Manual Completo de Python: De Cero a Experto

Autor: Basdonax AI **Fecha:** Noviembre 2025

MÓDULO 1: NIVEL BÁSICO - Los Fundamentos de Python

¡Bienvenido/a a tu viaje en el mundo de Python! Este módulo es el punto de partida, diseñado para establecer una base sólida en los conceptos esenciales de la programación y el lenguaje Python. Al finalizar, serás capaz de escribir programas sencillos, entender cómo funcionan las variables y controlar el flujo de ejecución de tu código.

1. Introducción a Python

1.1. ¿Qué es Python y por qué aprenderlo?

Teoría: Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, creado por Guido van Rossum y lanzado por primera vez en 1991. Su filosofía se centra en la **legibilidad del código**, lo que lo hace ideal para principiantes. Es conocido por su sintaxis clara y concisa, que permite a los desarrolladores expresar conceptos en menos líneas de código que otros lenguajes.

¿Por qué es tan popular? Python se utiliza en casi todos los campos de la tecnología moderna:

- **Desarrollo Web:** Con frameworks como Django y Flask.
- **Análisis de Datos y Ciencia de Datos:** Con librerías como Pandas y NumPy.
- **Inteligencia Artificial y Machine Learning:** Con TensorFlow, PyTorch y Scikit-learn.
- **Automatización de Tareas (Scripting):** Para simplificar tareas repetitivas.

1.2. Instalación y configuración del entorno (Python 3.12+)

Teoría: Para empezar a programar, necesitamos el intérprete de Python. La versión recomendada es la 3.12 o superior. El intérprete es el programa que lee tu código Python y lo ejecuta. Además, utilizaremos un **Entorno de Desarrollo Integrado (IDE)** o un editor de código, como Visual Studio Code, para escribir y gestionar nuestros archivos.

Ejemplo práctico (Verificación de la instalación): Una vez instalado, puedes verificar la versión de Python desde la terminal o línea de comandos.

```
# Comando para verificar la versión de Python
python3 --version
```

Explicación línea por línea:

- `python3` : Invoca el intérprete de Python (en algunos sistemas, puede ser solo `python`).
- `--version` : Es un argumento que le pide al intérprete que muestre su versión instalada.

1.3. El primer programa: "Hola Mundo" y la función `print()`

Teoría: La función `print()` es la herramienta fundamental en Python para mostrar información en la consola. Todo lo que se encuentra dentro de los paréntesis y, si es texto, entre comillas, será impreso.

Ejemplo práctico:

```
# Este es nuestro primer programa en Python
print("¡Hola Mundo! Estoy aprendiendo Python con Basdonax AI.")
print(10 + 5)
```

Explicación línea por línea:

- `# Este es...` : Una línea que comienza con `#` es un **comentario**. Python la ignora; solo sirve para que los humanos entendamos el código.

- `print(...)`: Llama a la función `print`.
- `"¡Hola Mundo!..."`: Es una cadena de texto (string) que se mostrará exactamente como está escrita.
- `print(10 + 5)`: La función `print` también puede evaluar expresiones matemáticas antes de mostrar el resultado (15).

Ejercicio 1.3: Escribe un programa que imprima tu nombre completo en una línea y tu edad en la línea siguiente.

Solución 1.3:

```
# Solución al Ejercicio 1.3
print("Mi nombre es [Tu Nombre Completo]")
print(30) # Reemplaza 30 con tu edad real
```

---

2. Variables y Tipos de Datos

2.1. Concepto de variable y asignación

Teoría: Una **variable** es un nombre simbólico que hace referencia a un valor almacenado en la memoria de la computadora. Piensa en una variable como una caja etiquetada donde puedes guardar información. En Python, la asignación se realiza con el operador de igualdad (`=`).

Ejemplo práctico:

```
# Asignación de variables
nombre_usuario = "Ana López"
edad = 28
es_estudiante = True

print(nombre_usuario)
print(edad)
print(es_estudiante)
```

Explicación línea por línea:

- `nombre_usuario = "Ana López"`: Asigna la cadena de texto `"Ana López"` a la variable `nombre_usuario`.
- `edad = 28`: Asigna el número entero `28` a la variable `edad`.
- `es_estudiante = True`: Asigna el valor booleano `True` a la variable `es_estudiante`.
- Las llamadas a `print()` muestran el valor almacenado en cada variable.

2.2. Tipos de datos primitivos: `int`, `float`, `str`, `bool`

Teoría: Python maneja varios tipos de datos. Los más comunes son:

Tipo	Nombre Completo	Descripción	Ejemplo
`int`	Entero	Números sin decimales (positivos o negativos).	`10`, `-500`
`float`	Flotante	Números con parte decimal.	`3.14`, `-0.01`
`str`	Cadena de Texto	Secuencia de caracteres (debe ir entre comillas).	`"Python"`, `'Hola'`
`bool`	Booleano	Representa valores de verdad: `True` o `False`.	`True`, `False`

Puedes usar la función `type()` para verificar el tipo de dato de una variable.

Ejemplo práctico:

```
numero_entero = 100
numero_decimal = 10.5
texto = "Basdonax"
verdadero_falso = False

print(type(numero_entero))
print(type(numero_decimal))
print(type(texto))
print(type(verdadero_falso))
```

2.3. Conversión de tipos (`casting`)

Teoría: A veces, necesitamos cambiar el tipo de dato de una variable. Esto se llama **conversión de tipos** o *casting*. Python proporciona funciones como `int()`, `float()`, y `str()` para realizar estas conversiones.

Ejemplo práctico:

```
numero_como_texto = "123"
numero_entero = int(numero_como_texto) # Convierte "123" a 123 (entero)

numero_decimal = 9.99
numero_entero_truncado = int(numero_decimal) # Convierte 9.99 a 9 (trunca la
parte decimal)

print(f"El tipo de {numero_entero} es {type(numero_entero)}")
print(f"El tipo de {numero_entero_truncado} es
{type(numero_entero_truncado)}")
```

2.4. Entrada de datos por el usuario: la función `input()`

Teoría: La función `input()` detiene la ejecución del programa y espera a que el usuario escriba algo y presione Enter. **Importante:** `input()` siempre devuelve el valor como una **cadena de texto (`str`)**, incluso si el usuario ingresa un número. Si necesitas un número, deberás usar la conversión de tipos (`int()` o `float()`).

Ejemplo práctico:

```
nombre = input("Por favor, ingresa tu nombre: ")
anio_nacimiento = input("Ingresa tu año de nacimiento: ")

# Convertimos el año a entero para poder hacer cálculos
edad_calculada = 2025 - int(anio_nacimiento)

print(f"Hola, {nombre}. Tienes aproximadamente {edad_calculada} años.")
```

Explicación línea por línea:

- `nombre = input(...)`: Muestra el mensaje y guarda la respuesta del usuario como `str` en la variable `nombre`.
- `int(anio_nacimiento)`: Convierte la cadena de texto ingresada (ej. "1995") a un número entero (ej. 1995).
- `f"Hola, {nombre}..."`: Es una **f-string** (cadena formateada), una forma moderna y legible de incluir variables dentro de una cadena de texto.

Ejercicio 2.4: Crea un programa que solicite al usuario dos números. Luego, suma esos números y muestra el resultado en la consola. Asegúrate de manejar la conversión de tipos correctamente.

Solución 2.4:

```
# Solución al Ejercicio 2.4
num1_str = input("Ingresa el primer número: ")
num2_str = input("Ingresa el segundo número: ")

# Convertir a float para permitir números decimales
num1 = float(num1_str)
num2 = float(num2_str)

suma = num1 + num2

print(f"La suma de {num1} y {num2} es igual a {suma}.")
```

---

3. Operadores y Expresiones

3.1. Operadores aritméticos

Teoría: Los operadores aritméticos se utilizan para realizar cálculos matemáticos.

