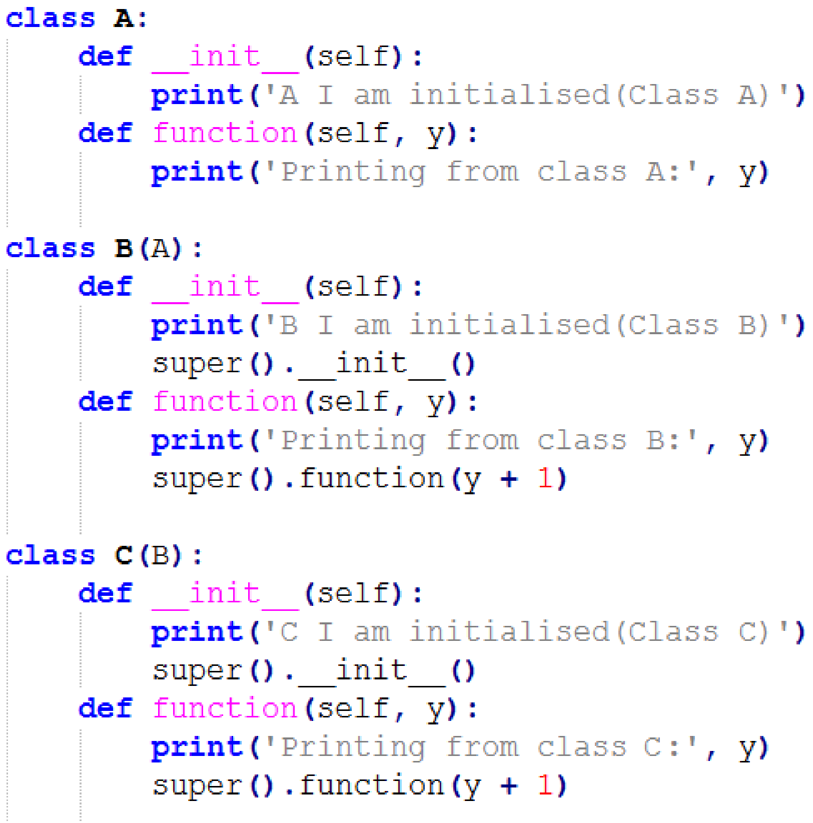
```python
class A():
    def function(self):
        print('function of class A')

class B(A):
    def function(self):
        print('function of class B')
        super(B, self).function()

class C(B):
    def function(self):
        print('function of class C')
        super(C, self).function()
```

Output:



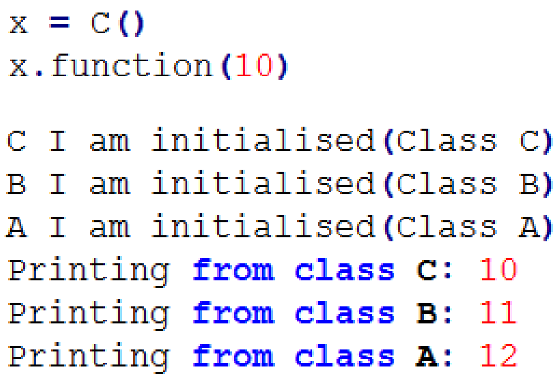
```
x = C()
x.function()

function of class C
function of class B
function of class A
```

Super Keyword in Python



```python
class A:
    def __init__(self):
        print('A I am initialised(Class A)')
    def function(self, y):
        print('Printing from class A:', y)

class B(A):
    def __init__(self):
        print('B I am initialised(Class B)')
        super().__init__()
    def function(self, y):
        print('Printing from class B:', y)
        super().function(y + 1)

class C(B):
    def __init__(self):
        print('C I am initialised(Class C)')
        super().__init__()
    def function(self, y):
        print('Printing from class C:', y)
        super().function(y + 1)
```

Output:



```
x = C()
x.function(10)

C I am initialised(Class C)
B I am initialised(Class B)
A I am initialised(Class A)
Printing from class C: 10
Printing from class B: 11
Printing from class A: 12
```

Lecture Agenda

- ✓ Section 1: Inheritance
- ✓ Section 2: Access Modifiers
- ✓ Section 3: Function Overriding
- ✓ Section 4: Multiple Inheritance
- **Section 5: Composition Relationship**
- Section 6: Aggregation Relationship

Composition Relationship

- Composition implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

- We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result.

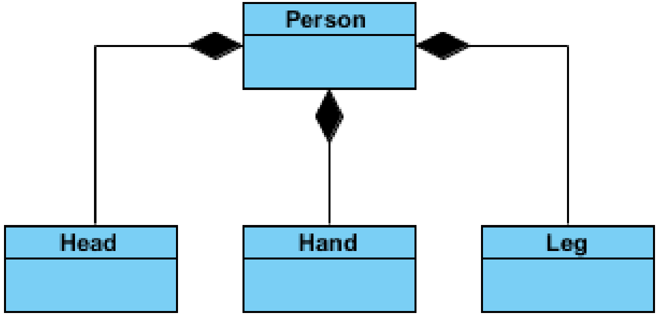
- Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

  - It represents part-of relationship.

  - In composition, both the entities are dependent on each other.

  - When there is a composition between two entities,

  the composed object cannot exist without the other entity.



Lecture Agenda

- ✓ Section 1: Inheritance
- ✓ Section 2: Access Modifiers
- ✓ Section 3: Function Overriding
- ✓ Section 4: Multiple Inheritance
- ✓ Section 5: Composition Relationship
- Section 6: Aggregation Relationship

Aggregation Relationship

- Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

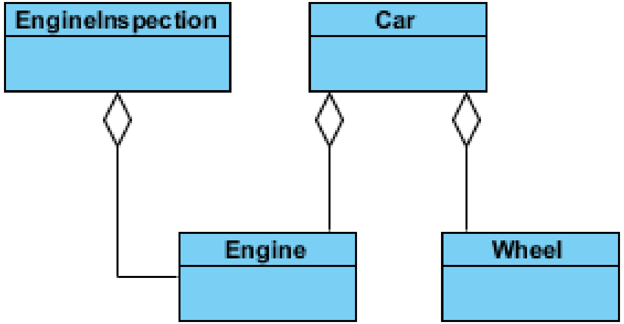
- Aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.

- It is a special form of Association where:

  - It represents Has-A relationship.

  - It is a unidirectional association, one way relationship. i.e., department can have students but vice versa is not possible and thus unidirectional in nature.

  - In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity



Lecture Agenda

- ✓ Section 1: Inheritance
- ✓ Section 2: Access Modifiers
- ✓ Section 3: Function Overriding
- ✓ Section 4: Multiple Inheritance
- ✓ Section 5: Composition Relationship
- ✓ Section 6: Aggregation Relationship

DO
MORE.