Ejercicios de Programación orientada a objetos

Programación — DAW

Ricardo Pérez López
IES Doñana

Curso 2021/2022

1. Diseñar y codificar un modelo orientado a objetos de un banco donde hay cuentas corrientes que tienen un titular y unos movimientos. Los titulares son clientes del banco. Los clientes del banco pueden ser titulares de varias cuentas al mismo tiempo. Los movimientos pertenecen a una sola cuenta. Para ello:

   a) Crear la clase `Cliente` con los atributos `__dni`, `__nombre` y `__apellidos`.

   b) Crear la clase `Movimiento` con los atributos `__concepto` y `__cantidad`. Los movimientos son inmutables.

   c) Crear la clase `Cuenta` con los atributos `__numero`, `__titular`, `__movimientos` y `__saldo`.

      i. No se puede cambiar el número de una cuenta.

      ii. Se puede añadir un movimiento a una cuenta, pero no cambiar ni eliminar movimientos.

      iii. Tampoco se puede modificar directamente el saldo, sino que se debe actualizar directamente a partir de los movimientos que se realicen en la cuenta.

   d) Crear un módulo `principal.py` que use las clases anteriores para representar un modelo dinámico de objetos donde el cliente Antonio Martínez tiene dos cuentas corrientes, cada una de ellas con tres movimientos. Imprimir por pantalla el saldo actual de cada cuenta.

   e) ¿Cómo se podría implementar la generación automática e incremental del número de cuenta cuando se crea una cuenta nueva? Codificarlo. (*Indicación*: Usar atributos estáticos.)

f) ¿Cómo se podría implementar la colección de cuentas del banco de forma que se pueda acceder de forma eficiente a una cuenta concreta a partir de su número? Codificarlo.

2. Diseñar y codificar un modelo orientado a objetos de una tienda online donde hay clientes, artículos y facturas. Para ello:

   a) Crear la clase `Cliente` con los atributos `__dni`, `__nombre` y `__apellidos`.

   b) Crear la clase `Articulo` con los atributos `__codigo`, `__denominacion` y `__precio`.

   c) Crear la clase `Factura` con los atributos `__numero`, `__cliente` y `__lineas`.

      i. Cada línea de factura debe almacenar un artículo y una cantidad.

      ii. El total de la factura no se debe almacenar, sino que se debe calcular automáticamente sumando el precio de cada artículo multiplicado por la cantidad.

      iii. Las líneas de una factura pertenecen a esa factura.

      iv. Las líneas de una factura se pueden añadir o eliminar de una factura, pero no modificar.

   d) Crear un módulo `principal.py` que use las clases anteriores para representar un modelo dinámico de objetos donde existe una factura del cliente Rosa González que ha comprado dos televisores de 399 € cada uno y una tarjeta gráfica de 239 €. Imprimir por pantalla todos los datos de la factura como si fuera una factura real, incluyendo el importe total de la misma.

3. Diseñar y codificar un modelo orientado a objetos de una biblioteca donde hay socios que pueden llevarse libros prestados durante quince días como máximo. Para ello:

   a) Crear la clase `Lector` con los atributos `__numero`, `__nombre`, `__apellidos` y `__moroso` (éste último es un *booleano* que por defecto estará a `False`).

   b) Crear la clase `Libro` con los atributos `__codigo`, `__titulo`, `__autor` y `__editorial`.

   c) Crear la clase `Prestamo` con los atributos `__lector`, `__libro`, `__fecha_prestamo` y `__fecha_devolucion`.

      i. Los préstamos se crean con `__fecha_devolucion` vacía (es decir, a `None`).

      ii. Las fechas se pueden almacenar como objetos de la clase `datetime` del módulo `datetime`.

      iii. Cuando un lector devuelve un libro, se introduce la fecha de devolución en el atributo `__fecha_devolucion` del préstamo correspondiente.

      iv. El resto de los atributos de un préstamo no se pueden modificar.

v. Si se devuelve un libro pasados más de quince días desde su préstamo, se debe marcar al lector como moroso.

vi. Se debe impedir que un libro ya prestado se pueda volver a prestar. Para ello hay dos técnicas:

A. Crear y usar un atributo *booleano* `__prestado` en la clase `Libro`, de forma que se ponga a `True` cuando se preste y a `False` cuando se devuelva.