Operador	Nombre	Descripción	Ejemplo	Resultado
`+`	Suma	Suma dos valores.	`5 + 3`	`8`
`-`	Resta	Resta dos valores.	`10 - 4`	`6`
`*`	Multiplicación	Multiplica dos valores.	`4 * 5`	`20`
`/`	División	Divide dos valores (siempre devuelve un `float`).	`10 / 3`	`3.333...`
`//`	División Entera	Divide y devuelve solo la parte entera.	`10 // 3`	`3`
`%`	Módulo	Devuelve el resto de la división.	`10 % 3`	`1`
`**`	Exponenciación	Eleva un número a la potencia de otro.	`2 ** 3`	`8`

Ejemplo práctico:

```
a = 15
b = 4

print(f"Suma: {a + b}")
print(f"División: {a / b}")
print(f"Módulo (Resto): {a % b}")
```

3.2. Operadores de comparación

Teoría: Los operadores de comparación se utilizan para comparar dos valores y siempre devuelven un valor booleano (`True` o `False`).

Operador	Descripción	Ejemplo
`==`	Igual a	`5 == 5` (True)
`!=`	Diferente de	`5 != 10` (True)
`>`	Mayor que	`10 > 5` (True)
`<`	Menor que	`5 < 10` (True)
`>=`	Mayor o igual que	`5 >= 5` (True)
`<=`	Menor o igual que	`5 <= 10` (True)

Ejemplo práctico:

```
saldo = 1000
gasto = 500

print(f"¿El saldo es mayor que el gasto? {saldo > gasto}")
print(f"¿El saldo es igual a 1000? {saldo == 1000}")
```

3.3. Operadores lógicos (`and` , `or` , `not`)

Teoría: Los operadores lógicos combinan expresiones booleanas para obtener un único resultado booleano.

Operador	Descripción	Resultado
`and`	Devuelve `True` si **ambas** expresiones son verdaderas.	`True and False` (False)
`or`	Devuelve `True` si **al menos una** expresión es verdadera.	`True or False` (True)
`not`	Invierte el valor booleano.	`not True` (False)

Ejemplo práctico:

```
tiene_licencia = True
es_mayor_edad = True

puede_conducir = tiene_licencia and es_mayor_edad
print(f"¿Puede conducir? {puede_conducir}")

es_dia_libre = False
es_fin_semana = True

puede_descansar = es_dia_libre or es_fin_semana
print(f"¿Puede descansar? {puede_descansar}")
```

Ejercicio 3.3: Una persona puede obtener un descuento si es **mayor de 60 años** O si es **menor de 18 años**. Crea un programa que solicite la edad y determine si la persona aplica para el descuento.

Solución 3.3:

```
# Solución al Ejercicio 3.3
edad_str = input("Ingresa tu edad: ")
edad = int(edad_str)

aplica_descuento = (edad > 60) or (edad < 18)

print(f"¿Aplica para el descuento? {aplica_descuento}")
```

---

4. Estructuras de Control de Flujo

4.1. Condicionales: `if`, `elif`, `else`

Teoría: Las estructuras condicionales permiten que el programa tome decisiones y ejecute diferentes bloques de código según si una condición es verdadera o falsa.

- `if`: Ejecuta un bloque de código si la condición es `True`.
- `elif` (else if): Se evalúa si la condición `if` anterior fue `False`. Puedes tener múltiples `elif`.

- `else`: Ejecuta un bloque de código si ninguna de las condiciones anteriores fue `True`.

Ejemplo práctico:

```
nota = 75

if nota >= 90:
    print("¡Excelente! Aprobaste con A.")
elif nota >= 70:
    print("Buen trabajo. Aprobaste con B.")
else:
    print("Necesitas repasar. No aprobaste.")
```

Explicación: Python evalúa las condiciones en orden. Si `nota >= 90` es `True`, ejecuta ese bloque y salta el resto. Si es `False`, pasa a evaluar `nota >= 70`, y así sucesivamente. El bloque `else` es el camino por defecto.

4.2. Bucles: `while`

Teoría: El bucle `while` ejecuta un bloque de código **mientras** una condición sea verdadera. Es crucial asegurarse de que la condición eventualmente se vuelva falsa para evitar un **bucle infinito**.

