Advanced Oops

Interview Questions

(Practice Project)

PW SKILLS

1. What is a metaclass in Python?

Answer:
In Python, a metaclass is a class that defines the behavior of other classes. It's often described as "a class of a class" because it's used to create and customize classes themselves. When you define a class, Python uses a metaclass to create that class object. By default, Python uses the `type` metaclass, but you can create custom metaclasses to modify class creation behavior.

Metaclasses allow you to:
- Modify class creation
- Add or modify attributes and methods
- Implement class decorators
- Enforce coding standards or patterns
- Create abstract base classes

Here's a simple example of a metaclass:

```
```python
class MyMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['custom_attribute'] = 'Added by metaclass'
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MyMetaclass):
    pass

print(MyClass.custom_attribute)  # Output: Added by
metaclass
```
```

In this example, `MyMetaclass` adds a custom attribute to any class that uses it as its metaclass.

2. How do you define a class using a custom metaclass?

Answer:
To define a class using a custom metaclass, you use the `metaclass` keyword argument in the class definition.

Here's the general syntax:

```
```python
class MyMetaclass(type):
    def __new__(cls, name, bases, attrs):
        # Custom metaclass behavior
        return super().__new__(cls, name, bases, attrs)
class MyClass(metaclass=MyMetaclass):
    # Class definition
    pass
```
```

You can also use the metaclass for all classes in a module by setting the `__metaclass__` attribute at the module level (in Python 2) or by using the `metaclass` parameter in the base class for Python 3.

Here's a more practical example:

```python
class LoggingMetaclass(type):
    def __new__(cls, name, bases, attrs):
        for attr_name, attr_value in attrs.items():
            if callable(attr_value):
                attrs[attr_name] =
cls.log_method(attr_value)
        return super().__new__(cls, name, bases, attrs)

    @staticmethod
    def log_method(method):
        def wrapper(*args, **kwargs):
            print(f"Calling method: {method.__name__}")
            return method(*args, **kwargs)
        return wrapper

class MyClass(metaclass=LoggingMetaclass):
    def some_method(self):
        print("Doing something")

obj = MyClass()
obj.some_method()
# Output:
# Calling method: some_method
# Doing something
```

In this example, the `LoggingMetaclass` adds logging functionality to all methods of classes that use it.

3. What is multiple inheritance?

Answer:
Multiple inheritance is a feature in object-oriented programming where a class can inherit attributes and methods from more than one parent class. This allows a derived class to combine and reuse code from multiple base classes.

In Python, multiple inheritance is implemented by listing all parent classes in the class definition, separated by commas. Here's a basic example:

```python
class A:
    def method_a(self):
        print("Method A")

class B:
    def method_b(self):
        print("Method B")

class C(A, B):
    def method_c(self):
        print("Method C")

obj = C()
obj.method_a()  # Output: Method A
obj.method_b()  # Output: Method B
obj.method_c()  # Output: Method C
```

In this example, class `C` inherits from both `A` and `B`, so it has access to methods from both parent classes.

Multiple inheritance can be powerful, but it also introduces complexity, especially when dealing with method resolution order (MRO) and the potential for naming conflicts between parent classes.

4. What's the difference between `@classmethod` and `@staticmethod`?

Answer:
Both `@classmethod` and `@staticmethod` are decorators used to define methods that don't require access to instance-specific data, but they have some key differences:

`@classmethod`:
- Receives the class as the implicit first argument (conventionally named `cls`)
- Can access and modify class state
- Can be called on both the class and its instances
- Often used for alternative constructors or methods that operate on the class itself

`@staticmethod`:
- Doesn't receive any implicit first argument
- Cannot access or modify class or instance state (unless explicitly passed)
- Can be called on both the class and its instances
- Behaves like a plain function that's defined inside the class for namespace purposes

Here's an example illustrating the differences:

```python
class MyClass:
    class_attribute = 0

    def __init__(self, value):
        self.instance_attribute = value

    @classmethod
    def class_method(cls):
        cls.class_attribute += 1
        return cls.class_attribute

    @staticmethod
    def static_method(x, y):
        return x + y

    def instance_method(self):
        return self.instance_attribute
# Usage
print(MyClass.class_method())  # Output: 1
print(MyClass.static_method(3, 4))  # Output: 7

obj = MyClass(5)
print(obj.class_method())  # Output: 2
print(obj.static_method(3, 4))  # Output: 7
print(obj.instance_method())  # Output: 5
```

In this example, `class_method` can modify the class state, `static_method` behaves like a regular function, and `instance_method` requires an instance to be called.

