1. What is 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 resemble real-life objects and their interactions.

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

2. What are the advantages of OOPs?

The advantages of OOPs:

- Easier way to analyse and solve bugs
- Reusability of code through inheritance
- Effective problem solving
- Elimination of code redundancy

3. What are the principles of OOPs?

The principles of OOPs involve,

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user. This is one of the core concepts of object-oriented programming (OOP) languages.

Polymorphism is an OOPs concept. The word polymorphism means having many forms. It refers to the use of a single type entity (method, operator, or object) to represent different types in different scenarios

4. What are Classes?

A

`class` is a prototype from which objects are created. Classes can be used to bundle related attributes and methods. An instance of a `class` is an Object.

Code

PYTHON

```python
class Mobile:
    def __init__(self, model, storage):
        self.model = model
        self.storage = storage


obj = Mobile("iPhone 12 Pro", "128GB")
print(obj.model)
```

Output

```
iPhone 12 Pro
```

5. What is `__init__` / `constructor` in Python?

The

`__init__` is a special method used to initialize values to attributes.

Code

PYTHON

```python
class Mobile:
    def __init__(self, model, camera):
        self.model = model
        self.camera = camera
    def make_call(self, number):
        print("calling..{}".format(number))

mobile = Mobile("Nikon", "D850")
mobile.make_call("12345")
```

In the above example, the

`model` and `camera` attributes are initialized with the values that are passed to the `__init__` method.

6. What is `self` in OOPs?

In Python, the

`self` is the first parameter of methods that represents the instance of the class. Therefore, to call attributes and methods of a class, the programmer need to use `self` within the class.

`self` is not a keyword and has no special meaning in Python. Writing this parameter as self is a convention. We can use other names but it is highly discouraged.

Code

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

    def info(self):
```

```
        print("My name is {}".format(self.name))

    def make_sound(self):
        print("Bow Wow")

dog1 = Dog('Rex', 2)

dog1.info()
```

Collapse

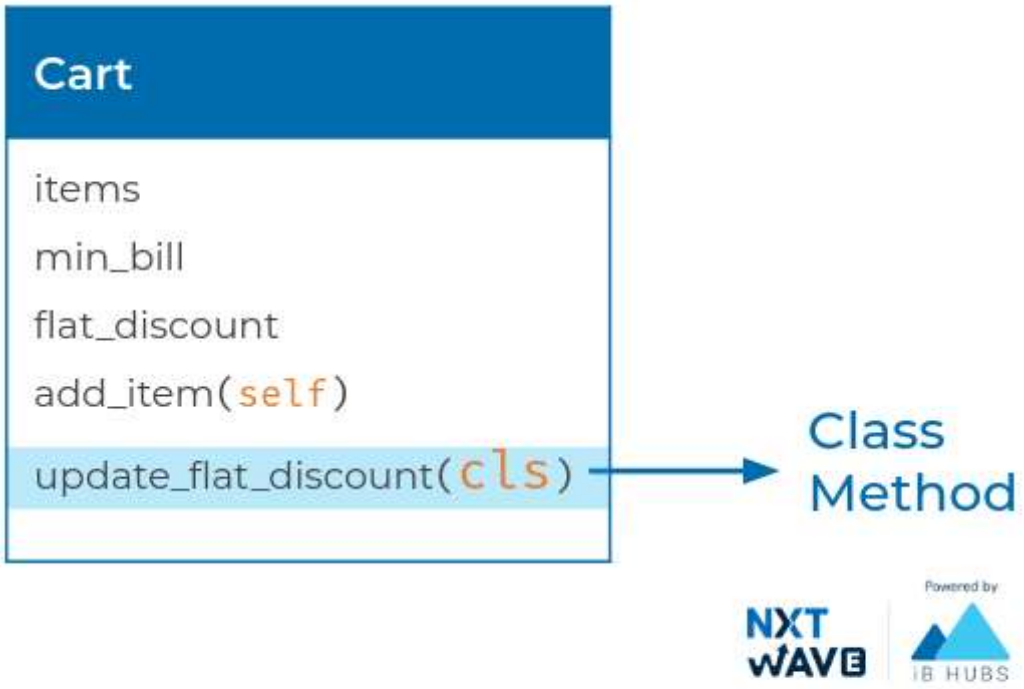
Output

```
My name is Rex
```

7. What are 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

```
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)
```

Output

```
25
```

`@classmethod` decorator marks the method below it as a class method.

8. What is Inheritance?

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

The class whose attributes and methods are inherited is known as the Super/Base/Parent Class. And the class that inherits the attributes and methods from the parent class is the Sub/Derived/Child Class.

9. Write an example program to show Inheritance?

Code

PYTHON

```
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))
        print("You Saved: {}".format(self.you_save))
        print("Ratings: {}".format(self.ratings))

class ElectronicItem(Product):
    def set_warranty(self, warranty_in_months):
        self.warranty_in_months = warranty_in_months

    def get_warranty(self):
        return self.warranty_in_months

e = ElectronicItem("TV", 45000, 40000, 3.5)
e.display_product_details()
```

Collapse

Output

