

[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

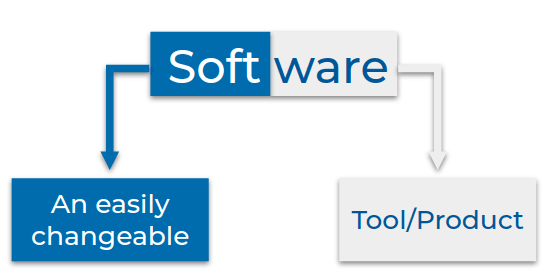
**Introduction to OOP**

Good Software

Before jumping into Object Oriented Programming, let’s understand the word **Software**.

Software is an easily changeable tool/product that performs a specific task.

The ease of changing or making changes to the software is referred as its softness.



* A good software should keep the users happy, by delivering what they need.
* A good software should also keep the developers happy. Ease of making changes (softness) keeps developers happy. So is should be:
  + Easy to understand and make changes.
  + Easy to fix bugs and add new features within the scope.

Object-Oriented Programming System (OOPS) is a way of *approaching*, *designing*, *developing* software that is easy to change.

**Note**

Keep in mind that building software differs from solving the coding questions (that you are doing in practice).

A fundamental difference is that you move on to the next problem once you solve a coding problem. But with software, once you build it, you are often required to add new features and fix bugs that need you to make changes to the code written by you or your colleagues. So unlike the coding questions, with software, you have to keep working with the same code for a long time.

Therefore, ease of understanding (code-readability) and ease of making changes (code-maintainability) to the code become crucial in software development.

The techniques and concepts you learn in this topic are developed over time based on software developers' experiences to make working code easier.

OOPs

Object-Oriented Programming is a way of approaching, designing and developing software, so that the components of the software and the interactions between them resembles real-life objects and their interactions.

Proper usage of OOPS concepts helps us build well-organized systems that are easy to use and extend.

Describing Real-Life Objects

In Object Oriented Programming, we model software after real-life objects. To be good at modeling software after real-life objects, we should be able to properly describe them.

Let us try describing these real-life objects



The following description is a **bad way of describing**, as the information of an object scattered and unorganized.



Object 1 is a car and it has four tyres.

Object 2 is a dog and it has four legs.

Object 1 has four doors.

Object 1 can make sound.

Object 2, barks.

Object 1 is in blue color.

Object 2 is in brown color.

Organized Description

* In the below description we are grouping the information related to an object.



Object 1 is a car and it has four tyres.

Object 1 has four doors.

Object 1 can make sound.

Object 1 is in blue color.

Object 2 is a dog and it has four legs. Object 2, barks.

Object 2 is in brown color

* In the below approach, we further organize the information into
  + What the object is?
  + What the object has?
  + What the object can do?



Object 1 is a car

Object 1 has

Four tyres

Four seats

Four doors

and so on ...

Object 1 can

Sound horn

Move

Accelerate

The above description shows a simple framework where we describe object by specifying the properties that the object has and actions that the object can do.

Organized Description should be

* A clear separation of objects.
* A clear grouping of what object has and what it does.

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

# Describing Similar Objects

Sometimes, the objects that we are describing are so similar that only values of the properties differ.





Object 3 is a Mobile

Properties

camera : 13 MP

storage : 16 GB

battery life : 21 Hrs

ram : 3 GB

and so on ...

Object 4 is a Mobile

Properties

camera : 64 MP

In this case, the objects that we describe have completely the same set of properties like camera, storage, etc.

## Template

For objects that are very similar to each other (objects that have the same set of actions and properties), we can create a standard **Form** or **Template** that can be used to describe different objects in that category.

Mobile Template



Model :

Camera:

Storage:

Does it have a Face Unlock? Yes | No

Filled Template



Model : iPhone 12 Pro

Camera: 64MP

Storage: 128GB

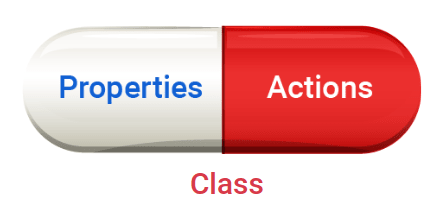
Does it have a Face Unlock? Yes

## Bundling Data

While modeling real-life objects with object oriented programming, we ensure to bundle related information together to clearly separate information of different objects.

Bundling of related properties and actions together is called Encapsulation.

