SUPER() in Classes:

Certainly! `super()` in Python is used to access methods and properties from a parent class. It's particularly useful in situations where you're extending a class (subclassing) and want to invoke the methods or properties of the parent class.

Here's how `super()` works:

CODE:

```
class Parent:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print("Hello,", self.name)


class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Calls the constructor of the
Parent class
        self.age = age

    def greet(self):
        super().greet()  # Calls the greet() method of the
Parent class
        print("How are you?")


# Creating an instance of the Child class
child = Child("Alice", 10)
child.greet()
```

Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\37.py
Hello, Alice
How are you?
```

In this example:

`Parent` class has an `__init__()` method that initializes the `name` attribute and a `greet()` method that prints a greeting message.

`Child` class inherits from `Parent` and extends it by adding an `age` attribute. It also overrides the `greet()` method.

In the `__init__()` method of `Child`, `super().__init__(name)` is used to call the constructor of the `Parent` class, ensuring that the `name` attribute is initialized properly.

Similarly, in the `greet()` method of `Child`, `super().greet()` is used to call the `greet()` method of the `Parent` class to print the initial greeting message.

Using `super()` helps in maintaining code consistency, especially in complex inheritance hierarchies, and ensures that the behavior defined in the parent classes is retained while extending or modifying it in the subclasses.

DIAMOND SHAPE PROBLEM IN MULTIPLE INHERITANCE IN PYTHON:

The Diamond Shape Problem is a common issue that arises in programming languages that support multiple inheritance, including Python. It occurs when a class inherits from two or more classes that have a common ancestor. As a result, there can be ambiguity in method resolution or attribute lookup, leading to unexpected behavior.

Here's a simplified example to illustrate the Diamond Shape Problem:

CODE:

```
class A:
    def method(self):
        print("Method from class A")

class B(A):
    def method(self):
        print("Method from class B")

class C(A):
    def method(self):
        print("Method from class C")

class D(B, C):
```


```
    pass

# Creating an instance of class D and calling the method
d = D()
d.method()
```

OUTPUT:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\37_1.py
Method from class B
```

In this example, class `D` inherits from both classes `B` and `C`, which both inherit from class `A`. When you call the `method()` on an instance of class `D`, which implementation of `method()` should be invoked? Is it from class `B` or class `C` or class `A`?

DUNDER METHODS IN PYTHON:

Dunder methods, short for "double underscore" methods, are special methods in Python that have names surrounded by double underscores on both sides. These methods are also known as magic methods or special methods. Dunder methods allow classes to define how they interact with Python's built-in functionality, such as arithmetic operations, comparison operators, and built-in functions like `len()`, `str()`, `repr()`, etc.

Here are a few commonly used dunder methods:

1) `__init__(self, ...)`: **The constructor method, called when an object is instantiated.**
2) `__str__(self)`: **Called by the `str()` function and `print()` to return a string representation of an object.**
3) `__repr__(self)`: **Called by the `repr()` function to return a string representation of an object for debugging purposes.**
4) `__len__(self)`: **Called by the `len()` function to return the length of an object.**
5) `__getitem__(self, key)`: **Called to retrieve an item from an object using square bracket notation (`obj[key]`).**
6) `__setitem__(self, key, value)`: **Called to set an item in an object using square bracket notation (`obj[key] = value`).**
7) `__delitem__(self, key)`: **Called to delete an item from an object using the `del` statement (`del obj[key]`).**

8) `__iter__(self)`: **Called when an object is iterated over, for example, in a `for` loop.**
9) `__next__(self)`: **Called to return the next item in an iterator.**
10) `__eq__(self, other)`: **Called when testing for equality using the `==` operator.**
11) `__lt__(self, other)`, `__le__(self, other)`, `__gt__(self, other)`,
12) `__ge__(self, other)`: **Comparison methods for less than, less than or equal to, greater than, and greater than or equal to, respectively.**

These dunder methods allow Python classes to define custom behavior for various operations and integrate seamlessly with Python's syntax and built-in functions. By implementing these methods, you can make your custom objects behave like built-in types, providing a more intuitive and consistent interface for users of your code.

ABSTRACT BASE CLASS & @abstractmethod:

Abstract Base Classes (ABCs) in Python are a way of defining abstract interfaces for classes. They provide a blueprint for other classes to follow by specifying a set of methods that must be implemented by any concrete subclass. ABCs help enforce a common interface across different implementations, making code more predictable and maintainable.

In Python, the `abc` module provides support for defining abstract base classes. The `@abstractmethod` decorator is used to mark methods within an ABC that must be implemented by concrete subclasses. If a subclass fails to implement one or more abstract methods defined in the ABC, Python raises a `TypeError` at runtime.

