1. Práctica POO

TUIA - Programación 2

Programación Orientada a Objetos

Ejercicio 1

Considere las clases `Point` y `Rectangle` que vimos en la clase de teoría.

```
class Point:
    """ representación de un punto en un plano cartesiano 2D """

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def __str__(self) -> str:
        return '(' + str(self.x) + ', ' + str(self.y) + ')'

    def __eq__(self, other) -> bool:
        if not isinstance(other, Point):
            return NotImplemented
        return self.x == other.x and self.y == other.y

    def __add__(self, other: 'Point') -> 'Point':
        return Point(self.x + other.x, self.y + other.y)


class Rectangle:
    def __init__(self, width: float, height: float, corner: Point) -> None:
        self.width = width
        self.height = height
        self.corner = corner
```

1. Defina métodos `__str__` y `__eq__` apropiados para `Rectangle`.
2. Defina una función llamada `mover_rectángulo` que tome un rectángulo y dos parámetros `dx` y `dy`. Esta función debería cambiar de posición el rectángulo sumando `dx` a la coordenada `x` de la esquina superior izquierda y del mismo modo sumar `dy` a la coordenada `y` de la esquina superior izquierda. Haga las dos versiones, una función pura y una función modificadora.
3. Escriba código para crear algunas instancias de puntos y rectángulos y pruebe los métodos y funciones que escribió.
4. Realice un pequeño diagrama de estados mostrando el estado global del programa y de los objetos

involucrados.

Ejercicio 2

Defina, en la clase `Point`, un método `distancia()` que nos de la distancia euclídea entre dos puntos.

Nota: Recordar que la distancia euclídea entre dos puntos $A = (x_1, y_1)$ y $B = (x_2, y_2)$ se calcula como: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Ejercicio 3

Defina la clase `Automovil` que contenga (al menos) los siguientes atributos:

- `patente` (string)
- `marca` (string)
- `kilometros_recorridos` (float)
- `litros_nafta` (float)

La clase deberá proveer un constructor que permita inicializar los atributos, siendo obligatorios `patente` y `marca`. `kilometros_recorridos` y `litros_nafta`, se pueden especificar o no. Si no se especifican, se inicializarán por defecto en 0.

La clase tendrá además un método llamado `avanzar()` que recibirá como argumento el número de kilómetros a conducir y sumará los kilómetros recorridos al valor del atributo `kilometros_recorridos`. El método también restará al valor de `litros_nafta` la cantidad consumida (se calcula el consumo de gasolina como `8.8` litros por cada `100` kms recorridos).

La clase también contendrá otro método llamado `cargar_nafta()` que recibirá como argumento los litros introducidos que deberán sumarse a la variable `litros_nafta`.

Por último, será necesario controlar que el método avanzar nunca obtendrá un número negativo en la gasolina. En dicho caso, deberá mostrar el siguiente mensaje: `"Es necesario cargar nafta para recorrer la cantidad indicada de kilómetros"`.

Ejemplos de uso:

```
auto = Automovil("AEF-202", "Peugeot")
auto.cargar_nafta(10)
print(auto.kilometros_recorridos)  # Debería mostrar 0
print(auto.litros_nafta)  # Debería mostrar 10
auto.avanzar(50)
print(auto.kilometros_recorridos)  # Debería mostrar 50
print(auto.litros_nafta)  # Debería mostrar 5.6
auto.avanzar(100)  # Debería mostrar un mensaje de error: "nafta insuficiente"
auto.avanzar(40)
print(auto.kilometros_recorridos)  # Debería mostrar 90
print(auto.litros_nafta)  # Debería mostrar 2.08
```

Ejercicio 4

Defina una clase `Robot` que simule los movimientos de un robot y calcule la posición en la que se encuentra cada momento. El robot se moverá por un tablero infinito de coordenadas `X` e `Y`, podrá realizar los siguientes movimientos:

- Avanzar hacia adelante (A).
- Retroceder (R).

- Avanzar hacia la izquierda (I) o hacia la derecha (D).

