Advanced Oops

Interview Questions 2

(Practice Project)

PW SKILLS

1. Explain the concept of metaclasses in Python. Provide an example of how to use a metaclass to modify class behavior.

Answer: Metaclasses in Python:
Metaclasses are classes for classes. They define how a class behaves. The metaclass can be used to modify the class creation process.

Example:

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

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

class MyClass(metaclass=MetaLogger):
    def some_method(self):
        pass
```

This metaclass adds logging to all methods of the class.

2. Describe the difference between `__new__` and `__init__` methods in Python. When would you use `__new__` instead of `__init__`?

Answer:
`__new__` vs `__init__`:
- `__new__` is called to create a new instance of the class. It's a static method that returns the instance.
- `__init__` is called to initialize the newly created instance.

Use `__new__` when you need to control the creation of the instance itself, like in implementing singletons or modifying instance creation based on input parameters.

3. Explain the descriptor protocol in Python. Implement a descriptor that validates an attribute to ensure it's always a positive integer.

Answer:
Descriptors define how attribute access is handled. They implement `__get__`, `__set__`, or `__delete__` methods.

Example of a descriptor for positive integers:

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

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int) or value ⩽ 0:
            raise ValueError("Value must be a positive
integer")
        instance.__dict__[self.name] = value

class MyClass:
    age = PositiveInteger('age')
```

4. What are abstract base classes (ABCs) in Python? Implement an ABC for a data structure that enforces certain method implementations in its subclasses.

Answer:

Abstract Base Classes (ABCs):
ABCs define a common API for a set of subclasses. They can't be instantiated and may have abstract methods.

Example:

```python
from abc import ABC, abstractmethod

class DataStructure(ABC):
    @abstractmethod
    def add(self, item):
        pass
 @abstractmethod
    def remove(self, item):
        pass

class List(DataStructure):
    def add(self, item):
        # Implementation
        pass

    def remove(self, item):
        # Implementation
        pass
```

5. Describe the Method Resolution Order (MRO) in Python. How does it work with multiple inheritance? Provide an example that demonstrates a complex MRO scenario.

Answer:
MRO defines the order in which Python searches for methods in a class hierarchy. It uses the C3 linearization algorithm.

Example:

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

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

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

class D(B, C):
    pass

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

6. Explain the concept of mixin classes in Python. Implement a mixin that adds logging functionality to any class it's mixed into.

Answer:
Mixins are classes that provide methods to other classes without being their parent class.

Example:

```python
class LoggingMixin:
    def log(self, message):
        print(f"Log: {message}")

class MyClass(LoggingMixin):
    def some_method(self):
        self.log("Method called")
```

7. What are class decorators? Implement a class decorator that adds a `timestamp` attribute to all methods of a class, recording when they were last called.

Answer:
Class decorators modify or enhance classes. They are applied using the `@decorator` syntax above the class definition.

Example:

```python
import time

def add_timestamp(cls):
    class Wrapper(cls):
        def __getattribute__(self, name):
            attr = super().__getattribute__(name)
            if callable(attr):
                def wrapper(*args, **kwargs):
                    setattr(attr, 'last_called',
time.time())
                    return attr(*args, **kwargs)
                return wrapper
            return attr
    return Wrapper

@add_timestamp
class MyClass:
    def some_method(self):
        pass
```

8. Describe the difference between `__getattr__` and `__getattribute__` in Python. When would you use one over the other?

Answer:
__getattr__` vs `__getattribute__`:
- `__getattr__` is called when an attribute is not found through normal lookup.
- `__getattribute__` is called for every attribute access, even for existing attributes.

Use `__getattr__` for implementing fallback behavior for missing attributes. Use `__getattribute__` when you need to control all attribute access, but be careful to avoid infinite recursion.

9. Explain the concept of context managers in Python. Implement a context manager that measures and prints the execution time of the code within its context.

Answer:
Context managers define actions to be taken when entering and exiting a context (using the `with` statement).

Example:

```python
import time

class TimerContext:
    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        end = time.time()
        print(f"Execution time: {end - self.start}
seconds")

with TimerContext():
    # Code to be timed
    pass
```

10. What is the difference between shallow copy and deep copy in Python? Implement a custom class that properly handles both shallow and deep copying of its instances.

Answer:

- Shallow copy creates a new object but references the same memory addresses for nested objects.
- Deep copy creates a new object and recursively copies all nested objects.

Example:

```python
import copy

class CustomClass:
    def __init__(self, value, nested):
        self.value = value
        self.nested = nested

    def __copy__(self):
        return CustomClass(self.value, self.nested)

    def __deepcopy__(self, memo):
        return CustomClass(copy.deepcopy(self.value, memo),
                           copy.deepcopy(self.nested,
memo))
```

11. Describe the concept of method chaining in OOP. Implement a class that allows for method chaining to perform a series of operations on data.

Answer:

Method chaining allows multiple method calls in a single line, where each method returns self.

Example:

```python
class Calculator:
    def __init__(self):
        self.result = 0

    def add(self, value):
        self.result += value
    return self

    def subtract(self, value):
        self.result -= value
        return self

    def value(self):
        return self.result

calc = Calculator().add(5).subtract(2).add(10)
print(calc.value())  # Output: 13
```

12. Explain the concept of slots in Python classes. When and why would you use them? Provide an example demonstrating their benefits.

Answer:
`__slots__` is a class variable that limits the attributes an instance can have, saving memory and improving access speed.

Example:

```python
class SlottedClass:
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y

# This will raise an AttributeError
# slotted_instance = SlottedClass(1, 2)
# slotted_instance.z = 3
```

13. Implement a singleton pattern in Python using a metaclass. Ensure that it's thread-safe.

Answer:
A singleton ensures only one instance of a class exists.

Example:

```python
import threading
class Singleton(type):
    _instances = {}
    _lock = threading.Lock()

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

class MyClass(metaclass=Singleton):
    pass
```

14. Describe the concept of duck typing in Python. How does it relate to SOLID principles, particularly the Liskov Substitution Principle?

Answer:
Duck typing is a concept where the type or class of an object is less important than the methods it defines. "If it walks like a duck and quacks like a duck, it's a duck."

The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Duck typing aligns with LSP by focusing on behavior rather than explicit types, allowing for more flexible and substitutable code.

15. Implement a custom iterator class that generates prime numbers up to a specified limit. Ensure it's memory efficient for large limits.

Answer:
Custom Iterator for Prime Numbers:

```python
class PrimeIterator:
    def __init__(self, limit):
        self.limit = limit
        self.current = 2
  def __iter__(self):
        return self
```

```
 def __next__(self):
        while self.current < self.limit:
            if self.is_prime(self.current):
                prime = self.current
                self.current += 1
                return prime
            self.current += 1
        raise StopIteration

    @staticmethod
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                return False
        return True
# Usage
for prime in PrimeIterator(20):
    print(prime)
```