Boletín POO

3027 - CarritoCompra

Definir la clase **CarritoCompra** que contiene 2 variables de instancia:

1. Artículos y su cantidad
2. Artículos y su precio

(Nota: usar diccionarios para crear estos atributos de clase)

Implementar los siguientes métodos:

- **__init__(self, …)**: inicia atributos de la clase.
- **agregar_articulo(self, articulo, precio, cantidad)**: añade al carrito la cantidad de artículos especificada.
- **eliminar_articulo(self, articulo)**: borra del carrito la el artículo especificado.
- **precio_total(self)**: retorna el importe del carrito.
- **listar_articulos(self)**: retorna una lista con los artículos del carrito.
- **listar_carrito(self)**: retorna una lista donde cada elemento es una tupla (articulo,precio,cantidad)
- **costo_total_articulo(self,articulo)**: retorna el importe para un artículo del carrito.
- **obtener_cantidad(self, articulo)**: retorna la cantidad que hay en el carrito para ese artículo.
- **contar_articulos_distintos(self)**: retorna la cantidad de artículos distintos que hay en el carrito.
- **__str__(self)**: personaliza la representación del objeto de la clase.

```
## pruebas / casos de uso
carrito1 = CarritoCompra()
print("Listar carrito: ",carrito1.listar_carrito())
print("- "*20)
print("Agregamos 2 articulos")
carrito1.agregar_articulo(articulo="zumo naranja", cantidad=2,
precio=4)
carrito1.agregar_articulo(articulo="pan", cantidad=1.5,
precio=3)
carrito1.agregar_articulo(articulo="manzana", cantidad=2,
precio=5)
print("Listar carrito: ",carrito1.listar_carrito())
print("- "*20)
print("La cantidad de zumo naranja seleccionado
es:",carrito1.obtener_cantidad("zumo naranja"))
```

```
print("El costo total del
pan:",carrito1.costo_total_articulo("pan"))
print("- "*20)
print("Eliminamos el pan del carrito")
carrito1.eliminar_articulo("pan")
print("Listar carrito: ",carrito1.listar_carrito())
print("- "*20)
print("El número de artículos distintos en el
carrito:",carrito1.contar_articulos_distintos())
print("Estado del carrito:")
print(carrito1)
```

```
-----------------

istar carrito:  []
- - - - - - - - - - - - - - - - - - - -
Agregamos 2 articulos
Listar carrito:  [('zumo naranja', 2.0, 4), ('pan', 1.5, 3),
('manzana', 2.0, 5)]
- - - - - - - - - - - - - - - - - - - -
La cantidad de zumo naranja seleccionado es: 2.0
El costo total del pan: 4.5
- - - - - - - - - - - - - - - - - - - -
Eliminamos el pan del carrito
Listar carrito:  [('zumo naranja', 2.0, 4), ('manzana', 2.0, 5)]
- - - - - - - - - - - - - - - - - - - -
El número de artículos distintos en el carrito: 2
Estado del carrito:
Los artículos en el carrito son: ['zumo naranja', 'manzana']
El costo actual del carrito es: 18.0
```

3028 - GeneradorClave

Definir la clase **GeneradorClave** que tiene los siguientes atributos:

1. **longitud**: longitud de la clave
2. **caracteresEspeciales**: indica si puede contener
3. **numeros**: indica si puede contener
4. **mayusculas**: indica si puede contener
5. **minusculas**: indica si puede contener

Implementar los siguiente métodos:

- **__init__(self, …)**: inicia atributos de la clase.