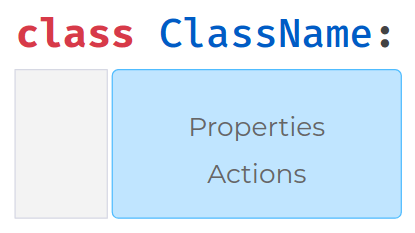
Classes can be used to bundle related properties and actions.



## Defining a Class

To create a class, use the keyword

class



## Special Method

In Python, a special method

\_\_init\_\_

is used to assign values to properties.

#### Code



1

2

3

4

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera = camera

PYTHON

## Properties & Values

#### Code



1

2

3

4

5

6

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera = camera

def make\_call(self, number):

print("calling..")

PYTHON

In the above example,

model

and

camera

are the properties and values are which passed to the

\_\_init\_\_

method.

## Action



1

2

3

4

5

6

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera = camera

def make\_call(self, number):

print("calling..")

PYTHON

In the above example, the below function is an action



1

2

def make\_call(self, number):

print("calling..")

PYTHON

In OOP terminology, properties are referred as **attributes** actions are referred as **methods**

### **Using a Class**

To use the defined class, we have to instantiate it. A class is like a blueprint, while its instance is based on that class with actual values.

## Instance of Class

Syntax for creating an instance of class looks similar to function call.

An instance of class is **Object**.

#### Code



1

2

3

4

5

6

7

8

9

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera= camera

mobile\_obj = Mobile(

"iPhone 12 Pro",

"12 MP")

print(mobile\_obj)

PYTHON

## Class Object

An object is simply a collection of attributes and methods that act on those data.



1

2

3

4

5

6

7

8

9

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera= camera

mobile\_obj = Mobile(

"iPhone 12 Pro",

"12 MP")

print(mobile\_obj)

PYTHON

## Method Arguments & Return Values

Similar to functions, Methods also support positional, keyword & default arguments and also return values.

#### Code



1

2

3

4

5

6

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera= camera

def make\_call(self,number):

return "calling..{}".format(number)

PYTHON

## Instance Methods of Class

For instance method, we need to first write

self

in the function definition and then the other arguments.

#### Code



1

2

3

4

5

6

7

8

9

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera= camera

def make\_call(self,number):

print("calling..{}".format(number))

mobile\_obj = Mobile("iPhone 12 Pro", "12 MP")

mobile\_obj.make\_call(9876543210)

PYTHON

#### Output



calling..9876543210

## Multiple Instances

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera= camera

def make\_call(self,number):

print("calling..{}".format(number))

mobile\_obj1 = Mobile("iPhone 12 Pro", "12 MP")

print(id(mobile\_obj1))

mobile\_obj2 = Mobile("Galaxy M51", "64 MP")

print(id(mobile\_obj2))

PYTHON

#### Output



139685004996560

139685004996368

## Type of Object

The class from which object is created is considered to be the type of object.

#### Code



1

2

3

4

5

6

7

class Mobile:

def \_\_init\_\_(self, model, camera):

self.model = model

self.camera = camera

obj\_1 = Mobile("iPhone 12 Pro", "12 MP")

print(type(obj\_1))

PYTHON

#### Output



<class '\_\_main\_\_.Mobile'>

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

# Classes and Objects

### **Attributes of an Object**

Attributes can be set or accessed using

.

(dot) character.

#### Code



1

2

3

4

5

6

7

8

class Mobile:

def \_\_init\_\_(self, model, storage):

self.model = model

self.storage = storage

obj = Mobile("iPhone 12 Pro", "128GB")

print(obj.model)

PYTHON

#### Output



iPhone 12 Pro

### **Accessing in Other Methods**

We can also access and update properties in other methods.

#### Code



1

2

3

4

5

6

7

8

9

10

class Mobile:

def \_\_init\_\_(self, model):

self.model = model

def get\_model(self):

print(self.model)

obj\_1 = Mobile("iPhone 12 Pro")

obj\_1.get\_model()

PYTHON

#### Output



iPhone 12 Pro

### **Updating Attributes**

It is recommended to update attributes through methods.

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Mobile:

def \_\_init\_\_(self, model):

self.model = model

def update\_model(self, model):

self.model = model

obj\_1 = Mobile("iPhone 12")

print(obj\_1.model)

obj\_1.update\_model("iPhone 12 Pro")

PYTHON

#### Output



iPhone 12

iPhone 12 Pro

## Modeling Class

Let’s model the scenario of shopping cart of ecommerce site.