Here's an example illustrating the usage of ABCs and `@abstractmethod`:

CODE:

```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def perimeter(self):
```


```
        pass

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

    def area(self):
        return 3.14 * self.radius * self.radius

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

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

    def perimeter(self):
        return 4 * self.side

# Creating instances of Circle and Square
circle = Circle(5)
square = Square(4)

print("Circle Area:", circle.area())
print("Circle Perimeter:", circle.perimeter())

print("Square Area:", square.area())
print("Square Perimeter:", square.perimeter())
```

OUTPUT:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\37_2.py
Circle Area: 78.5
Circle Perimeter: 31.400000000000002
Square Area: 16
Square Perimeter: 16
```

In this example:

`Shape` is an abstract base class that defines two abstract methods: `area()` and `perimeter()`.

`Circle` and `Square` are concrete subclasses of `Shape`. They implement the `area()` and `perimeter()` methods, as required by the ABC.

If you attempt to instantiate a subclass without implementing all the abstract methods, Python will raise a `TypeError`.

Abstract base classes and `@abstractmethod` provide a powerful mechanism for defining interfaces and enforcing adherence to those interfaces in Python code. They promote code reusability, maintainability, and help catch errors at compile time rather than runtime.

SETTERS & PROPERTY DECORATORS:

In Python, setters and property decorators are used to implement property accessors and mutators, allowing you to define custom behavior for getting and setting attributes of an object.

Property Decorator: The `property` decorator allows you to define a method as a property of a class. It provides a way to customize attribute access, allowing you to execute code when accessing or assigning values to an attribute.

Here's how you can use the `property` decorator:

CODE:

```
class MyClass:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        if value < 0:
            raise ValueError("x must be non-negative")
        self._x = value

obj = MyClass()
```


```
obj.x = 10  # Calls the setter method
print(obj.x)  # Calls the getter method
```

OUTPUT:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\37_3.py
10
```

In this example, `x` is a property of the `MyClass` class. The `@property` decorator marks the `x()` method as a getter, and `@x.setter` decorator marks the `x()` method as a setter. When you access `obj.x`, it calls the getter method, and when you assign a value to `obj.x`, it calls the setter method.

Setters: Setters are methods that are used to set the value of an attribute in a class. They allow you to perform validation or execute additional code when setting the value of an attribute. Here's how you can define a setter without using the property decorator:

CODE:

```
class MyClass:
    def __init__(self):
        self._x = 0

    def set_x(self, value):
        if value < 0:
            raise ValueError("x must be non-negative")
        self._x = value

    def get_x(self):
        return self._x

obj = MyClass()
obj.set_x(10)  # Calls the setter method
print(obj.get_x())  # Calls the getter method
```

OUTPUT:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\37_4.py
10
```

In this example, `set_x()` is a setter method that sets the value of `_x`, and `get_x()` is a getter method that returns the value of `_x`.

Using property decorators and setters allows you to define properties with custom behavior, such as validation or computed attributes, making your classes more flexible and easier to work with.

OBJECT INTROSPECTION:

Object introspection in Python refers to the ability to examine the attributes and methods of an object at runtime. It allows you to programmatically inspect the structure, properties, and behavior of objects in your code. Python provides several built-in functions and techniques for object introspection:

`dir()` Function: The `dir()` function returns a list of attributes and methods of an object. It provides a comprehensive view of the object's structure.

```
class MyClass:
    def __init__(self):
        self.x = 10
        self.y = 20

obj = MyClass()
print(dir(obj))
```

type() Function: The type() function returns the type of an object. It's useful for determining the class of an object.

```
obj = 10
print(type(obj))  # Output: <class 'int'>
```

isinstance() Function: The isinstance() function checks if an object is an instance of a particular class or type.

```
obj = 10
print(isinstance(obj, int))  # Output: True
```

getattr() Function: The getattr() function retrieves the value of an attribute of an object by name.

```
class MyClass:
    def __init__(self):
        self.x = 10

obj = MyClass()
print(getattr(obj, 'x'))  # Output: 10
```

hasattr() Function: The hasattr() function checks if an object has a particular attribute.

```
class MyClass:
    def __init__(self):
        self.x = 10

obj = MyClass()
print(hasattr(obj, 'x'))  # Output: True
```

setattr() Function: The setattr() function sets the value of an attribute of an object by name.

```
class MyClass:
    def __init__(self):
        self.x = 10

obj = MyClass()
setattr(obj, 'y', 20)
print(obj.y)  # Output: 20
```

OUTPUT: