Programación Orientada a Objetos en Python

Clases, Métodos de Instancia, Métodos de Clase y Estáticos

Python Full Stack - Clase 17

🎯 Objetivos de Hoy

¿Qué aprenderemos?

- 🔄 Repasar conceptos de clases, constructores y atributos
- 🎯 Dominar los métodos de instancia y el uso de `self`
- 🏛️ Entender atributos de clase vs atributos de instancia
- 🔧 Implementar métodos de clase con `@classmethod`
- ⚡ Crear métodos estáticos con `@staticmethod`
- 🎨 Diferenciar cuándo usar cada tipo de método
- 💳 Practicar con ejercicios de tarjetas de crédito
- 🔗 Aplicar conceptos avanzados de OOP

🔄 Repaso: Clases, Constructor y Atributos

🔄 ¿Qué recordamos de la clase anterior?

Conceptos clave que ya dominamos:

- ✅ **Clase:** Plantilla para crear objetos
- ✅ **Instancia:** Objeto específico creado de una clase
- ✅ **Constructor** `__init__` : Se ejecuta automáticamente al crear un objeto
- ✅ **Atributos de instancia:** Variables que pertenecen a cada objeto (`self.atributo`)
- ✅ `self` : Referencia al objeto actual

🔄 Recordando la Sintaxis Básica

```
class Usuario:
    def __init__(self, nombre, apellido, email):
        # Atributos de instancia
        self.nombre = nombre
        self.apellido = apellido
        self.email = email
        self.limite_credito = 30000
        self.saldo_pagar = 0

# Crear instancias
usuario1 = Usuario("María", "González", "maria@email.com")
usuario2 = Usuario("Pedro", "Martínez", "pedro@email.com")

# Acceder a atributos
print(usuario1.nombre)  # María
print(usuario2.saldo_pagar)  # 0
```

Cada instancia tiene sus propios valores de atributos.

🔄 El Constructor `__init__`

¿Qué hace el constructor?

```
class Usuario:
    def __init__(self, nombre, apellido, email):
        # Este código se ejecuta AUTOMÁTICAMENTE
        # cuando creas una nueva instancia
        self.nombre = nombre
        self.apellido = apellido
        self.email = email
```

El constructor inicializa los atributos del objeto cuando se crea.

Analogía: Es como llenar un formulario cuando te registras en un sitio web. Los campos son los atributos, y el constructor los completa con tus datos.

🔄 Concepto de `self`

`self` es como decir "yo mismo" o "este objeto"

```
class Usuario:
    def __init__(self, nombre):
        # self se refiere al objeto que se está creando
        self.nombre = nombre  # Este objeto tendrá este nombre

    def saludar(self):
        # self se refiere al objeto que llama al método
        print(f"Hola, soy {self.nombre}")
```

Cuando llamas: `usuario1.saludar()`

Python internamente hace: `Usuario.saludar(usuario1)`

Entonces: `self = usuario1`

🎯 Métodos de Instancia

🎯 ¿Qué son los Métodos de Instancia?

Definición Simple

Los **métodos de instancia** son funciones que pertenecen a una clase y actúan sobre una instancia específica. Tienen acceso a los atributos y otros métodos de esa instancia.

Características:

- ✅ Siempre reciben `self` como primer parámetro
- ✅ Pueden acceder a los atributos de la instancia (`self.atributo`)
- ✅ Se llaman desde una instancia: `objeto.metodo()`
- ✅ Cada instancia puede tener comportamientos diferentes

🎯 Analogía del Mundo Real

Piensa en un auto:

```
Auto (Instancia)
├── Atributos (Características)
│   ├── Color: Rojo
│   ├── Marca: Toyota
│   └── Velocidad: 0 km/h
└── Métodos (Acciones)
    ├── acelerar() → Aumenta la velocidad de ESTE auto
    ├── frenar() → Disminuye la velocidad de ESTE auto
    └── obtener_velocidad() → Muestra la velocidad de ESTE auto
```

Cada auto tiene sus propios atributos y puede realizar sus propias acciones.