The features a cart should have

* can add an item
* can remove an item from cart
* update quantity of an item
* to show list of items in cart
* to show total price for the items in the cart

#### Code



1

2

3

4

5

6

7

8

9

10

11

12

class Cart:

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

self.items = {}

self.price\_details = {"book": 500, "laptop": 30000}

def add\_item(self, item\_name, quantity):

self.items[item\_name] = quantity

def remove\_item(self, item\_name):

del self.items[item\_name]

PYTHON

#### Output



31500

['book']

1000

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

# Attributes & Methods

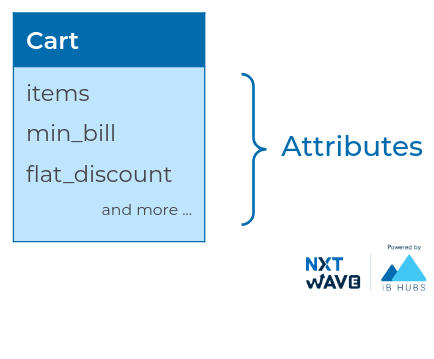
### **Shopping Cart**

* Users can add different items to their shopping cart and checkout.
* The total value of the cart should be more than a minimum amount (Rs. 100/-) for the checkout.
* During Offer Sales, all users get a flat discount on their cart and the minimum cart value will be Rs. 200/-.

### **Attributes**

Broadly, attributes can be categorized as

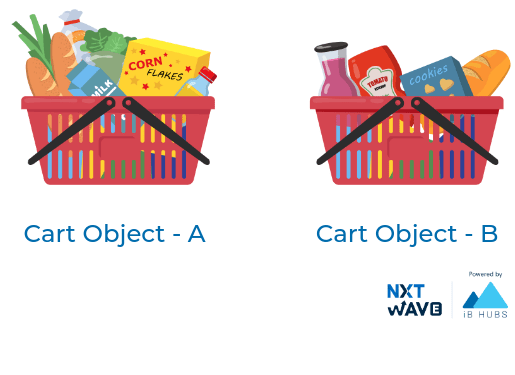
* Instance Attributes
* Class Attributes



### **Instance Attributes**

Attributes whose value can differ for each instance of class are modeled as instance attributes.

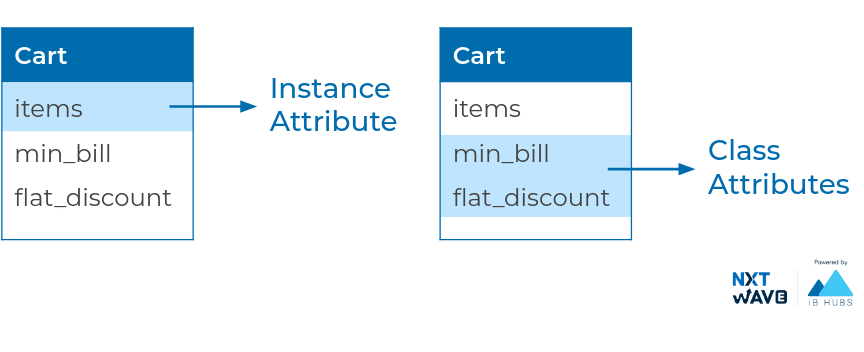
*Ex*: Items in Cart



### **Class Attributes**

Attributes whose values stay common for all the objects are modelled as Class Attributes.

*Ex*: Minimum Cart Bill, Flat Discount



### **Accessing Instance Attributes**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Cart:

flat\_discount = 0

min\_bill = 100

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

self.items = {}

def add\_item(self,..):

self.items[item\_name] = quantity

def display\_items(self):

print(items)

a = Cart()

a.display\_items()

PYTHON

#### Output



NameError: name 'items' is not defined

Instance attributes can only be accessed using instance of class.

## Self

self

passed to method contains the object, which is an instance of class.