5. How do you stack multiple decorators on a single function?

Answer:
To stack multiple decorators on a single function, you simply apply them one after another, with the topmost decorator being applied last. The order of decorators matters, as each decorator modifies the function returned by the decorator below it.

Here's an example of stacking multiple decorators:

```
```python
def decorator1(func):
    def wrapper(*args, **kwargs):
        print("Decorator 1 - Before")
        result = func(*args, **kwargs)
        print("Decorator 1 - After")
        return result
    return wrapper

def decorator2(func):
    def wrapper(*args, **kwargs):
        print("Decorator 2 - Before")
        result = func(*args, **kwargs)
        print("Decorator 2 - After")
        return result
    return wrapper

@decorator1
@decorator2
def my_function():
    print("Original function")

my_function()
```


Output:
```
Decorator 1 - Before
Decorator 2 - Before
Original function
Decorator 2 - After
Decorator 1 - After
```
```

In this example, `decorator2` is applied first, then `decorator1`. The execution order is:
1. `decorator1` starts
2. `decorator2` starts
3. Original function executes
4. `decorator2` finishes
5. `decorator1` finishes

You can stack as many decorators as needed, and they will be applied in the order they are listed, from bottom to top.

6. Explain the difference between `__new__` and `__init__` methods in a metaclass.

Answer:

In a metaclass, `__new__` and `__init__` serve different purposes and are called at different times during class creation:

`__new__`:

- Is called to create and return the new class object
- Receives the metaclass as its first argument, followed by the name of the class being created, its base classes, and a dictionary of attributes
- Is responsible for creating and returning the actual class object
- Can modify the class before it's created
- `__init__`:
- Is called to initialize the newly created class object
- Receives the already created class object as its first argument, followed by the same arguments as `__new__`
- Can modify the class after it's been created, but cannot replace the class object itself

Here's an example illustrating the difference:

```
```python
class MyMetaclass(type):
    def __new__(cls, name, bases, attrs):
        print(f"Creating class {name}")
        new_attrs = {key.upper(): value for key, value in
attrs.items()}
       return super().__new__(cls, name, bases, new_attrs)

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

class MyClass(metaclass=MyMetaclass):
    x = 1
    y = 2

print(MyClass.X)  # Output: 1
print(MyClass.initialized)  # Output: True
```
```

In this example, `__new__` modifies the class attributes (converting keys to uppercase) before the class is created, while `__init__` adds a new attribute after the class has been created.

7. How does the C3 linearization algorithm work in Python's MRO?

Answer:

The C3 linearization algorithm is used by Python to determine the Method Resolution Order (MRO) in cases of multiple inheritance. It ensures a consistent and predictable order for method lookup. The algorithm works as follows:

1. Start with the class for which you're computing the MRO.
2. For each parent class (from left to right in the inheritance list):
   a. Add the parent class to the MRO if it's not already there and all its parents are already in the MRO.
   b. If it can't be added, move to the next parent.
3. Repeat step 2 until all parent classes have been processed.
4. If any classes remain, raise an error (this indicates an impossible inheritance structure).

The C3 algorithm ensures that:

- A subclass appears before its parents
- The order of parents in the inheritance list is preserved
- The algorithm is monotonic (a class's MRO is an extension of its parents' MROs)

Here's an example:

```
```python
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass

print(D.mro())
# Output: [<class '__main__.D'>, <class '__main__.B'>,
<class '__main__.C'>, <class '__main__.A'>, <class
'object'>]
```
```

In this case, the C3 algorithm ensures that `B` comes before `C` in `D`'s MRO, respecting the order in which they were inherited.