El robot tendrá un método llamado `mueve()` que recibirá la orden como parámetro y otro método, `posicion_actual()`, que indicará su posición en las coordenadas `X` e `Y`. Al crear el robot, este se inicializará a las coordenadas `(0, 0)`.

Puedes utilizar el siguiente código para probar la clase creada:

```
mi_robot = Robot()
orden = input("Introduce la orden: ")
while orden != 'fin':
    mi_robot.mueve(orden)
    print(mi_robot.posicion_actual())
    orden = input("Introduce la orden: ")
```

Ejercicio 5

Mejora el robot que acabamos de implementar:

- Ahora `mover` puede recibir un solo movimiento, o bien, una secuencia de movimientos (por ejemplo, `"RRAARDDI"`). **Atención**: si la secuencia de movimientos contiene algún movimiento inválido, debe informarlo antes de realizar cualquier movimiento.
- Agregar un método `obtener_historico_de_movimientos` que devuelva el historial de movimientos que realizó el robot.
- Agregar un método `como_volver` que indique la secuencia de movimientos que deberíamos realizar para regresar a la coordenada `(0, 0)`.

Composición y Herencia

Ejercicio 6

Crear las clases `Materia` y `Carrera`.

Cada materia tiene: - Un código de materia. - Un nombre de la materia. - Una cantidad de créditos que aporta.

Una carrera puede pensarse como una lista de materias.

La clase debería comportarse de forma que el siguiente ejemplo tenga sentido:

```
>>> analisis2 = Materia("61.03", "Análisis 2", 8)
>>> fisica2 = Materia("62.01", "Física 2", 8)
>>> algo1 = Materia("75.40", "Algoritmos 1", 6)
>>> c = Carrera([analisis2, fisica2, algo1])
>>> print(c)
Créditos: 0 -- Promedio: N/A -- Materias aprobadas:
>>> c.aprobar("95.14", 7)
Error: La materia 75.14 no es parte del plan de estudios
>>> c.aprobar("75.40", 10)
>>> c.aprobar("62.01", 7)
>>> print(c)
Créditos: 14 -- Promedio: 8.5 -- Materias aprobadas: 75.40 Algoritmos 1 (10) 62.01 Física 2 (7)
```

Ejercicio 7

Encontrar los errores en el siguiente código y proponer soluciones:

```
class Cosa:
    def __init__(self, valor):
        self.valor = valor


class Coleccion:
    def __init__(self):
        self.coleccion = []

    def agregar_cosa(cosa: Cosa):
        coleccion.append(cosa)


cosa = Cosa()
coleccion = Coleccion()
coleccion.agregar_cosa(cosa)
```

Ejercicio 8

Considere la siguiente jerarquía de clases:

```
                          |--- Felinos
Animales --- Mamíferos ---|--- Cánidos
                          |--- Primates --- Hacker
```

Programe un conjunto de seis clases que modele esta taxonomía utilizando clases. Luego, agregue un método `speak` a cada clase imprimiendo un mensaje apropiado a cada clase (por ejemplo, una instancia de animal podría imprimir `"Soy un animal"`).

Luego, agregue un método `talk` a la clase `Animal`, que simplemente delegue el funcionamiento en `speak`. ¿Qué ocurre al llamar a `talk` en una subclase? ¿Qué ocurre si borramos el método `speak` de la clase `Hacker`?

Ejercicio 9

Complete la funcionalidad de la clase `Jugador`, implementando los siguientes métodos:

- `golpeado`: quita vida al jugador.
- `golpear`: quita vida al enemigo y lo agrega a la lista de enemigos golpeados.

```
class Entidad:
    def __init__(self, vida_inicial: int):
        self.vida = vida_inicial


class Enemigo(Entidad):
    pass


class Jugador(Entidad):
    def __init__(self, vida_inicial: int):
```

```
        super().__init__(vida_inicial)
        self.enemigos_golpeados = []

    def golpeado(self, cuanto):
        """Completar."""
        pass

    def golpear(self, enemigo, cuanto):
        """Completar."""
        pass
```