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Cart:

flat\_discount = 0

min\_bill = 100

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

self.items = {}

def add\_item(self,item\_name, quantity):

self.items[item\_name] = quantity

def display\_items(self):

print(self)

a = Cart()

PYTHON

#### Output



<\_\_main\_\_.Cart object at 0x7f6f83c9dfd0>

<\_\_main\_\_.Cart object at 0x7f6f83c9dfd0>

### **Accessing Using Self**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Cart:

flat\_discount = 0

min\_bill = 100

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

self.items = {}

def add\_item(self, item\_name,quantity):

self.items[item\_name] = quantity

def display\_items(self):

print(self.items)

a = Cart()

a.add\_item("book", 3)

PYTHON

#### Output



{"book": 3}

### **Accessing Using Object**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Cart:

flat\_discount = 0

min\_bill = 100

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

self.items = {}

def add\_item(self, item\_name,quantity):

self.items[item\_name] = quantity

def display\_items(self):

print(self.items)

a = Cart()

a.add\_item("book", 3)

PYTHON

#### Output



{'book': 3}

### **Accessing Using Class**

#### Code



1

2

3

4

5

6

7

8

9

10

class Cart:

flat\_discount = 0

min\_bill = 100

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

self.items = {}

def add\_item(self, item\_name,quantity):

self.items[item\_name] = quantity

def display\_items(self):

print(self.items)

print(Cart.items)

PYTHON

### **Output**



AttributeError: type object 'Cart' has no attribute 'items'

### **Accessing Class Attributes**

#### **Example 1**

#### Code



1

2

3

4

5

6

7

class Cart:

flat\_discount = 0

min\_bill = 100

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

self.items = {}

print(Cart.min\_bill)

PYTHON

### **Output**



100

#### **Example 2**

#### Code



1

2

3

4

5

6

7

8

9

10

class Cart:

flat\_discount = 0

min\_bill = 100

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

self.items = {}

def print\_min\_bill(self):

print(Cart.min\_bill)

a = Cart()

a.print\_min\_bill()

PYTHON

### **Output**



100

### **Updating Class Attribute**

#### Code



1

2

3

4

5

6

7

8

9

10

class Cart:

flat\_discount = 0

min\_bill = 100

def print\_min\_bill(self):

print(Cart.min\_bill)

a = Cart()

b = Cart()

Cart.min\_bill = 200

print(a.print\_min\_bill())

print(b.print\_min\_bill())

PYTHON

### **Output**



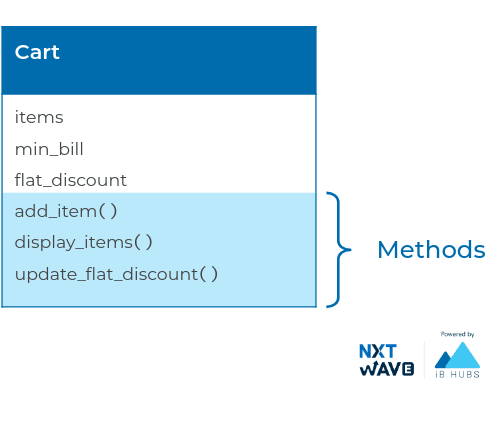
200

200

### **Method**

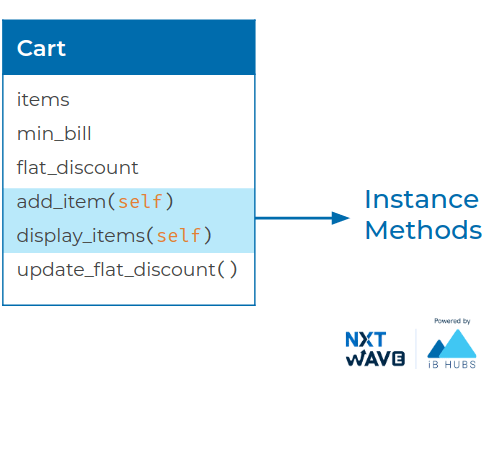
Broadly, methods can be categorized as

* Instance Methods
* Class Methods
* Static Methods



### **Instance Methods**

Instance methods can access all attributes of the instance and have self as a parameter.



#### **Example 1**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Cart:

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

self.items = {}

def add\_item(self, item\_name,quantity):

self.items[item\_name] = quantity

def display\_items(self):

print(self.items)

a = Cart()

a.add\_item("book", 3)

a.display\_items()

PYTHON

### **Output**



{'book': 3}

#### **Example 2**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Cart:

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

self.items = {}

def add\_item(self, item\_name,quantity):

self.items[item\_name] = quantity

self.display\_items()

def display\_items(self):

print(self.items)

a = Cart()

a.add\_item("book", 3)

PYTHON

### **Output**



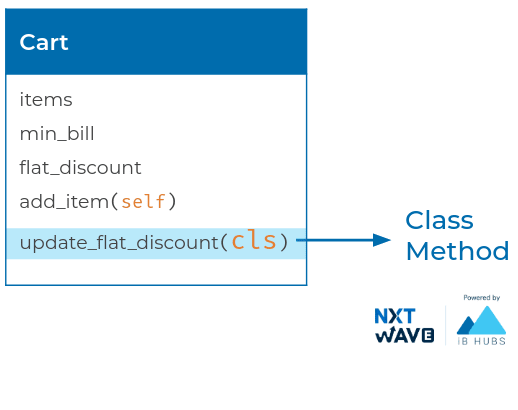
{'book': 3}

### **Class Methods**

Methods which need access to class attributes but not instance attributes are marked as Class Methods. For class methods, we send

cls

as a parameter indicating we are passing the class.



