

Manual de práctica

Calculadora básica con POO.



Fundamentos de Python

Calculadora básica con POO

Introducción

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza objetos y clases para organizar el código de manera modular, reutilizable y escalable. Esta metodología permite modelar elementos del mundo real de forma más intuitiva y eficiente, facilitando el mantenimiento y la extensión del software.

En esta práctica, aprenderás a distinguir los principios de la POO para desarrollar una calculadora básica en Python. A través de este proyecto, podrás definir clases, utilizar propiedades para validar datos, implementar métodos para operaciones matemáticas y gestionar un historial de operaciones.

¡Comencemos!

Recursos necesarios

	Equipo <ul style="list-style-type: none">• Equipo de cómputo con conexión a internet• Windows 10 ,x86_64• 8Gb de RAM
	<ul style="list-style-type: none">• Visual Studio• Python

Etapas

Antes de iniciar, es necesario que identifiques las etapas que integran esta práctica.



Desarrollo de la práctica

Etapa 1. Presentación del caso

Calculadora básica

Eres desarrollador de una aplicación de calculadora básica y necesitas implementar un sistema que permita a los usuarios realizar operaciones matemáticas simples. Tu tarea será desarrollar el código en Python, para crear una calculadora capaz de interpretar las expresiones matemáticas que haya ingresado el usuario; ejecutar las operaciones correspondientes y mantener un historial de todas las operaciones realizadas durante la sesión.

La calculadora debe cumplir los siguientes requisitos:

- Debe manejar las cuatro operaciones aritméticas fundamentales: suma, resta, multiplicación y división.
- El sistema debe leer las expresiones matemáticas en formato de texto, interpretar correctamente los números y operadores, validar que los datos ingresados sean correctos, ejecutar el cálculo correspondiente y mostrar el resultado inmediatamente al usuario.
- Debe incluir funcionalidades de historial, que permitan al usuario visualizar todas las operaciones realizadas durante la sesión.
- Debe ejecutarse en un bucle continuo, hasta que el usuario decida salir, mediante un comando especial.
- El programa debe manejar la interacción con el usuario con mensajes claros, procesar comandos especiales y gestionar errores (división entre cero, entradas no válidas, etc.).



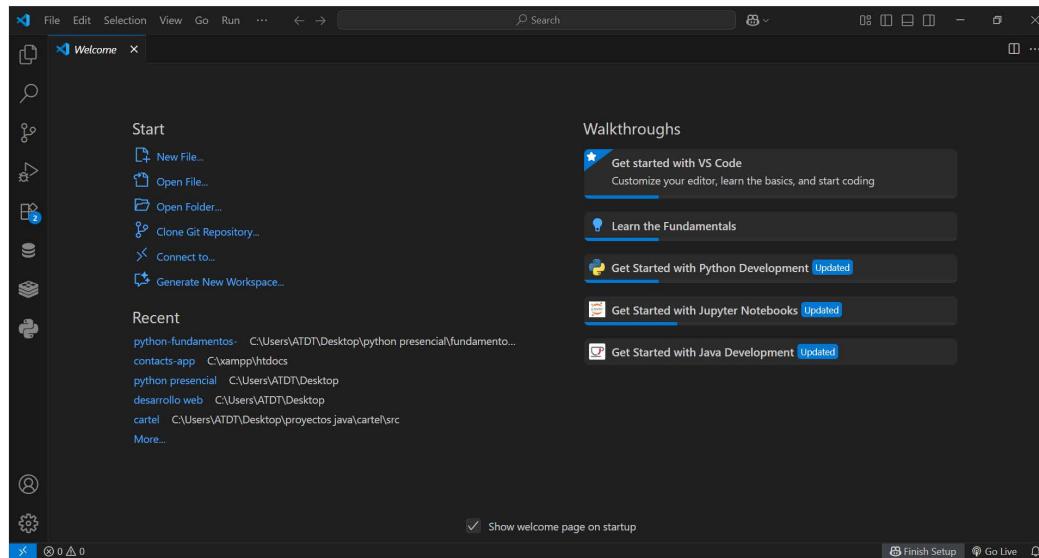
Para saber más

A lo largo de esta práctica aprenderás y desarrollarás habilidades importantes como:

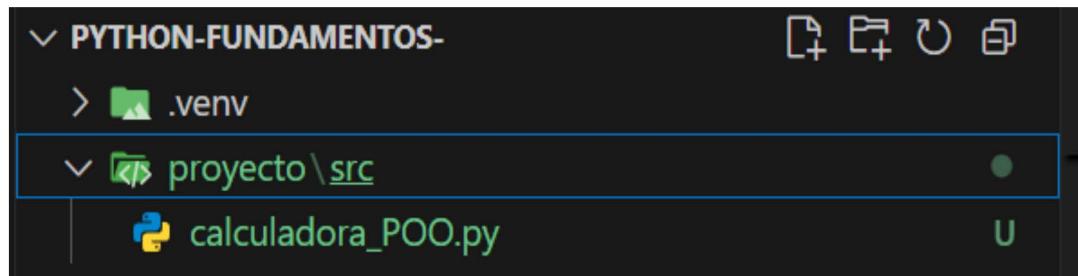
- Pensamiento algorítmico - Dividir problemas complejos
- UX básica - Ponerse en lugar del usuario
- Iteración - Mejorar progresivamente

Etapa 2. Preparación del entorno de desarrollo

1. **Inicia** los programas Visual Studio y Python.



2. **Crea** una carpeta desde Visual Studio dando clic en “New folder” y asignale el nombre *Proyectos*, posteriormente, **guarda** en la carpeta el programa con el que trabajarás, con el nombre *calculadora_POO.py*.



Etapa 3. Desarrollo del código

Definición de la clase “Calculadora”

1. **Crea** una clase llamada “Calculadora” que servirá como base para construir tu programa. Para iniciar la clase, **escribe** el siguiente código.

- *class*: Palabra clave para definir una nueva clase.
- *Calculadora*: Nombre de la clase (por convención, se usa *PascalCase*).

```
class Calculadora:
```

2. **Define** un método constructor (*__init__*) que reciba dos parámetros (*numero1* y *numero2*). Este método servirá para inicializar los valores de la calculadora.

- *__init__*: Método especial llamado “constructor”, se ejecuta al crear una nueva instancia.
- *self*: Referencia al atributo de la instancia actual de la clase (obligatorio en todos los métodos).
- *numero1, numero2*: Parámetros con valores por defecto (0).

```
def __init__(self, numero1, numero2):
```



Notas

Para esta práctica sólo será necesario que funcione con dos valores (*numero1*, *numero2*).

3. Escribe el siguiente código para guardar los valores en los atributos de la clase.

- Crea un atributo protegido con el valor del parámetro.
- El guión bajo(_) indica que es un atributo "protegido" (convención Python).

```
self._numero1 = numero1  
self._numero2 = numero2
```

4. Agrega una lista vacía llamada “_historial”. Esta lista se usará para almacenar el historial de las operaciones que se realicen.

```
self._historial = []
```

```
proyecto4 > src > 📁 calculadora_POO.py > 🏷 Calculadora > 🎯  
1 ↴ class Calculadora:  
2     numero1 = 0  
3     numero2 = 0  
4  
5 ↴     def __init__(self, numero1, numero2):  
6         self._numero1 = numero1  
7         self._numero2 = numero2  
8         self._historial = []
```

Propiedades (Getters y Setters)

5. Crea una propiedad llamada “numero1” que retorne el valor del atributo `_numero1`. Esta propiedad permitirá consultar el primer número de la calculadora sin necesidad de modificarlo directamente.

- Decorador que convierte el método en una propiedad de solo lectura.
- `def numero1:` Getter que retorna el valor actual de `_numero1`.

```
@property
def numero1(self):
    return self._numero1
```

6. Define un setter para la propiedad `numero1`, de manera que permita asignar un nuevo valor al atributo `_numero1`. Esto servirá para modificar el primer número de la calculadora de forma controlada.

