Aproximación teórica y secuencia didáctica para la mediación didáctica de la matemática con Python

Tito Amaury Tapia Bastidas
Uniputumayo

Octubre 2025

Índice general

Listings

Presentación

En el presente documento se diseña una **aproximación teórica y práctica para la mediación didáctica de la matemática con Python**. Su propósito principal es **convertirse en un instrumento de análisis, validación y aplicación didáctica**, tanto para los expertos evaluadores como para los docentes e investigadores interesados en la integración del pensamiento computacional en la enseñanza de la matemática.

En ese sentido, desde la perspectiva epistemológica y metodológica, este trabajo se fundamenta en el **paradigma sociocrítico** y en la **investigación-acción** como ejes que articulan teoría y práctica. La programación en Python se asume aquí no solo como un recurso técnico, sino como un *mediador epistémico* que favorece la comprensión conceptual, la reflexión crítica y la transformación de la práctica docente.

La estructura del documento se organiza en tres partes complementarias:

1. **Fundamentación teórica:** presenta el marco conceptual y filosófico que sustenta la mediación didáctica, destacando el valor del error como fuente de conocimiento y la pertinencia del enfoque sociocrítico en la formación tecnológica.

2. **Secuencia didáctica:** desarrolla un conjunto de capítulos que integran el aprendizaje progresivo de Python con el análisis matemático, articulando ejercicios, ejemplos y reflexiones orientadas a la resolución de errores algebraicos comunes.

3. **Instrumento de validación:** se destina a la evaluación del diseño por parte de jueces expertos, conforme a los criterios de rigor científico propuestos por Lincoln y Guba (1985): credibilidad, transferibilidad, dependencia y confirmabilidad.

En estel documento si tiene la intención, en consecuencia, **evidenciar la coherencia entre la teoría, la práctica y la reflexión docente**, mostrando cómo la mediación didáctica con Python puede contribuir a la resignificación del aprendizaje matemático en programas tecnológicos. Cada capítulo, ejercicio y reflexión fueron construidos a partir de la experiencia formativa en el Programa de Desarrollo de Software de la Universidad del Putumayo, con el propósito de fortalecer la autonomía, la creatividad y el pensamiento crítico de los estudiantes.

Por lo tanto este material es también una herramienta abierta para el diálogo académico. Su lectura, análisis y validación por expertos permitirán consolidar un modelo de mediación didáctica adaptable a diversos contextos educativos y alineado con los desafíos contemporáneos de la enseñanza de las matemáticas.

Capítulo 1

Fundamentación teórica

1.1. El error como dato formativo

El error, en el contexto educativo y específicamente en la enseñanza de la matemática, ha sido tradicionalmente interpretado como una manifestación del fracaso o de la ausencia de conocimiento. Sin embargo, desde una perspectiva contemporánea y crítica del aprendizaje, el error se comprende como un **dato formativo**: una evidencia observable que refleja los procesos de pensamiento, las concepciones previas y las estrategias cognitivas que el estudiante pone en juego al enfrentarse a una tarea matemática.

Ahora bien, autores como Bachelard (1938) y Astolfi (1999) concluyen que el conocimiento científico progresa a través de la *superación de los obstáculos epistemológicos*. En este sentido, el error se convierte en un insumo estructural del proceso de aprendizaje, porque al ser identificado, analizado y comprendido, permite reorganizar el conocimiento y generar nuevas formas de comprensión.

Sine embargo, en el campo de la didáctica de la matemática, Radatz (1979) propone una clasificación de los errores según su origen: errores conceptuales, procedimentales, lingüísticos, y de aplicación de reglas. En ese orden, esta tipología ha orientado el desarrollo de estrategias pedagógicas centradas en el análisis del error como una vía para acceder al pensamiento del estudiante. Así, el error no se considera un desvío del conocimiento, sino una manifestación de cómo el sujeto **construye y negocia significados** en su tránsito hacia la comprensión matemática.

Desde el enfoque sociocrítico y la investigación-acción (Carr & Kemmis, 1988), el análisis del error se entiende además como un acto de reflexión emancipadora: identificar los errores implica revisar las condiciones institucionales, discursivas y pedagógicas que los generan. El docente deja de ser un simple evaluador del desempeño para asumir el rol de **investigador de su propia práctica**, promoviendo la autorregulación cognitiva y la autonomía de los estudiantes.

En este marco, la mediación didáctica con Python adquiere un valor particular, ya que el lenguaje de programación ofrece un entorno donde los errores son *visibles, verificables y corregibles*. Cada instrucción escrita en Python puede ser comprendida como una **hipótesis cognitiva** que el estudiante pone a prueba frente a un resultado inmediato. El error de sintaxis, de lógica o de procedimiento no se penaliza, sino que se convierte en una oportunidad para analizar la estructura del pensamiento matemático y su traducción algorítmica.

De esta forma, el error cumple una triple función:

1. **Cognitiva:** permite detectar y comprender las estrategias de razonamiento del

estudiante.

2. **Didáctica:** orienta la intervención del docente y la selección de mediaciones adecuadas.

3. **Epistemológica:** posibilita la reconstrucción del conocimiento a partir del conflicto cognitivo.

El reconocimiento del error como dato formativo transforma la evaluación en un proceso reflexivo, donde el objetivo no es únicamente medir el resultado, sino comprender *cómo y por qué* el estudiante llega a él. Así, el error se revaloriza como una **manifestación del pensamiento en desarrollo**, un punto de partida para la mediación pedagógica y la reconstrucción crítica del saber matemático.

En el marco de la mediación didáctica con Python, cada error constituye un objeto de análisis que puede ser modelado, simulado y verificado. Esta dinámica convierte al lenguaje de programación en un espacio dialógico entre el estudiante y el conocimiento, permitiendo que el error se transforme en comprensión y la práctica en investigación.

1.2. Síntesis del texto base (Pérez Zárate)

El artículo de Juana Inés Pérez Zárate, titulado *"Errores algebraicos más comunes que cometen los alumnos de bachillerato"*, constituye un referente fundamental para comprender las dificultades que enfrentan los estudiantes en la manipulación simbólica del álgebra escolar. La autora parte de una afirmación esencial: **todo conocimiento está mezclado con errores y prejuicios**, lo cual obliga a reconocer el error como parte constitutiva del aprendizaje y no como un elemento ajeno a él.

Pérez Zárate plantea que, históricamente, la enseñanza de la matemática ha estado centrada en la transmisión de procedimientos más que en la comprensión de significados. En consecuencia, los estudiantes suelen memorizar reglas sin interiorizar sus fundamentos conceptuales, lo que genera confusiones recurrentes cuando deben aplicar dichos procedimientos en nuevos contextos. Esta brecha entre la instrucción y la comprensión se manifiesta en la aparición de errores típicos, que funcionan como indicadores del modo en que el estudiante construye sentido en torno a las operaciones algebraicas.

En su análisis, la autora identifica una serie de **errores sistemáticos** vinculados con:

- la jerarquía de operaciones y el uso de signos,
- la multiplicación y división de monomios y polinomios,
- la suma y resta de fracciones algebraicas,
- la aplicación de las leyes de los exponentes, y
- la interpretación incorrecta de raíces y desarrollos binomiales.

Pérez Zárate enfatiza que estos errores no deben ser considerados como simples equivocaciones, sino como *manifestaciones del proceso cognitivo* mediante el cual el estudiante intenta comprender el lenguaje algebraico. Cada error, por tanto, refleja una estructura

de pensamiento parcial, incompleta o alternativa, que el docente puede aprovechar para orientar la enseñanza hacia la reconstrucción del conocimiento.

Asimismo, la autora resalta la importancia del papel del maestro como **investigador de su práctica**. El docente no solo enseña, sino que también observa, analiza e interpreta los modos en que sus estudiantes piensan y aprenden. Desde esta mirada, el aula se convierte en un espacio de investigación continua, donde el error es una evidencia empírica que puede ser estudiada, categorizada y transformada en una oportunidad de aprendizaje significativo.

En sintonía con el enfoque de esta investigación, retomamos la propuesta de Pérez Zárate para plantear una **mediación didáctica apoyada en Python**, que permite representar, simular y verificar los procedimientos algebraicos. Mediante el uso de bibliotecas como `SymPy`, los estudiantes pueden **visualizar los errores algebraicos**, contrastar sus resultados con los de un modelo simbólico y reflexionar sobre el porqué de sus discrepancias. Esta metodología convierte al error en un objeto de estudio interactivo y reproducible, fortaleciendo la comprensión conceptual y el pensamiento computacional.

La síntesis del texto de Pérez Zárate nos permite comprender que la persistencia de los errores algebraicos no responde únicamente a fallas de memorización, sino a una carencia de mediaciones significativas. Integrar Python en el proceso de enseñanza-aprendizaje abre la posibilidad de convertir esos errores en oportunidades de reflexión y reconstrucción cognitiva, transformando la experiencia matemática en un ejercicio activo de comprensión y crítica.

1.3. Mediación didáctica y paradigma sociocrítico

La **mediación didáctica**, desde la perspectiva sociocultural de Lev Vigotsky (1978), se entiende como el proceso mediante el cual el aprendizaje se produce gracias a la interacción entre el sujeto, los otros y los instrumentos culturales que actúan como mediadores. El conocimiento no se transfiere de manera directa, sino que se construye a través de la comunicación, el diálogo y la acción conjunta. En este contexto, los signos, el lenguaje y las herramientas tecnológicas se convierten en vehículos para la interiorización del pensamiento y la transformación de la experiencia.

Aplicado a la enseñanza de la matemática, Python se concibe como un **instrumento de mediación simbólica y cognitiva**. El lenguaje de programación permite representar operaciones, visualizar procesos y comprobar hipótesis, convirtiendo la actividad matemática en una experiencia interactiva. Cada fragmento de código escrito por el estudiante funciona como una forma de expresión del pensamiento algebraico, un signo cultural que traduce el razonamiento lógico a estructuras computacionales verificables. De este modo, Python no sólo facilita la ejecución de algoritmos, sino que actúa como un mediador epistémico que potencia la comprensión y la reflexión sobre los propios procedimientos.

Desde el **paradigma sociocrítico**, desarrollado por Carr y Kemmis (1988), la educación se concibe como una práctica social orientada a la transformación y a la emancipación del sujeto. El conocimiento no es un producto neutral, sino el resultado de un proceso dialéctico entre teoría y práctica, entre reflexión y acción. El docente-investigador asume, por tanto, el papel de un sujeto crítico que analiza su práctica, detecta problemáticas y construye alternativas pedagógicas en colaboración con los participantes. Esta mirada implica que toda acción educativa debe generar conciencia, promover la autonomía y

favorecer la participación activa de los aprendices.