- **Mayusculas(self, mayusculas)**: modifica la variable de instancia mayúsculas.
- **Minusculas(self, minusculas)**: modifica la variable de instancia minúsculas.
- **Numeros(self, numeros)**: modifica la variable de instancia numeros.
- **CaracteresEspeciales (self, caracteresEspeciales)**: modifica la variable de instancia caracteresEspeciales.
- **Modificar_longitud(self, longitud)**
- **generarClave(self)**: genera la clave teniendo en cuenta las variables de instancia.
- **valoraClave(self)**: retorna un número que es la puntuación de la clave por su robustez con valores del 1 al 5 teniendo en cuenta,
  - > 10 caracters -> +1 punto
  - Al menos una mayúscula -> +1 punto
  - Al menos una minúscula -> +1 punto
  - Al menos un número -> +1 punto
  - Al menos un car. especial -> +1 punto
- **guardaClave(self,clave,ruta)**: guarda la clave en el archivo que se indica en la ruta.
- **leeClave(self,ruta)**: retorna la clave contenida en el archivo que se indica en la ruta.

```
## pruebas / casos de uso
generador_clave = GeneradorClaves(12)
clave1 = generador_clave.generarClave()
print(clave1)
print("Robustez de la clave1 (0 es debil-5 es muy robusta): " +
str(generador_clave.valoraClave(clave1)))
print("- "*20)
generador_clave.Mayuscula(False)
generador_clave.Minusculas(False)
generador_clave.CaracteresEspeciales(True)
clave2 = generador_clave.generarClave()
print(clave2)
print("Robustez de la clave2 (0 es debil-5 es muy robusta): " +
str(generador_clave.valoraClave(clave2)))
print("- "*20)
print("Guardando la clave2 ...")
generador_clave.guardarClave(clave2)
print("Clave2 guardada ok!")
print("Leyendo la clave2 ...")
print(generador_clave.leerClave(clave2))

---------------------------------

YC)eh7[/90y3
```

```
Robustez de la clave1 (0 es debil-5 es muy robusta): 5
- - - - - - - - - - - - - - - - - - - - -
1)@4/8!8;#10
Robustez de la clave2 (0 es debil-5 es muy robusta): 3
- - - - - - - - - - - - - - - - - - - - -
Guardando la clave2 ...
Clave2 guardada ok!
Leyendo la clave2 ...
1)@4/8!8;#10
```

3029 - MiDiccionario

Definir la clase **MiDiccionario** que debe contener el constructor y los siguientes métodos:

- **__init__(self)**: inicia una variable de instancia con un diccionario vacío.
- **agregar_elemento(self, clave, valor)**
- **borrar_elemento(self, clave, valor)**
- **__iter__(self)**: retorna un iterador del objeto de la clase.
- **__getitem__(self,clave)**: retorna el elemento de la clave.
- **__setitem__(self,clave,valor)**: modifica el elemento de la clave.
- **__len__(self)**: retorna el número de elementos del objeto de la clase.
- **__str__(self)**: personaliza la representación del objeto de la clase.
- **listar_claves(self)**: retorna una lista con todas las claves
- **listar_elementos(self)**: retorna una lista con todas los elementos
- **borrar_diccionario(self)**: deja el diccionario vacío.
- **contiene_clave(self, clave)**: retorna true si la clave existe en el objeto de la clase.

```
## pruebas / casos de uso
## creación de una instancia
dicc_1 = MiDiccionario()
## llamada de métodos
dicc_1.agregar("fruta", "manzana")
dicc_1.agregar("vegetal", "zanahoria")
dicc_1.agregar("carne", "pavo")
print(dicc_1)
print("- "*20)
## creación de un iterador
iter_dicc1 = iter(dicc_1)
```

```
print(f"primera iteracion:", next(iter_dicc1))
print(f"segunda iteracion:", next(iter_dicc1))
print(f"- "*20)
print(f"El numero de elementos del diccionario es: ",
len(dicc_1))
print(f"Las claves del diccionario es: ",
dicc_1.listar_claves())
print(f"Los valores del diccionario son: ",
dicc_1.listar_valores())
print(f"fruta esta el diccionario:
{dicc_1.contiene_clave("fruta")}")
print(f"La lista de elementos del diccionario es: ",
dicc_1.listar_elementos())
print(f"- "*20)
dicc_1.borrar_diccionario()
print("Despues de limpiar el diccionario", dicc_1)
```

3030 - Bombilla

En una casa hay muchas bombillas, que se apagan y se encienden. Para hacer esto, crear una clase **Bombilla** con una variable privada que indica si está apagada o encendida. Un método me dirá el estado de la bombilla. Existe un interruptor general, de forma que si este se apaga, todas las bombillas quedan apagadas, y si este se activa, las bombillas están como antes.