8. What is the diamond problem in multiple inheritance, and how does Python resolve it?

Answer:
The diamond problem occurs in multiple inheritance when a class inherits from two classes that have a common ancestor. This creates an ambiguity about which implementation of a method to use if it's defined in multiple parent classes.

Consider this structure:
```
```

```
  A
 / \
B   C
 \ /
  D
```

If `B` and `C` both override a method from `A`, and `D` inherits from both `B` and `C`, which version of the method should `D` use?

Python resolves this using the C3 linearization algorithm to determine the Method Resolution Order (MRO). The MRO provides a consistent order for method lookup, avoiding ambiguity.

Here's an example:

```python
class A:
    def method(self):
        print("A's method")

class B(A):
    def method(self):
        print("B's method")

class C(A):
    def method(self):
        print("C's method")

class D(B, C):
    pass

d = D()
d.method()  # Output: B's method
print(D.mro())
# Output: [<class '__main__.D'>, <class '__main__.B'>,
<class '__main__.C'>, <class '__main__.A'>, <class
'object'>]
```

In this case, `B`'s method is called because `B` comes before `C` in the MRO. The `super()` function can be used to call methods from parent classes in a way that respects the MRO.

9. Write a metaclass that adds a `log` method to all classes using it.

Answer:

Here's an implementation of a metaclass that adds a `log` method to all classes using it:

```python
import logging

class LoggingMetaclass(type):
    def __new__(cls, name, bases, attrs):
        # Add the log method to the class
        def log(self, message):
            logging.info(f"{self.__class__.__name__}:
{message}")

        attrs['log'] = log
        return super().__new__(cls, name, bases, attrs)

# Example usage
class MyClass(metaclass=LoggingMetaclass):
    def some_method(self):
        self.log("some_method was called")

# Set up logging
logging.basicConfig(level=logging.INFO)
# Test the class
obj = MyClass()
obj.log("This is a log message")
obj.some_method()
```

This metaclass adds a `log` method to every class that uses it. The `log` method uses Python's built-in `logging` module to log messages with the class name as a prefix.

When you run this code, you'll see output like:
```
**INFO:root:MyClass:** This is a log message
**INFO:root:MyClass:** some_method was called
```
This metaclass allows any class using it to easily log messages without having to implement the logging functionality in each class individually.

10. How can you use decorators to implement memoization?

Answer:
Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again. Decorators are an excellent way to implement memoization in Python. Here's an example:

```python
from functools import wraps

def memoize(func):
    cache = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]

    return wrapper
# Example usage
@memoize
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(100))  # This would be very slow without
memoization
```

In this example, the `memoize` decorator creates a cache dictionary. When the decorated function is called, it checks if the result for the given arguments is already in the cache. If so, it returns the cached result. If not, it calls the function, stores the result in the cache, and then returns it.

This implementation dramatically speeds up recursive functions like Fibonacci, reducing time complexity from O(2^n) to O(n).

11. Explain how to create a decorator that takes arguments.

Answer:
To create a decorator that takes arguments, you need to add an extra level of nesting. The outermost function takes the decorator arguments, the middle function is the actual decorator, and the innermost function is the wrapper around the decorated function. Here's the general structure:

```python
def decorator_with_args(decorator_arg1, decorator_arg2):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Use decorator_arg1 and decorator_arg2
            # Call func(*args, **kwargs) as needed
            pass
        return wrapper
    return decorator
```

Here's a practical example of a decorator that repeats a function a specified number of times:

```python
from functools import wraps

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

# Usage
@repeat(times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
# Output:
# Hello, Alice!
# Hello, Alice!
# Hello, Alice!
```

In this example, `repeat` is a decorator factory that takes an argument `times`. It returns the actual decorator, which then wraps the function to be decorated.

12. What's the purpose of the `functools.wraps` decorator?

Answer:
The `functools.wraps` decorator is used to preserve the metadata of the original function when it's decorated. When you create a decorator, you're essentially replacing the original function with a new one (the wrapper). This can lead to loss of important metadata such as the function's name, docstring, and argument list.

Here's an example to illustrate the problem and how `functools.wraps` solves it:

```python
from functools import wraps

def decorator_without_wraps(func):
    def wrapper(*args, **kwargs):
"""This is the wrapper function"""
        return func(*args, **kwargs)
    return wrapper
```

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

@decorator_without_wraps
def hello_world():
    """This is the hello_world function"""
    print("Hello, World!")

