📘 Guía de Examen: Programación Orientada a Objetos en Python

Este documento resume los conceptos fundamentales de POO en Python basados en el ejemplo de la clase `CuentaCorriente`.

1. Estructura Básica y Constructor

En Python, el constructor siempre se llama `__init__`. El primer parámetro de cualquier método de instancia debe ser `self` (equivale al `this` de Java).

```
class CuentaCorriente:
    def __init__(self, codigo, titular, saldo=5000):
        self.__codigo = codigo   # Atributo "privado"
        self.__saldo = saldo     # Atributo "privado"
```

2. Niveles de Visibilidad (Encapsulamiento)

Python no tiene modificadores de acceso estrictos como `private` o `public`, pero usa convenciones de guiones bajos:

Sintaxis	Convención	Descripción
`self.dato`	**Público**	Accesible desde cualquier lugar.
`self._dato`	**Protegido**	Indica que no debería usarse fuera de la clase (pero se puede).
`self.__dato`	**Privado**	Python cambia el nombre internamente (*Name Mangling*) para dificultar el acceso.

> **Truco de examen:** Para acceder a un dato `__saldo` desde fuera (aunque no se deba):
>
> ```
> objeto._NombreClase__saldo
> ```

3. Atributos y Métodos de Clase (Static)

- **Atributo de Clase:** Se define fuera del `__init__`. Es compartido por todas las instancias (como `static` en Java).
- **@classmethod:** Recibe `cls` (la clase) como primer argumento. Se usa para métodos que afectan a la clase en general.
- **@staticmethod:** No recibe ni `self` ni `cls`. Es una función normal que vive dentro de la clase por organización.

```
    __numCuentas = 0 # Atributo estático

    @classmethod
    def getNumCuenta(cls):
        return cls.__numCuentas

    @staticmethod
    def devolverDatoSucursal():
        print("Dirección de la oficina...")
```

4. Getters y Setters (Decorador @property)

Es la forma "pythónica" de gestionar atributos privados. Permite usar métodos como si fueran variables normales.

```
    @property # Getter
    def saldo(self):
        return self.__saldo

    @saldo.setter # Setter (debe llamarse igual que el getter)
    def saldo(self, nuevo_saldo):
        self.__saldo = nuevo_saldo

# Uso:
c1.saldo = 1000  # Llama al setter automáticamente
print(c1.saldo)  # Llama al getter automáticamente
```

5. Métodos Mágicos (Dunder Methods)

Son métodos que empiezan y terminan con doble guion bajo (`__`). Permiten sobrecargar operadores y comportamientos.

- `__str__(self)` : El `toString()` de Java. Debe devolver un **string**. Se activa al hacer `print(objeto)` o `str(objeto)` .
- `__add__(self, otro)` : Define qué pasa al usar el símbolo `+` entre dos objetos.
- `__len__(self)` : Para usar `len(objeto)` .
- `__eq__(self, otro)` : Para comparar igualdad `==` .

6. Polimorfismo / Sobrecarga manual

Python no permite definir dos funciones con el mismo nombre y distintos parámetros. Se simula usando `isinstance()` o valores por defecto.

```
def funcionSobreCargada(valor):
    if isinstance(valor, int):
        # Lógica para enteros
    elif isinstance(valor, str):
```

```
        # Lógica para strings
```

7. Clases Vacías y Atributos Dinámicos

Puedes crear una clase mínima con `pass` y añadirle atributos en tiempo de ejecución (algo que no se puede hacer en lenguajes estáticos como Java).

```
class Alumno:
    pass

a1 = Alumno()
a1.nombre = "Juan" # Atributo creado al vuelo
```