3031 - CambioMonedas

Crear una clase **CambioMonedas** cuyos atributos son **importeCompra** y **dineroCliente**. Crear un método **mostrarCambio()** que muestre la cantidad de billetes y monedas que tendrá el cambio. La prueba será la siguiente

```
c1 = CambioMonedas(215.35, 250)
c1.mostrar_cambio()
c2 = CambioMonedas(100, 80)
c2.mostrar_cambio()
c3 = CambioMonedas(75.75, 90)
c3.mostrar_cambio()
```

y mostrará por consola:

```
Cambio resultante de 34.65:
Billetes 20€: 1
```

```
Billetes 10€: 1
Monedas 2€: 2
Monedas 50 cent: 1
Monedas 10 cent: 1
Monedas 5 cent: 1
* * * * * * * * * * *
Error!! Faltan 20
Cambio resultante de 0.00:
* * * * * * * * * * *
Cambio resultante de 14.25:
Billetes 10€: 1
Monedas 2€: 2
Monedas 20 cent: 1
Monedas 5 cent: 1
* * * * * * * * * * *
```

3032 - Calendario

Definir la clase **Calendario** con los atributos **dia**, **mes** y **anio**, y los métodos **__init__**, **incrementarDia()**, **mostrar()**, e **iguales()** con los parámetros que se indican a continuación:

```
class Calendario:
    def __init__(self, dia, mes, anio):
    def incrementarDia(self): #incrementa 1 día
    def incrementarMes(self): #incrementa 1 mes
    def incrementarAnio(self, n): #incrementa n años
    def mostrar(self): #muestra la fecha
    def iguales(self, otraFecha): #true si las fechas son
iguales
```

Prueba:

```
c1 = Calendario(28, 12, 2024)
c1.incrementarDia()
c1.incrementarDia()
c1.incrementarDia()
c1.mostrar()
c2 = Calendario(4, 1, 2025)
c2.mostrar()
print("Fechas iguales:", c1.iguales(c2))
c1.incrementarDia()
c1.mostrar()
c2.mostrar()
print("Fechas iguales:", c1.iguales(c2))
```

Boletín POO

3033 - Colores

En el momento de decorar una casa, se plantea el problema de elegir la paleta de colores que vamos a utilizar en nuestra decoración. Existe una solución, algo atrevida, que consiste en utilizar colores al azar.

Define la clase **Colores** con un atributo **colores** que es una cadena como la siguiente: "AZUL:ROJO:AMARILLO:VERDE:BLANCO:NEGRO", Aunque a esta cadena podemos añadirle colores sumándole nuevos colores.

La clase tendrá los siguientes métodos:
tabla_colores(self, n): devuelve una tabla con los n colores que necesitemos elegidos al azar sin repeticiones.
sumar_color(self, name): añade nuevos colores a los existentes.

```
# Prueba
tc = Colores()
tc.suma_color("violeta")
tc.suma_color("magenta")
tc.suma_color("naranja")
print(tc.tabla_colores(3))
```

3034 - SintonizadorFM

Define una clase **SintonizadorFM** que permite controlar un sintonizador de emisoras FM; se desea dotar al controlador de una interfaz que permite subir (up) o bajar (down) la frecuencia (en saltos de 0,5 Mhz) y mostrar la frecuencia sintonizada en un momento dado (display).

Suponemos que el rango de frecuencias va de 80Mhz a 108Mhz.
Al inicio el controlador sintoniza la frecuencia que se le indique en el contructor * (80 Mhz por defecto). Si bajando el sintonizador llego al límite, paso al extremo contrario. Probar su funcionamiento.

```
#Prueba
azul = SintonizadorFM(107)
azul.up()
azul.up()
azul.up()
azul.up()
azul.display()
```

```
verde = SintonizadorFM(200)
verde.display()
```

3035 - Trenes

Nos han encargado definir los paquetes y clases necesarias (solo implementar los atributos y los constructores) para gestionar una empresa ferroviaria, en la que se distinguen dos grandes grupos: el personal y la maquinaria. En el primero se ubican todos los empleados de la empresa, que se clasifican en 3 grupos: **maquinistas**, **mecánicos** y **jefes de estación**. De cada uno de ellos hay que guardar lo siguiente:

1. **Maquinistas**: nombre, DNI, sueldo y rango.
2. **Mecánico**: Nombre, tfno y especialidad
3. **Jefes de estación**: Nombre, DNI y fecha antigüedad.