En este marco, la mediación didáctica con Python materializa la lógica de la *investigación-acción*: cada sesión de clase se convierte en un espacio de exploración y reconstrucción del conocimiento, donde los errores son analizados como datos y las hipótesis de los estudiantes se ponen a prueba mediante la ejecución del código. El proceso de depuración o corrección de errores (*debugging*) adquiere una dimensión pedagógica, pues enseña a revisar, interpretar y mejorar los propios razonamientos.

Por su parte, los criterios de **rigurosidad cualitativa** propuestos por Lincoln y Guba (1985) —*credibilidad, transferibilidad, dependencia y confirmabilidad*— proporcionan el soporte metodológico para validar esta experiencia didáctica. En el contexto de la mediación con Python:

- La **credibilidad** se asegura mediante la triangulación entre teoría, práctica y reflexión docente.
- La **transferibilidad** se demuestra al aplicar los mismos principios a diferentes contextos y niveles de complejidad.
- La **dependencia** se garantiza por la coherencia interna del proceso de enseñanza y por la replicabilidad de los ejercicios en Python.
- La **confirmabilidad** se alcanza al conservar registros, códigos y evidencias que permiten verificar el proceso de aprendizaje.

Así, la mediación didáctica con Python se erige como una práctica coherente con los principios del paradigma sociocrítico: reflexiva, participativa y transformadora. El aula se convierte en un laboratorio de pensamiento donde los estudiantes aprenden a *dialogar con el error*, a comprender la lógica subyacente de sus procesos matemáticos y a construir conocimiento a través de la acción compartida. De esta manera, la tecnología deja de ser un simple recurso instrumental para consolidarse como un **agente epistemológico** que impulsa la autonomía, la crítica y la comprensión profunda en la enseñanza de la matemática.

1.4. Errores algebraicos a trabajar

En correspondencia con el estudio de Juana Inés Pérez Zárate *"Errores algebraicos más comunes que cometen los alumnos de bachillerato"* (s. f.), se identifican los **diez errores algebraicos más representativos** que afectan el aprendizaje y la comprensión simbólica de los estudiantes. Estos errores se asumen en esta investigación como **unidades de análisis didáctico**, al permitir observar la lógica del razonamiento del estudiante y diseñar estrategias de mediación apoyadas en Python y su biblioteca simbólica `SymPy`.

Cada error se relaciona con una línea mínima de intervención computacional, cuyo propósito es posibilitar la verificación, comparación y reconstrucción conceptual de los procedimientos algebraicos. La tabla siguiente presenta una síntesis de los errores seleccionados y la propuesta de mediación.

#	**Error algebraico según Pérez Zárate (s. f.)**	**Línea breve de mediación con Python / SymPy**
1	Cambiar sólo el primer signo al anteponer el negativo a un polinomio.	Construir polinomios simbólicos y comparar $-(p(x))$ con `expand(-(p))` para evidenciar el cambio de todos los signos.
2	Multiplicar monomios sin multiplicar los coeficientes numéricos.	Crear monomios aleatorios y verificar productos con `simplify()` y evaluación numérica.
3	Ignorar la jerarquía de operaciones (sumar antes que multiplicar o potenciar).	Analizar expresiones con `sympify()` y comparar resultados con y sin paréntesis.
4	Dividir el denominador entre cada término del numerador.	Representar $\frac{ax+b}{cx}$ y contrastar con `apart()` o `cancel()` para visualizar la fracción reducida correcta.
5	Sumar fracciones algebraicas sumando numeradores y denominadores directamente.	Calcular el MCM con `lcm()` y validar la equivalencia mediante `equals()`.
6	Confundir las leyes de los exponentes (potencia de potencia y multiplicación).	Usar `expand_power_base()` y `powsimp()` para mostrar la regla correcta.
7	Omitir el signo negativo en raíces cuadradas o aplicarlas sólo a la parte literal.	Explorar dominios reales y complejos con `sqrt()` y supuestos simbólicos (`symbols(..., real=True)`).
8	Desarrollar $(a+b)^2$ como a^2+b^2, omitiendo el término doble.	Utilizar `expand((a+b)**2)` y contrastar con ejemplos numéricos.
9	Elevar $(a+b)^3$ como a^3+b^3, ignorando los términos intermedios.	Aplicar `expand((a+b)**3)` y analizar el patrón de Pascal para comprender la estructura del cubo perfecto.
10	Separar la raíz cuadrada de un polinomio en raíces de cada término.	Contrastar `sqrt(a+b)` vs. `sqrt(a)+sqrt(b)` y discutir las condiciones en que no es válida la operación.

Cuadro 1.1: Errores algebraicos seleccionados con base en Pérez Zárate (s. f.) y su mediación con Python/`SymPy`.

Estos diez errores constituyen el núcleo del trabajo empírico de la secuencia didáctica. Cada uno será modelado y analizado mediante ejercicios prácticos en Python, de modo que el estudiante pueda detectar, simular y corregir sus propios razonamientos algebraicos. La mediación computacional convierte el error en un punto de partida para la comprensión, la metacognición y la transformación del pensamiento matemático.

Capítulo 2

Arquitectura de la secuencia didáctica

- Diseñar una secuencia didáctica que integre el aprendizaje de la matemática con el uso de Python como mediador epistémico y tecnológico.
- Promover la comprensión conceptual y la reflexión crítica sobre los errores algebraicos mediante la experimentación con código.
- Fomentar el desarrollo del pensamiento computacional y la autonomía del estudiante en la resolución de problemas matemáticos.
- Articular la práctica docente con los principios del paradigma sociocrítico y la metodología de investigación-acción.
- Generar evidencias analíticas que sirvan para la validación de la mediación didáctica por parte de expertos.

2.1. Ciclo didáctico: diagnóstico, exploración, formalización y transferencia

La secuencia didáctica se estructura bajo un **ciclo reflexivo de cuatro fases**, que orienta el proceso de enseñanza-aprendizaje desde la identificación del problema hasta la aplicación autónoma del conocimiento. Este modelo, inspirado en la lógica de la *investigación-acción* (Carr & Kemmis, 1988), concibe al aula como un laboratorio pedagógico en el que la práctica se analiza, transforma y valida de manera continua.

1. Diagnóstico

En esta primera fase se identifican los **errores algebraicos recurrentes** y las concepciones previas de los estudiantes. A través de ejercicios iniciales y preguntas abiertas, el docente observa las estrategias utilizadas para resolver problemas algebraicos, determinando las necesidades de mediación. Los resultados de este diagnóstico orientan el diseño de actividades y la selección de los temas que se trabajarán mediante Python.

2. Exploración

En esta etapa, el estudiante **interactúa con el lenguaje Python** para representar operaciones y verificar sus procedimientos. El error se introduce como objeto de análisis: cada resultado inesperado o mensaje del intérprete se convierte en una oportunidad para reflexionar sobre el significado matemático subyacente. El docente guía la interpretación, fomenta la formulación de hipótesis y promueve el diálogo entre pares, construyendo un aprendizaje colaborativo.

3. Formalización

Aquí se sistematizan los conceptos trabajados y se consolidan las relaciones entre los procedimientos algebraicos y sus representaciones simbólicas. La biblioteca `SymPy` se emplea para **expresar formalmente las operaciones**, factorizar, simplificar y resolver ecuaciones, permitiendo visualizar la lógica interna del álgebra. El estudiante asume un rol activo: documenta su código, explica los resultados y genera conexiones entre la teoría matemática y la práctica computacional.

4. Transferencia

Finalmente, el conocimiento adquirido se aplica a nuevos contextos o problemas más complejos. El estudiante desarrolla **proyectos autónomos o ejercicios integradores**, donde debe reconocer, prevenir y corregir errores algebraicos utilizando Python. Esta fase fortalece la metacognición, la autocrítica y la capacidad de extrapolar los aprendizajes a situaciones reales, en coherencia con los principios del paradigma sociocrítico.

El ciclo no es lineal ni cerrado: cada fase puede retroalimentarse a partir de los resultados obtenidos, generando una espiral reflexiva que une teoría, práctica y evaluación. De esta manera, la secuencia didáctica se convierte en un proceso dinámico de construcción y validación del conocimiento, donde Python actúa como mediador del pensamiento matemático y del desarrollo profesional docente.

Capítulo 3

Python básico

3.1. Tipos y operaciones

- Reconocer los tipos de datos fundamentales en Python y su importancia en la representación matemática.
- Comprender la relación entre las operaciones aritméticas y los procesos lógicos del lenguaje de programación.
- Desarrollar la capacidad de interpretar los resultados de una instrucción como evidencia de razonamiento matemático.

Python, al ser un lenguaje de alto nivel, ofrece una sintaxis simple y expresiva que facilita la comprensión de los conceptos matemáticos subyacentes. El trabajo con tipos de datos —`int`, `float`, `str`, `bool`— permite establecer correspondencias entre los objetos matemáticos y sus representaciones computacionales. Esta relación convierte al código en una forma de *mediación simbólica*, donde cada operación ejecutada se traduce en una hipótesis matemática que puede ser verificada inmediatamente.

Tipos de datos primitivos

```
# Identificacion de tipos
a = 5             # Entero
b = 3.14          # Flotante
c = "Python"      # Cadena de texto
d = True          # Booleano

print(type(a), type(b), type(c), type(d))
```

Listing 3.1: Ejemplo 1. Identificacion de tipos de datos

Operaciones aritméticas básicas

```
# Operaciones con enteros y flotantes
x = 10
y = 4
```

```
4
5  suma = x + y
6  resta = x - y
7  producto = x * y
8  division = x / y        # division real
9  potencia = x ** y
10 modulo = x % y
11
12 print("Suma:", suma)
13 print("Resta:", resta)
14 print("Producto:", producto)
15 print("Division:", division)
16 print("Potencia:", potencia)
17 print("Modulo:", modulo)
```

Listing 3.2: Ejemplo 2. Operaciones aritmeticas basicas

Reflexión didáctica

Cada resultado generado por el intérprete de Python puede interpretarse como una **evidencia cognitiva** del procedimiento seguido. Cuando el estudiante obtiene un valor inesperado, puede analizar el error directamente en el código, reconociendo su causa y corrigiéndolo en tiempo real.

Capítulo 4

Estructuras de datos

4.1. Listas: creación, indexación y slicing

- Crear y manipular listas en Python para organizar información.
- Acceder a elementos por índice y aplicar *slicing* para obtener sublistas.
- Utilizar métodos frecuentes de listas en tareas de procesado de datos.

Las listas en Python son estructuras *ordenadas* y *mutables* que permiten almacenar elementos heterogéneos. Su manejo correcto favorece la organización de colecciones y el razonamiento sobre secuencias, índices y subestructuras.

