Classes and Objects

Friday, 4 November 2022 3:30 pm

OOP is a real world approach to solve a problem.

Main Concepts of OOP:

1. Classes and Objects
2. Inheritance
3. Polymorphism
4. Abstraction
5. Interfaces
6. Encapsulation

Python may not contain all the OOP concepts. Such as,

Class

A class is a blueprint from which objects are created (a collection of objects).
A class defines a new datatype from which objects of that type are also created.

A class contains attributes (class variables) and methods (functions).

Objects

An object is an instance of a class. It may act like a real-world object and has a state and behavior associated with it.

State --> attributes of the object
Behavior --> methods of the object

Constructor in Python

A constructor immediately initializes an object's attributes upon creation.
It is defined by the __init__ method in Python

Additional note: __self__ method in a class refers to the object calling that class

```python
class Cat:
    species = 'mammal'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_oldest_cat(self,*args):
        # *args because we need to compare all objects age variable
        return max(args)

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def get_name(self):
        return self.name
```

```python
# Instantiate the Cat object with 3 cats
Lulu = Cat("Lulu",5)
Chuchu = Cat("Chuchu",7)
Momo = Cat("Momo",6)

# Print out: "The oldest cat is x years old."
print(f"Oldest cat is {Lulu.get_oldest_cat(Lulu.age,Chuchu.age,Momo.age)}")
print(Lulu) # Printing objects invokes __str__ method that converts object to string
print(Lulu.get_name())
```

Inheritance

Friday, 4 November 2022 3:30 pm

Inheritance is the ability of one class to inherit the properties from another class. The class that inherits properties is called the **child class (sub class)** and the class from which the properties are being inherited is called the **parent class (super class).**

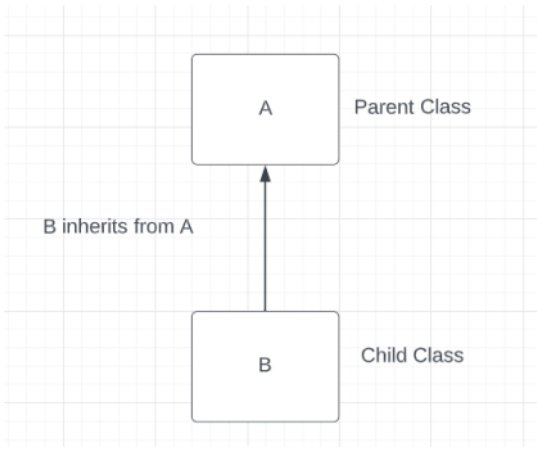
We can reuse attributes and methods of the parent class. Moreover, we can add new methods in the child classes as well.

The benefits of inheritance are:

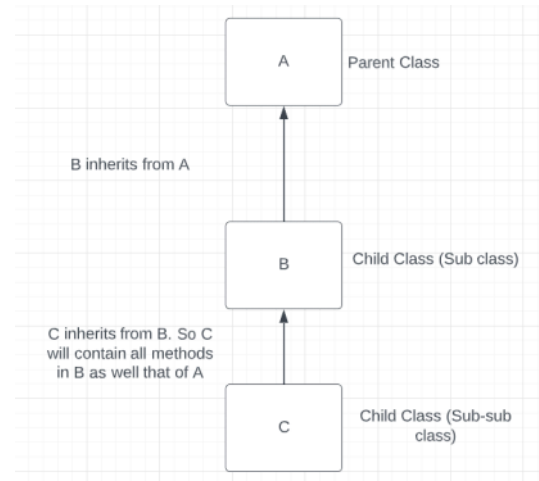
- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of inheritance

