What Is Inheritance?

Inheritance allows features defined in one class to be inherited and reused in the

definition of another class.

Inheritance is a way to form new classes using classes that have already been defined.

The newly formed classes are called derived classes, the classes that we derive from are called base classes.

Important benefits of inheritance are code reuse and reduction of complexity of a program.

The derived classes (descendants) override or extend the functionality of base classes (ancestors).

A class that is defined as extending a parent class has the following syntax:

class SubClassName(BaseClassName):

class-body

Note that the parent class is specified by providing the name of that class in

round brackets after the name of the new (child) class

class Person(object): #base class

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def birthday(self):

print('Happy birthday you were', self.age)

self.age += 1

print('You are now', self.age)

# We could now define the class Employee as being a class whose definition

# builds on (or inherits from) the class Person

class Employee(Person): #derrive class

def \_\_init\_\_(self, name, age, id):

super().\_\_init\_\_(name, age)

self.id = id

def calculate\_pay(self, hours\_worked):

rate\_of\_pay = 7.50

if self.age >= 21:

rate\_of\_pay += 2.50

return hours\_worked \* rate\_of\_pay

def \_\_str\_\_(self):

return "Person "+ self.name +"has the age" + int(self.age)

e = Employee("Asha",28,123)

print(e.name)

e1 = Employee("Dheeraj",19,234)

e.birthday()

print(e.calculate\_pay(30))

print(issubclass(Employee,Person)) #From person employee is extended and not the other way

print(issubclass(Person,Employee))

Asha

Happy birthday you were 28

You are now 29

300.0

True

False

Here we do several things:

1. The class is called Employee but it extends Person. This is indicated by including the name of the class being inherited in parentheses after the name of the class being defined (e.g. Employee(Person)) in the class declaration.
2. Inside the **init** method we reference the **init**() method defined in the class Person and used to initialise instances of that class (via the super().**init**() reference. This allows whatever initialisation is required for Person to happen. This is called from within the Employee class’s **init**() which then allows any initialisation required by the Employee to occur. Note that the call to the super().**init**() initialiser can come anywhere within the Employee.**init**() method; but by convention it comes first to ensure that whatever the Person class does during initialisation does not over write what happens in the Employee class.
3. All instances of the class Person have a name, and age and have the behaviour birthday().
4. All instances of the class Employee have a name, and age and an id and have the behaviours birthday() and calculate\_pay(house\_worked).
5. The method calculate\_pay() defined in the Employee class can access the attributes name and age just as it can access the attribute id. In fact, it uses the employee’s age to determine the rate of pay to apply.

We can go further, and we can subclass Employee, for example with the class SalesPerson

**class** SalesPerson(Employee):

**def** \_\_init\_\_(self, name, age, id, region, sales):

super().\_\_init\_\_(name, age, id)

self.region **=** region

self.sales **=** sales

**def** bonus(self):

**return** self.sales **\*** 0.5

​

Now we can say that the class SalesPerson has a name, an age and an id as well as a region and a sales total. It also has the methods birthday(), calculate\_pay(hourse\_worked) and bonus().

In this case the SalesPerson.**init**() method calls the Employee. **init**() method as that is the next class up the hierarchy and thus we want to run that classes initialisation behaviour before we set up the SalesPerson class (which of course in turn runs the Person classes initialisation behaviour).

print('Person')

p **=** Person('John', 54)

print(p)

print('-' **\*** 25)

print('Employee')

e **=** Employee('Denise', 51, 7468)

e.birthday()

print('e.calculate\_pay(40):', e.calculate\_pay(40))

print('-' **\*** 25)

print('SalesPerson')

s **=** SalesPerson('Phoebe', 21, 4712, 'UK', 30000.0)

s.birthday()

print('s.calculate\_pay(40):', s.calculate\_pay(40))

print('s.bonus():', s.bonus())

Person

<\_\_main\_\_.Person object at 0x08F2F0B0>

-------------------------

Employee

Happy birthday you were 51

You are now 52

e.calculate\_pay(40): 400.0

-------------------------

SalesPerson

Happy birthday you were 21

You are now 22

s.calculate\_pay(40): 400.0

s.bonus(): 15000.0

It is important to note that we have not done anything to the class Person by defining Employee and SalesPerson; that is it is not affected by those class definitions. Thus, a Person does not have an employee id. Similarly, neither an Employee nor a Person have a region or a sales total.