B. Comprobar en los objetos `Prestamo` si ese libro está prestado y aún no se ha devuelto.

d) Crear un módulo `principal.py` que use las clases anteriores para representar un modelo dinámico de objetos donde haya dos lectores (Ana García y Roberto Sánchez) y tres libros («*El camino*» de Miguel Delibes, «*Cien años de soledad*» de Gabriel García Márquez y «*Rayuela*» de Julio Cortázar).

e) Prestar dos libros a un lector y el otro al otro lector.

f) Intentar prestar un libro que ya está prestado.

g) Intentar devolver un libro prestado con más de quince días de préstamo.

4. Traducir a clases y objetos todo el código que tenemos actualmente en `https://github.com/iesdonana/vampiro.git`, teniendo en cuenta que:

a) Las localidades deben ser objetos de la clase `Localidad`.

b) Es posible que no sea necesario hacer una clase `Localidades`.

c) El jugador debe ser un único objeto de la clase `Jugador` (a ésto se le llama *Singleton*).

d) Las conexiones de una localidad se deben almacenar dentro de la localidad.

e) Cada conexión puede ser un objeto de una clase `Conexion`, o puede que no merezca la pena crear una clase para eso. En tal caso, las conexiones serían parejas de elementos (que se podrían representar con cualquier estructura tipo lista, tupla, diccionario…) que contenga una dirección y una localidad de destino.

f) Los ítems (objetos que aparecen en el juego, como el crucifijo o la ristra de ajos) deben ser objetos de la clase `Item`.

g) Las colecciones deben ser objetos de la clase `Coleccion`.

h) Los ítems pueden estar en una localidad o en el inventario del jugador. Para ello, hay que usar objetos de la clase `Coleccion`.

i) Un *token* representa una palabra con significado propio y distinto de otras palabras. Cada *token* debe ser un objeto de la clase `Token`.

j) Cada grupo de palabras del mismo tipo («verbo», «nombre», etc.) debe ser un objeto de la clase `Vocabulario`. Así, debe haber un vocabulario que contenga todos los verbos, otro que contenga todos los nombres, etc.

   Cada objeto de la clase `Vocabulario` debe inicializarse con un diccionario que asocie cada *token* con una lista de lexemas que sean sinónimos.

   Por ejemplo:

```
ABRIR = Token()
ARRIBA = Token()
verbos = Vocabulario({
    ABRIR: ['ABRIR', 'ABRE'],
    ARRIBA: ['ARRIBA', 'SUBIR', 'SUBE']
})
CRUCIFIJO = Token()
AJOS = Token()
nombres = Vocabulario({
    CRUCIFIJO: ['CRUCIFIJO', 'CRUZ'],
    AJOS: ['AJOS', 'AJO', 'RISTRA']
})
```

k) La función `interpretar` debe implementarse como un método estático de una clase (que puede ser la misma clase `Vocabulario`) que se encargue de analizar sintácticamente la entrada del jugador a partir de los vocabulario de verbos y de nombres. Ese método debe devolver los *tokens* del verbo y el nombre encontrados, o un *token* especial (llamado *token nulo*) que represente que no se ha encontrado el verbo o el nombre.

5. Crear la clase `Persona` con un método `compara_edad` que compare la edad de una persona con la de otra.

   Ejemplos:

```
>>> p1 = Person('Samuel', 24)
>>> p2 = Person('Jael', 36)
>>> p3 = Person('Liliana', 24)
>>> p1.compara_edad(p2)
'Jael es más viejo que yo.'
>>> p2.compara_edad(p1)
'Samuel es más joven que yo.'
>>> p1.compara_edad(p3)
'Liliana tiene la misma edad que yo.'
```

   Fuente:

   `https://edabit.com/challenge/JFLADuABfkeoz8mqN`

6. Crear la clase `Empleado` con atributos `nombre` y `apellidos`. A partir de ellos, crear los atributos `nombre_completo` y `email` de la siguiente forma:

   - El `nombre_completo` será simplemente el nombre y los apellidos unidos con un espacio en blanco.
   - El `email` se forma con el nombre y los apellidos (todo en minúsculas) unidos con un `.` intermedio y seguido de `@company.com`.