Creación e inspección básica

```
# Creacion de listas
numeros = [10, 20, 30, 40, 50]
mixta   = ["azul", 3, 2.5, True]

# Longitud y tipo
print(len(numeros), type(numeros))

# Acceso por indice (0-based) y desde el final
print(numeros[0], numeros[-1])      # 10 50

# Reemplazo (mutabilidad)
numeros[1] = 25
print(numeros)                      # [10, 25, 30, 40, 50]
```

Listing 4.1: Listado 5. Creacion e inspeccion de listas

Indexación y *slicing*

```
a = [9,8,7,6,5,4,3,2,1,0]

# Slicing: a[inicio:fin:paso]
print(a[1:5])       # indices 1..4 -> [8,7,6,5]
```

```
print(a[:4])        # desde el inicio -> [9,8,7,6]
print(a[6:])        # desde 6 hasta el final -> [3,2,1,0]
print(a[::2])       # paso 2 -> [9,7,5,3,1]
print(a[::-1])      # reverso -> [0,1,2,3,4,5,6,7,8,9]
```

Listing 4.2: Listado 6. Indexacion y slicing

Métodos frecuentes de listas

```
b = [3, 1, 4, 1, 5, 9]
b.append(2)                 # agrega al final
b.insert(1, 7)              # inserta en indice 1
b.extend([8, 3])            # concatena
print(b)

b.remove(1)                 # elimina primera aparicion
ultimo = b.pop()            # saca ultimo elemento
print("ultimo:", ultimo, "lista:", b)

print("indice de 4:", b.index(4))
print("conteo de 3:", b.count(3))

b.sort()                    # orden ascendente
print("ordenada:", b)
b.reverse()                 # invierte in-place
print("reversa:", b)
```

Listing 4.3: Listado 7. Metodos tipicos de listas

Listas anidadas y *slicing* 2D sencillo

```
mat = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Acceso fila/columna
print(mat[0][1])      # 2
print(mat[2][2])      # 9

# Extraer una columna con comprension
col2 = [fila[1] for fila in mat]
print("col2:", col2)  # [2,5,8]

# Submatriz por slicing de filas
sub = mat[0:2]          # filas 0 y 1
print(sub)
```

Listing 4.4: Listado 8. Listas anidadas e indexacion bidimensional

Para evitar errores de codificación con `listings`, se han eliminado tildes y eñes dentro de los bloques de código. En el texto normal (fuera de `lstlisting`) puedes usar acentos sin problema.

Reflexión didáctica

El trabajo con listas permite observar regularidades y patrones en secuencias, favorecer la comprensión de índices positivos/negativos y razonar sobre subestructuras mediante *slicing*. Estas ideas son base para analizar errores algebraicos posteriores y para modelar procesos con `SymPy`.

Capítulo 5

Control de flujo

5.1. For anidados y patrones

- Comprender el funcionamiento del ciclo `for` y su papel en la automatización de procesos repetitivos.
- Implementar estructuras anidadas para construir patrones numéricos y geométricos.
- Relacionar la iteración con los procesos de generalización en el razonamiento matemático.

Los `for` permiten recorrer secuencias y ejecutar acciones repetitivas, controlando de manera explícita el número de iteraciones. Su uso desarrolla en el estudiante la noción de *regularidad y estructura*, esenciales en la comprensión de conceptos como progresiones, tablas y simetrías.

Estructura básica del ciclo `for`

```
# Recorrer una lista
frutas = ["manzana", "pera", "uva"]
for f in frutas:
    print("Fruta:", f)

# Recorrer un rango de numeros
for i in range(1, 6):
    print("Iteracion:", i)
```

Listing 5.1: Listado 9. Uso basico del ciclo for

Iteraciones anidadas

```
# Generar una tabla de multiplicar 1..3
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} x {j} = {i*j}")
```

```
    print("---")
```

Listing 5.2: Listado 10. For anidado para tabla de multiplicar

En un ciclo anidado, el segundo `for` se ejecuta completamente por cada repetición del primero. Esto permite representar estructuras bidimensionales o combinaciones ordenadas de elementos.

Generacion de patrones numericos

```
# Patron triangular de numeros
n = 5
for i in range(1, n+1):
    for j in range(1, i+1):
        print(j, end=" ")
    print()

# Patron inverso
for i in range(n, 0, -1):
    print("*" * i)
```

Listing 5.3: Listado 11. Patrones numericos simples

Este tipo de patrones favorece la comprensión de estructuras recursivas y secuencias crecientes o decrecientes, relacionadas con nociones de combinatoria y aritmética progresiva.

Uso combinado de rangos y condiciones

```
# Imprimir numeros pares e impares hasta 10
for i in range(1, 11):
    if i % 2 == 0:
        print(i, "es par")
    else:
        print(i, "es impar")
```

Listing 5.4: Listado 12. Iteracion con condicionales dentro del ciclo

La combinacion de ciclos y decisiones permite explorar propiedades numéricas (paridad, divisibilidad, residuos) y conectar el pensamiento algorítmico con el matemático.

Reflexion didactica

El control de flujo mediante estructuras `for` y sus variantes anidadas ofrece un espacio para que el estudiante experimente la idea de **invarianza y repeticion con sentido**. A través de la ejecucion del codigo, los patrones emergen visual y numericamente, posibilitando que el error sea detectado de manera inmediata y comprendido como parte del proceso formativo. El docente, desde el enfoque sociocritico, puede usar estos ejercicios para fomentar la explicacion colaborativa y la reconstruccion del razonamiento matematico.

> Para evitar errores de compilacion, los listados eliminan tildes y eñes dentro del codigo, manteniendo el texto explicativo con acentos normales.

Capítulo 6

Contadores y acumuladores

- Diferenciar los conceptos de contador y acumulador dentro de un algoritmo.
- Aplicar estructuras repetitivas para calcular frecuencias y sumas sucesivas.
- Interpretar el valor de las variables como memoria del proceso iterativo.
- Usar el error de programacion como herramienta de aprendizaje reflexivo.

Los contadores y acumuladores son estructuras lógicas que permiten almacenar resultados parciales a lo largo de un ciclo. Su comprension ayuda a visualizar el flujo del razonamiento paso a paso, ya que cada iteracion modifica el estado interno del programa. En terminos didacticos, esta practica fortalece la capacidad de seguir secuencias, detectar errores y justificar resultados intermedios.

Contador basico

```
# Contar numeros pares en una lista
numeros = [1, 2, 3, 4, 5, 6, 7, 8]
contador = 0

for n in numeros:
    if n % 2 == 0:
        contador += 1

print("Cantidad de pares:", contador)
```

Listing 6.1: Listado 31. Contador de elementos pares

Acumulador numerico

```
# Calcular la suma total de una lista
valores = [3, 5, 7, 2]
acumulador = 0

for v in valores:
    acumulador += v
```

```
7
8 print("Suma total:", acumulador)
```

Listing 6.2: Listado 32. Suma acumulativa de elementos

Combinacion de contador y acumulador

```
1  # Calcular promedio de numeros pares
2  datos = [4, 9, 12, 5, 8]
3  suma = 0
4  conteo = 0
5
6  for d in datos:
7      if d % 2 == 0:
8          suma += d
9          conteo += 1
10
11 promedio = suma / conteo
12 print("Promedio de pares:", promedio)
```

Listing 6.3: Listado 33. Promedio de numeros pares

Ejemplo aplicado: conteo de ocurrencias

```
1  # Contar vocales en una palabra
2  palabra = "programacion"
3  contador = 0
4  for letra in palabra:
5      if letra in "aeiou":
6          contador += 1
7  print("Cantidad de vocales:", contador)
```

Listing 6.4: Listado 34. Conteo de ocurrencias de vocales

Reflexion didactica

El uso de contadores y acumuladores permite a los estudiantes representar simbolicamente procesos iterativos y observar la transformacion progresiva de una variable. Cada ejecucion ofrece un registro del pensamiento secuencial y la logica de repeticion. El docente puede aprovechar los errores de sintaxis o de logica (por ejemplo, no inicializar el acumulador o confundir el operador `+=`) como oportunidades para discutir la importancia de la consistencia y del control de flujo.

> Se han eliminado acentos y eñes dentro de los bloques de codigo para evitar errores de codificacion con el paquete `listings`. Los textos explicativos mantienen la ortografia normal en espanol.

Capítulo 7

Ciclo `while`

- Comprender la estructura del ciclo `while` como herramienta de repeticion controlada por condicion.
- Aplicar bucles `while` para resolver problemas que dependen de una prueba logica de salida.
- Desarrollar habilidades de analisis del flujo de control y prevencion de errores logicos (bucles infinitos).
- Relacionar el uso de `while` con procesos matematicos iterativos y experimentacion simbolica.

El ciclo `while` ejecuta un bloque de instrucciones mientras se cumpla una condicion booleana. A diferencia del ciclo `for`, no requiere conocer de antemano el numero de iteraciones, sino que depende del estado logico de una variable. Esta caracteristica lo convierte en una herramienta poderosa para modelar situaciones en las que el resultado se alcanza mediante ensayo, aproximacion o verificacion continua.

Estructura general

```
# Ejemplo basico
contador = 1
while contador <= 5:
    print("Iteracion:", contador)
    contador += 1
print("Fin del ciclo")
```

Listing 7.1: Listado 41. Estructura basica del ciclo while

Uso con entrada del usuario

```
# Repetir hasta recibir dato valido
valor = int(input("Digite un numero positivo: "))
while valor <= 0:
    print("Debe ser positivo.")
    valor = int(input("Digite nuevamente: "))
```

```
print("Numero aceptado:", valor)
```

Listing 7.2: Listado 42. Validacion de entrada con while

Este tipo de control es comun en algoritmos de validacion y en procesos de retroalimentacion inmediata, donde el estudiante puede detectar de forma iterativa su error de ingreso o razonamiento.