La estructura básica:

```
class Usuario:
    def __init__(self, nombre, apellido, email):
        self.nombre = nombre
        self.apellido = apellido
        self.email = email
        self.saldo_pagar = 0

    # Método de instancia
    def hacer_compra(self, monto):
        # self se refiere a la instancia que llama al método
        self.saldo_pagar += monto
        print(f"{self.nombre} hizo una compra de ${monto}")
```

Partes importantes:

- `def` → Define una función (método)
- `hacer_compra` → Nombre del método
- `self` → SIEMPRE va primero (referencia al objeto)

🎯 Llamar Métodos de Instancia

Usamos la notación de punto:

```
class Usuario:
    def __init__(self, nombre, apellido, email):
        self.nombre = nombre
        self.apellido = apellido
        self.saldo_pagar = 0

    def hacer_compra(self, monto):
        self.saldo_pagar += monto

# Crear instancias
maria = Usuario("María", "González", "maria@email.com")
pedro = Usuario("Pedro", "Martínez", "pedro@email.com")

# Llamar métodos (cada instancia actúa independientemente)
maria.hacer_compra(150)  # María hace una compra
pedro.hacer_compra(200)  # Pedro hace una compra diferente

print(maria.saldo_pagar)  # 150
print(pedro.saldo_pagar)  # 200
```

🎯 ¿Cómo Funciona `self` Internamente?

El envío implícito de `self`:

```
# Cuando escribes esto:
maria.hacer_compra(150)

# Python internamente hace esto:
Usuario.hacer_compra(maria, 150)
#                    ↑
#                Esto es self
```

`self` se pasa automáticamente cuando llamas al método desde una instancia.

No necesitas pasar `self` manualmente - Python lo hace por ti.

🎯 Ejemplo Completo: Clase Usuario con Métodos

```
class Usuario:
    def __init__(self, nombre, apellido, email):
        self.nombre = nombre
        self.apellido = apellido
        self.email = email
        self.limite_credito = 30000
        self.saldo_pagar = 0

    def hacer_compra(self, monto):
        """Agrega una compra al saldo a pagar"""
        self.saldo_pagar += monto
        return self  # Retornamos self para encadenar métodos

    def pagar_tarjeta(self, monto):
        """Reduce el saldo a pagar"""
        if monto <= self.saldo_pagar:
            self.saldo_pagar -= monto
            print(f"Pago de ${monto} realizado")
        else:
            print("El monto excede el saldo a pagar")
        return self

    def mostrar_saldo(self):
        """Muestra el saldo actual"""
        print(f"{self.nombre} tiene un saldo de ${self.saldo_pagar}")
        return self
```

🎯 Usando los Métodos

```
# Crear usuario
miyagi = Usuario("Nariyoshi", "Miyagi", "miyagi@codingdojo.la")

# Llamar métodos individuales
miyagi.hacer_compra(150)
miyagi.hacer_compra(300)
miyagi.mostrar_saldo()  # Muestra: Nariyoshi tiene un saldo de $450

# Encadenar métodos (porque retornan self)
miyagi.hacer_compra(100).pagar_tarjeta(50).mostrar_saldo()
# Resultado: Pago de $50 realizado
#            Nariyoshi tiene un saldo de $500
```

Nota: Para encadenar métodos, cada método debe retornar `self`.

💪 Ejercicio: Métodos de Instancia

Crea un archivo `01-metodos-instancia.py` con:

1. Crea una clase `Perro` con:
   - Atributos: `nombre`, `raza`, `edad`, `energia` (valor inicial: 100)
   - Método `ladrar()`: Imprime "{nombre} está ladrando"
   - Método `correr()`: Reduce energía en 20 y muestra la energía restante
   - Método `dormir()`: Aumenta energía en 50 (máximo 100)
2. Crea dos instancias de `Perro` con diferentes nombres
3. Haz que ambos perros ladren, corran y duerman
4. Muestra la energía de cada perro

Pistas:

- Usa `self` para acceder a los atributos
- Recuerda usar `print()` para mostrar mensajes
- Puedes usar `min()` para limitar valores máximos

⏱️ **Tiempo:** 10 minutos

🏛️ Atributos de Clase vs Atributos de Instancia

🏛️ Dos Tipos de Atributos

¿Cuál es la diferencia?

Hasta ahora hemos usado **atributos de instancia**, que pertenecen a cada objeto individual. Pero también existen **atributos de clase**, que son compartidos por todas las instancias.