- Decorador para definir el setter de la propiedad `numero1`
- `nuevo_numero1`: Nuevo valor que se quiere asignar.

```
@numero1.setter
def numero1(self, nuevo_numero1):
```

7. Dentro del setter de `número1`, **agrega** una validación para asegurar que solo se asignen números enteros o decimales. Para ello, **escribe** el siguiente condicional: el `if` verifica si el nuevo valor es entero o float, si es válido, asigna el valor a la variable de clase `self._numero1`.

```
if type(nuevo_numero1) in (int, float):  
    self._numero1 = nuevo_numero1
```

8. Si no es número, el programa deberá mostrar un error con el mensaje “*Debe ser un número*”, para ello, **escribe** la siguiente línea de código.

- `raise ValueError`: Lanza un mensaje de error si el valor no es numérico.
- El mensaje describe el error específico.

```
else:  
    raise ValueError("Debe ser un número")
```

9. **Repite** el proceso (5 - 8) para el *numero2*.

```
10 |         """
11 |         """-----setter and getter-----"""
12 |     @property
13 |     def numero1(self):
14 |         return self._numero1
15 |
16 |     @numero1.setter
17 |     def numero1(self, nuevo_numero1):
18 |         if type(nuevo_numero1) in (int, float):
19 |             self._numero1 = nuevo_numero1
20 |         else:
21 |             raise ValueError("Debe ser un número")
22 |     @property
23 |     def numero2(self):
24 |         return self._numero2
25 |
26 |     @numero2.setter
27 |     def numero2(self, nuevo_numero2):
28 |         if type(nuevo_numero2) in (int, float):
29 |             self._numero2 = nuevo_numero2
30 |         else:
31 |             raise ValueError("Debe ser un número")
32 |
33 |
```

Implementación de operaciones matemáticas

10. **Crea** un método llamado “sumar” que realice la operación de suma entre los atributos *_numero1* y *_numero2*. El resultado se guardará en la variable “resultado”.

- *numero1 + numero2*: Realiza la suma aritmética.
- *resultado*: Almacena el resultado en una variable libre.

```
def sumar(self):  
    resultado = self._numero1 + self._numero2
```



Notas

Repite el mismo procedimiento para crear los métodos restar, multiplicar y dividir, cambiando únicamente la operación que se realiza entre `_numero1` y `_numero2`.

11. Dentro del método `sumar`, **utiliza** el método `_registrar_operacion` pasando como argumentos el símbolo '+' y la variable `resultado`. Luego, devuelve el valor de `resultado`. Realiza el mismo proceso para las demás operaciones.

- `registrar_operacion`: Llama al método privado para guardar en el historial.
- `return`: Retorna el resultado de la operación.

```
self._registrar_operacion('+', resultado)  
return resultado
```

Método privado para registro en historial

12. **Define** el método privado `_registrar_operacion` que reciba `operador` y `resultado`. Aquí escribirás el código que almacenará cada operación en la lista de historial de la calculadora.

- `registrar_operacion`: El guión bajo indica que es método privado (solo para uso interno).
- `operador`: Símbolo de la operación (+, -, *, /)
- `resultado`: Resultado numérico de la operación.

```
def _registrar_operacion(self, operador, resultado):
```

13. Dentro del método `_registrar_operacion`, **escribe** el código que agregue un diccionario a la lista `_historial`. El diccionario debe contener la operación realizada como texto y el resultado.

- `historial.append`: Agrega un nuevo elemento al final de la lista
- `{...}`: Crea un diccionario con dos claves: 'operación' y 'resultado'
- Como valor para la clave `operador` se concatenan las variables de la operación solicitada, creando una *string* con la expresión completa.

```
self._historial.append({  
    'operacion': f"{self._numero1} {operador} {self._numero2}",  
    'resultado': resultado})
```

Visualización del historial

14. **Crea** un método llamado "ver_historial" que revise si la lista `_historial` está vacía. Si no hay operaciones registradas, debe mostrar el mensaje "No hay operaciones en el historial" y salir del método.

- El `if` se usa para verificar si el historial está vacío.
- Si está vacío, muestra el mensaje y termina la ejecución con `return`.

```
def ver_historial(self):  
    if not self._historial:  
        print("No hay operaciones en el historial.")  
    return
```

- 15.** Dentro del método `ver_historial`, **escribe** el código que recorra la lista `_historial` y muestre todas las operaciones realizadas junto con sus resultados. Usa un contador para numerar cada operación.