Ejemplo práctico:

```
contador = 1
while contador <= 5:
    print(f"Contando: {contador}")
    contador = contador + 1 # Esto asegura que el bucle termine
```

Explicación línea por línea:

- `contador = 1`: Inicializa la variable de control.
- `while contador <= 5:`: La condición. El bucle se repite mientras `contador` sea menor o igual a 5.
- `contador = contador + 1`: Incrementa el contador en 1 en cada iteración. Esto es lo que garantiza que la condición se vuelva falsa después de 5 repeticiones.

4.3. Bucles: `for` y la función `range()`

Teoría: El bucle `for` se utiliza para iterar sobre una secuencia (como una lista de números o una cadena de texto). La función `range()` es muy común con `for`, ya que genera una secuencia de números.

- `range(n)`: Genera números desde 0 hasta `n-1`.
- `range(inicio, fin)`: Genera números desde `inicio` hasta `fin-1`.
- `range(inicio, fin, paso)`: Genera números con un incremento o decremento específico.

Ejemplo práctico:

```
# Iterar sobre una secuencia de números
for i in range(3): # Genera 0, 1, 2
    print(f"Iteración número {i}")

# Iterar sobre una cadena de texto
palabra = "Python"
for letra in palabra:
    print(f"Letra: {letra}")
```

Ejercicio 4.3: Utiliza un bucle `for` y la función `range()` para imprimir todos los números pares desde 2 hasta 10 (inclusive).

Solución 4.3:

```
# Solución al Ejercicio 4.3
# Usamos range(2, 11, 2) para empezar en 2, terminar antes de 11, y avanzar
de 2 en 2.
for numero in range(2, 11, 2):
    print(numero)
```

---

5. Estructuras de Datos Fundamentales (I)

5.1. Listas (`list`): Creación, acceso, modificación y métodos básicos

Teoría: Una **lista** es una colección ordenada y mutable (modificable) de elementos. Los elementos pueden ser de cualquier tipo de dato y se definen entre corchetes `[]`.

Ejemplo práctico:

```
frutas = ["manzana", "banana", "cereza"]

# Acceso (la indexación comienza en 0)
print(f"La primera fruta es: {frutas[0]}")

# Modificación
frutas[1] = "uva"
print(f"Lista modificada: {frutas}")

# Métodos básicos
frutas.append("naranja") # Añade un elemento al final
frutas.remove("manzana") # Elimina el primer elemento que coincida
print(f"Lista final: {frutas}")
```

5.2. Tuplas (`tuple`): Inmutabilidad y usos

Teoría: Una **tupla** es una colección ordenada e **inmutable** (no se puede modificar después de su creación) de elementos. Se definen entre paréntesis `()`. Se usan para datos que no deben cambiar, como coordenadas geográficas o fechas.

Ejemplo práctico:

```
coordenadas = (10.5, 20.3)

# Acceso (igual que las listas)
print(f"Latitud: {coordenadas[0]}")

# Intento de modificación (esto causaría un error)
# coordenadas[0] = 11.0
# print(coordenadas)
```

Ejercicio 5.2: Crea una lista llamada `tareas` con tres tareas pendientes. Luego, usa un método de lista para añadir una cuarta tarea y finalmente, usa un bucle `for` para imprimir cada tarea de la lista.

Solución 5.2:

```
# Solución al Ejercicio 5.2
tareas = ["Comprar víveres", "Estudiar Python", "Hacer ejercicio"]

# Añadir una tarea
tareas.append("Llamar al doctor")

# Imprimir cada tarea
print("\nLista de Tareas:")
for tarea in tareas:
    print(f"- {tarea}")
```

---

6. Evaluación Parcial del Nivel Básico

Instrucciones: Responde las siguientes preguntas y resuelve los ejercicios de codificación.

Cuestionario de Opción Múltiple

1. ¿Qué tipo de dato devuelve la función `input()` en Python? a) `int` b) `float` c) `str` d) `bool`
2. ¿Cuál es el resultado de la expresión `10 // 3`? a) 3.333... b) 3 c) 1 d) 0
3. ¿Cuál de las siguientes estructuras de datos es **inmutable**? a) `list` b) `dict` c) `tuple` d) `set`

Ejercicios de Codificación Cortos

1. **Calculadora de Área:** Escribe un programa que solicite el radio de un círculo (como número decimal) y calcule e imprima su área. (Fórmula: Área = $\pi \cdot r^2$. Usa `3.14159` como valor de π).
2. **Clasificador de Edad:** Solicita la edad de una persona. Si la edad es mayor o igual a 18, imprime "Es adulto". Si es menor de 18, imprime "Es menor de

edad”.