@decorator_with_wraps
def goodbye_world():
    """This is the goodbye_world function"""
    print("Goodbye, World!")

print(hello_world.__name__)  # Output: wrapper
print(hello_world.__doc__)   # Output: This is the wrapper
function

print(goodbye_world.__name__)  # Output: goodbye_world
print(goodbye_world.__doc__)   # Output: This is the
goodbye_world function
```

As you can see, without `@wraps`, the decorated function loses its original name and docstring. With `@wraps`, these are preserved, which is crucial for introspection, debugging, and generating accurate documentation.

13. How would you implement a singleton pattern using a metaclass?

Answer:
The singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. Here's how you can implement it using a metaclass:

```
```python
class Singleton(type):
_instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args,
**kwargs)
        return cls._instances[cls]

class MyClass(metaclass=Singleton):
    def __init__(self):
        self.value = None

 def set_value(self, value):
        self.value = value

# Usage
a = MyClass()
b = MyClass()

print(a is b)  # Output: True

a.set_value(5)
print(b.value)  # Output: 5
```
```

In this implementation, the `Singleton` metaclass overrides the `__call__` method. When a class using this metaclass is instantiated, `__call__` checks if an instance already exists. If not, it creates one; otherwise, it returns the existing instance.

This ensures that no matter how many times you try to create an instance of `MyClass`, you always get the same object.

14. Write a class decorator that counts the number of instances created.

Answer:
Here's a class decorator that counts the number of instances created for a class:

```
```python
def instance_counter(cls):
    original_init = cls.__init__
    instance_count = 0
def new_init(self, *args, **kwargs):
        nonlocal instance_count
        instance_count += 1
        self.instance_number = instance_count
        original_init(self, *args, **kwargs)

    cls.__init__ = new_init

    def get_instance_count(cls):
        return instance_count

    cls.get_instance_count =
classmethod(get_instance_count)

    return cls

# Usage
@instance_counter
class MyClass:
    def __init__(self, value):
        self.value = value
# Create instances
obj1 = MyClass(1)
obj2 = MyClass(2)
obj3 = MyClass(3)

print(obj1.instance_number)  # Output: 1
print(obj2.instance_number)  # Output: 2
print(obj3.instance_number)  # Output: 3
print(MyClass.get_instance_count())  # Output: 3
```
```

This decorator does the following:
1. It keeps track of the original `__init__` method.
2. It defines a new `__init__` method that increments the instance count and assigns an instance number to each new instance.
3. It adds a class method `get_instance_count` to retrieve the total number of instances created.

This implementation allows you to track how many instances have been created and gives each instance a unique number.

15. Explain how to use `super()` in a multiple inheritance scenario.

Answer:

`super()` is a built-in function in Python that's used to call methods from parent classes. In a multiple inheritance scenario, `super()` becomes particularly useful as it follows the Method Resolution Order (MRO) to determine which parent class's method to call.

Here's an example to illustrate its use:

```python
class A:
    def method(self):
        print("A's method")

class B(A):
    def method(self):
        print("B's method")
        super().method()

class C(A):
    def method(self):
        print("C's method")
        super().method()

class D(B, C):
    def method(self):
        print("D's method")
        super().method()

d = D()
d.method()
print(D.mro())
```

Output:

```
D's method
B's method
C's method
A's method
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

In this example:

1. `D` inherits from both `B` and `C`.
2. When `d.method()` is called, it first prints "D's method".
3. `super().method()` in `D` calls `B`'s method (next in the MRO).
4. `super().method()` in `B` calls `C`'s method (next in the MRO).
5. `super().method()` in `C` calls `A`'s method.

`super()` ensures that all methods in the inheritance chain are called in the correct order, following the MRO. This is particularly useful for cooperative multiple inheritance, where each class in the inheritance hierarchy needs to do its part and then pass control to the next class.

Using `super()` in this way helps avoid the diamond problem and ensures that each method in the inheritance chain is called exactly once, in the order specified by the MRO.