🏛️ Atributos de Instancia

Pertenecen a cada objeto individual

```
class Usuario:
    def __init__(self, nombre, email):
        # Atributos de instancia (diferentes para cada usuario)
        self.nombre = nombre
        self.email = email
        self.saldo_pagar = 0

usuario1 = Usuario("María", "maria@email.com")
usuario2 = Usuario("Pedro", "pedro@email.com")

# Cada usuario tiene su propio saldo
usuario1.saldo_pagar = 100
print(usuario2.saldo_pagar)  # 0 (no afecta a usuario2)
```

Cada instancia tiene sus propios valores.

🏛️ Atributos de Clase

Compartidos por todas las instancias

```python
class TarjetaCredito:
    # Atributo de clase (compartido por todas las instancias)
    banco = "Banco Internacional de Programadores"

    def __init__(self, limite_credito, saldo_pagar):
        # Atributos de instancia (diferentes para cada tarjeta)
        self.limite_credito = limite_credito
        self.saldo_pagar = saldo_pagar

# Crear tarjetas
tarjeta1 = TarjetaCredito(20000, 0)
tarjeta2 = TarjetaCredito(30000, 500)

# Todas las tarjetas tienen el mismo banco
print(tarjeta1.banco)  # Banco Internacional de Programadores
print(tarjeta2.banco)  # Banco Internacional de Programadores
```

Se define fuera del `__init__` y se comparte entre todas las instancias.

🏛️ Tabla Comparativa

Característica	Atributos de Instancia	Atributos de Clase
Definición	Dentro de `__init__` con `self.`	Fuera de `__init__`, sin `self.`
Pertenencia	Cada objeto tiene su propia copia	Compartido por todos los objetos
Cambio	Solo afecta a ese objeto	Afecta a todos los objetos
Acceso	`objeto.atributo`	`objeto.atributo` o `Clase.atributo`
Uso	Datos únicos de cada objeto	Datos comunes a todos los objetos

🏛️ Modificar Atributos de Clase

Cambiar para una instancia específica:

```
class TarjetaCredito:
    banco = "Banco Internacional de Programadores"

    def __init__(self, limite_credito):
        self.limite_credito = limite_credito

tarjeta1 = TarjetaCredito(20000)
tarjeta2 = TarjetaCredito(30000)

# Cambiar solo para tarjeta1
tarjeta1.banco = "Banco Nacional de Python"

print(tarjeta1.banco)  # Banco Nacional de Python
print(tarjeta2.banco)  # Banco Internacional de Programadores
```

Solo afecta a esa instancia específica.

🏛️ Modificar Atributos de Clase para Todos

Cambiar para toda la clase:

```
class TarjetaCredito:
    banco = "Banco Internacional de Programadores"

    def __init__(self, limite_credito):
        self.limite_credito = limite_credito

tarjeta1 = TarjetaCredito(20000)
tarjeta2 = TarjetaCredito(30000)

# Cambiar para toda la clase
TarjetaCredito.banco = "Banco Comercial de Desarrolladores"

print(tarjeta1.banco)  # Banco Comercial de Desarrolladores
print(tarjeta2.banco)  # Banco Comercial de Desarrolladores
```

Afecta a todas las instancias existentes y futuras.

🏛️ Ejemplo Completo: Atributos de Clase

```
class TarjetaCredito:
    # Atributo de clase
    banco = "Banco Internacional de Programadores"
    todas_las_tarjetas = []  # Lista compartida

    def __init__(self, limite_credito, saldo_pagar):
        # Atributos de instancia
        self.limite_credito = limite_credito
        self.saldo_pagar = saldo_pagar

        # Agregar esta tarjeta a la lista compartida
        TarjetaCredito.todas_las_tarjetas.append(self)
# Crear tarjetas
tarjeta1 = TarjetaCredito(20000, 0)
tarjeta2 = TarjetaCredito(30000, 500)
print(len(TarjetaCredito.todas_las_tarjetas))  # Ver todas las tarjetas creadas
```

Útil para rastrear todas las instancias creadas.

🔧 Métodos de Clase (`@classmethod`)

🔧 ¿Qué son los Métodos de Clase?

Definición Simple

Los **métodos de clase** son métodos que pertenecen a la clase misma, no a una instancia específica. Reciben `cls` (referencia a la clase) en lugar de `self`.

