Section A3 Inheritance、composition、aggregation

Inheritance

To create a class that inherits the properties and methods from another class, send the parent class as a parameter when constructing the child class.

```
class Person:
    def __init__(self, fname, lname):
        self.first_name = fname
        self.last_name = lname

    def print_name(self):
        print(self.first_name, self.last_name)
```

```
class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
        self.first_name = lname
        self.last_name = fname
```

the child's __init__() method overrides the inheritance of the __init__() method of its parent.
To keep the inheritance of the parent's __init__() method, you need to add a call to the parent's __init__() method, as shown in the example below.

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

m = Student("Mike", "Olsen")
m.print_name()
```

we also have the super() method that will allow the child class to inherit all the properties and methods from the parent. The only difference of this approach (compared to the one we saw above) is that you do not need to specifically mention the name of the parent element in the super() method. It will automatically inherit the properties and methods from the parent.

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

Multiple Inheritance
you can inherit from more than one superclass. This is called *multiple inheritance*. Multiple inheritance allows you to design classes that inherit behaviour and attributes from more than one class.

```python
# defining multiple inheritance
class Airplane(Vehicle,FlyingVehicle):
    def __init__(self, speed, doors, passengers, capacity, propulsion_mechanism, engines, air_type):

        # we first call the __init__ function of the first superclass
        Vehicle.__init__(self, speed,doors, passengers, capacity)

        # then of the second super class
        FlyingVehicle.__init__(self, engines, propulsion_mechanism)

        # then we initialize the rest of the attributes
        self.air_type=air_type
        self.passengers=passengers
        self.capacity=capacity

    def __repr__(self):
        return "I am an Airplane"

    #we can call methods from both the superclasses
    def airborne(self):
        super().fly()

    def ground(self):
        super().travel()
```

Method Resolution Order
all objects are instances of the object class.

In such multiple inheritance examples, any specified attribute(s) or method(s) is first searched in the current class (i.e., MultiDerived class). If the specified attribute(s) or method(s) is not found, the search continues to its parent classes in a depth-first and left-right manner without searching the same class twice.
The search order specified above is also called linearisation of the MultiDerived class, and the set of rules used to identify this order is called **Method Resolution Order (MRO)**. MRO of a class can be viewed using the __mro__ attribute (which returns a tuple) or the mro() method (which returns a list).

```python
class Base1:
    name = "String 1"

class Base2:
    name = "String 2"

class MultiDerived(Base1, Base2):
    pass

print(MultiDerived.name)