#### Code



1

2

3

4

5

6

7

8

9

10

class Cart:

flat\_discount = 0

min\_bill = 100

@classmethod

def update\_flat\_discount(cls,

new\_flat\_discount):

cls.flat\_discount = new\_flat\_discount

Cart.update\_flat\_discount(25)

print(Cart.flat\_discount)

PYTHON

#### Output



25

@classmethod

decorator marks the method below it as a class method.

We will learn more about decorators in upcoming sessions.

### **Accessing Class Method**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Cart:

flat\_discount = 0

min\_bill = 100

@classmethod

def update\_flat\_discount(cls, new\_flat\_discount):

cls.flat\_discount = new\_flat\_discount

@classmethod

def increase\_flat\_discount(cls, amount):

new\_flat\_discount = cls.flat\_discount + amount

cls.update\_flat\_discount(new\_flat\_discount)

PYTHON

#### Output



50

### **Static Method**

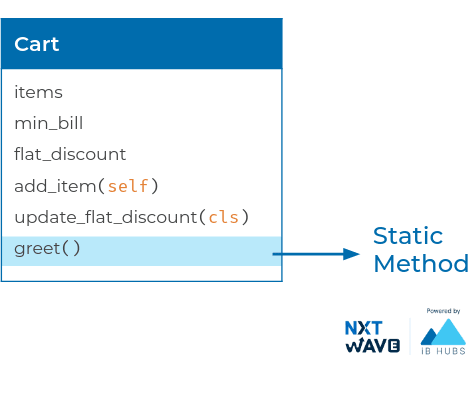
We might need some generic methods that don’t need access to either instance or class attributes. These type of methods are called Static Methods.

Usually, static methods are used to create utility functions which make more sense to be part of the class.

@staticmethod

decorator marks the method below it as a static method.

We will learn more about decorators in upcoming sessions.



#### Code



1

2

3

4

5

6

7

class Cart:

@staticmethod

def greet():

print("Have a Great Shopping")

Cart.greet()

PYTHON

#### Output



Have a Great Shopping

### **Overview of Instance, Class & Static Methods**

| Instance Methods | Class Methods | Static Methods |
| --- | --- | --- |
| self as parameter | cls as parameter | No cls or self as parameters |
| No decorator required | Need decorator @classmethod | Need decorator @staticmethod |
| Can be accessed through object(instance of class) | Can be accessed through class | Can be accessed through class |

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

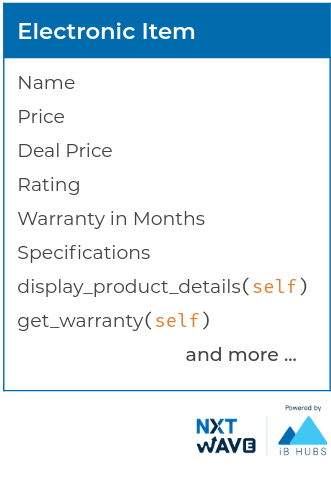
# Inheritance

### **Products**

Lets model e-commerce site having different products like Electronics, Kids Wear, Grocery, etc.

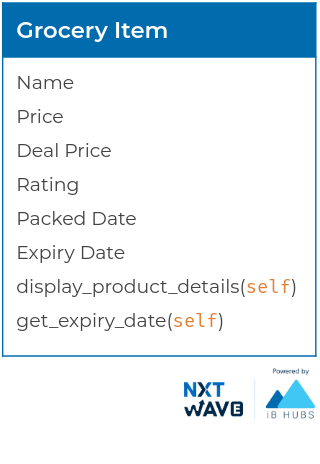
### **Electronic Item**

Following are few attributes & methods for an Electronic product.