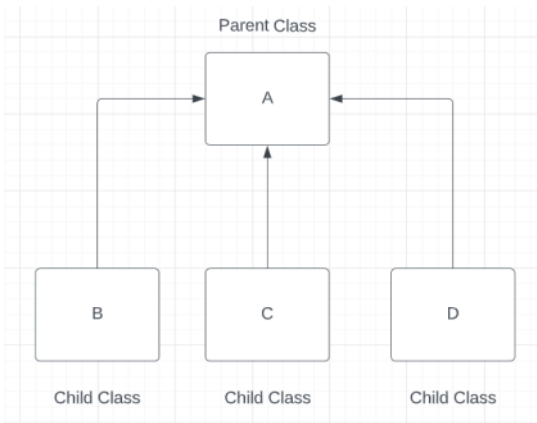
1) Single Inheritance - one child class inherits from parent class



2) Multilevel inheritance



3) Hierarchical inheritance - more than one child class inherits from a parent class



Things to remember:

1) super() --> access all features of the parent class (attributes + methods). e.g

```
super().__init__(person_name,year_of_birth) # call parent class constructor
```

2) If there is no constructor in child class, python goes to the one in parent class.
3) If child class has a constructor, then it will override parent class constructor. Same applied for methods **(method overriding)**

```python
class Person:

    # Set type for the variables (better), and you can set default values
    def __init__(self, person_name: str, year_of_birth: int):
        self.__person_name = person_name
        self.__year_of_birth = year_of_birth

    def get_name(self):
        return self.__person_name

    def get_year_of_birth(self):
        return self.__year_of_birth

    # check if user has entered a valid name - replaced later with regular expressions
    def __has_any_number(self, name:str):
        return "0" in name

    def set_new_name(self, new_name:str):
        if self.__has_any_number(new_name):
            print("You are not allowed to set this name")
        self.__person_name = new_name

    def person_details(self):
        return f"Person => Name = {self.__person_name}, YOB = {self.__year_of_birth}"
```

```python
class Student(Person):

    def __init__(self, person_name:str, year_of_birth:int, email_id:str, student_id:str):
        super().__init__(person_name,year_of_birth) # call parent class constructor
        self.email_id = email_id
        self.student_id = student_id

    # override method from parent class (method overriding)
    def person_details(self):
        return f"Student => Email = {self.email_id} , ID - {self.student_id}"

    # if we directly want to print object details (without mentioning method again and again)
    # we will override dunder methods (__example__)

    def __str__(self):
        return f"Email = {self.email_id} , ID - {self.student_id}"

    #This dunder method helps in debugging (representing object)
    def __repr__(self):
        return f"Email = {self.email_id} , ID - {self.student_id}"
```

```python
s1 = Student("Jawwad",1998,"jk@hotmail.com","EK564")
print(s1.person_details())
print(s1) # print object details (overwrite dunder methods)
```

Interfaces and Abstract classes

Monday, 7 November 2022 1:49 pm

Abstraction means hiding away information or abstracting way information (lower level information) and giving access for only what's necessary (higher level information)

In abstraction, lower level details are hidden and presented at a higher level.

There is no Encapsulation nor Abstraction Python.

Though abstract classes exist in Django (Python Web Framework), but they function in a different way.

Exceptions and their handling

Monday, 7 November 2022 1:49 pm

Errors in Python can be of two types i.e**. Syntax errors and Exceptions**.
Errors are the problems in a program due to which the program will stop the execution. On the other hand, **exceptions are raised** when some internal events occur which changes the normal flow of the program.

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Exceptions: Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Some common Exceptions:

- Type error --> operation between different types
- Name error --> variable not defined
- Index error --> index out of range
- Key error --> key does not exist
- Zero Division

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Therefore,

```
try:
    # test block of code for error
except:
    # handle error if it occurs
else:
    # block of code to be executed if no error was raised (optional)
```

```
OP > demo_oop.py > ...
while True:
    try:
        age = int(input("Enter your age = "))
    except Exception as ex:
        print(f"Error: {repr(ex)}") # repr shows exception class name
    else:
        print("Thank you for entering your age")
        break
```

Raise an Exception

Sometimes, we can define our own set of exceptions

```
x = 'hello'
if type(x) is not x:
    raise TypeError("x should be an int")
```