Contador inverso y condiciones compuestas

```
# Contador regresivo
n = 5
while n > 0:
    print("Cuenta atras:", n)
    n -= 1
print("Despegue!")

# Ejemplo con condicion compuesta
x = 0
while x < 10 and x != 6:
    print("x vale", x)
    x += 2
```

Listing 7.3: Listado 43. Contador inverso y condiciones logicas

Acumulacion con while

```
# Acumular valores hasta exceder 50
total = 0
numero = 1
while total <= 50:
    total += numero
    numero += 1
print("Total:", total, "Numero final:", numero)
```

Listing 7.4: Listado 44. Suma hasta alcanzar un limite

Uso de banderas de control

```
# Ejemplo con variable de control
continuar = True
contador = 0
while continuar:
    print("Iteracion:", contador)
    contador += 1
    if contador >= 3:
        continuar = False
print("Fin del proceso")
```

Listing 7.5: Listado 45. Uso de bandera para control de flujo

Romper el ciclo: instruccion `break`

```
# Terminar ciclo cuando se cumple una condicion
suma = 0
while True:
    n = int(input("Digite un numero (0 para salir): "))
    if n == 0:
        break
    suma += n
print("Suma total:", suma)
```

Listing 7.6: Listado 46. Uso de break para detener el ciclo

Reflexion didactica

El ciclo `while` permite modelar procesos de razonamiento condicional semejantes a los que se emplean en la resolucion de problemas matematicos abiertos: el estudiante prueba, verifica y ajusta hasta que se cumpla una condicion de salida. En este sentido, cada ejecucion se convierte en un acto de autoevaluacion, donde el error deja de ser penalizado y se transforma en parte natural del aprendizaje. El docente, desde el enfoque sociocritico, puede orientar la discusion sobre el significado de las condiciones logicas, la convergencia de los procesos y la nocion de limite.

Los bloques de codigo se presentan sin tildes ni eñes para evitar conflictos de codificacion con el paquete `listings`. El texto explicativo mantiene acentos y ortografia normal en espanol.

Capítulo 8

Comprensiones y `math`

- Aplicar comprensiones de listas como forma sintetica de crear secuencias en Python.
- Utilizar el modulo `math` para realizar operaciones numericas avanzadas.
- Comprender la relacion entre expresion matematica y expresion algorítmica.
- Promover la exploracion experimental de funciones y formulas dentro de Python.

Las **comprensiones de listas** son una forma compacta de generar listas mediante expresiones dentro de corchetes. Representan un proceso matematico en forma simbólica, al condensar en una sola linea lo que tradicionalmente se escribe con un ciclo. Por su parte, el modulo `math` amplía las capacidades aritméticas del lenguaje, incorporando funciones de raiz, potencia, logaritmo y trigonometría.

Comprensiones simples

```
# Crear una lista de cuadrados
cuadrados = [x**2 for x in range(1, 6)]
print(cuadrados)   # [1, 4, 9, 16, 25]
```

Listing 8.1: Listado 51. Comprension basica de lista

Comprensiones con condicion

```
# Filtrar numeros pares al crear la lista
pares = [x for x in range(1, 11) if x % 2 == 0]
print(pares)   # [2, 4, 6, 8, 10]
```

Listing 8.2: Listado 52. Comprension con condicion

Comprension doble (anidada)

```
# Generar pares ordenados (x, y)
pares_ordenados = [(x, y) for x in range(1, 4) for y in range(1, 4)
    ]
print(pares_ordenados)
```

Listing 8.3: Listado 53. Comprension anidada

Uso del modulo `math`

El modulo `math` contiene funciones y constantes para el calculo numerico. Permite traducir formulas matematicas en operaciones programadas, fortaleciendo la comprension del vinculo entre simbolismo y proceso.

```
import math

print("Pi:", math.pi)
print("Raiz cuadrada de 16:", math.sqrt(16))
print("Seno de 30 grados:", math.sin(math.radians(30)))
print("Logaritmo natural de e:", math.log(math.e))
```

Listing 8.4: Listado 54. Operaciones con el modulo math

Ejemplo aplicado: lista de raices cuadradas

```
import math

numeros = [1, 4, 9, 16, 25]
raices = [math.sqrt(n) for n in numeros]
print(raices)    # [1.0, 2.0, 3.0, 4.0, 5.0]
```

Listing 8.5: Listado 55. Lista de raices cuadradas

Ejemplo: valores de una funcion trigonometrica

```
import math

angulos = [0, 30, 45, 60, 90]
senos = [math.sin(math.radians(a)) for a in angulos]
print("Senos:", senos)
```

Listing 8.6: Listado 56. Evaluacion de una funcion trigonometrica

Combinacion de comprension y validacion logica

```
import math

# Generar lista de numeros cuyo cuadrado sea menor que 50
valores = [x for x in range(1, 20) if x**2 < 50]
```

```
print(valores)
```

Listing 8.7: Listado 57. Filtrado con condicion matematica

Reflexion didactica

El uso de comprensiones y del modulo `math` aproxima al estudiante a una forma de pensamiento simbólico que integra el lenguaje natural de la matemática con el lenguaje formal del código. La posibilidad de verificar de manera inmediata los resultados numéricos estimula la exploración, la comparación de procedimientos y la autonomía cognitiva. En el marco de la mediacion didactica, esta experiencia permite observar cómo el error en la expresión o en la fórmula se convierte en una oportunidad de reconstrucción conceptual, favoreciendo el desarrollo del pensamiento computacional.

Los listados de codigo se presentan sin tildes ni eñes para evitar conflictos con el paquete `listings`. El texto explicativo mantiene la ortografia en espanol con acentos normales.

Capítulo 9

Condicionales

- Comprender la estructura de control condicional en Python como herramienta de decision logica.
- Aplicar instrucciones `if`, `elif` y `else` en la resolucion de problemas numericos y simbolicos.
- Analizar el papel de las condiciones logicas en la verificacion de procedimientos matematicos.
- Promover la interpretacion del error logico como parte del razonamiento computacional.

La estructura condicional constituye un pilar del pensamiento algoritmico. En Python, las decisiones se expresan mediante sentencias `if`, que evalúan una proposicion booleana y ejecutan acciones distintas segun su valor de verdad. Esta logica reproduce la estructura de razonamiento matematico basada en la comprobacion de casos y condiciones.

Estructura basica de decision

```
# Verificar si un numero es positivo
x = int(input("Digite un numero: "))
if x > 0:
    print("El numero es positivo")
```

Listing 9.1: Listado 58. Estructura if basica

Decision con alternativa

```
# Verificar si un numero es par o impar
n = int(input("Digite un numero: "))
if n % 2 == 0:
    print("El numero es par")
else:
    print("El numero es impar")
```

Listing 9.2: Listado 59. Uso de if-else

Decision multiple

```
# Clasificar un numero segun su valor
x = int(input("Digite un numero: "))

if x < 0:
    print("Negativo")
elif x == 0:
    print("Cero")
else:
    print("Positivo")
```

Listing 9.3: Listado 60. Uso de if-elif-else

Condiciones compuestas

```
# Determinar si un numero esta dentro de un rango
n = int(input("Digite un numero entre 1 y 10: "))

if n >= 1 and n <= 10:
    print("Dentro del rango")
else:
    print("Fuera del rango")
```

Listing 9.4: Listado 61. Condiciones con operadores logicos

Condicionales anidados

```
# Evaluar rendimiento segun nota
nota = float(input("Digite la nota: "))

if nota >= 3.0:
    if nota >= 4.5:
        print("Desempeno excelente")
    else:
        print("Aprobado")
else:
    print("Reprobado")
```

Listing 9.5: Listado 62. Estructuras anidadas

Uso de operadores logicos combinados

```
a = True
b = False

print("and:", a and b)
print("or :", a or b)
print("not:", not a)
```

Listing 9.6: Listado 63. Operadores logicos combinados

Aplicacion matematica simple

```
# Funcion definida por tramos
x = float(input("Digite el valor de x: "))

if x < 0:
    y = -x
elif 0 <= x <= 2:
    y = x**2
else:
    y = 2*x + 1

print("f(x) =", y)
```

Listing 9.7: Listado 64. Evaluacion de funcion por tramos

Validacion de divisiones

```
# Evitar division por cero
a = int(input("Numerador: "))
b = int(input("Denominador: "))

if b != 0:
    print("Resultado:", a / b)
else:
    print("Error: no se puede dividir entre cero")
```

Listing 9.8: Listado 65. Comprobacion antes de dividir

Deteccion de raices reales

```
# Verificar si una ecuacion cuadratica tiene raices reales
a = 1
b = 2
c = 5

discriminante = b**2 - 4*a*c

if discriminante > 0:
    print("Dos raices reales")
elif discriminante == 0:
    print("Una raiz real")
else:
    print("Raices imaginarias")
```

Listing 9.9: Listado 66. Discriminante de una ecuacion cuadratica

Ejemplo con multiples condiciones

```
# Clasificar por grupo de edad
edad = int(input("Digite la edad: "))

if edad < 12:
    print("Nino")
elif edad < 18:
    print("Adolescente")
elif edad < 60:
    print("Adulto")
else:
    print("Adulto mayor")
```

Listing 9.10: Listado 67. Clasificacion segun edad

Condicional con expresiones logicas complejas

```
x = 7
y = 3

if (x > 5 and y < 5) or (x + y == 10):
    print("Condicion verdadera")
else:
    print("Condicion falsa")
```

Listing 9.11: Listado 68. Condicion combinada compleja

Uso del operador ternario

```
# Determinar el mayor de dos numeros
a = 8
b = 12

mayor = a if a > b else b
print("Mayor:", mayor)
```

Listing 9.12: Listado 69. Expresion condicional abreviada

Comparacion encadenada

```
x = 5
if 0 < x < 10:
    print("x esta entre 0 y 10")
```

Listing 9.13: Listado 70. Comparacion encadenada

Reflexion didactica

El manejo de estructuras condicionales fortalece la comprension del razonamiento matematico basado en casos y pruebas logicas. Cada bloque de decision permite visualizar la relacion entre proposiciones, condiciones y resultados, lo que refuerza la nocion de coherencia argumentativa. Desde la mediacion didactica con Python, el estudiante aprende a reconocer el error logico no como una falla, sino como una evidencia del proceso de razonamiento en desarrollo. La interpretacion de mensajes de error y la validacion de condiciones promueven la autoevaluacion y la construccion de autonomia cognitiva.

Los bloques de codigo se presentan sin tildes ni eñes para evitar conflictos de codificacion. El texto explicativo mantiene la ortografia normal en espanol.

Capítulo 10

Funciones

- Comprender el concepto de funcion en Python como estructura modular del pensamiento algoritmico.
- Definir funciones con y sin parametros, diferenciando el alcance de las variables.
- Aplicar funciones para resolver problemas numericos y simbolicos con retorno de resultados.
- Analizar el valor didactico de la modularidad y la reutilizacion de codigo en el aprendizaje matematico.

Las **funciones** representan uno de los conceptos mas importantes de la programacion estructurada. Permiten agrupar instrucciones bajo un mismo nombre y ejecutarlas de manera repetida sin reescribir el codigo. En terminos didacticos, constituyen una forma de **abstraccion** y **modularizacion** del pensamiento, similar al proceso matematico de definir operaciones o procedimientos generales.