In terms of behaviour, instances of all three classes can run the method birthday(), but

• only Employee and SalesPerson objects can run the method calculcate\_pay() and

• only SalesPerson objects can run the method bonus().

*# Lets see few more examples*

**class** Animal(object):

**def** \_\_init\_\_(self):

print("Animal created")

​

**def** whoAmI(self):

print("Animal")

​

**def** eat(self):

print("Eating")

​

​

**class** Dog(Animal):

**def** \_\_init\_\_(self):

Animal.\_\_init\_\_(self)

**def** whoAmI(self):

print("Dog")

​

**def** bark(self):

print("Woof!")

​

d **=** Dog()

d.whoAmI()

*# d.eat(2)*

d.eat()

d.bark()

Animal created

Dog

Eating

Woof!

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class.

It is shown by the eat() method. The derived class modifies existing behavior of the base class.

shown by the whoAmI() method.

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

*# Every class in Python extends one or more superclasses. This is true even of the*

*# class Person shown below:*

**class** Person:

**def** \_\_init\_\_(self, name, age):

self.name **=** name

self.age **=** age

*# This is because if you do not specify a superclass explicitly Python automatically*

*# adds in the class object as a parent class. Thus the above is exactly the same as the*

*# following listing which explicitly lists the class object as the superclass of Person:*

​

**class** Person(object):

**def** \_\_init\_\_(self, name, age):

self.name **=** name

self.age **=** age

​

**Terminology Around Inheritance**

The following terminology is commonly used with inheritance in most object oriented languages including Python: Class A class defines a combination of data and procedures that operate on that data. Subclass A subclass is a class that inherits from another class. For example, an Employee might inherit from a class Person. Subclasses are, of course, classes in their own right. Any class can have any number of subclasses. Superclass A superclass is the parent of a class. It is the class from which the current class inherits. For example, Person might be the superclass of Employee. In Python, a class can have any number of superclasses. Single or multiple inheritance Single and multiple inheritance refer to the number of super classes from which a class can inherit. For example, Java is a single inheritance system, in which a class can only inherit from one class. Python by contrast is a multiple inheritance system in which a class can inherit from one or more classes.

**Types of Hierarchy**

In most object-oriented systems there are two types of hierarchy; one refers to inheritance (whether single or multiple) and the other refers to instantiation. The inheritance hierarchy has already been described. It is the way in which one class inherits features from a superclass. The instantiation hierarchy relates to instances or objects rather than classes and is important during the execution of the object. There are two types of instance relationships: one indicates a part-of relationship, while the other relates to a using relationship (it is referred to as an isa relationship).

  Person                       Car

    |                           |

    |                           |

  Student                       Engine

​

The difference between an is-a relationship and a part-of relationship is often

confusing

Student is-a type of Person whereas an Engine is part-of a Car. It does not make sense to say that a

student is part-of a person or that an engine is-a type of car!

​

*# This is- part of is written in 2 ways ie. has-a relation , is-part of relation know as*

*# Aggregation and composition*

​

*#in this example we can not apply inheritence as Salary is not Employee or Employee is not Salary*

*#Salary is just part of Employee , so we can use composition*

*# wheer we can delegate the task to other class , by creating object with in the class*

*# Example for Compositon*

​

**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:

**def** \_\_init\_\_(self,name,age,pay,bonus):

self.name **=** name

self.age **=** age

self.sal\_obj **=** Salary(pay,bonus) *#instantiating Salary class inside Employee class , Salary class is content and*

*#Employee class is Container.*

*# s = Salary(100,200)*

emp **=** Employee("Asha",30,10000,20000)

print(emp.sal\_obj.annual\_salary())

​

**del** emp

​

*#Properties of Composition*

*# The relation between associated classess is defined by the keyword "part-of" , salary is part of Employee*