Características:

- ✅ Se definen con el decorador `@classmethod`
- ✅ Reciben `cls` como primer parámetro (no `self`)
- ✅ Pueden acceder a atributos de clase (`cls.atributo`)
- ✅ NO pueden acceder a atributos de instancia directamente
- ✅ Se llaman desde la clase: `Clase.metodo()` o `objeto.metodo()`

🔧 Analogía del Mundo Real

Piensa en una fábrica de autos:

```
Fábrica de Autos (Clase)
├── Método de Clase: cambiar_nombre_fabrica()
│   → Afecta a TODOS los autos (pasados y futuros)
│
└── Autos Individuales (Instancias)
    ├── Auto 1: Color rojo, Velocidad 60 km/h
    └── Auto 2: Color azul, Velocidad 80 km/h
```

Los métodos de clase actúan sobre la clase completa, no sobre autos individuales.

🔧 Sintaxis: Crear un Método de Clase

Usamos el decorador `@classmethod`:

```
class TarjetaCredito:
    # Atributo de clase
    banco = "Banco Internacional de Programadores"

    def __init__(self, limite_credito, saldo_pagar):
        self.limite_credito = limite_credito
        self.saldo_pagar = saldo_pagar

    # Método de clase
    @classmethod
    def cambiar_banco(cls, nuevo_nombre):
        # cls se refiere a la clase TarjetaCredito
        cls.banco = nuevo_nombre
        print(f"El banco cambió a: {nuevo_nombre}")
```

Partes importantes:

- `@classmethod` → Decorador que indica que es un método de clase
- `cls` → Referencia a la clase (como `self` pero para la clase)
- `cls.banco` → Accede al atributo de clase

🔧 Llamar Métodos de Clase

Desde la clase o desde una instancia:

```python
class TarjetaCredito:
    banco = "Banco Internacional de Programadores"

    @classmethod
    def cambiar_banco(cls, nuevo_nombre):
        cls.banco = nuevo_nombre

# Llamar desde la clase (más común)
TarjetaCredito.cambiar_banco("Banco Nacional de Python")

# También puedes llamar desde una instancia
tarjeta1 = TarjetaCredito(20000, 0)
tarjeta1.cambiar_banco("Banco Comercial")  # También funciona

# Verificar el cambio
print(TarjetaCredito.banco)  # Banco Comercial
```

Ambas formas funcionan, pero llamar desde la clase es más claro.

🔧 Ejemplo Completo: Métodos de Clase

```python
class TarjetaCredito:
    banco = "Banco Internacional de Programadores"
    todas_las_tarjetas = []

    def __init__(self, limite_credito, saldo_pagar):
        self.limite_credito = limite_credito
        self.saldo_pagar = saldo_pagar
        TarjetaCredito.todas_las_tarjetas.append(self)

    @classmethod
    def cambiar_banco(cls, nuevo_nombre):
        """Cambia el banco para todas las tarjetas"""
        cls.banco = nuevo_nombre

    @classmethod
    def total_saldos(cls):
        """Calcula el total de saldos de todas las tarjetas"""
        total = 0
        for tarjeta in cls.todas_las_tarjetas:
            total += tarjeta.saldo_pagar
        return total

# Usar los métodos de clase
tarjeta1 = TarjetaCredito(20000, 500)
tarjeta2 = TarjetaCredito(30000, 1000)

# Cambiar banco para todos
TarjetaCredito.cambiar_banco("Banco Nacional")

# Ver total de saldos
total = TarjetaCredito.total_saldos()
print(f"Total de saldos: ${total}")  # $1500
```

🔧 ¿Cuándo Usar Métodos de Clase?

Casos de uso comunes:

✅ **Cambiar configuración global** de la clase
✅ **Crear instancias con lógica especial** (factories)
✅ **Acceder a atributos de clase** que son compartidos
✅ **Realizar operaciones** sobre todas las instancias
✅ **Manipular listas compartidas** de instancias

No uses métodos de clase si necesitas acceder a atributos de instancia específicos.

Crea un archivo `02-metodos-clase.py` con:

1. Crea una clase `Estudiante` con:
   - Atributo de clase: `escuela = "Escuela de Programadores"`
   - Atributo de clase: `todos_los_estudiantes = []`
   - Constructor que recibe: `nombre`, `edad`, `nota`
   - En el constructor, agrega cada estudiante a `todos_los_estudiantes`
2. Crea un método de clase `cambiar_escuela()` que cambie el nombre de la escuela
3. Crea un método de clase `promedio_general()` que calcule el promedio de notas de todos los estudiantes
4. Crea 3 estudiantes y prueba los métodos de clase

Pistas:

- Usa `@classmethod` y `cls` para los métodos de clase

⚡ Métodos Estáticos (`@staticmethod`)

⚡ ¿Qué son los Métodos Estáticos?

