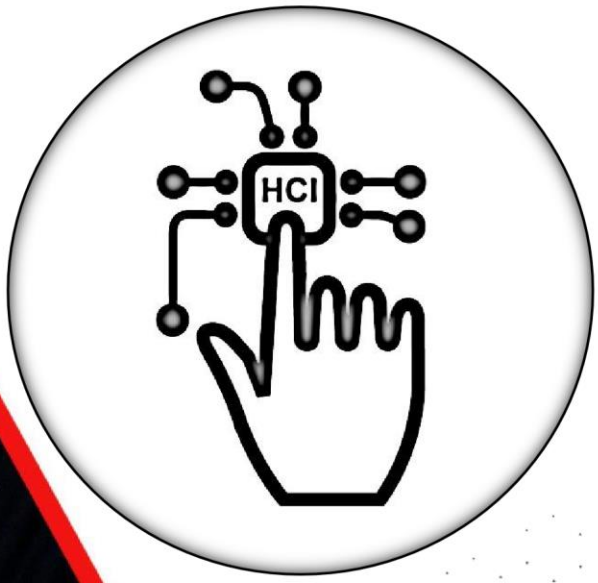
Assignment

Abbottabad University Of Science And Technology

Departement Of Computer Science



Name : M Hassan Ashraf

Roll No : 2023132

Subject : OOP

Class : CS – 2nd A

Submitted to:

Sir Jamal Abdul Ahad

Q1: Logging Metaclass: Create a metaclass that automatically logs the creation and initialization of all classes it defines.

Solution:

```
class LoggingMeta(type):
    def __new__(cls, name, bases, dct):
        # Log the creation of the class
        print(f"Creating class: {name}")
        return super().__new__(cls, name, bases, dct)

    def __init__(cls, name, bases, dct):
        # Log the initialization of the class
        print(f"Initializing class: {name}")
        super().__init__(name, bases, dct)

# Applying the metaclass to all classes
class MyClass1(metaclass=LoggingMeta):
    def __init__(self):
        print("MyClass1 initialized")

class MyClass2(metaclass=LoggingMeta):
    def __init__(self):
        print("MyClass2 initialized")

# Example usage
obj1 = MyClass1()
obj2 = MyClass2()
```

Q2: Singleton Metaclass: Implement a metaclass that ensures only one instance of a class can be created

Solution:

```
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        # If an instance doesn't exist, create one and store it in _instances
        if cls not in cls._instances:
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]

# Example usage
class MyClass(metaclass=SingletonMeta):
    def __init__(self, value):
        self.value = value

# Creating instances
obj1 = MyClass(1)
obj2 = MyClass(2)

# Checking if both instances refer to the same object
print(obj1 is obj2)  # Output: True
```

Q3: Attribute Validation Metaclass: Design a metaclass that performs custom validation on attributes declared during class creation.

Solution:

```
class AttributeValidationMeta(type):
    def __new__(cls, name, bases, dct):
        # Perform custom validation on attributes
        for attribute_name, attribute_value in dct.items():
            if isinstance(attribute_value, int) and attribute_value < 0:
                raise ValueError(f"Invalid value for attribute '{attribute_name}': {attribute_value}. Must be a non-negative integer.")

        # Create the class using the default behavior
        return super().__new__(cls, name, bases, dct)
```

```
# Example usage
class MyClass(metaclass=AttributeValidationMeta):
    positive_number = 42
    negative_number = -10  # This will raise a ValueError during class creation

# Creating an instance (not relevant to the metaclass)
obj = MyClass()
```

Q4: Metaclass for Multiple Inheritance: Create a metaclass that manages complex inheritance structures and enforces specific rules or restrictions on inheriting multiple parent classes.

Solution:

```
class MultipleInheritanceMeta(type):
    def __new__(cls, name, bases, dct):
        # Check if there is a common base class among parent classes
        common_base = None
        for base_cls in bases:
            if common_base is None:
                common_base = set(base_cls.__bases__)
            else:
                common_base &= set(base_cls.__bases__)

        if not common_base:
            raise TypeError(f"Classes with MultipleInheritanceMeta must have a common base class.")

        # Create the class using the default behavior
        return super().__new__(cls, name, bases, dct)

# Example usage
class BaseA:
    pass

class BaseB:
    pass
```

```
class CommonBase:
    pass

class MyClass(BaseA, BaseB, CommonBase, metaclass=MultipleInheritanceMeta):
    pass

# This will raise a TypeError because MyClass doesn't have a common base class among its parents
```