*# salary obj is created inside emp class, so if we delete employee object then Salary obj is also deleted*

140000

*#Example for Aggregation*

**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:

**def** \_\_init\_\_(self,name,age,salary):

self.name **=** name

self.age **=** age

self.sal\_obj **=** salary

​

salary **=** Salary(10000,20000)

​

emp **=** Employee("Asha",30,salary)

print(emp.sal\_obj.annual\_salary())

**del** emp

*# The properties of Aggregation*

*# The relation between associated classess is defined by the keyword "has-a" , Employee has a salary*

*# The associated classes are unidirectional, we are passing salary object to employee*

*# salary obj and emp obj are created independentely ,if is deleted other still survives*

​

140000

**Purpose of Subclasses**

Subclasses are used to refine the behaviour and data structures of a superclass. A parent class may define some generic/shared attributes and methods; these can then be inherited and reused by several other (sub) classes which add subclass specific attributes and behaviour. In fact, there are only a small number of things that a subclass should do relative to its parent or super class. If a proposed subclass does not do any of these then your selected parent class is not the most appropriate super class to use. A subclass should modify the behaviour of its parent class or extend the data held by its parent class. This modification should refine the class in one or more of these ways:

• Changes to the external protocol or interface of the class, that is it should extend the set of methods or attributes provided by the class.

• Changes in the implementation of the methods; i.e. the way in which the behaviour provided by the class are implemented.

• Additional behaviour that references inherited behaviour. If a subclass does not provide one or more of the above, then it is incorrectly placed. For example, if a subclass implements a set of new methods, but does not refer to the attributes or methods of the parent class, then the class is not really a subclass of the parent (it does not extend it).

**Overriding Methods**

Overriding occurs when a method is defined in a class (for example, Person) and also in one of its subclasses (for example, Employee). It means that instances of Person and Employee both respond to requests for this method to be run but each has their own implementation of the method.

For example, let us assume that we define the method **str**() in these classes (so that we have a string representation of these objects to use with the print function). The pseudo code definition of this in Person might be:

**class** Person:

**def** \_\_init\_\_(self, name, age):

self.name **=** name

self.age **=** age

**def** \_\_str\_\_(self):

**return** self.name **+** ' is ' **+** str(self.age)

​

**class** Employee(Person):

**def** \_\_init\_\_(self, name, age, id):

super().\_\_init\_\_(name, age)

self.id **=** id

**def** \_\_str\_\_(self): *#if this method was not written then its base class str method would be invoked, so overriding of str*

**return** self.name **+** ' is ' **+** str(self.age) **+** ' - id ' **+** str(self.id) **+** ')'

*# def display(self):*

*# pass*

Instances of these classes will both be convertible to a string using **str**() but the version used by instances of Employee will differ from that used with instances of Person,

p **=** Person('John', 54)

print(p)

e **=** Employee('Denise', 51, 1234)

print(e)

John is 54

Denise is 51 - id 1234)

**Extending Superclass Methods**

However, in the previous section we had to duplicated the code in Person down in Employee so that we could convert the name and age attributes into strings. However we can avoid this duplication by invoking the parent class’s method from within the child class version (as we in fact did for the **init**() initialiser). For example: In this version of the code the Employee classes version of the **str**() method first calls the parent classes version of this method and then adds the location information to the string returned from that. This means that we only have one location that converts name and age into a string.

**class** Person:

**def** \_\_init\_\_(self, name, age):

self.name **=** name

self.age **=** age

**def** \_\_str\_\_(self):

**return** self.name **+** ' is ' **+** str(self.age)

**class** Employee(Person):

**def** \_\_init\_\_(self, name, age, id):

super().\_\_init\_\_(name, age)

self.id **=** id

**def** \_\_str\_\_(self):

**return** super().\_\_str\_\_() **+** '-id(' **+** str(self.id) **+** ')'

p **=** Person('John', 54)

print(p)

e **=** Employee('Denise', 51, 1234)

print(e)

John is 54

Denise is 51-id(1234)