Definición Simple

Los **métodos estáticos** son funciones auxiliares que pertenecen a la clase pero NO tienen acceso ni a atributos de instancia ni a atributos de clase. Solo reciben los argumentos que les pasas.

Características:

- ✅ Se definen con el decorador `@staticmethod`
- ✅ NO reciben `self` ni `cls`
- ✅ NO pueden acceder a atributos de instancia
- ✅ NO pueden acceder a atributos de clase
- ✅ Son funciones auxiliares "organizadas" dentro de la clase
- ✅ Se llaman desde la clase o desde una instancia

⚡ Analogía del Mundo Real

Piensa en una calculadora:

```
Calculadora (Clase)
├── Métodos Estáticos (Herramientas útiles)
│   ├── sumar(a, b) → Solo usa los números que le das
│   ├── multiplicar(a, b) → No necesita saber sobre la calculadora
│   └── es_par(numero) → Función auxiliar simple
│
└── No necesita conocer el estado de la calculadora
```

Son funciones útiles que están organizadas dentro de la clase, pero no dependen de ella.

⚡ Sintaxis: Crear un Método Estático

Usamos el decorador `@staticmethod`: Solo recibe los argumentos que necesita, nada más.

```
class TarjetaCredito:
    def __init__(self, limite_credito, saldo_pagar):
        self.limite_credito = limite_credito
        self.saldo_pagar = saldo_pagar
    def hacer_compra(self, monto):
        # Usar método estático para validar
        if TarjetaCredito.puede_comprar(self.limite_credito, self.saldo_pagar,
                                        monto):
            self.saldo_pagar += monto
        else:
            print("Tarjeta Rechazada, has alcanzado tu límite")
    @staticmethod  # Método estático (no usa self ni cls)
    def puede_comprar(limite, saldo_utilizado, monto):
        """Verifica si se puede realizar la compra"""
        return (saldo_utilizado + monto) <= limite
```

⚡ Llamar Métodos Estáticos

Desde la clase o desde una instancia:

```
class TarjetaCredito:
    @staticmethod
    def puede_comprar(limite, saldo_utilizado, monto):
        return (saldo_utilizado + monto) <= limite

# Llamar desde la clase (más común)
resultado = TarjetaCredito.puede_comprar(20000, 5000, 3000)
print(resultado)  # True

# También puedes llamar desde una instancia
tarjeta1 = TarjetaCredito(20000, 5000)
resultado = tarjeta1.puede_comprar(20000, 5000, 3000)
print(resultado)  # True
```

Ambas formas funcionan igual porque no dependen del objeto.

⚡ Ejemplo Completo: Métodos Estáticos

```
class TarjetaCredito:
    def __init__(self, limite_credito, saldo_pagar):
        self.limite_credito = limite_credito
        self.saldo_pagar = saldo_pagar

    def hacer_compra(self, monto):
        # Usar método estático para validar
        if TarjetaCredito.puede_comprar(self.limite_credito,
                                        self.saldo_pagar,
                                        monto):
            self.saldo_pagar += monto
            print(f"Compra de ${monto} realizada")
        else:
            print("Tarjeta Rechazada, has alcanzado tu límite")
        return self

    @staticmethod
    def puede_comprar(limite, saldo_utilizado, monto):
        """Verifica si se puede realizar la compra"""
        return (saldo_utilizado + monto) <= limite

    @staticmethod
    def calcular_interes(monto, tasa):
        """Calcula el interés sobre un monto"""
        return monto * tasa

# Usar métodos estáticos
tarjeta = TarjetaCredito(20000, 5000)
tarjeta.hacer_compra(3000)  # Compra realizada

# También se pueden usar directamente
interes = TarjetaCredito.calcular_interes(1000, 0.015)
print(f"Interés: ${interes}")  # $15.0
```

⚡ ¿Cuándo Usar Métodos Estáticos?

Casos de uso comunes:

✅ **Funciones auxiliares** que están relacionadas con la clase
✅ **Validaciones** que no dependen del estado del objeto
✅ **Cálculos** que solo necesitan los argumentos dados
✅ **Organizar código** relacionado pero independiente
✅ **Evitar repetición** (D.R.Y. - Don't Repeat Yourself)

No uses métodos estáticos si necesitas acceder a `self` o `cls`.