Definicion basica de funcion

```
# Definicion y llamado de una funcion simple
def saludo():
    print("Bienvenido al aprendizaje con Python")

saludo()
```

Listing 10.1: Listado 71. Funcion sin parametros

Funcion con parametros

```
# Funcion con parametros numericos
def suma(a, b):
    resultado = a + b
    return resultado

print("Suma:", suma(5, 3))
```

Listing 10.2: Listado 72. Funcion con parametros

Funcion con retorno y documentacion

```
def area_triangulo(base, altura):
    """
    Calcula el area de un triangulo.
    Formula: (base * altura) / 2
    """
    return (base * altura) / 2

print(area_triangulo(4, 5))
```

Listing 10.3: Listado 73. Funcion documentada con retorno

Parametros por defecto

```
def potencia(base, exponente=2):
    return base ** exponente

print(potencia(3))      # usa exponente=2
print(potencia(2, 5))   # exponente personalizado
```

Listing 10.4: Listado 74. Parametros con valores por defecto

Funciones con multiples retornos

```
def operaciones(a, b):
    suma = a + b
    resta = a - b
    producto = a * b
    return suma, resta, producto

x, y, z = operaciones(6, 3)
print("Resultados:", x, y, z)
```

Listing 10.5: Listado 75. Multiples valores de retorno

Funciones lambda o anonimas

```
# Funcion anonima para duplicar un valor
doble = lambda x: x * 2
print(doble(7))
```

Listing 10.6: Listado 76. Expresion lambda

Uso de funciones dentro de otras funciones

```python
def cuadrado(x):
    return x * x

def suma_de_cuadrados(a, b):
    return cuadrado(a) + cuadrado(b)

print(suma_de_cuadrados(3, 4))
```

Listing 10.7: Listado 77. Funcion dentro de otra

Recursion: funcion que se llama a si misma

```python
# Calcular factorial de un numero
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)

print("Factorial de 5:", factorial(5))
```

Listing 10.8: Listado 78. Ejemplo de recursion simple

Funcion recursiva con visualizacion del proceso

```python
def factorial_trazado(n):
    print("Calculando factorial(", n, ")")
    if n == 1:
        return 1
    else:
        resultado = n * factorial_trazado(n-1)
        print("factorial(", n, ") =", resultado)
        return resultado

factorial_trazado(4)
```

Listing 10.9: Listado 79. Seguimiento del proceso recursivo

Funciones de orden superior

```python
# Aplicar una funcion a cada elemento
numeros = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)
```

Listing 10.10: Listado 80. Uso de map con funcion lambda

Uso de filter y reduce

```
from functools import reduce

numeros = [2, 5, 7, 8, 9, 10]
pares = list(filter(lambda x: x % 2 == 0, numeros))
suma = reduce(lambda a, b: a + b, numeros)

print("Pares:", pares)
print("Suma total:", suma)
```

Listing 10.11: Listado 81. Filter y reduce

Funciones como objetos

```
def cuadrado(x):
    return x * x

def aplicar_funcion(func, lista):
    return [func(x) for x in lista]

resultado = aplicar_funcion(cuadrado, [1, 2, 3, 4])
print(resultado)
```

Listing 10.12: Listado 82. Asignar y pasar funciones como argumentos

Funciones con manejo de excepciones

```
def dividir(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: division entre cero")
        return None

print(dividir(10, 0))
```

Listing 10.13: Listado 83. Manejo de errores dentro de una funcion

Funciones para calculos matematicos

```
def media(lista):
    return sum(lista) / len(lista)

valores = [4, 6, 8, 10]
print("Media:", media(valores))
```

Listing 10.14: Listado 84. Funcion para calcular media aritmetica

Funcion que retorna otra funcion

```
def generar_potencia(n):
    def elevar(x):
        return x ** n
    return elevar

cuadrado = generar_potencia(2)
print(cuadrado(5))
```

Listing 10.15: Listado 85. Funcion que retorna otra funcion

Funciones lambda combinadas

```
doble = lambda x: x * 2
cuadrado = lambda x: x ** 2
combinada = lambda x: cuadrado(doble(x))
print(combinada(3))
```

Listing 10.16: Listado 86. Composicion con lambdas

Funciones con valores simbolicos

```
from sympy import symbols, sin, diff

x = symbols('x')
f = sin(x)
df = diff(f, x)
print("f(x) =", f)
print("f'(x) =", df)
```

Listing 10.17: Listado 87. Funcion simbolica con SymPy

Reflexion didactica

El estudio de las funciones en Python promueve la comprension de la abstraccion matematica aplicada al contexto digital. Definir una funcion implica reconocer relaciones de dependencia entre variables, generalizar procedimientos y transferir conocimiento a nuevas situaciones. Desde la mediacion didactica, el docente puede utilizar la depuracion de errores de sintaxis, de parametros o de logica como oportunidades para desarrollar pensamiento metacognitivo, autonomia y coherencia en la construccion del conocimiento.

> Los bloques de codigo se presentan sin tildes ni eñes para evitar errores de codificacion en `listings`. El texto explicativo mantiene la ortografia normal en espanol.

Capítulo 11

Introduccion a SymPy

- Introducir el uso de la libreria SymPy para el calculo simbolico en Python.
- Diferenciar entre el tratamiento numerico y el simbolico de las expresiones matematicas.
- Aplicar operaciones basicas de algebra, factorizacion, sustitucion y evaluacion simbolica.
- Comprender el valor didactico del trabajo algebraico asistido por computadora.

SymPy es una libreria de Python dedicada al calculo simbolico, lo que significa que permite trabajar con expresiones matematicas como objetos manipulables y no solo como valores numericos. Esto la convierte en una herramienta fundamental para la mediacion didactica del pensamiento algebraico, ya que posibilita analizar, expandir, simplificar y resolver expresiones de manera formal.

Importacion y definicion de simbolos

```
from sympy import symbols

x, y = symbols('x y')
expr = 3*x + 2*y
print(expr)
```

Listing 11.1: Listado 88. Importacion y creacion de simbolos

Operaciones basicas

```
from sympy import symbols

x, y = symbols('x y')
expr1 = x + y
expr2 = x * y
expr3 = x**2 + 2*x + 1
print(expr1, expr2, expr3)
```

Listing 11.2: Listado 89. Operaciones simbolicas basicas

Sustitucion de valores

```
from sympy import symbols

x = symbols('x')
expr = x**2 + 3*x + 2
resultado = expr.subs(x, 4)
print("Resultado:", resultado)
```

Listing 11.3: Listado 90. Sustitucion de valores numericos

Evaluacion numerica

```
from sympy import symbols

x = symbols('x')
expr = (x**2 + 1) / 3
valor = expr.subs(x, 5).evalf()
print("Valor numerico:", valor)
```

Listing 11.4: Listado 91. Evaluacion numerica con evalf

Expansion y factorizacion

```
from sympy import symbols, expand, factor

x = symbols('x')
expr = (x + 2)*(x + 3)
print("Expandido:", expand(expr))
print("Factorizado:", factor(expand(expr)))
```

Listing 11.5: Listado 92. Expandir y factorizar expresiones

Simplificacion algebraica

```
from sympy import symbols, simplify

x = symbols('x')
expr = (x**2 - 4)/(x - 2)
print("Simplificado:", simplify(expr))
```

Listing 11.6: Listado 93. Simplificacion algebraica

Expresiones racionales y fracciones

```
from sympy import symbols, Rational

x = symbols('x')
expr = Rational(3, 4)*x + Rational(2, 5)
print(expr)
```

Listing 11.7: Listado 94. Operaciones con fracciones algebraicas

Derivacion simbolica

```
from sympy import symbols, diff

x = symbols('x')
f = x**3 + 2*x**2 + 3*x + 4
df = diff(f, x)
print("f'(x) =", df)
```

Listing 11.8: Listado 95. Derivada de una expresion simbolica

Integracion simbolica

```
from sympy import symbols, integrate

x = symbols('x')
f = x**2 + 3*x + 2
int_f = integrate(f, x)
print("Integral indefinida:", int_f)
```

Listing 11.9: Listado 96. Integracion de una funcion

Evaluacion de integrales definidas

```
from sympy import symbols, integrate

x = symbols('x')
f = x**2
res = integrate(f, (x, 0, 3))
print("Integral definida:", res)
```

Listing 11.10: Listado 97. Integral definida en un intervalo

Límites

```
from sympy import symbols, limit

x = symbols('x')
```

```
expr = (x**2 - 4)/(x - 2)
limite = limit(expr, x, 2)
print("Limite:", limite)
```

Listing 11.11: Listado 98. Calculo de limites

Ecuaciones y soluciones simples

```
from sympy import symbols, Eq, solve

x = symbols('x')
ec = Eq(3*x + 2, 11)
sol = solve(ec)
print("Solucion:", sol)
```

Listing 11.12: Listado 99. Resolver ecuaciones lineales

Ecuaciones cuadraticas

```
from sympy import symbols, Eq, solve

x = symbols('x')
ec = Eq(x**2 - 5*x + 6, 0)
sol = solve(ec)
print("Raices:", sol)
```

Listing 11.13: Listado 100. Resolver ecuacion cuadratica

Ecuaciones con multiples variables

```
from sympy import symbols, Eq, solve

x, y = symbols('x y')
ec1 = Eq(2*x + y, 10)
ec2 = Eq(x - y, 2)
sol = solve((ec1, ec2), (x, y))
print(sol)
```

Listing 11.14: Listado 101. Ecuacion con dos variables

Uso de funciones simbolicas

```
from sympy import symbols, Function, diff

x = symbols('x')
f = Function('f')(x)
expr = x**2 + 2*x + 1
print("Derivada:", diff(expr, x))
```

Listing 11.15: Listado 102. Definir y derivar una funcion simbolica

Evaluacion numerica de funciones simbolicas

```
from sympy import symbols, sin

x = symbols('x')
f = sin(x)
valor = f.subs(x, 3.14/2).evalf()
print("Valor aproximado:", valor)
```

Listing 11.16: Listado 103. Evaluacion simbolica numerica

Uso de expresiones compuestas

```
from sympy import symbols, sin, cos

x = symbols('x')
expr = sin(x)**2 + cos(x)**2
print("Resultado:", expr.simplify())
```

Listing 11.17: Listado 104. Composicion de expresiones simbolicas

Expresion simbolica en varias etapas

```
from sympy import symbols, expand, factor, simplify

x = symbols('x')
expr = (x + 1)**3
print("Expandido:", expand(expr))
print("Factorizado:", factor(expand(expr)))
print("Simplificado:", simplify(expand(expr)))
```

Listing 11.18: Listado 105. Manipulacion progresiva de una expresion

Visualizacion basica de una expresion

```
from sympy import symbols, plot

x = symbols('x')
expr = x**2 - 4
plot(expr)
```

Listing 11.19: Listado 106. Visualizacion de expresion simbolica

Evaluacion combinada

```
from sympy import symbols, diff, integrate, sin

x = symbols('x')
```

```
f = sin(x)
df = diff(f, x)
intf = integrate(f, x)
print("f'(x) =", df)
print("Integral(f) =", intf)
```

Listing 11.20: Listado 107. Combinacion de operaciones simbolicas

Reflexion didactica

El trabajo con SymPy permite que el estudiante explore la matematica desde un enfoque constructivo y reflexivo. Cada operacion ejecutada se convierte en un acto de comprobacion simbolica que refuerza la comprension de las reglas algebraicas y diferenciales. Desde la mediacion didactica, la libreria ofrece un entorno de experimentacion controlada, donde los errores de logica o de notacion son oportunidades para analizar los fundamentos del lenguaje matematico y del razonamiento formal. En coherencia con el paradigma sociocritico, el docente se posiciona como orientador del proceso reflexivo, promoviendo la autonomia y la capacidad de interpretar los resultados de manera critica.