   Ejemplos:

   ```
   >>> emp1 = Empleado('Juan', 'Pérez')
   >>> emp2 = Empleado('María',  'García')
   >>> emp3 = Empleado('Antonio', 'González')
   >>> emp1.nombre_completo
   'Juan Pérez'
   >>> emp2.email
   'maría.garcía@company.com'
   >>> emp3.nombre
   'Antonio'
   ```

   Fuente:

   `https://edabit.com/challenge/gB7nt6WzZy8TymCah`

7. Las instancias de la clase `Empleado` tienen los atributos `nombre`, `apellidos` y `salario`. Crear, además, el método estático `desde_cadena` que analiza una cadena que contiene esos tres valores separados por un guion y los asigna a sus atributos correspondientes.

   Ejemplos:

   ```
   >>> emp1 = Empleados('María', 'García', 60000)
   >>> emp2 = Empleados.desde_cadena('Juan-Pérez-55000')
   >>> emp1.nombre
   'María'
   >>> emp1.salario
   60000
   >>> emp2.nombre
   'Juan'
   >>> emp2.salario
   55000
   ```

   Fuente:

   `https://edabit.com/challenge/j2HauiSdDadkjxjsQ`

Soluciones

1. a)
```
class Cliente:
    def __init__(self, dni, nombre, apellidos):
        self.__dni = dni
        self.__nombre = nombre
        self.__apellidos = apellidos

    def dni(self):
        return self.__dni

    def nombre(self):
        return self.__nombre

    def apellidos(self):
        return self.__apellidos
```

   b)
```
class Movimiento:
    def __init__(self, concepto, cantidad):
        self.__concepto = concepto
        self.__cantidad = cantidad

    def concepto(self):
        return self.__concepto

    def cantidad(self):
        return self.__cantidad
```

   c)
```
class Cuenta:
    def __init__(self, numero, titular):
        self.__numero = numero
        self.__titular = titular
        self.__movimientos = []
        self.__saldo = 0

    def numero(self):
        return self.__numero

    def titular(self):
        return self.__titular

    def saldo(self):
        return self.__saldo
```

```
    def nuevo_movimiento(self, concepto, cantidad):
        self.__movimientos.append(Movimiento(concepto, cantidad))
        self.__saldo += cantidad
```

d)
```
antonio = Cliente('38475923M', 'Antonio', 'Martínez')
c1 = Cuenta(1, antonio)
c1.nuevo_movimiento('Ingreso', 100)
c1.nuevo_movimiento('Retirada', -30)
c1.nuevo_movimiento('Ingreso', 20)
c2 = Cuenta(2, antonio)
c2.nuevo_movimiento('Ingreso', 500)
c2.nuevo_movimiento('Ingreso', 80)
c2.nuevo_movimiento('Retirada', -40)
print(c1.saldo())
print(c2.saldo())
```

e) En la clase `Cuenta` guardamos un diccionario como una variable de clase, cuya clave sería el número de la cuenta y cuyo valor sería la cuenta correspondiente a ese número. Además, creamos un método estático que devuelva la cuenta a partir de su número:

```
class Cuenta:
    __cuentas = {}

    def __init__(self, numero, titular):
        self.__numero = numero
        self.__titular = titular
        self.__movimientos = []
        self.__saldo = 0
        Cuenta.__cuentas[numero] = self

    @staticmethod
    def buscar(numero):
        return Cuenta.__cuentas.get(numero)
```

Y luego se podría buscar una cuenta así:

```
c = Cuenta.buscar(2)
```