⚡ Comparación: Los Tres Tipos de Métodos

Característica	Método de Instancia	Método de Clase	Método Estático
Decorador	Ninguno	`@classmethod`	`@staticmethod`
Primer parámetro	`self`	`cls`	Ninguno
Accede a instancia	✅ Sí (`self.atributo`)	❌ No	❌ No
Accede a clase	✅ Sí (`Clase.atributo`)	✅ Sí (`cls.atributo`)	❌ No

¡Hora de practicar!

Crea un archivo `03-metodos-estaticos.py` con:

1. Crea una clase `Matematicas` con métodos estáticos:
   - `es_par(numero)`: Retorna True si el número es par
   - `es_primo(numero)`: Retorna True si el número es primo
   - `calcular_promedio(numeros)`: Calcula el promedio de una lista de números
2. Prueba cada método estático desde la clase
3. Crea una instancia y prueba llamar los métodos desde la instancia también

Pistas:

- Usa `@staticmethod` para cada método
- No uses `self` ni `cls`
- Para números primos, verifica divisibilidad del 2 hasta número-1

Aplicando Todo Juntos

🎨 Ejemplo Completo: Clase TarjetaCredito

```
class TarjetaCredito:
    # Atributo de clase
    banco = "Banco Internacional de Programadores"
    todas_las_tarjetas = []

    def __init__(self, limite_credito, saldo_pagar):
        # Atributos de instancia
        self.limite_credito = limite_credito
        self.saldo_pagar = saldo_pagar
        TarjetaCredito.todas_las_tarjetas.append(self)

    # Método de instancia
    def hacer_compra(self, monto):
        if TarjetaCredito.puede_comprar(self.limite_credito,
                                        self.saldo_pagar,
                                        monto):
            self.saldo_pagar += monto
        else:
            print("Tarjeta Rechazada")
        return self

    # Método de clase
    @classmethod
    def cambiar_banco(cls, nuevo_nombre):
        cls.banco = nuevo_nombre

    @classmethod
    def total_saldos(cls):
        total = 0
        for tarjeta in cls.todas_las_tarjetas:
            total += tarjeta.saldo_pagar
        return total

    # Método estático
    @staticmethod
    def puede_comprar(limite, saldo_utilizado, monto):
        return (saldo_utilizado + monto) <= limite
```

🎨 Usando la Clase Completa

```
# Crear tarjetas
tarjeta1 = TarjetaCredito(20000, 0)
tarjeta2 = TarjetaCredito(30000, 500)

# Usar método de instancia
tarjeta1.hacer_compra(1000).hacer_compra(500)

# Usar método de clase
TarjetaCredito.cambiar_banco("Banco Nacional")
print(TarjetaCredito.banco)  # Banco Nacional

# Usar método estático
puede = TarjetaCredito.puede_comprar(20000, 5000, 3000)
print(puede)  # True

# Ver total de saldos (método de clase)
total = TarjetaCredito.total_saldos()
print(f"Total: ${total}")  # $2000
```

1. Crea una clase `TarjetaCredito` completa con:
    - Atributo de clase: `banco = "Mi Banco"`
    - Atributo de clase: `todas_las_tarjetas = []`
    - Constructor: `limite_credito`, `saldo_pagar`
    - Método de instancia `hacer_compra(monto)`: Valida y agrega compra
    - Método de instancia `pagar_tarjeta(monto)`: Reduce el saldo
    - Método de clase `cambiar_banco(nombre)`: Cambia el banco
    - Método de clase `total_saldos()`: Suma todos los saldos
    - Método estático `puede_comprar(limite, saldo, monto)`: Valida compra
2. Crea 3 tarjetas y prueba todos los métodos
3. Cambia el banco y verifica que afecta a todas las tarjetas

Pistas:

📋 Resumen de Conceptos Clave

📋 Resumen: Tipos de Métodos

Tipo	Decorador	Primer Parámetro	Acceso a Instancia	Acceso a Clase
Instancia	Ninguno	`self`	✅ Sí	✅ Sí
Clase	`@classmethod`	`cls`	❌ No	✅ Sí
Estático	`@staticmethod`	Ninguno	❌ No	❌ No