```
Product: TV
Price: 45000
Deal Price: 40000
You Saved: 5000
Ratings: 3.5
```

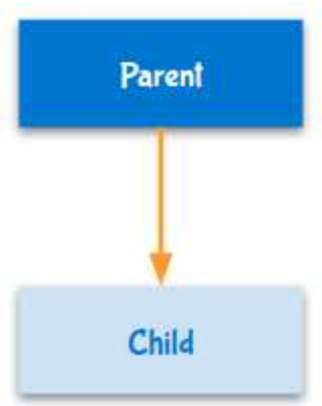
In the above example, the child class **ElectronicItem** inherits all the attributes and methods from the parent class **Product**.

10. What are the types of Inheritance?

There are 5 types of Inheritance in Python:

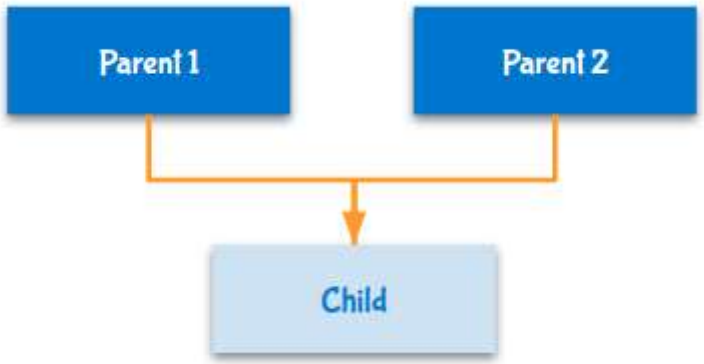
1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Single Inheritance:



When a Child class inherits attributes and methods from a single Parent class, it is called Single Inheritance.

Multiple Inheritance:

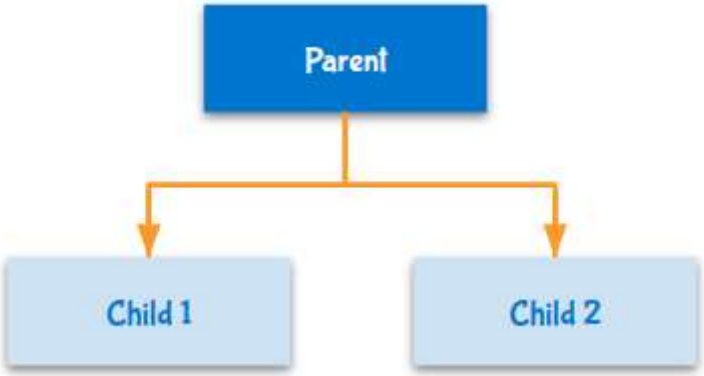


When a Child class inherits attributes and methods from more than one Parent class, it is called Multiple Inheritance.

Multilevel Inheritance:

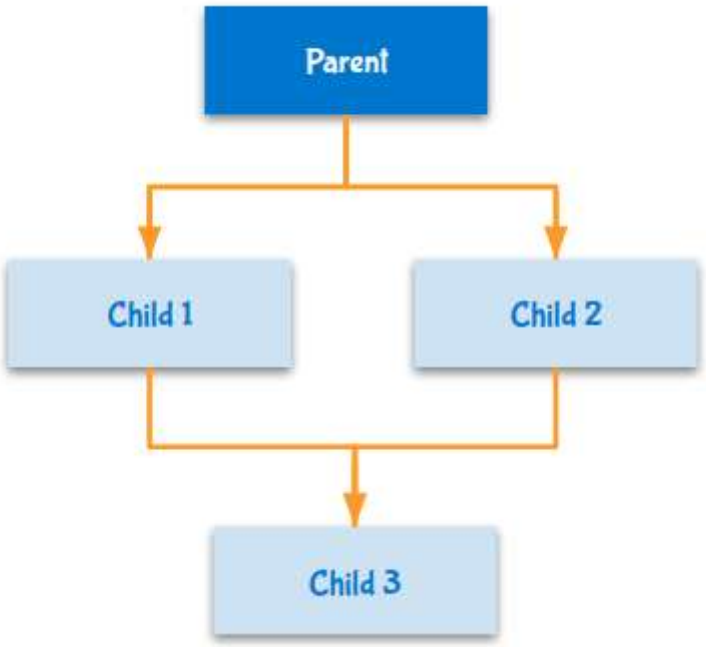
When a Child class inherits attributes and methods from another Child class, it is called Multilevel Inheritance.

Hierarchical Inheritance:



When more than one Child class inherits attributes and methods from a single Parent class, it is called Hierarchical Inheritance.

Hybrid Inheritance:



Any combination of more than one type of inheritance is called Hybrid Inheritance.

11. What is Encapsulation?

While modelling objects with object-oriented programming, we bundle related information together to clearly separate information of different objects.

The bundling of related attributes and methods together is called Encapsulation.

Classes can be used to bundle related attributes and methods.

12. What is Method Overriding?

Method Overriding is an OOPs concept related to Inheritance. When a child class method overrides the parent class method of the same name, parameters and return type, it is known as Method Overriding.

Method Overriding allows us to change the implementation of a function in the child class that is defined in the parent class.

Code

```
class Product:

    def __init__(self, name, price, deal_price, rating):
        self.name = name
        self.price = price
        self.deal_price = deal_price
        self.rating = rating
```

```
        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))
        print("You Saved: {}".format(self.you_save))
        print("Rating: {}".format(self.rating))

    def get_deal_price(self):
        return self.deal_price

class ElectronicItem(Product):

    def display_product_details(self):
        super().display_product_details()
        print("Warranty {} months".format(self.warranty_in_months))

    def set_warranty(self, warranty_in_months):
        self.warranty_in_months = warranty_in_months

    def get_warranty(self):
        return self.warranty_in_months

e = ElectronicItem("Laptop",45000, 40000,3.5)
e.set_warranty(10)
e.display_product_details()
```

Collapse

Output

```
Product: Laptop
Price: 45000
Deal Price: 40000
You Saved: 5000
Rating: 3.5
Warranty 10 months
```

In the above example, the

`display_product_details()` method in the **ElectronicItem** class overrides the `display_product_details()` method of the **Product** class.

MARK AS COMPLETED

Submit Feedback

Notes Discussions