Herramientas de Big Data

Taller 1: Big Data Tools

Joao Muñoz Obando (0000281951)
Ingeniería Informática

Felipe Abella Ballesteros (0000291513)
Ingeniería informática

Lucas Santiago Reales Caicedo (0000297528)
Ingeniería informática

Juan Esteban Peña (303610)
Ingeniería Informática

Universidad de La Sabana
Maestría en Analítica Aplicada
Profesor: Hugo Franco, Ph.D.
25 de agosto de 2025

1. OOP for Big Data

El entendimiento de las clases en Python, cómo funcionan, buenas prácticas que se deben usar, son puntos clave para la iniciación en Big Data.

En este taller se nos ha dado el código para una calculadora básica en python, a través de varios retos, nuestro equipo tendrá que arreglar el código, refactorizando malas prácticas, añadiendo nuevas funcionalidades y en general, robusteciendo la calculadora dada; El taller consiste de siete retos los cuales deben resolverse de manera adecuada e idonea.

1.1. Refactoriar el constructor

El método init actual de la calculadora hace dos cosas, la inicialización de atributos y la lógica de cálculo. Esto viola el principio de los constructores, el cual dice que el método init debe establecer el estado inicial del objeto, nada más.

Refactorizar el método init de la calculadora programada en python para que solo inicialice el operando1, operando2 y los atributos de operación, remueva la lógica condicional que se encarga del cálculo al instanciar el objeto.

1.1.1. Método de solución

Para la solución del reto, se removio la lógica de calculo del constructor init. dejando solamente la inicialización de las variables operando1, operando2, operacion y current potencia.

La lógica de calculo se movio a otro metodo aparte denominado operación, el cual tiene como atributos, self y operacion, los cuales son de tipo object y string respectivamente. Dentro de este metodo se presenta una secuencia de condicionales los cuales imprimiran la operación especificada en el atributo operacion del metodo definino.

Algoritmo 1. Solución del problema (pseudocódigo)

```
def __init__(self,operando1=0, operando2=0, operacion=None):
    self.operando1=operando1
    self.operando2=operando2
    self.operacion=operacion
    self.current_potencia = 0
    #Removed logic from constructor, moved to a method appart.

def operacion(self, operacion): #New method for operations
    if operacion is not None:
        if operacion=='+':
            print(self.suma())
        elif operacion=='-':
            print(self.resta())
        elif operacion=='*':
            print(self.multiplicacion())
        elif operacion=='/':
            print(self.division())
```

```
elif operacion=='^':
    print(self.potencia())
elif operacion=='^2':
    print(self.sqrt())
```

Según el archivo presentado, Çalculadora(1).py.[es]te metodo se implementa con exito, haciendo que al instanciar el objeto de calculadora, no necesitemos poner un parametro adicional para la operación.

1.1.2. Resultados

De esta forma, al ejecutar la siguiente linea:

```
calc=Calculadora(3.12,6.0)
```

La calculadora no imprime automaticamente el tipo de operación y no necesita de un atributo de operación para su inicialización. Completando el reto.

1.1.3. Discusión

Al analizar los resultados, se observa que la refactorización del codigo fue exitosa pues el metodo init cumple su unica función que es la de inicializar el objeto, no maneja cualquier otro tipo de lógia. Para eso se utilizan los otros metodos creados dentro de la calculadora.

Con esto, seguímos una linea base de buenas practicas de metodos y en especial, de como el metodo init debe ser creado y manipulado.

1.2. Consistencia en operaciones

En la versión inicial, el método `resta` asignaba explícitamente el símbolo de la operación al atributo `self.operacion`, mientras que los métodos `suma`, `multiplicacion` y `division` no lo hacían. Esta inconsistencia podía generar errores, especialmente al invocar el método `__str__`, el cual depende de la variable `self.operacion`.

1.2.1. Método de solución

Se modificaron los métodos `suma`, `multiplicacion` y `division` para que de manera uniforme asignen el valor de la operación en el atributo `self.operacion`. De esta manera, cada vez que se ejecuta una operación, el estado de la calculadora refleja correctamente la última operación realizada.

Variables:

- `self.operando1: float` — primer operando.
- `self.operando2: float` — segundo operando.
- `self.operacion: string` — símbolo de la operación matemática.

Algoritmo 2. Solución del problema (pseudocódigo)

```
def suma(self, operando2=None):
    self.operacion = "+"
    if self.operando1 is not None and operando2 is not None:
        self.operando2 = operando2
        self.operando1 = self.operando1 + self.operando2
    return self

def multiplicacion(self, operando2=None):
    self.operacion = "*"
    if self.operando1 is not None and operando2 is not None:
        self.operando2 = operando2
        self.operando1 = self.operando1 * self.operando2
    return self

def division(self, operando2=None):
    self.operacion = "/"
    if self.operando1 is not None and operando2 is not None:
        self.operando2 = operando2
        self.operando1 = self.operando1 / self.operando2
    return self
```

1.2.2. Resultados

Ahora, al invocar cualquier operación, el atributo `self.operacion` se actualiza correctamente, lo cual garantiza que el método `__str__` siempre muestre el estado real de la operación realizada.