📋 Resumen: Atributos

Tipo	Definición	Pertenencia	Ejemplo
Instancia	Dentro de `__init__` con `self.`	Cada objeto	`self.nombre`
Clase	Fuera de `__init__` sin `self.`	Compartido	`banco = "..."`

📋 Resumen: Sintaxis Clave

```python
class MiClase:
    # Atributo de clase
    atributo_clase = "valor"

    def __init__(self, param):
        # Atributo de instancia
        self.atributo_instancia = param

    # Método de instancia
    def metodo_instancia(self):
        return self.atributo_instancia

    # Método de clase
    @classmethod
    def metodo_clase(cls):
        return cls.atributo_clase

    # Método estático
    @staticmethod
    def metodo_estatico(param):
        return param * 2
```

📋 Lo Más Importante

Conceptos que debes recordar:

1. ✅ **Métodos de instancia** → Actúan sobre un objeto específico (`self`)
2. ✅ **Métodos de clase** → Actúan sobre la clase completa (`cls` , `@classmethod`)
3. ✅ **Métodos estáticos** → Funciones auxiliares sin acceso a clase/instancia (`@staticmethod`)
4. ✅ **Atributos de instancia** → Únicos para cada objeto (`self.atributo`)
5. ✅ **Atributos de clase** → Compartidos por todos (`Clase.atributo`)
6. ✅ `self` → Referencia al objeto actual
7. ✅ `cls` → Referencia a la clase (en métodos de clase)

1. Olvidar decoradores

```
# ❌ Error (no es método de clase)
def metodo_clase(cls):
    cls.atributo = valor

# ✅ Correcto
@classmethod
def metodo_clase(cls):
    cls.atributo = valor
```

2. Confundir `self` y `cls`

```
# ❌ Error en método de clase
@classmethod
def metodo(self):  # Debe ser cls, no self
    pass

# ✅ Correcto
@classmethod
def metodo(cls):
```

3. Intentar acceder a instancia desde método estático

```
# ❌ Error
@staticmethod
def metodo():
    print(self.atributo)  # No existe self

# ✅ Correcto
@staticmethod
def metodo(valor):
    print(valor)  # Solo usa los argumentos
```

📚 Recursos para Seguir Aprendiendo

Documentación y tutoriales:

- 📖 Python Docs - Classes - Documentación oficial
- 🎓 Real Python - Instance, Class, and Static Methods - Guía detallada
- 📚 MDN - OOP Concepts - Conceptos generales
- 🎮 Python Tutor - Visualiza la ejecución de código
- 💻 Codecademy - Python Classes - Práctica interactiva

💪 Práctica para Casa

Proyecto sugerido: Sistema de Biblioteca

Crea un sistema completo con:

1. 📚 **Clase** `Libro` con:
    - Atributos: título, autor, año, disponible (True/False)
    - Métodos de instancia: prestar(), devolver()
    - Método estático: validar_año(año) - verifica si el año es válido

1. 👤 **Clase** `Usuario` con:
    - Atributos: nombre, email, libros_prestados (lista)
    - Métodos de instancia: tomar_prestado(libro), devolver_libro(libro)
    - Método de clase: contar_usuarios() - cuenta cuántos usuarios hay
2. 🏛️ **Clase** `Biblioteca` con:
    - Atributo de clase: nombre = "Biblioteca Central"
    - Métodos de clase: cambiar_nombre(nombre)

Crea varias instancias y prueba todas las funcionalidades.

💡 Consejos Finales

Buenas prácticas:

✅ **Usa métodos de instancia** cuando necesites trabajar con datos del objeto
✅ **Usa métodos de clase** cuando necesites trabajar con datos compartidos
✅ **Usa métodos estáticos** para funciones auxiliares relacionadas
✅ **No abuses de métodos estáticos** - si no necesita estar en la clase, déjalo fuera
✅ **Nombres descriptivos** - `cambiar_banco()` es mejor que `cb()`
✅ **Documenta tus métodos** - usa docstrings para explicar qué hacen
✅ **Practica encadenar métodos** - retorna `self` cuando tenga sentido

🎉 ¡Felicidades!

Ya dominas métodos de instancia, clase y estáticos

Ahora puedes crear clases más poderosas y organizadas 🚀

? Preguntas

¿Alguna duda sobre métodos de clase o estáticos?

🙌 ¡Excelente Trabajo!

¡Nos vemos en la siguiente clase con asociación entre clases!

No olvides completar todos los ejercicios 📝