> Los listados de codigo se presentan sin tildes ni eñes para garantizar compatibilidad total con pdfLaTeX. El texto explicativo mantiene la ortografia normal en espanol.

Capítulo 12

Simplificacion y desarrollo en SymPy

- Aplicar tecnicas de simplificacion y desarrollo algebraico usando la libreria SymPy.
- Analizar el proceso de factorizacion, expansion y sustitucion como herramientas de comprension simbolica.
- Explorar los errores comunes de manipulacion algebraica mediante comprobacion computacional.
- Fortalecer la comprension conceptual de la equivalencia algebraica y el razonamiento simbolico.

En el ambito de la matematica escolar, las operaciones de simplificacion y desarrollo de expresiones suelen ser fuente de errores recurrentes. Con SymPy, estos procesos pueden visualizarse paso a paso, permitiendo comprobar la validez de las transformaciones y detectar inconsistencias en los procedimientos. El lenguaje simbolico computacional actua como mediador entre la regla formal y la intuicion del estudiante, promoviendo la verificacion y la autocorreccion.

Simplificacion algebraica directa

```
from sympy import symbols, simplify

x = symbols('x')
expr = (x**2 - 4)/(x - 2)
print("Simplificado:", simplify(expr))
```

Listing 12.1: Listado 113. Simplificacion directa de expresiones

Expansion y factorizacion

```
from sympy import symbols, expand, factor

x = symbols('x')
expr = (x + 2)*(x + 3)
print("Expandido:", expand(expr))
```

```
print("Factorizado:", factor(expand(expr)))
```

Listing 12.2: Listado 114. Expansion y factorizacion de polinomios

Desarrollo de binomios

```
from sympy import symbols, expand

x, y = symbols('x y')
print("Cuadrado:", expand((x + y)**2))
print("Cubo:", expand((x + y)**3))
```

Listing 12.3: Listado 115. Desarrollo de binomios cuadrados y cubicos

Simplificacion de fracciones algebraicas

```
from sympy import symbols, simplify

x = symbols('x')
expr = (x**2 - 1)/(x - 1)
print("Simplificado:", simplify(expr))
```

Listing 12.4: Listado 116. Fracciones algebraicas equivalentes

Sustitucion de expresiones

```
from sympy import symbols

x, y = symbols('x y')
expr = x**2 + 2*x + 1
expr_sub = expr.subs(x, y + 1)
print("Nueva expresion:", expr_sub)
```

Listing 12.5: Listado 117. Sustitucion de variables en expresiones

Comprobacion de equivalencia algebraica

```
from sympy import symbols, simplify

x = symbols('x')
expr1 = (x + 1)**2
expr2 = x**2 + 2*x + 1
print("Equivalentes:", simplify(expr1 - expr2) == 0)
```

Listing 12.6: Listado 118. Verificacion de igualdad algebraica

Expresiones con raices y potencias

```
from sympy import symbols, sqrt, simplify

x = symbols('x')
expr = sqrt(x**2)
print("Simplificado:", simplify(expr))
```

Listing 12.7: Listado 119. Simplificacion de radicales y potencias

Desarrollo de productos notables

```
from sympy import symbols, expand

x, y = symbols('x y')
print("Diferencia de cuadrados:", expand((x + y)*(x - y)))
print("Trinomio cuadrado perfecto:", expand((x + y)**2))
```

Listing 12.8: Listado 120. Identidades de productos notables

Composicion y combinacion de operaciones

```
from sympy import symbols, expand, factor, simplify

x = symbols('x')
expr = (x + 1)**4 - (x - 1)**4
simplificado = simplify(expr)
print("Simplificado:", simplificado)
print("Factorizado:", factor(simplificado))
```

Listing 12.9: Listado 121. Composicion de simplificacion y factorizacion

Reflexion didactica

El uso de SymPy para simplificar y desarrollar expresiones algebraicas permite que el estudiante reconozca la logica detras de las reglas formales del algebra. Cada operacion ejecutada constituye una comprobacion visual del principio de equivalencia y una oportunidad para analizar los errores comunes que suelen cometerse en el papel. Desde la mediacion didactica, el lenguaje simbolico computacional promueve la comprension profunda del proceso de transformacion algebraica, integrando el razonamiento matematico con el pensamiento computacional. En coherencia con el paradigma sociocritico, se fomenta la autonomia, la argumentacion y la conciencia del propio proceso de aprendizaje.

> Los bloques de codigo se presentan sin tildes ni eñes para asegurar compatibilidad con `listings`. El texto explicativo mantiene la ortografia en espanol con acentos normales.

Capítulo 13

Ecuaciones y sistemas con SymPy

- Resolver ecuaciones algebraicas y sistemas de ecuaciones utilizando SymPy.
- Diferenciar entre la solucion simbolica y la aproximacion numerica.
- Comprender el significado didactico de la resolucion computacional como verificacion del razonamiento algebraico.
- Promover el uso del error de ejecucion o de planteamiento como punto de partida para la reflexion matematica.

El estudio de las ecuaciones constituye el eje articulador del razonamiento algebraico. SymPy permite representar ecuaciones de forma simbolica y resolverlas mediante metodos exactos, ofreciendo al estudiante la posibilidad de comprobar sus resultados y reflexionar sobre el proceso seguido. Desde una perspectiva de mediacion didactica, esta herramienta transforma el aula en un espacio de investigacion, donde la validacion y la correccion se integran al aprendizaje.

Ecuacion lineal simple

```
from sympy import symbols, Eq, solve

x = symbols('x')
ec = Eq(3*x + 2, 11)
sol = solve(ec)
print("Solucion:", sol)
```

Listing 13.1: Listado 123. Ecuacion lineal basica

Ecuacion cuadratica

```
from sympy import symbols, Eq, solve

x = symbols('x')
ec = Eq(x**2 - 5*x + 6, 0)
sol = solve(ec)
print("Raices:", sol)
```

Listing 13.2: Listado 124. Ecuacion cuadratica

Ecuacion con parametros

```
from sympy import symbols, Eq, solve

x, a = symbols('x a')
ec = Eq(a*x + 2, 10)
sol = solve(ec, x)
print("Solucion en funcion de a:", sol)
```

Listing 13.3: Listado 125. Ecuacion con parametro simbolico

Ecuaciones racionales

```
from sympy import symbols, Eq, solve

x = symbols('x')
ec = Eq((x + 1)/(x - 2), 3)
sol = solve(ec)
print("Solucion:", sol)
```

Listing 13.4: Listado 126. Ecuacion racional simple

Ecuaciones con radicales

```
from sympy import symbols, Eq, sqrt, solve

x = symbols('x')
ec = Eq(sqrt(x + 1), x - 1)
sol = solve(ec)
print("Soluciones:", sol)
```

Listing 13.5: Listado 127. Ecuacion con raiz cuadrada

Ecuaciones exponenciales y logaritmicas

```
from sympy import symbols, Eq, exp, log, solve

x = symbols('x')
ec1 = Eq(exp(x), 5)
ec2 = Eq(log(x), 3)
print("Solucion exp:", solve(ec1))
print("Solucion log:", solve(ec2))
```

Listing 13.6: Listado 128. Ecuacion exponencial y logaritmica

Sistema de ecuaciones lineales

```
from sympy import symbols, Eq, solve

x, y = symbols('x y')
ec1 = Eq(2*x + y, 10)
ec2 = Eq(x - y, 2)
sol = solve((ec1, ec2), (x, y))
print("Solucion:", sol)
```

Listing 13.7: Listado 129. Sistema lineal de dos ecuaciones

Sistema con tres variables

```
from sympy import symbols, Eq, solve

x, y, z = symbols('x y z')
ec1 = Eq(x + y + z, 6)
ec2 = Eq(2*x - y + z, 3)
ec3 = Eq(x - 2*y + 3*z, 7)
sol = solve((ec1, ec2, ec3), (x, y, z))
print(sol)
```

Listing 13.8: Listado 130. Sistema con tres variables

Sistemas no lineales

```
from sympy import symbols, Eq, solve

x, y = symbols('x y')
ec1 = Eq(x**2 + y**2, 25)
ec2 = Eq(x - y, 3)
sol = solve((ec1, ec2), (x, y))
print(sol)
```

Listing 13.9: Listado 131. Sistema no lineal simbolico

Verificacion y evaluacion numerica

```
from sympy import symbols, Eq, solve

x = symbols('x')
ec = Eq(x**2 - 2, 0)
sol = solve(ec)
print("Soluciones simbolicas:", sol)
print("Valores numericos:", [s.evalf() for s in sol])
```

Listing 13.10: Listado 132. Verificacion de soluciones y evaluacion numerica

Reflexion didactica

El uso de SymPy para la resolucion de ecuaciones y sistemas favorece la comprension del proceso algebraico como un conjunto de transformaciones verificables. Cada solucion simbolica representa una evidencia del razonamiento deductivo, mientras que la evaluacion numerica permite contrastar los resultados y reforzar la nocion de exactitud frente a aproximacion. En terminos didacticos, esta experiencia impulsa al estudiante a revisar su planteamiento, interpretar errores de logica y validar sus resultados de manera autonoma. Desde el paradigma sociocritico, el aula se transforma en un espacio de investigacion compartida, donde la tecnologia actua como mediador del pensamiento matematico y de la construccion colectiva del conocimiento.

Los listados de codigo se presentan sin tildes ni eñes para asegurar compatibilidad con `listings`. El texto explicativo mantiene la ortografia normal en espanol.

Capítulo 14

Banco de ejercicios por error

Uso del banco

El presente banco de ejercicios se construyo a partir de los diez errores algebraicos identificados en la investigacion de Perez Zarate (s. f.), que sirven como insumo de diagnostico y mediacion didactica. Cada seccion presenta un error especifico con ejemplos de diagnostico, actividades de verificacion simbolica y numerica en Python mediante la libreria `SymPy`. El docente puede utilizar este material en dos momentos:

- **Antes de la intervencion** (diagnostico): para identificar la presencia del error conceptual o procedimental.
- **Durante la mediacion** (formalizacion y transferencia): para guiar la discusion, la comprobacion computacional y la correccion del razonamiento algebraico.