- Se inicializa con el valor 1 (la numeración empezará desde 1) para llevar la cuenta de la posición/número de cada operación en el historial.
- Iterar sobre cada operación almacenada en el historial.
- Mostrar cada operación numerada con su resultado.

```
print("\n--- Historial de Operaciones ---")
contador = 1
for operacion in self._historial:
    print(f"{contador}. {operacion['operacion']} =
{operacion['resultado']}")
```

Función interpretadora de expresiones

- 16. Crea** una función llamada “`interpretar_expresión`” que reciba como parámetro una cadena `expresión`. Luego, recorre los operadores '+', '-', '*' y '/' para identificar cuál se utiliza en la expresión.

- `for` se usa para iterar, por cada operador posible, operaciones básicas.

```
def interpretar_expresion(expresion):
    for operador in ['+', '-', '*', '/']:
```

17. Verifica si el operador actual se encuentra en la expresión. Si es así, divide la expresión en partes, usando el operador como un separador.

- El *if* verifica si el operador está presente en la expresión.
- *expresion.split*: Divide la expresión en partes, usando el operador como separador.

```
if operador in expression:  
    partes = expresion.split(operador)
```

18. Dentro del bloque que verifica las partes de la expresión, **convierte** ambos elementos a números (float) y elimina espacios en blanco con *strip()*. Luego, devuelve los dos números y el operador encontrado.

- Keyword *if* y *len*: Verifican que haya exactamente 2 partes (número - operador).
- *Partes[n]*: Elimina espacios en blanco alrededor del primer número y convierte el *string* a número decimal usando *float*.
- *return*: Retorna los valores interpretados.

```
if len(partes) == 2:  
    num1 = float(partes[0].strip())  
    num2 = float(partes[1].strip())  
    return num1, num2, operador
```

```
70
71 ✓ def interpretar_expresion(expresion):
72     """Interpreta la expresión matemática ingresada"""
73 ✓     for operador in ['+', '-', '*', '/']:
74 ✓         if operador in expresion:
75             partes = expresion.split(operador)
76 ✓             if len(partes) == 2:
77                 num1 = float(partes[0].strip())
78                 num2 = float(partes[1].strip())
79             return num1, num2, operador
```

Programa principal (main)

19. **Crea** una función *main* que inicialice un objeto de la clase “Calculadora”. Esto servirá como punto de entrada para usar todas las funcionalidades de la calculadora.

- Se crea una función llamada *main*, que significa **función principal** que inicializa todo el programa y que contendrá las instancias (objetos) de las diferentes clases.
- *calc*: Crea una nueva instancia de la clase *Calculadora*.

```
def main():
    calc = Calculadora()
```

20. **Muestra** instrucciones iniciales al usuario, acerca de cómo usar la calculadora.

```
print("Calculadora Básica. Escribe 'salir' para terminar o
'historial' para ver operaciones.\n")
```

21. Crea un bucle `while True`, que permita al usuario ingresar operaciones continuamente. Usa `input` para capturar la expresión que el usuario escriba.

- `while True`: Bucle infinito que mantiene el programa ejecutando hasta que el usuario lo termine.
- Se captura la entrada del usuario y la almacena en la variable `entrada`.

```
while True:  
    entrada = input("Ingresa la operación (ejemplo: 5 + 5): ")
```

22. Escribe la instrucción para que el usuario pueda decidir cuándo salir de la calculadora.

- `entrada` y los métodos, se usan para eliminar espacios y convierte a minúsculas, para preparar la comparación.
- "salir": Compara con el comando de salida.
- `break`: Rompe el bucle `while`, terminando el programa.

```
if entrada.strip().lower() == "salir":  
    print("¡Hasta pronto!")  
    break
```

23. Verifica si la entrada del usuario es “historial” (ignorando mayúsculas y espacios). Si es así, llama al método *ver_historial* del objeto *calc* y continúa con la siguiente iteración del bucle.

