Class Assessments

1. Define a Simple Class

Assessment: Define a class Person with attributes name and age.

Solution:

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

# Test
person = Person("Alice", 30)
print(person.name)  # Output: Alice
print(person.age)   # Output: 30
```

2. Class with Method

Assessment: Define a class Rectangle with attributes width and height, and a method area that calculates the area of the rectangle.

Solution:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Test
rect = Rectangle(4, 5)
print(rect.area())  # Output: 20
```

Intermediate Level

3. Inheritance

Assessment: Define a class Animal with a method speak. Define a subclass Dog that overrides the speak method.

Solution:

```
class Animal:
    def speak(self):
        return "Animal speaks"
```

```python
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Test
dog = Dog()
print(dog.speak())  # Output: Woof!
```

4. Class with Class Method

Assessment: Define a class Circle with a class method from_diameter that creates a Circle instance using the diameter.

Solution:

```python
import math

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

    @classmethod
    def from_diameter(cls, diameter):
        radius = diameter / 2
        return cls(radius)

    def area(self):
        return math.pi * (self.radius ** 2)

# Test
circle = Circle.from_diameter(10)
print(circle.radius)  # Output: 5.0
print(circle.area())  # Output: 78.53981633974483
```

5. Property Decorators

Assessment: Define a class Student with attributes name and _grade. Use a property decorator to create a getter and setter for grade.

Solution:

```python
class Student:
    def __init__(self, name, grade):
        self.name = name
        self._grade = grade

    @property
    def grade(self):
```

```
        return self._grade

    @grade.setter
    def grade(self, value):
        if 0 <= value <= 100:
            self._grade = value
        else:
            raise ValueError("Grade must be between 0 and 100")

# Test
student = Student("Bob", 85)
print(student.grade)  # Output: 85
student.grade = 95
print(student.grade)  # Output: 95
```

Advanced Level

6. Static Method

Assessment: Define a class MathUtils with a static method is_even that checks if a number is even.

Solution:

```
class MathUtils:
    @staticmethod
    def is_even(number):
        return number % 2 == 0

# Test
print(MathUtils.is_even(4))  # Output: True
print(MathUtils.is_even(5))  # Output: False
```

7. Class with Magic Methods

Assessment: Define a class Vector with attributes x and y. Implement the **add** method to add two vectors.

Solution:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

```
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

# Test
v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3)  # Output: Vector(4, 6)
```

8. Encapsulation

Assessment: Define a class BankAccount with a private attribute _balance. Provide methods to deposit, withdraw, and check the balance.

Solution:

```
class BankAccount:
    def __init__(self, initial_balance=0):
        self._balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
        else:
            raise ValueError("Deposit amount must be positive")

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
        else:
            raise ValueError("Invalid withdrawal amount")

    def get_balance(self):
        return self._balance

# Test
account = BankAccount(100)
account.deposit(50)
print(account.get_balance())  # Output: 150
account.withdraw(30)
print(account.get_balance())  # Output: 120
```

9. Multiple Inheritance

Assessment: Define two classes Flyable and Swimmable, each with a method move. Define a class Duck that inherits from both and overrides the move method.

Solution:

```
class Flyable:
    def move(self):
        return "Flying"

class Swimmable:
    def move(self):
        return "Swimming"

class Duck(Flyable, Swimmable):
    def move(self):
        return "Flying and Swimming"

# Test
duck = Duck()
print(duck.move())  # Output: Flying and Swimming
```

10. Abstract Base Class

Assessment: Define an abstract base class Shape with an abstract method area. Define two subclasses Square and Circle that implement the area method.

Solution:

```
from abc import ABC, abstractmethod
import math

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

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

    def area(self):
        return self.side ** 2

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

    def area(self):
        return math.pi * (self.radius ** 2)
```

```
# Test
square = Square(4)
circle = Circle(3)
print(square.area())  # Output: 16
print(circle.area())  # Output: 28.274333882308138
```