14.1. Error 1: cambio de signos en polinomios

Diagnostico

Los estudiantes suelen cambiar solo el signo del primer termino al anteponer un signo negativo a un polinomio.

Mediacion con Python/SymPy

```
from sympy import symbols, expand
x = symbols('x')
expr = -(x**2 - 3*x + 2)
print("Expresion correcta:", expand(expr))
```

Listing 14.1: Correccion de signos en un polinomio

14.2. Error 2: multiplicacion de monomios

Diagnostico

Error frecuente: multiplicar las partes literales y conservar el mismo coeficiente numerico.

Mediacion

```python
from sympy import symbols
x, y = symbols('x y')
expr = (3*x)*(2*y)
print(expr)
```

Listing 14.2: Multiplicacion correcta de monomios

14.3. Error 3: jerarquia de operaciones

Diagnostico

Los estudiantes resuelven operaciones sin respetar el orden jerarquico.

Mediacion

```python
print(3 + 2 * 5)     # primero multiplicacion
print((3 + 2) * 5)   # uso correcto de parentesis
```

Listing 14.3: Demostracion de la jerarquia de operaciones

14.4. Error 4: fracciones con polinomios

Diagnostico

Se dividen los terminos del denominador individualmente, sin considerar la fraccion completa.

Mediacion con Python/SymPy

```python
from sympy import symbols, cancel, apart
x = symbols('x')
expr = (x**2 + 3*x + 2)/(x + 1)
print("Forma cancelada:", cancel(expr))
print("Forma separada:", apart(expr))
```

Listing 14.4: Uso de cancel y apart

14.5. Error 5: suma o resta de fracciones algebraicas

Diagnostico

Se suman numeradores y denominadores directamente, sin calcular el minimo comun multiplo.

Mediacion

```
from sympy import symbols, simplify
x = symbols('x')
expr = 1/(x+1) + 1/(x+2)
print("Resultado correcto:", simplify(expr))
```

Listing 14.5: Suma correcta de fracciones algebraicas

14.6. Error 6: leyes de exponentes

Diagnostico

Confusion entre potencia de potencia y multiplicacion de potencias.

Mediacion

```
from sympy import symbols, expand_power_base, powsimp
x, a, b = symbols('x a b')
expr = (x**a)**b
print("Expansion:", expand_power_base(expr))
print("Simplificacion:", powsimp(expr))
```

Listing 14.6: Verificacion de leyes de los exponentes

14.7. Error 7: raiz cuadrada y signo

Diagnostico

Omitir el signo negativo al aplicar raiz cuadrada en un numero o expresion.

Mediacion

```
from sympy import symbols, sqrt
x = symbols('x', real=True)
expr = sqrt(x**2)
print(expr)
```

Listing 14.7: Verificacion del signo en la raiz cuadrada

14.8. Error 8: $(a+b)^2 \neq a^2 + b^2$

Diagnostico

Aplicacion incorrecta del binomio cuadrado perfecto, omitiendo el termino $2ab$.

Mediacion

```
from sympy import symbols, expand
a, b = symbols('a b')
print("Correcto:", expand((a+b)**2))
```

Listing 14.8: Comparacion simbolica del desarrollo correcto

14.9. Error 9: $(a+b)^3 \neq a^3 + b^3$

Diagnostico

Los estudiantes elevan cada termino al cubo, ignorando los terminos intermedios.

Mediacion

```
from sympy import symbols, expand
a, b = symbols('a b')
print(expand((a+b)**3))
```

Listing 14.9: Expansion de un binomio al cubo

14.10. Error 10: $\sqrt{a^2 + b^2}$

Diagnostico

Generalizacion incorrecta de la propiedad $\sqrt{a^2} = a$, aplicada a sumas internas.

Mediacion

```
from sympy import symbols, sqrt
a, b = symbols('a b', positive=True)
expr1 = sqrt(a**2 + b**2)
expr2 = a + b
print("Simbolico:", expr1.equals(expr2))
print("Evaluacion numerica:", expr1.subs({a:3,b:4}).evalf())
```

Listing 14.10: Contraejemplo numerico de raiz de suma

Capítulo 15

Resumen ejecutivo para jueces expertos

Propósito del documento

El presente resumen ejecutivo tiene como finalidad ofrecer a los jueces expertos una visión clara, sintética y rigurosa del proceso de validación del material didáctico y de los instrumentos de análisis diseñados en el marco de la investigación titulada:

"Construcción de una Aproximación Teórica para la Mediación Didáctica de la Matemática en Contextos de Formación Tecnológica: El Papel de Python como Recurso Pedagógico".

Este documento permite valorar la coherencia interna, la pertinencia científica y la validez didáctica del proyecto, conforme a los criterios de rigurosidad cualitativa establecidos por **Lincoln y Guba (1985)**: credibilidad, transferibilidad, dependencia y confirmabilidad:contentReference[oaicite:0]index=0.

Criterios de Rigurosidad Cualitativa

1. Credibilidad (Valor de Verdad)

Evalúa el grado de confianza en los hallazgos del estudio y su correspondencia con la realidad percibida por los participantes. Se garantiza mediante la triangulación de fuentes (documentos, observaciones y ejercicios), la revisión por pares y la validación empírica de los instrumentos Python (ejecución y depuración de código).

Aspectos a valorar:

- Coherencia entre los objetivos, categorías teóricas y resultados observables.
- Claridad conceptual en la descripción de los errores algebraicos.
- Correspondencia entre los códigos en Python y las interpretaciones cualitativas.

Calificación (1–5): _____ **Observaciones:** _____

2. Transferibilidad (Aplicabilidad)

Se refiere a la posibilidad de que los resultados sean aplicables a contextos similares. La investigación detalla descripciones densas del escenario (UNIPUTUMAYO, estudiantes de tecnología en software) y la replicabilidad de los instrumentos. **Aspectos a valorar:**

- Claridad en la descripción del contexto institucional y de la población.
- Adaptabilidad del modelo de mediación didáctica a otros programas tecnológicos.
- Coherencia entre la teoría y la práctica observable en la secuencia didáctica.

Calificación (1–5): _____ **Observaciones:** _____

3. Dependencia (Consistencia o Auditabilidad)

Evalúa la estabilidad del proceso y la trazabilidad de las decisiones metodológicas. Se demuestra mediante la documentación de cada fase (diagnóstico, exploración, formalización y transferencia):contentReference[oaicite:1]index=1. **Aspectos a valorar:**

- Claridad del procedimiento metodológico.
- Documentación del proceso de validación y corrección de errores.
- Existencia de una pista de auditoría (archivos, códigos, registros de campo).

Calificación (1–5): _____ **Observaciones:** _____

4. Confirmabilidad (Neutralidad)

Evalúa la objetividad del estudio y la independencia de los resultados frente a los sesgos del investigador. Se fundamenta en la evidencia empírica (salidas de código, entrevistas, triangulación) y en la reflexividad del autor. **Aspectos a valorar:**

- Claridad de los datos que sustentan los hallazgos.
- Transparencia interpretativa en los análisis.
- Neutralidad y equilibrio en la presentación de resultados.

Calificación (1–5): _____ **Observaciones:** _____

Estructura de Evaluación Integral

A continuación se presentan los apartados del documento a evaluar y la ponderación sugerida para la calificación final.

Sección del Documento	**Calificación (1–5)**	**Observaciones del Experto**
Fundamentación teórica y paradigma sociocrítico	___	
Sistema de errores algebraicos y mediación Python	___	
Secuencia didáctica y ciclo reflexivo	___	
Instrumentos de programación y validación técnica	___	
Rigor metodológico y coherencia con Guba y Lincoln	___	
Conclusiones y aplicabilidad	___	
Promedio global de validación		

Síntesis cualitativa de la evaluación

Espacio para el experto:

> *"La mediación didáctica con Python propuesta en este trabajo constituye un modelo innovador que articula pensamiento matemático y pensamiento computacional, con un enfoque coherente y transformador."*

Conclusión general del juez experto: _____

Firma y datos del evaluador

- Nombre y título: _____
- Institución: _____
- Área de especialidad: _____
- Fecha de revisión: _____
- Firma: _____

—

Referencias

- Carr, W., & Kemmis, S. (1988). *Teoría crítica de la enseñanza: La investigación-acción en la formación del profesorado*. Martínez Roca.
- Gandica de Roa, E. (2023). *Criterios de rigurosidad científica en la investigación cualitativa*. Universidad de la Amazonia.
- Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic Inquiry*. Sage Publications.
- Lincoln, Y. S., & Guba, E. G. (1988). *The Paradigm Dialog*. Sage Publications.
- Brousseau, G. (1986). *Fondements et méthodes de la didactique des mathématiques*. *Recherches en Didactique des Mathématiques*, 7(2), 33–115.
- Meurer, A., Smith, C. P., Paprocki, M., et al. (2017). *SymPy: Symbolic mathematics in Python*. *PeerJ Computer Science*, 3(e103). https://doi.org/10.7717/peerj-cs.103

Instrumento de validación por juicio de expertos

Propósito del anexo

En este anexo se presenta los materiales utilizados para la validación por juicio de expertos de los instrumentos de mediación didáctica basados en Python.

El objetivo es comprobar la pertinencia, claridad y coherencia teórico-práctica de los ejercicios y programas desarrollados en el marco de la investigación titulada: *"Construcción de una Aproximación Teórica para la Mediación Didáctica de la Matemática en Contextos de Formación Tecnológica: El Papel de Python como Recurso Pedagógico"*.

Criterios de rigurosidad cualitativa (Lincoln & Guba, 1985)

Criterio	Aspectos a evaluar	CalificaciónObservaciones del experto (1–5)
Credibilidad	Claridad conceptual, coherencia teórica y correspondencia entre los códigos Python y las interpretaciones cualitativas.	
Transferibilidad	Aplicabilidad del modelo de mediación a otros contextos educativos.	
Dependencia	Consistencia metodológica y trazabilidad del proceso de validación.	
Confirmabilidad	Neutralidad interpretativa y evidencia empírica de los resultados.	

Instrucciones de aplicación

Los expertos deben evaluar cada bloque de ejercicios Python considerando:

- Claridad de la instrucción y propósito didáctico.
- Correspondencia con los errores algebraicos comunes observados.
- Pertinencia tecnológica y facilidad de ejecución.
- Valor pedagógico en la mediación del aprendizaje matemático.