print(MultiDerived.__mro__)
print(MultiDerived.mro())
```

```
String 1
(<class '__main__.MultiDerived'>, <class '__main__.Base1'>, <class '__main__.Base2'>, <class 'object'>)
[<class '__main__.MultiDerived'>, <class '__main__.Base1'>, <class '__main__.Base2'>, <class 'object'>]
```

子类不能直接继承和访问父类的这些私有属性和方法。但是，实际上，Python执行了一个名为名称改写的过程，私有属性或方法名前的双下划线会被改写为 _ClassName__attribute_or_method。这意味着，尽管您不能直接通过 self.__private_var 访问它，但您可以通过 _ClassName__private_var 来访问它。尽管这样做是可能的，但通常不建议这样做，因为它违反了封装原则。如果您需要在子类中访问某些属性或方法，最好将它们声明为受保护的（只有一个下划线，如 _protected_var）而不是私有的。

the Protected Access Modifier

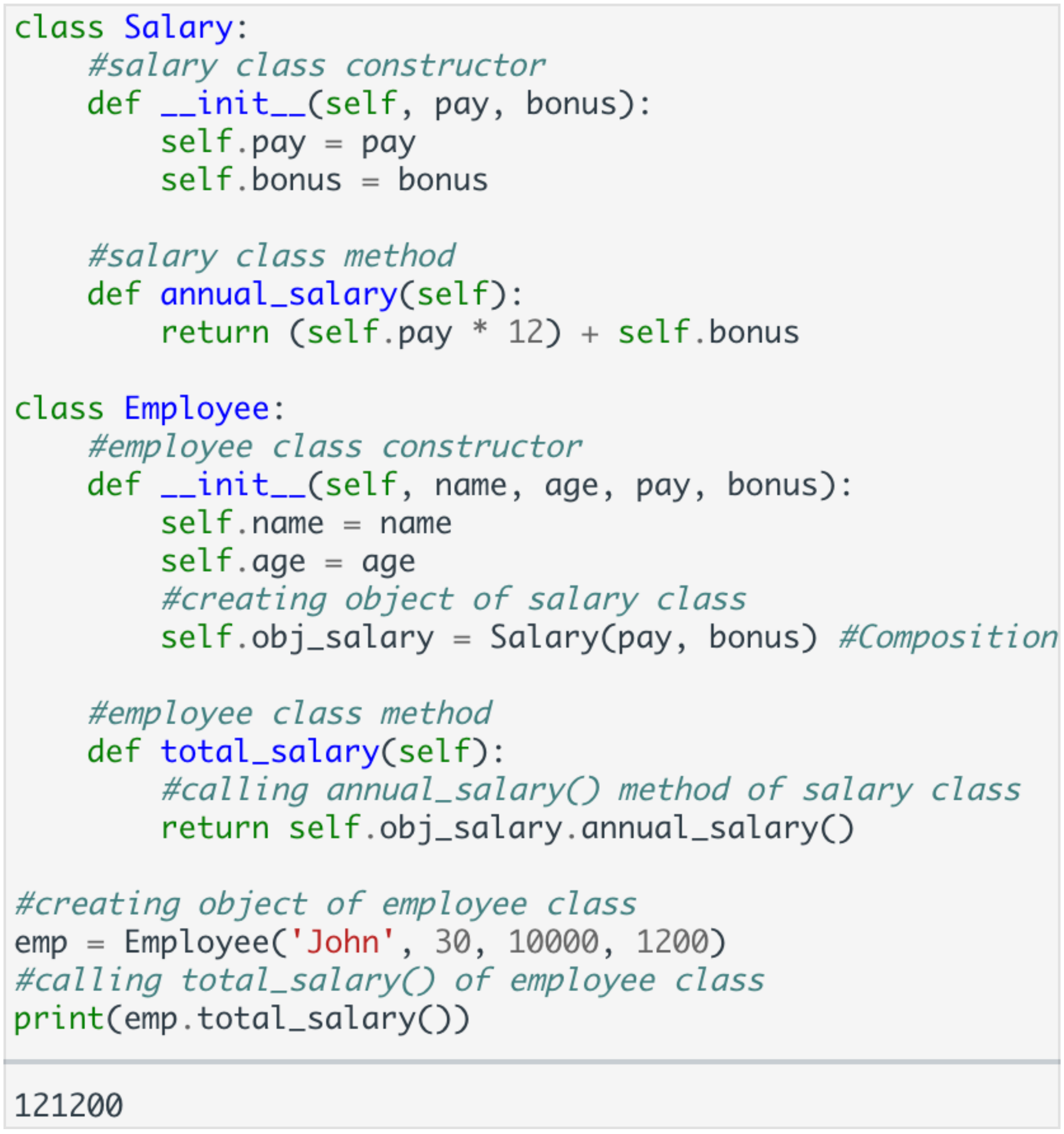
Protected variables can be accessed within the same package. To create protected variables, you need to prefix the variable name with a single underscore ('_').

Inheritance Types

- Single Inheritance
- Multi-Level Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Composition

Composition established “part of” relationship between objects.



```python
class Salary:
    #salary class constructor
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    #salary class method
    def annual_salary(self):
        return (self.pay * 12) + self.bonus

class Employee:
    #employee class constructor
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age
        #creating object of salary class
        self.obj_salary = Salary(pay, bonus) #Composition

    #employee class method
    def total_salary(self):
        #calling annual_salary() method of salary class
        return self.obj_salary.annual_salary()

#creating object of employee class
emp = Employee('John', 30, 10000, 1200)
#calling total_salary() of employee class
print(emp.total_salary())
```

```
121200
```

Aggregation

composition lets us delegate some responsibility from one class to another class. In other words, one class acts like a *Container* and the other class acts like *Content*. When there is a composition relationship between two classes, the content object cannot exist without the container object.
Aggregation represents "has a" relationship. It is a weak form of composition. In other words, the aggregation relationship is a form of composition, where objects are loosely coupled. If you delete the container object, the contents objects can live (without the container object). This means there are no objects or classes that own another object. Instead, it only creates a reference.

- The composition relationship represents “part of” relationship. In the composition relationship, both the associate class objects cannot survive independently.
- The aggregation relationship represents “has a” relationship. In the aggregation relationship, both the associate class objects can survive

individually.

```python
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay * 12) + self.bonus

class Employee:
    #'salary' is the object from the salary class that we pass as an argument
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        #assign the salary object to 'obj_salary'
        self.obj_salary = salary #Aggregation

    def total_salary(self):
        return self.obj_salary.annual_salary()

#instantiating the salary class
salary = Salary(10000, 1200)
#pass the salary object to the constructor of the employee class
emp = Employee('John', 30, salary)
print(emp.total_salary())
```

```
121200
```

What we have done differently is that instead of using the Salary class inside the Employee class, we now create an instance of the Salary class and pass this instance as an argument to the constructor of the Employee class.