الاسم : احمد السيد عبده على فرج | سكشن 1

Chapter 2

1. Vector3D Class with Operator Overloading

```
Python
class Vector3D:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __add__(self, other):
        return Vector3D(self.x + other.x, self.y +
other.y, self.z + other.z)

    def __sub__(self, other):
        return Vector3D(self.x - other.x, self.y -
other.y, self.z - other.z)

    def __mul__(self, other):
        # Dot product: x1*x2 + y1*y2 + z1*z2
        return self.x * other.x + self.y * other.y +
self.z * other.z

    def __repr__(self):
        return f"Vector3D({self.x}, {self.y}, {self.z})"

# Test
v1 = Vector3D(1, 2, 3)
v2 = Vector3D(4, 5, 6)
print(f"Add: {v1 + v2}")
print(f"Sub: {v1 - v2}")
print(f"Dot: {v1 * v2}")
```

2. Positive Number Descriptor

```
Python
class Positive:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        if instance is None: return self
        return instance.__dict__.get(self.name)

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError(f"{self.name} must be non-
negative")
        instance.__dict__[self.name] = value

class BankAccount:
    balance = Positive("balance")

    def __init__(self, amount):
        self.balance = amount

# Test
acc = BankAccount(100)
print(acc.balance)
# acc.balance = -50
```

3. Point Class with __slots__

```
Python
class Point:
    __slots__ = ('x', 'y')

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

p = Point(10, 20)
```

Explanation: When __slots__ is defined, Python does not create a __dict__ for instances. Instead, it allocates a fixed amount of memory for only the specified attributes (x and y). Attempting to add a new attribute (z) fails because there is no dynamic dictionary to store it, saving memory and speeding up access.

4. Disassembling a Simple Function

```
Python
import dis

def calculate_sum(a, b):
    return a + b

# dis.dis(calculate_sum)
```

Analysis:

- **LOAD_FAST**: Pushes a local variable (like a or b) onto the stack. It's "fast" because it uses an array index rather than a dictionary lookup.

- **BINARY_ADD**: Pops the top two items from the stack, adds them together, and pushes the result back onto the stack.
- **RETURN_VALUE**: Pops the top item from the stack and returns it to the caller.
- **Relation to PVM**: This confirms the Python Virtual Machine (PVM) is **stack-based**. Instructions don't operate directly on registers or variables; they push data to a stack, operate on the top elements, and push results back.