Advanced Oops

Reading Material

PW SKILLS

Topics Covered in This Chapter:

1. Metaclasses
- Understanding Metaclasses
- Creating Custom Metaclasses
- Use Cases for Metaclasses
- The __new__ Method,__Init__Method
- Metaclass Inheritance

2. Multiple Inheritance and MRO (Method Resolution Order)
- Basics of Multiple Inheritance
- Method Resolution Order (MRO)
- The C3 Linearization Algorithm
- super() Function in Multiple Inheritance
- Diamond Problem and Its Resolution

3. Decorators and Class Decorators
- Function Decorators
- Class Decorators
- Decorators with Arguments
- Stacking Decorators
- Wrapping Decorators
- Built-in Decorators (@property, @classmethod, @staticmethod)

4. Interview Questions (Easy/Medium/Hard)
- Theoretical Questions-20 Questions
- Code-based Questions-20 Questions

5. Multiple Choice Questions (Minimum 25-30 MCQ)

Font Size: 11
Font Style: Arial
Code Font Style: Roboto Mono : Example ———————> git clone <repository URL>

1. Metaclasses

Understanding Metaclasses:

Imagine you're a wizard who can create magical creatures. Now, what if you could create a spell that determines how all future magical creatures are made? That's kind of what a metaclass does in Python!

A metaclass is a class that defines how other classes are created. It's like a class factory - it makes classes!

In Python, everything is an object, even classes. And just like objects are instances of classes, classes themselves are instances of metaclasses. The default metaclass in Python is called `type`.

Creating Custom Metaclasses:

Let's create a simple metaclass:

```python
class MyMetaclass(type):
    def __new__(cls, name, bases, attrs):
        # Add a new method to the class
        attrs['greet'] = lambda self: f"Hello from {name}!"
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MyMetaclass):
    pass

obj = MyClass()
print(obj.greet())  # Output: Hello from MyClass!
```

In this example, `MyMetaclass` adds a `greet` method to any class that uses it as a metaclass.

Use Cases for Metaclasses:

1. Adding methods or attributes to classes automatically
2. Implementing singleton pattern
3. Registering classes (like in some database ORMs)
4. Modifying class creation process

The __new__ Method vs __init__ Method:

- `__new__` is called to create the class object. It's where you can modify the class before it's created.
- `__init__` is called to initialize the class object after it's been created.

```python
class MyMetaclass(type):
    def __new__(cls, name, bases, attrs):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, attrs)

    def __init__(cls, name, bases, attrs):
        print(f"Initializing class {name}")
        super().__init__(name, bases, attrs)

class MyClass(metaclass=MyMetaclass):
    pass
# Output:
# Creating class MyClass
# Initializing class MyClass
```

Metaclass Inheritance:

Metaclasses can inherit from each other, just like regular classes:

```python
class MetaA(type):
    def show(cls):
        print("Method from MetaA")

class MetaB(MetaA):
    def display(cls):
        print("Method from MetaB")

class MyClass(metaclass=MetaB):
 pass

MyClass.show()    # Output: Method from MetaA
MyClass.display() # Output: Method from MetaB
```

Metaclasses are a powerful feature, but they're rarely needed in everyday programming. They're mostly used in advanced frameworks and libraries.

2. Multiple Inheritance and MRO (Method Resolution Order)

Basics of Multiple Inheritance:

Multiple inheritance is when a class inherits from more than one parent class. It's like having superpowers from both your mom and dad!

```python
class Flying:
    def fly(self):
        return "I can fly!"

class Swimming:
    def swim(self):
        return "I can swim!"

class Duck(Flying, Swimming):
    pass

donald = Duck()
print(donald.fly())  # Output: I can fly!
print(donald.swim()) # Output: I can swim!
```

Method Resolution Order (MRO):

When a class inherits from multiple parents, Python needs to know in which order to look for methods. This order is called the Method Resolution Order (MRO).

```python
# Defining Class A as the base class with a method show()
class A:
    def show(self):
        print("Method from Class A")

# Class B inherits from Class A and overrides the show()
method
class B(A):
    def show(self):
        print("Method from Class B")

# Class C also inherits from Class A and overrides the
show() method
class C(A):
    def show(self):
        print("Method from Class C")

# Class D inherits from both Class B and Class C
class D(B, C):
    # Class D does not define its own show() method, so it
will use the show() method from its parents according to
MRO
    pass

# Create an instance of Class D
d = D()

# Calls the show() method, Python follows MRO to determine
which show() method to execute
d.show()# Method from Class B

# Printing the Method Resolution Order (MRO) of class D
# This prints the order in which Python will search for
methods and attributes in Class D
print("MRO of class D:", [x.__name__ for x in D.__mro__])
#output:['D', 'B', 'C', 'A', 'object']
```

Explanation:
According to the MRO, Python first checks if Class D has a show() method. If not, it checks Class B (since B is listed before C in the class definition of D). Python finds and executes the show() method from Class B.
The C3 Linearization Algorithm:

The C3 Linearization Algorithm:

Python uses the C3 linearization algorithm to determine the MRO. It's a bit complex, but the main idea is to keep the inheritance graph consistent and avoid conflicts.

