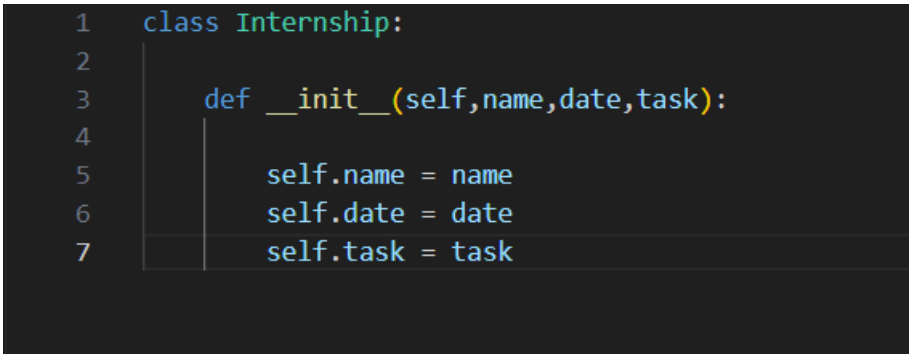
OOPS Concepts in Python Documentation

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to organize and structure code. Python is a versatile programming language that fully supports OOP principles. This documentation provides an overview of essential OOP concepts in Python.

1. Class & Objects:

   In Python, a class is a blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class will have.

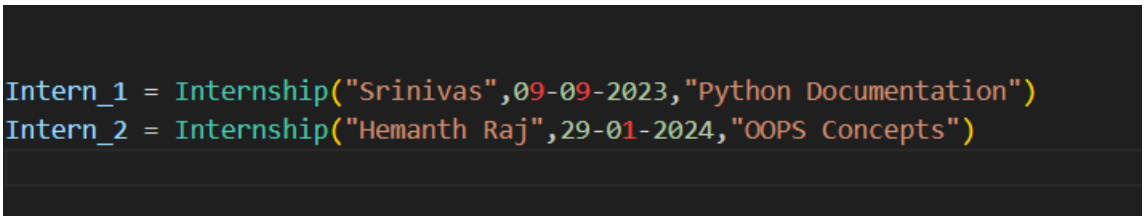


```python
class Internship:

    def __init__(self,name,date,task):

        self.name = name
        self.date = date
        self.task = task
```

   Objects are the instances of the class. They encapsulate the data and behavior.



```python
Intern_1 = Internship("Srinivas",09-09-2023,"Python Documentation")
Intern_2 = Internship("Hemanth Raj",29-01-2024,"OOPS Concepts")
```

2. Methods & Class Attributes:

Classes have variables which can be defined as Attributes

```
class Internship:

    pie = 3.14
```

,and they also have functions which can be defined as Methods.

```
    def details_info(self):
        print(f"{self.name} - {self.task}")


Intern_1.details_info()
```

3. Class Methods & Static Methods:

We can use two kinds of methods other than regular methods(functions). We use class decorators to declare the class method which uses class as an argument so that we can change the class variables dynamically from a method.

```
    @classmethod
    def change_pie(cls,add):
        cls.pie = cls.pie + add


Internship.change_pie(2)
```

4. Encapsulation:

Access Modifiers: Python uses access modifiers like public, protected, and private to control the visibility of attributes and methods.

Getter and Setter Methods: Encapsulation involves using getter and setter methods to access and modify private attributes, ensuring controlled access.

```python
class Internship:

    def __init__(self,name,date,task):

        self.name = name   #Public
        self._date = date  #Protected
        self.__task = task #Private
```

```python
class Internship:

    def __init__(self,name,date,task):

        self.name = name   #Public
        self._date = date  #Protected
        self.__task = task #Private

    def get_name(self):

        return self._name

    def set_name(self, new_name):

        if len(new_name) > 0:

            self._name = new_name
```

5. Inheritance:

Creating a Subclass: Inheritance allows a new class (subclass) to inherit attributes and methods from an existing class (parent class).

Overriding Methods: Subclasses can provide a specific implementation for a method defined in the parent class.

Types:

Single level Inheritance
Multiple Inheritance
Multilevel Inheritance
Hierarchical Inheritance
Hybrid Inheritance

```python
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Animal speaks
dog.bark()   # Output: Dog barks
```

6. Polymorphism:

Polymorphism allows objects to be treated as instances of their parent class, promoting code flexibility. It can be achieved through method overloading and method overriding.

Method Overloading: Polymorphism allows the definition of multiple methods with the same name but different parameters.

Method Overriding: Different classes can have methods with the same name, providing specific implementations.

```python
def calculate_area(shape):
    return shape.calculate_area()

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius ** 2

class Square:
    def __init__(self, side_length):
        self.side_length = side_length

    def calculate_area(self):
        return self.side_length ** 2

circle = Circle(5)
square = Square(4)

print(calculate_area(circle))  # Output: 78.5
print(calculate_area(square))  # Output: 16
```

7. Abstraction:

Abstraction involves hiding the complex implementation details of an object and exposing only the necessary features or functionalities. It is Achieved through abstract classes and abstract methods.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```