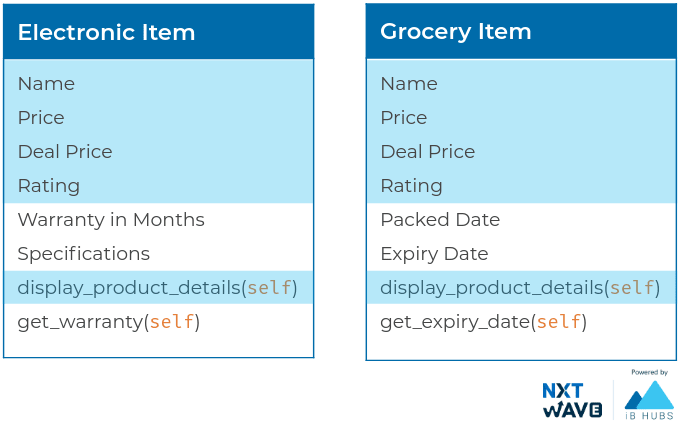
### **Grocery Item**

Similarly, attribute & methods for a Grocery item.



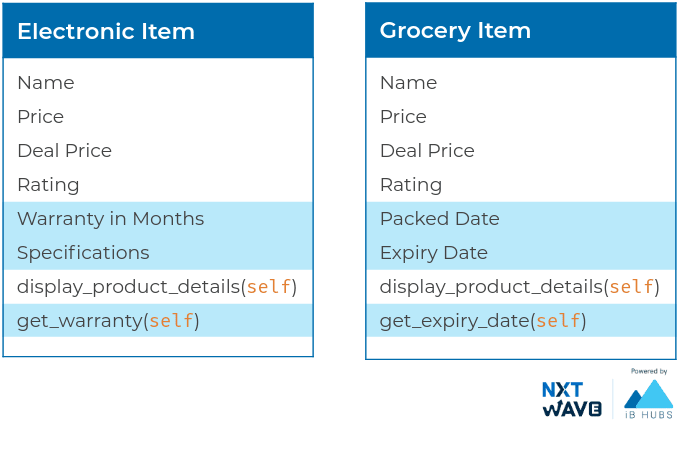
### **Common Attributes & Methods**

All these products Electronics, Kids Wear, Grocery etc.. have few common attributes & methods.



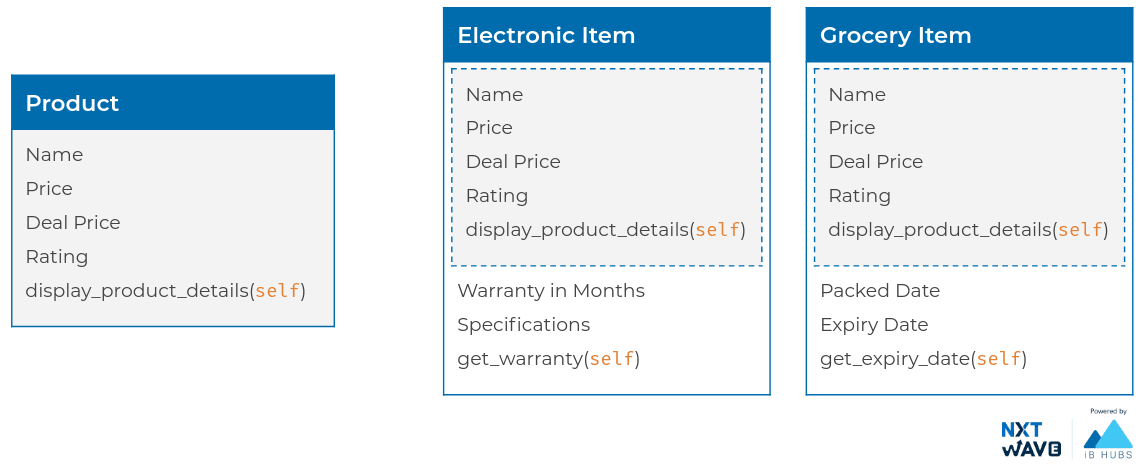
### **Specific Attributes & Methods**

Also, each product has specific attributes & methods of its own.