1.2.3. Discusión

La homogeneización en la asignación del atributo `self.operacion` asegura consistencia en la representación interna del objeto y previene errores al imprimir el estado del mismo. Esta mejora incrementa la robustez de la clase.

1.3. Mejora de la representación en cadena

En la versión inicial, el método `__str__` fallaba cuando no se definía una operación, ya que intentaba concatenar un `None` con cadenas, produciendo un error.

1.3.1. Método de solución

Se modificó el método `__str__` para que maneje la excepción mediante un bloque `try-except`. En caso de que no exista una operación definida, el método retorna la cadena descriptiva `Çalculadora en espera... No operación definida."`.

Algoritmo 3. Solución del problema (pseudocódigo)

```
def __str__(self):
    try:
        return str(self.operando1) + self.operacion + str(self.operando2)
    except Exception as err:
        return "Calculadora en espera... No operaci n definida."
```

1.3.2. Resultados

Si se crea un objeto sin especificar la operación:

```
calc = Calculadora(5, 3)
print(calc)
```

La salida será:

```
Calculadora en espera... No operación definida.
```

1.3.3. Discusión

Con esta mejora, la calculadora se vuelve más tolerante a estados incompletos y proporciona un mensaje informativo al usuario en lugar de un error. Esto incrementa la usabilidad de la clase y sigue el principio de "fail gracefully".

1.4. Operaciones con un solo operando

En la versión original, solo se soportaban operaciones binarias (suma, resta, multiplicación, división). No se incluían operaciones de un solo operando, como potenciación o raíz cuadrada.

1.4.1. Método de solución

Se agregaron dos nuevos métodos:

- `potencia(self, exponente: float)` — eleva `operando1` a la potencia indicada.
- `sqrt(self)` — calcula la raíz cuadrada de `operando1`, importando el módulo `math`.

Algoritmo 4. Solución del problema (pseudocódigo)

```
def potencia(self, exponente):
    self.operacion = "^"
    self.current_potencia = self.operando1 ** exponente
    return self.current_potencia

def sqrt(self):
    self.operacion = "^2"
    return math.sqrt(self.operando1)
```

1.4.2. Resultados

Al ejecutar:

```
calc = Calculadora(9)
print(calc.potencia(2)) # 81
print(calc.sqrt())      # 3.0
```

La calculadora ahora soporta operaciones unarias.

1.4.3. Discusión

La inclusión de operaciones de un solo operando extiende la funcionalidad de la calculadora, acercándola más al comportamiento de una calculadora científica. Esta mejora abre la puerta a añadir otras funciones matemáticas avanzadas como seno, coseno o tangente, ya implementadas en la clase hija.

1.5. Encadenamiento de métodos

En la versión inicial de la calculadora, para realizar una serie de operaciones continuas era necesario instanciar un nuevo objeto o actualizar manualmente los operandos. Esto dificultaba la ejecución fluida de cálculos consecutivos, reduciendo, principalmente, la eficiencia del código.

1.5.1. Método de solución

Se modificaron los métodos `suma`, `resta`, `multiplicacion` y `division` para que retornen el objeto `self` en lugar del resultado numérico. El valor calculado se almacena en `self.operando1`, permitiendo que pueda ser utilizado de inmediato en la siguiente operación.

Con este cambio se habilita el encadenamiento de métodos, donde múltiples operaciones pueden ejecutarse de manera secuencial en una sola instrucción.

Algoritmo 5. Solución del problema (pseudocódigo)

```
def suma(self, operando2=None):
    self.operacion = "+"
    if operando2 is not None:
        self.operando2 = operando2
    if not self._validar_operandos():
        return self
    self._resultado = self.operando1 + self.operando2
    self.operando1 = self._resultado
    return self

def multiplicacion(self, operando2=None):
    self.operacion = "*"
    if operando2 is not None:
        self.operando2 = operando2
    if not self._validar_operandos():
        return self
```

```
        self._resultado = self.operando1 * self.operando2
        self.operando1 = self._resultado
        return self
```

1.5.2. Resultados

Con esta implementación, es posible realizar operaciones continuas sin necesidad de reinstanciar el objeto. Por ejemplo:

```
calc = Calculadora(10)
resultado = calc.suma(5).multiplicacion(2)
print(resultado.operando1)  # 30
```

La salida confirma que el objeto mantiene el estado de la última operación realizada y lo utiliza para la siguiente.

1.5.3. Discusión

El encadenamiento de métodos mejora la legibilidad del código y ofrece una experiencia más cercana a los lenguajes de consulta o librerías modernas (como pandas). Además, mantiene la coherencia de la programación orientada a objetos: el objeto conserva su estado y permite construir secuencias de operaciones sin necesidad de crear nuevas instancias.