*#Constructors in inheritence*

*#base class constructor is availble to sub class*

**class** Father(object):

**def** \_\_init\_\_(self):

self.property **=** 88800

**def** display\_property(self):

print("Father's property",self.property)

**class** Son(Father):

**pass** *# if no constructor in derived class , base class constructor gets executed*

​

​

s **=** Son()

s.display\_property()

Father's property 88800

*#overriding base class constructor and overriding base class methods*

**class** Father:

**def** \_\_init\_\_(self):

self.property **=** 88800

**def** display\_property(self):

print("Father's property",self.property)

**class** Son(Father):

**def** \_\_init\_\_(self):

self.property **=** 22200

**def** display\_property(self):

print("Son's property",self.property) *# if constructor in derived class , base class constructor does not get executed*

​

​

s **=** Son()

s.display\_property()

Son's property 22200

*#super method , to access even the base class constructor*

**class** Father:

**def** \_\_init\_\_(self,propertyfather):

self.propertyfather **=** propertyfather

**def** display\_property(self):

print("Father's property",self.propertyfather)

**class** Son(Father):

**def** \_\_init\_\_(self,propertyson,propertyfather):

super().\_\_init\_\_(propertyfather)

self.propertyson **=** propertyson

**def** display\_property(self):

print("Son's property",self.propertyson,self.propertyfather) *# if constructor in derived class , base class constructor does not get executed*

​

s **=** Son(222,888)

s.display\_property()

Son's property 222 888

*#super method , to access even the base class method*

**class** Father:

**def** \_\_init\_\_(self,propertyfather):

self.propertyfather **=** propertyfather

**def** display\_property(self):

print("Father's property",self.propertyfather)

**class** Son(Father):

**def** \_\_init\_\_(self,propertyson,propertyfather):

super().\_\_init\_\_(propertyfather)

self.propertyson **=** propertyson

**def** display\_property(self):

super().display\_property()

print("Son's property",self.propertyson) *# if constructor in derived class , base class constructor does not get executed*

​

​

Multiple Inheritance

Python supports the idea of multiple inheritance; that is a class can inherit from one

or more other classes (many object-oriented languages limit inheritance to a single

class such as Java and C#).

                      class SubClassName(BaseClassName1, BaseClassName2, …

                              BaseClassNameN):

                          class-body

**class** Car(object):

**def** \_\_init\_\_(self,wheels**=**4):

self.wheels **=** wheels

*# We'll say that all cars, no matter their engine, have four wheels by default.*

​

**class** Gasoline(Car):

**def** \_\_init\_\_(self,engine**=**'Gasoline',tank\_cap**=**20):

Car.\_\_init\_\_(self)

self.engine **=** engine

self.tank\_cap **=** tank\_cap *# represents fuel tank capacity in gallons*

self.tank **=** 0

**def** refuel(self):

self.tank **=** self.tank\_cap

**class** Electric(Car):

**def** \_\_init\_\_(self,engine**=**'Electric',kWh\_cap**=**60):

Car.\_\_init\_\_(self)

self.engine **=** engine

self.kWh\_cap **=** kWh\_cap *# represents battery capacity in kilowatt-hours*

self.kWh **=** 0

**def** recharge(self):

self.kWh **=** self.kWh\_cap

**class** Hybrid(Gasoline, Electric):

**def** \_\_init\_\_(self,engine**=**'Hybrid',tank\_cap**=**11,kWh\_cap**=**5):

Gasoline.\_\_init\_\_(self,engine,tank\_cap)

Electric.\_\_init\_\_(self,engine,kWh\_cap)

prius **=** Hybrid()

print(prius.tank)

print(prius.kWh)

​

print(prius.recharge())

print(prius.kWh)

**Method Resolution Order (MRO)**

Things get complicated when you have several base classes and levels of inheritance. This is resolved using Method Resolution Order - a formal plan that Python follows when running object methods.