- *ver_historial*: Llama al método para mostrar historial.
- *continue*: Salta al siguiente ciclo del bucle.

```
if entrada.strip().lower() == "historial":  
    calc.ver_historial()  
    continue
```

24. Escribe el siguiente código, para que valide si es que existe un error, en caso de no encontrar uno de los valores.

- *interpretar_expresion*: Intenta interpretar la expresión.
- *if not resultado*: Retorna a None (expresión inválida).
- Muestra mensaje de error y continúa con siguiente iteración.

```
resultado = interpretar_expresion(entrada)  
if not resultado:  
    print("Expresión no válida. Usa el formato: número operador  
número (ej. 5 + 5)\n")  
    continue
```

25. Asigna los valores devueltos por *interpretar_expresion* a *num1*, *num2* y *operador*. Luego, utiliza los setters de la calculadora, para actualizar los atributos *numero1* y *numero2* del objeto *calc*.

- *num1, num2, operador*: Desempaquetá la tupla retornada.
- *calc.numero1 = num1*: Asigna valor usando setter.

```
num1, num2, operador = resultado
calc.numero1 = num1
calc.numero2 = num2
```

26. Aplica el siguiente código, para especificar el tipo de operación a realizar, detectando la instrucción del usuario.

- Ejecuta la operación correspondiente, según el operador seleccionado por el usuario.
- *print*: Muestra el resultado de la operación.

```
if operador == '+':
    print("Resultado:", calc.sumar())
elif operador == '-':
    print("Resultado:", calc.restar())
elif operador == '*':
    print("Resultado:", calc.multiplicar())
elif operador == '/':
    print("Resultado:", calc.dividir())
```

```
def main():
    calc = Calculadora()
    print("Calculadora Básica. Escribe 'salir' para terminar o 'historial' para ver operaciones.\n")

    while True:
        entrada = input("Ingresa la operación (ejemplo: 5 + 5): ")

        if entrada.strip().lower() == "salir":
            print("¡Hasta pronto!")
            break

        if entrada.strip().lower() == "historial":
            calc.ver_historial()
            continue

        resultado = interpretar_expresion(entrada)
        if not resultado:
            print("Expresión no válida. Usa el formato: número operador número (ej. 5 + 5)\n")
            continue
        num1, num2, operador = resultado
        calc.numero1 = num1
        calc.numero2 = num2
        if operador == '+':
            print("Resultado:", calc.sumar())
        elif operador == '-':
            print("Resultado:", calc.restar())
        elif operador == '*':
            print("Resultado:", calc.multiplicar())
        elif operador == '/':
            print("Resultado:", calc.dividir())
```

27. Al final del programa, **llama** a la función *main()* para iniciar la calculadora cuando se ejecute el programa.

- *main():* Llama a la función principal para iniciar el programa

```
l13
l14
l15
l16  main()
```

Ejecución del programa

28. Para **ejecutar** el código de manera correcta:

- Copia cada sección en orden.
- Mantén la indentación correcta (espacios en blanco al inicio de la línea de código, para indicar a qué bloque de código pertenece).
- Ejecuta el archivo completo.
- Prueba con diferentes expresiones matemáticas.

```
Calculadora Básica. Escribe 'salir' para terminar o 'historial' para ver operaciones.
```

```
Ingresa la operación (ejemplo: 5 + 5): 5*5
Resultado: 25.0
```

```
Ingresa la operación (ejemplo: 5 + 5): 3 / 9
Resultado: 0.3333333333333333
```

```
Ingresa la operación (ejemplo: 5 + 5): []
```



Reflexión

Después de terminar el desarrollo, surgen preguntas importantes que van más allá de que el programa “funcione”:

- ¿Debe la calculadora permitir operaciones inválidas, como la división por cero?

Actualmente, cada operación matemática sigue unas reglas fijas, ya definidas previamente, pero...

- ¿Dónde está el límite sobre lo que debe hacer la calculadora de forma automática?



Ejercicio de
reforzamiento

Si deseas seguir practicando tus conocimientos:

Replica este ejercicio modificando los *if* repetitivos por *match case* para que el código tenga una mejor simplicidad.

¡Listo!

Has concluido exitosamente la práctica sobre el desarrollo de una calculadora básica con POO.