Este patrón hace que la calculadora sea más extensible y fácil de usar para cálculos complejos, cumpliendo así con las buenas prácticas de diseño de clases en Python.

1.6. Calculadora Científica con Herencia

Implementar las funciones de seno, coseno y tangente en la Calculadora Científica utilizando herencia, esto para que nos permita reutilizar el código de una clase existente para crear una nueva con funcionalidades adicionales.

1.6.1. Método de solución

Se creó una nueva clase llamada `CalculadoraCientifica` que hereda de `Calculadora`. De esta manera, la nueva clase puede usar todos los métodos básicos (`suma`, `resta`, `multiplicacion`, `division`, `potencia`, `sqrt`) sin necesidad de reimplementarlos.

Además, se añadieron tres métodos nuevos que permiten realizar cálculos trigonométricos sobre `self.operando1` (en radianes):

- `seno(self)`
- `coseno(self)`
- `tangente(self)`

Cada método actualiza el valor de `operando1` con el resultado de la operación y retorna el objeto `self`, manteniendo la posibilidad de encadenamiento de métodos.

Algoritmo 6. Solución del problema (pseudocódigo)

```
class CalculadoraCientifica(Calculadora):

    def seno(self):
        self.operacion = "sin"
        self.operando1 = math.sin(self.operando1)
        return self

    def coseno(self):
        self.operacion = "cos"
        self.operando1 = math.cos(self.operando1)
        return self

    def tangente(self):
        self.operacion = "tan"
        self.operando1 = math.tan(self.operando1)
        return self
```

1.6.2. Resultados

Con esta implementación, la clase `CalculadoraCientifica` puede realizar tanto operaciones básicas como trigonométricas, y todas las funciones soportan encadenamiento.

Ejemplo de uso:

```
calc = CalculadoraCientifica(10)
resultado = calc.suma(5).multiplicacion(2).seno()
print(resultado.operando1)
```

La salida corresponde al valor de:

$$\sin((10 + 5) \times 2) = \sin(30)$$

1.6.3. Discusión

El uso de herencia permitió extender la clase base `Calculadora` sin duplicar código, fomentando la reutilización y la escalabilidad.

Además, gracias al patrón de encadenamiento de métodos, es posible construir secuencias de operaciones complejas en una sola línea, lo que mejora la legibilidad y la facilidad de uso.

1.7. Manejo robusto de errores

Hasta ahora, el código solo gestionaba la excepción `ZeroDivisionError` en el método `division`. Sin embargo, el manejo de errores para el resto de operaciones nos garantiza el correcto fucnionamiento incluso cuando el usuario proporcione datos inválidos.

1.7.1. Método de solución

Se añadió un bloque `try...except` en cada uno de los métodos principales: `suma`, `resta`, `multiplicacion` y `division`. El bloque rodea la línea donde ocurre la operación aritmética.

De esta forma:

- Si los operandos son válidos (números), la operación se ejecuta normalmente.
- Si ocurre un `TypeError` (por ejemplo, al intentar sumar un número con un string), se captura la excepción, se muestra un mensaje de error y se retorna `None`.

Algoritmo 7. Solución del problema (pseudocódigo)

```
def suma(self, operando2=None):
    self.operacion = "+"
    if operando2 is not None:
        self.operando2 = operando2
    try:
        self.operando1 = self.operando1 + self.operando2
        return self
    except TypeError:
        print("Error: Los operandos deben ser num ricos.")
        return None

def resta(self, operando2=None):
    self.operacion = "-"
    if operando2 is not None:
        self.operando2 = operando2
    try:
        self.operando1 = self.operando1 - self.operando2
        return self
    except TypeError:
        print("Error: Los operandos deben ser num ricos.")
        return None

def multiplicacion(self, operando2=None):
    self.operacion = "*"
    if operando2 is not None:
        self.operando2 = operando2
    try:
        self.operando1 = self.operando1 * self.operando2
        return self
    except TypeError:
        print("Error: Los operandos deben ser num ricos.")
        return None

def division(self, operando2=None):
    self.operacion = "/"
```

```
        if operando2 is not None:
            self.operando2 = operando2
        try:
            self.operando1 = self.operando1 / self.operando2
            return self
        except ZeroDivisionError:
            print("Error: Divisi n por cero.")
            return None
        except TypeError:
            print("Error: Los operandos deben ser num ricos.")
            return None
```

1.7.2. Resultados

Con este cambio, la calculadora ahora puede detectar cuando un usuario introduce valores no numéricos y responder de manera clara sin interrumpir la ejecución.

Ejemplo:

```
calc = Calculadora(10)
resultado = calc.suma("abc")
# Mensaje en pantalla: "Error: Los operandos deben ser num ricos."
print(resultado)  # None
```

1.7.3. Discusión

El manejo robusto de errores mejora significativamente el funcionamiento del programa. Ahora, la calculadora no se detiene ante entradas inválidas y comunica al usuario qué salió mal.

Referencias

- Hugo Franco Ph.D, & Autor2, E. Calculadora.py