​

​

​

**class** Car(object):

**def** move(self):

print('Car - move()')

**class** Toy(object):

**def** move(self):

print('Toy - move()')

**class** ToyCar( Car, Toy):

""" A Toy Car """

**pass**

tc **=** ToyCar()

tc.move()

Car - move()

We can say that the class ToyCar inherits all the attributes (data) and methods (behaviour) defined in classes Car, Toy and object. One of the fundamental questions that this raises is how is inheritance of behaviour managed within a multiple inheritance hierarchy. The challenge that multiple inheritance possesses is illustrated by adding a couple of methods to the class hierarchy we are looking at. In this example we have added the method move() to both the class Car and the class Toy:

The question here is which version of the method move() will be run when an instance of the ToyCar class is instantiated and we call toy\_car.move()? This illustrates (a simple version of) the so-called “diamond inheritance” problem. The issue is that with multiple base classes from which attributes or methods may be inherited, there is often ambiguity that must be resolved. Here, when we create an instance of the class ToyCar, and call the move() method, does this invoke the one inherited from the Car base class or from the Toy base class? The answer is that in Python 3, a breadth first search is used to find methods defined in parent classes; this means that when the method move() is called on ToyCar, it would first look in Car; it would then only look in Toy if it could not find a method move() in Car. If it cannot find the method in either Car or Toy it would then look in the class object. As a result, it will find the version in Car first and use that version.

*# However, if we alter the order in which the ToyCar inherits from the parent*

*# classes such that we swap Toy and Car around:*

**class** ToyCar(Toy, Car):

""" A Toy Car """

**pass**

tc **=** ToyCar()

tc.move()

​

​

*# Then the Toy class is searched first and the output is changed to Toy – move().*

Toy - move()

*# if classes B and C each derive from A, and class D derives from both B and C, which class is "first in line" when a method is called on D?*

*# Consider the following:*

**class** A(object):

num **=** 4

**class** B(A):

**pass**

​

**class** C(A):

**pass**

**class** D(B,C):

**pass**

​

d **=** D()

d.num

​

Out[3]:

4

**class** A(object):

**def** \_\_init\_\_(self):

self.num **=** 4

**class** B(A):

*# super().\_\_init\_\_()*

**pass**

​

**class** C(A):

**def** \_\_init\_\_(self):

self.num **=** 5

**class** D(B,C):

**pass**

d **=** D()

d.num

Out[12]:

5

Schematically, the relationship looks like this:

A

num=4

/ \

/ \

B C

pass num=5

\ /

\ /

D

pass

Here num is a class attribute belonging to all four classes. So what happens if we call D.num?

D.num

Out[14]:

5

You would think that D.num would follow B up to A and return 4. Instead, Python obeys the first method in the chain that defines num. The order followed is [D, B, C, A, object] where object is Python's base object class.

In our example, the first class to define and/or override a previously defined num is C

super() Python's built-in super() function provides a shortcut for calling base classes, because it automatically follows Method Resolution Order.

In its simplest form with single inheritance, super() can be used in place of the base class name :

**class** MyBaseClass:

**def** \_\_init\_\_(self,x,y):

self.x **=** x

self.y **=** y

**class** MyDerivedClass(MyBaseClass):

**def** \_\_init\_\_(self,x,y,z):

super().\_\_init\_\_(x,y)

self.z **=** z

Note that we don't pass self to super().**init**() as super() handles this automatically.

In a more dynamic form, with multiple inheritance like the "diamond diagram" shown above, super() can be used to properly manage method definitions:

**class** A:

**def** truth(self):

**return** 'All numbers are even'

**class** B(A):

**pass**

​

**class** C(A):

**def** truth(self):

**return** 'Some numbers are even'

​

**class** D(B,C):

**def** truth(self,num):

**if** num**%**2 **==** 0:

**return** A.truth(self)

**else**:

**return** super().truth()

d **=** D()

d.truth(6)

d.truth(5)

In the above example, if we pass an even number to d.truth(), we'll believe the A version of .truth() and run with it. Otherwise, follow the MRO and return the more general case

For more information on super() visit <https://docs.python.org/3/library/functions.html#super> and <https://rhettinger.wordpress.com/2011/05/26/super-considered-super/>

​