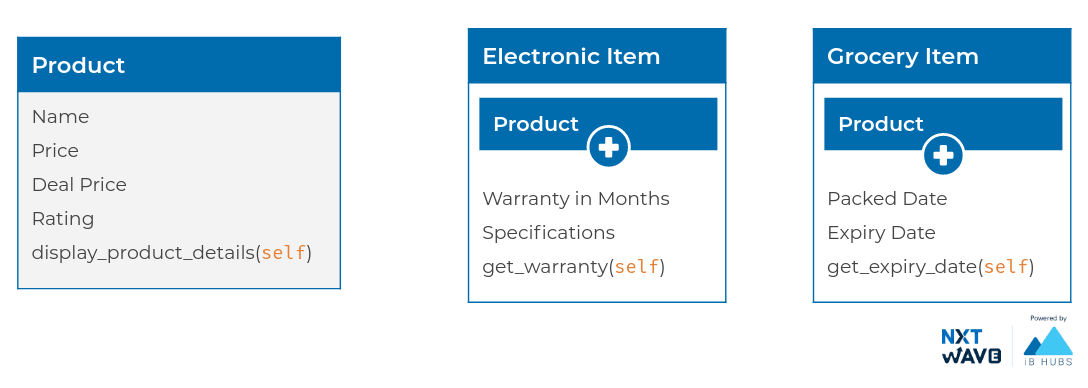


### **Electronic & Grocery Items**

Electronic Item & Grocery Item will have all attributes & methods which are common to all products. Lets Separate the common attributes & methods as Product



### **Modelling Classes**



### **Advantages of Modelling Classes as above**

* Reusability
* Clear Separation
* More Organized

### **Inheritance**

Inheritance is a mechanism by which a class inherits attributes and methods from another class.

With Inheritance, we can have

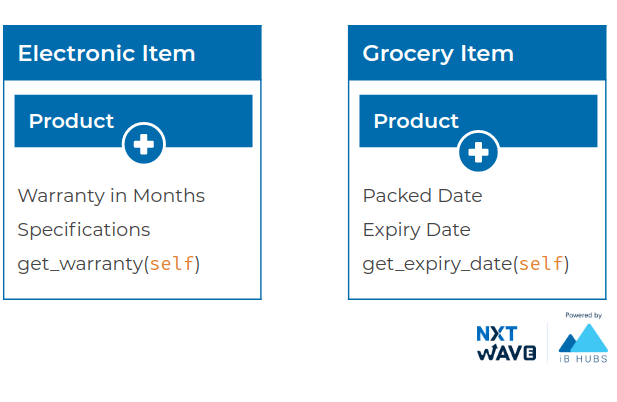
ElectronicItem

inherit the attributes & methods from

Product

instead of defining them again.

Product is Super/Base/Parent Class and ElectronicItem is Sub/Derived/Child Class.



### **Super Class**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

print("Price: {}".format(self.price))

print("Deal Price: {}".format(self.deal\_price))

PYTHON

#### Output



Product: Shoes

Price: 500

Deal Price: 250

You Saved: 250

Ratings: 3.5

### **Sub Class**

The subclass automatically inherits all the attributes & methods from its superclass.

### ***Example 1***

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

print("Price: {}".format(self.price))

print("Deal Price: {}".format(self.deal\_price))

PYTHON

#### Output



Product: TV

Price: 45000

Deal Price: 40000

You Saved: 5000

Ratings: 3.5

### ***Example 2***

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

print("Price: {}".format(self.price))

print("Deal Price: {}".format(self.deal\_price))

PYTHON

#### Output



Product: milk

Price: 25

Deal Price: 20

You Saved: 5

Ratings: 3

### ***Example 3***

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

print("Price: {}".format(self.price))

print("Deal Price: {}".format(self.deal\_price))

PYTHON

#### Output



24

In the above example, calling

set\_warranty

will create an attribute

warranty\_in\_months

.

### **Super Class & Sub Class**

Superclass cannot access the methods and attributes of the subclass.

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

print("Price: {}".format(self.price))

print("Deal Price: {}".format(self.deal\_price))

PYTHON

#### Output



AttributeError: 'Product' object has no attribute 'set\_warranty'

### **Sub Class Method**

#### Code



1

2

3

4

5

6

7

8

9

10

11

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

print("Price: {}".format(self.price))

print("Deal Price: {}".format(self.deal\_price))

PYTHON

#### Output



Product: TV

Price: 45000

Deal Price: 40000

You Saved: 5000

Ratings: 3.5

### **Calling Super Class Method**

We can call methods defined in superclass from the methods in the subclass.