Bloque A - Logica y Control (Python basico)

```python
# BLOQUE A - LOGICA Y CONTROL (Python basico)
# Objetivo: condicionales, bucles e IO para razonar paso a paso.

SEPARADOR = "-" * 48

# 1) Positivo, negativo o cero
print(SEPARADOR)
print("Ejercicio 1 - Positivo, Negativo o Cero")
n = float(input("Ingrese un numero: "))
if n > 0:
    print("El numero es positivo")
elif n < 0:
    print("El numero es negativo")
else:
    print("El numero es cero")

# 2) Par o impar
print(SEPARADOR)
print("Ejercicio 2 - Par o Impar")
num = int(input("Ingrese un numero entero: "))
print("Par" if num % 2 == 0 else "Impar")

# 3) Numeros del 1 al 10 (par/impar)
print(SEPARADOR)
print("Ejercicio 3 - Numeros del 1 al 10 (par/impar)")
for i in range(1, 11):
    etiqueta = "par" if i % 2 == 0 else "impar"
    print(f"{i} es {etiqueta}")

# 4) Suma acumulativa 1..n
print(SEPARADOR)
print("Ejercicio 4 - Suma acumulativa")
n = int(input("Ingrese un numero n: "))
suma = sum(range(1, n + 1))
print(f"La suma de 1 a {n} es: {suma}")

```

```python
# 5) Contador con while (1..10)
print(SEPARADOR)
print("Ejercicio 5 - Contador con while")
c = 1
while c <= 10:
    print(c)
    c += 1

# 6) Tabla de multiplicar
print(SEPARADOR)
print("Ejercicio 6 - Tabla de multiplicar")
t = int(input("Ingrese un numero: "))
for i in range(1, 11):
    print(f"{t} x {i} = {t * i}")

# 7) Suma de numeros positivos hasta negativo
print(SEPARADOR)
print("Ejercicio 7 - Suma de numeros positivos")
ac = 0.0
while True:
    x = float(input("Ingrese un numero (negativo para salir): "))
    if x < 0:
        break
    ac += x
print(f"Suma total: {ac}")

# 8) Contar vocales
print(SEPARADOR)
print("Ejercicio 8 - Contar vocales")
palabra = input("Ingrese una palabra: ")
vocales = "aeiouAEIOU"
cont = sum(1 for ch in palabra if ch in vocales)
print(f"La palabra '{palabra}' tiene {cont} vocales.")

# 9) Numeros primos en un rango [a, b]
print(SEPARADOR)
print("Ejercicio 9 - Numeros primos en un rango")
a = int(input("Inicio: "))
b = int(input("Fin: "))
print(f"Primos entre {a} y {b}:")
for val in range(a, b + 1):
    if val > 1 and all(val % d != 0 for d in range(2, int(val**0.5)
        + 1)):
        print(val)

# 10) Menu basico (suma/resta)
print(SEPARADOR)
print("Ejercicio 10 - Menu basico (suma/resta)")
print("1) Sumar dos numeros")
print("2) Restar dos numeros")
print("3) Salir")
```

```
op = input("Opcion: ").strip()
if op in {"1", "2"}:
    a = float(input("a: "))
    b = float(input("b: "))
    print("Resultado:", a + b if op == "1" else a - b)
else:
    print("Fin del bloque A")
print(SEPARADOR)
```

Listing 1: Bloque A - Logica y Control (Python basico)

Bloque B - Operaciones con math (10 ejercicios)

```
# BLOQUE B - OPERACIONES CON math
# Objetivo: vincular calculo aritmetico/geometrico con verificacion
    computacional.

import math
SEPARADOR = "-" * 48

# 1) Raiz cuadrada segura
print(SEPARADOR)
print("Ejercicio 1 - Raiz cuadrada segura")
x = float(input("Ingrese un numero: "))
if x >= 0:
    print("sqrt(x) =", math.sqrt(x))
else:
    print("Error: no existe raiz real para numero negativo")

# 2) Area de un triangulo (base, altura > 0)
print(SEPARADOR)
print("Ejercicio 2 - Area de un triangulo")
base = float(input("Base: "))
altura = float(input("Altura: "))
if base > 0 and altura > 0:
    print("Area =", (base * altura) / 2)
else:
    print("Datos no validos (base y altura deben ser positivos)")

# 3) Conversion grados a radianes
print(SEPARADOR)
print("Ejercicio 3 - Grados a radianes")
gr = float(input("Angulo en grados: "))
rad = math.radians(gr)
print("radianes =", rad)
if gr > 360:
    print("Aviso: el angulo supera una vuelta completa")

# 4) Redondeos (ceil, floor) y valor absoluto
print(SEPARADOR)
print("Ejercicio 4 - Redondeos y valor absoluto")
```

```
z = float(input("Numero decimal: "))
print("ceil(z) =", math.ceil(z))
print("floor(z) =", math.floor(z))
print("fabs(z) =", math.fabs(z))

# 5) Potencias de 3 hasta un limite n
print(SEPARADOR)
print("Ejercicio 5 - Potencias de 3")
lim = int(input("Exponente limite: "))
for i in range(lim + 1):
    print(f"3**{i} = {3 ** i}")

# 6) Tabla de raices cuadradas (1..10)
print(SEPARADOR)
print("Ejercicio 6 - Tabla de raices cuadradas (1..10)")
for i in range(1, 11):
    print(f"sqrt({i}) = {round(math.sqrt(i), 2)}")

# 7) Verificacion de numero primo (simple)
print(SEPARADOR)
print("Ejercicio 7 - Verificar si n es primo")
n = int(input("n: "))
if n <= 1:
    print("No es primo")
else:
    es_primo = all(n % d != 0 for d in range(2, int(math.sqrt(n)) +
        1))
    print("Es primo" if es_primo else "No es primo")

# 8) Factoriales acumulados (1..5) y suma
print(SEPARADOR)
print("Ejercicio 8 - Factoriales acumulados")
suma_fact = 0
for k in range(1, 6):
    fk = math.factorial(k)
    suma_fact += fk
    print(f"{k}! = {fk}")
print("Suma factorial =", suma_fact)

# 9) Potencia a^b (entrada controlada)
print(SEPARADOR)
print("Ejercicio 9 - Potencia a^b")
a = float(input("Base a: "))
b = float(input("Exponente b: "))
print("a**b =", math.pow(a, b))

# 10) Mini calculadora cientifica (menu)
print(SEPARADOR)
print("Ejercicio 10 - Calculadora cientifica (menu)")
while True:
```

```
    print("\n1) sqrt    2) potencia    3) log10    4) factorial
       5) salir")
    opcion = input("Opcion: ").strip()
    if opcion == "1":
        v = float(input("Numero: "))
        print("sqrt =", math.sqrt(v) if v >= 0 else "No real")
    elif opcion == "2":
        b1 = float(input("Base: "))
        e1 = float(input("Exponente: "))
        print("resultado =", math.pow(b1, e1))
    elif opcion == "3":
        v = float(input("Numero (>0): "))
        print("log10 =", math.log10(v) if v > 0 else "No definido")
    elif opcion == "4":
        t = int(input("Entero >= 0: "))
        print("factorial =", math.factorial(t) if t >= 0 else "No
           definido")
    elif opcion == "5":
        print("Fin del bloque B")
        break
    else:
        print("Opcion no valida")
print(SEPARADOR)
```

Listing 2: Bloque B - Operaciones con la libreria estandar math

Bloque C - Algebra Simbolica con SymPy (10 ejercicios)

```
# BLOQUE C - ALGEBRA SIMBOLICA CON SymPy
# Objetivo: verificar y explicar errores algebraicos frecuentes
   mediante
# operaciones simbolicas (expand, simplify, Eq).

import sympy as sp
from sympy import symbols, expand, simplify, Eq

SEPARADOR = "-" * 48
x, y, a, b = symbols("x y a b")

# 1) Distribucion del signo negativo
print(SEPARADOR)
print("Ejercicio 1 - Signo negativo")
expr = -(3*x + 2*y - 5)
print("Original: -(3x + 2y - 5)")
print("Distribuido:", expand(expr))

# 2) Exponentes: (x + y)^2 vs x^2 + y^2
print(SEPARADOR)
print("Ejercicio 2 - Exponentes (cuadrado de binomio)")
err = x**2 + y**2
ok = expand((x + y)**2)
```

```
print("Error comun:", err)
print("Forma correcta:", ok)

# 3) Formula (a - b)^2 = a^2 - 2ab + b^2
print(SEPARADOR)
print("Ejercicio 3 - Formula del binomio (a - b)^2")
print("Expansion:", expand((a - b)**2))

# 4) Jerarquia de operaciones
print(SEPARADOR)
print("Ejercicio 4 - Jerarquia de operaciones")
expr_txt = "2 + 3 * 4"
val = sp.sympify(expr_txt)
print(f"Evaluacion de '{expr_txt}':", val)

# 5) Fraccion algebraica: simplificacion
print(SEPARADOR)
print("Ejercicio 5 - Fraccion algebraica")
frac = (x + 6) / (3*x)
print("Simplificada:", simplify(frac))
print("Nota: los terminos del numerador se dividen por el mismo
   monomio del denominador.")

# 6) Leyes de los exponentes: (2x)^2
print(SEPARADOR)
print("Ejercicio 6 - Leyes de los exponentes")
res = expand((2*x)**2)
print("Resultado correcto de (2x)^2:", res)

# 7) Binomio al cubo: (x + y)^3
print(SEPARADOR)
print("Ejercicio 7 - Binomio al cubo")
print("Expansion:", expand((x + y)**3))

# 8) Verificacion de igualdad simbolica con Eq
print(SEPARADOR)
print("Ejercicio 8 - Verificar igualdad con Eq")
lhs = expand((x + y)**2)
rhs = x**2 + 2*x*y + y**2
print("Eq(lhs, rhs) =", Eq(lhs, rhs))

# 9) Sustitucion numerica para contrastar error vs correcto
print(SEPARADOR)
print("Ejercicio 9 - Sustitucion numerica (x=2, y=3)")
print("Error (x^2 + y^2) =", err.subs({x: 2, y: 3}))
print("Correcto ((x + y)^2) =", ok.subs({x: 2, y: 3}))

# 10) Resumen: simplificar, expandir, comparar
print(SEPARADOR)
print("Ejercicio 10 - Resumen de herramientas")
expr_demo = (x + 1)*(x - 1)
```

```
print("Original:", expr_demo)
print("Expand:", expand(expr_demo))
print("Simplify((x^2 - 1)/(x - 1)) ->", simplify((x**2 - 1)/(x - 1)
    ))
print("Fin del bloque C")
print(SEPARADOR)
```

Listing 3: Bloque C - Algebra Simbolica con SymPy