En la parte de maquinaria hay **trenes**, **locomotoras** y **vagones**, de cada uno de ellos hay que considerar:

1. **Vagones**: numero identificador, carga máxima (en kilos), carga actual y tipo de mercancía
2. **Locomotoras**: matricula, potencia del motor, antigüedad (año de fabricación), además cada locomotora tiene asignado un **mecánico** que la mantiene.
3. **Trenes**: está formado por una **locomotora** y un máximo de 5 **vagones**. Cada tren tiene asignado un **maquinista** responsable.

Prueba

```
mecanico1 = Mecanico("Pepe", "658878787", "MOTOR")
maquinista1 = Maquinista("Manuel", "45567678F", 3400,
"Secundario")
jefeEstacion1 = JefeEstacion("Sergio", "34345345f",
"25/12/2005")
locomotora1 = Locomotora("f456", 240, 2002, mecanico1)
tren1 = Tren(locomotora1, maquinista1)
tren1.enganchaVagon(567, 450, "Plátanos")
tren1.enganchaVagon(345, 345, "Plátanos")
tren1.enganchaVagon(567, 200, "Tomates")
tren1.enganchaVagon(567, 100, "Sandias")
tren1.enganchaVagon(567, 156, "Peras")
tren1.enganchaVagon(567, 345, "Plátanos")
print(mecanico1)
print(maquinista1)
```

```
print(jefeEstacion1)
print(tren1)
```

Resultado por consola

```
Error !! el tren no admite más vagones
Mecánico: Pepe MOTOR
Maquinista: Manuel 45567678F 3400.0
Jefe Estación: Sergio 34345345f 2005-12-25 00:00:00
Componentes del tren:
      Locomotora: f456
      Maquinista: Manuel
            Vagón 1 Mercancía: Plátanos
            Vagón 2 Mercancía: Plátanos
            Vagón 3 Mercancía: Tomates
            Vagón 4 Mercancía: Sandías
            Vagón 5 Mercancía: Peras
```

3036 - Cola

Es posible usar una lista como una **cola**, donde el primer elemento añadido es el primer elemento retirado («primero en entrar, primero en salir»); sin embargo, las listas no son eficientes para este propósito. Agregar y sacar del final de la lista es rápido, pero insertar o sacar del comienzo de una lista es lento (porque todos los otros elementos tienen que ser desplazados en uno). Para implementar una cola, puedes utilizar **collections.deque** el cual fue diseñado para añadir y quitar de ambas puntas de forma rápida.

Implementa una cola con los métodos insertar(), extraer(), numeroElementos() y mostrar()

prueba

```
c1 = Cola()
c1.insertar(25)
c1.insertar(212)
c1.insertar(13)
c1.insertar(2)
c1.insertar(22)
c1.insertar(11)
c1.insertar(4)
c1.insertar(23)
c1.mostrar()
print("Número de elementos: ",c1.numeroElementos())
print("Extrae:", c1.extraer())
print("Extrae:", c1.extraer())
```

```
print("Extrae:", c1.extraer())
c1.mostrar()
```

Salida por consola:

```
[25, 212, 13, 2, 22, 11, 4, 23]
Número de elementos: 8
Extrae: 25
Extrae: 212
Extrae: 13
[2, 22, 11, 4, 23]
```

3037 - CuentaCorriente

apartado a)

Diseña la clase **CuentaCorriente**, que guarde los datos: DNI, nombre y saldo. Las operaciones típicas con una cuenta corriente son:

- Crea una cuenta: es necesario el DNI y el nombre, el saldo será
- Sacar dinero: el método dirá si se puede, o sea si existe saldo
- Ingresar dinero: se incrementa el saldo
- Mostrar información: muestra la información disponible de la cuenta

apartado b)

Vamos a añadir el atributo banco a la clase. Como el banco es el mismo para todas las cuentas, el atributo será estático.

apartado c)

Existen gestores que administran las cuentas bancarias y atienden a sus propietarios. Cada cuenta, en caso de tenerlo, cuenta con un único gestor. Crear la clase **Gestor** de la que interesa guardar su nombre, tfno y el importe máximo autorizado con el que puede operar. Con respecto a los gestores, existen las siguientes restricciones:

- Un gestor tendrá siempre un nombre y un tfno
- Si no se asigna, el importe máximo autorizado por operación será de 10.000€