#### Code



1

2

3

4

5

6

7

8

9

10

11

12

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

print("Price: {}".format(self.price))

print("Deal Price: {}".format(self.deal\_price))

PYTHON

#### Output



Product: TV

Price: 45000

Deal Price: 40000

You Saved: 5000

Ratings: 3.5

Warranty 24 months

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

# Inheritance - Part 2

**How would you design and implement placing order with the details of all the products bought?**

## Composition

Modelling instances of one class as attributes of another class is called **Composition**

#### Code



1

2

3

4

5

6

7

8

9

10

11

12

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

PYTHON

#### Output



Product: Milk

Price: 40

Deal Price: 25

You Saved: 15

Ratings: 3.5

Quantity: 2

Product: TV

Price: 45000

Deal Price: 40000

You Saved: 5000

Ratings: 3.5

In the above example, we are modelling **Product** as attribute of **Order**

### **Overriding Methods**

Sometimes, we require a method in the instances of a sub class to behave differently from the method in instance of a superclass.

#### Code



1

2

3

4

5

6

7

8

9

10

11

12

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

PYTHON

#### Output



RecursionError: maximum recursion depth exceeded

Because

self.display\_product\_details()

in ElectronicItem class does not call the method in the superclass.

## Super

### **Accessing Super Class’s Method**

super()

allows us to call methods of the superclass (Product) from the subclass.

Instead of writing and methods to access and modify warranty we can override

\_\_init\_\_

Let's add warranty of ElectronicItem.

#### Code



1

2

3

4

5

6

7

8

9

10

11

12

class Product:

def \_\_init\_\_(self, name, price, deal\_price, ratings):

self.name = name

self.price = price

self.deal\_price = deal\_price

self.ratings = ratings

self.you\_save = price - deal\_price

def display\_product\_details(self):

print("Product: {}".format(self.name))

PYTHON

#### Output



Product: Laptop

Price: 45000

Deal Price: 40000

You Saved: 5000

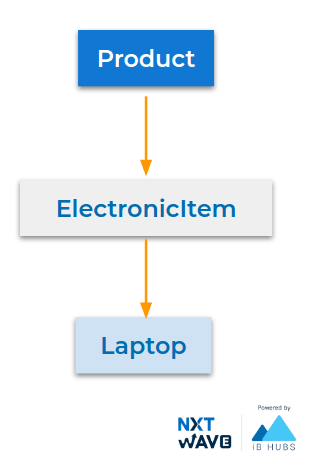
Ratings: 3.5

Warranty 10 months

## MultiLevel Inheritance

We can also inherit from a subclass. This is called **MultiLevel Inheritance**.

We can continue such inheritance to any depth in Python.





1

2

3

4

5

6

7

8

class Product:

pass

class ElectronicItem(Product):

pass

class Laptop(ElectronicItem):

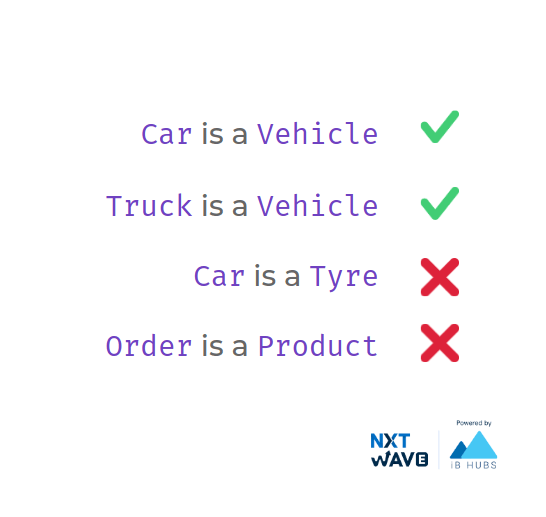
pass

PYTHON

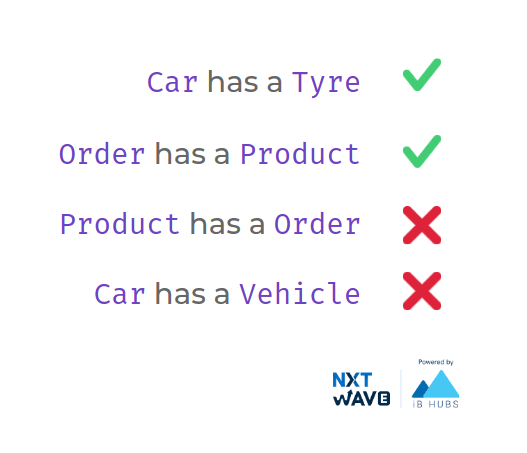
## Inheritance & Composition

### **When to use Inheritance?**

Prefer modeling with inheritance when the classes have an **IS-A** relationship.



### **When to use Composition?**

Prefer modeling with inheritance when the classes have an **HAS-A** relationship. 

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1