Note:
The C3 algorithm is a deterministic MRO algorithm that ensures a single consistent order in any inheritance hierarchy.

It enforces two key rules:

1. **Children precede their parents:** If class A inherits from class B, A must appear before B in the MRO.
2. **Preserve the order of appearance in the base class list:** If class C is defined as class C(B, A):, then B should appear before A in the MRO of C.

super() Function in Multiple Inheritance:

The `super()` function is used to call methods from parent classes. In multiple inheritance, it follows the MRO.

```python
class A:
    def greet(self):
        return "Hello from Dada ji"

class B(A):
    def greet(self):
        return super().greet() + " and Papa"

class C(A):
    def greet(self):
        return super().greet() + " and Chacha"

class D(B, C):
    def greet(self):
        return super().greet() + " and Gullak"
d = D()
print(d.greet())
# Output: Hello from Dada ji and Chacha and Papa and Gullak
```

Diamond Problem and Its Resolution:

The diamond problem occurs when a class inherits from two classes that have a common ancestor.

```python
class A:
    def greet(self):
        return "Hello from A"

class B(A):
    def greet(self):
        return "Hello from B"
```

```
class C(A):
    def greet(self):
        return "Hello from C"

class D(B, C):
    pass

d = D()
print(d.greet())  # Output: Hello from B
```

Python's MRO resolves this by following the order of inheritance and the C3 linearization algorithm.

3. Decorators and Class Decorators

Function Decorators:

A decorator is like a wrapper for your function. It allows you to add new functionality to an existing function without modifying its structure.

```
```python
def uppercase_decorator(func):
    def wrapper():
        result = func()
        return result.upper()
    return wrapper

@uppercase_decorator
def greet():
    return "hello, world!"

print(greet())  # Output: HELLO, WORLD!
```
```

Class Decorators:

Class decorators are similar to function decorators, but they work on classes instead of functions.

```
```python
def add_greeting(cls):
    cls.greet = lambda self: f"Hello from {cls.__name__}!"
    return cls

@add_greeting
class MyClass:
    pass

obj = MyClass()
print(obj.greet())  # Output: Hello from MyClass!
```
```

Decorators with Arguments:

You can also create decorators that accept arguments.

```python
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def say_hello():
    print("Hello!")
say_hello()
# Output:
# Hello!
# Hello!
# Hello!
```

Stacking Decorators:

You can apply multiple decorators to a single function. They are applied from bottom to top.

```python
def bold(func):
    def wrapper():
        return "<b>" + func() + "</b>"
    return wrapper

def italic(func):
    def wrapper():
        return "<i>" + func() + "</i>"
    return wrapper

@bold
@italic
def greet():
    return "Hello, world!"

print(greet())  # Output: <b><i>Hello, world!</i></b>
```

Wrapping Decorators:

To preserve the metadata of the original function, use the `@wraps` decorator from the `functools` module.

```python
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        """This is the wrapper function"""
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def greet(name):
    """This function greets someone"""
    return f"Hello, {name}!"

print(greet.__name__)  # Output: greet
print(greet.__doc__)   # Output: This function greets
someone
```

Built-in Decorators:

Python has some built-in decorators:

1. `@property`: Allows you to use a method like an attribute.

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

    @property
    def area(self):
        return 3.14 * self._radius ** 2

c = Circle(5)
print(c.area)  # Output: 78.5
```

2. `@classmethod`: Defines a method that works with the class, rather than instances.

```python
class Person:
    count = 0

    def __init__(self, name):
        self.name = name
        Person.count += 1

    @classmethod
    def number_of_people(cls):
        return cls.count

Person("Alice")
Person("Bob")
print(Person.number_of_people())  # Output: 2
```

3. `@staticmethod`: Defines a method that doesn't need access to the class or instance.

```python
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

print(MathOperations.add(5, 3))  # Output: 8
```

These advanced OOP concepts provide powerful tools for creating flexible and efficient code structures. While they may seem complex at first, with practice, they become valuable techniques in a programmer's toolkit.

Code File: Colab