3. **Contador Regresivo:** Usa un bucle `while` para imprimir los números del 5 al 1, y luego imprime “¡Despegue!”.

Soluciones a la Evaluación

Cuestionario:

1. c) `str`
2. b) 3
3. c) `tuple`

Ejercicios de Codificación:

1. **Calculadora de Área:**

```
PI = 3.14159
radio_str = input("Ingresa el radio del círculo: ")
radio = float(radio_str)


area = PI * (radio ** 2)


print(f"El área del círculo con radio {radio} es: {area}")
```

2. **Clasificador de Edad:**

```
edad_str = input("Ingresa tu edad: ")
edad = int(edad_str)


if edad >= 18:
    print("Es adulto")
else:
    print("Es menor de edad")
```

3. **Contador Regresivo:**

```python
contador = 5
while contador >= 1:
    print(contador)
    contador -= 1 # Equivalente a contador = contador - 1


print("¡Despegue!")
```

MÓDULO 2: NIVEL INTERMEDIO - Organización y Reutilización del Código

Has dominado los fundamentos de Python. En este módulo, profundizaremos en estructuras de datos más complejas, aprenderás a organizar tu código con funciones y a manejar los errores que inevitablemente surgen en la programación.

7. Estructuras de Datos Fundamentales (II)

7.1. Diccionarios (`dict`): Clave-valor, acceso y métodos

Teoría: Un **diccionario** es una colección de pares **clave-valor** no ordenada, mutable e indexada. Piensa en él como una agenda telefónica: en lugar de usar un índice numérico (como en las listas), usas una clave única (el nombre) para acceder a un valor (el número de teléfono). Se definen entre llaves `{}`.

Ejemplo práctico:

```python
persona = {
    "nombre": "Carlos",
    "edad": 35,
    "ciudad": "Santiago"
}

# Acceso al valor usando la clave
print(f"Nombre: {persona['nombre']}")

# Modificación
persona["edad"] = 36
print(f"Nueva edad: {persona['edad']}")

# Métodos básicos
print(f"Claves: {persona.keys()}")
print(f"Valores: {persona.values()}")
```

Explicación línea por línea:

- `persona = {...}`: Define un diccionario con tres pares clave-valor.
- `persona['nombre']`: Accede al valor asociado a la clave `"nombre"`.
- `persona.keys()`: Devuelve una vista de todas las claves del diccionario.

7.2. Conjuntos (`set`): Operaciones y usos

Teoría: Un **conjunto** es una colección de elementos no ordenada y no indexada, donde **no se permiten elementos duplicados**. Se definen entre llaves `{}` o usando la función `set()`. Son ideales para eliminar duplicados de una lista o para realizar operaciones matemáticas de conjuntos (unión, intersección, diferencia).

Ejemplo práctico:

```python
numeros_duplicados = [1, 2, 2, 3, 4, 4, 5]
conjunto_sin_duplicados = set(numeros_duplicados)
print(f"Conjunto sin duplicados: {conjunto_sin_duplicados}")

a = {1, 2, 3}
b = {3, 4, 5}

# Operación de unión
union = a.union(b)
print(f"Unión: {union}") # {1, 2, 3, 4, 5}

# Operación de intersección
interseccion = a.intersection(b)
print(f"Intersección: {interseccion}") # {3}
```

Ejercicio 7.2: Tienes una lista de estudiantes que asisten a la clase de Python y otra lista de estudiantes que asisten a la clase de SQL. Usa conjuntos para encontrar qué estudiantes asisten a **ambas** clases.

```python
estudiantes_python = ["Ana", "Beto", "Carla", "David"]
estudiantes_sql = ["Carla", "Elena", "David", "Felipe"]
```

Solución 7.2:

```python
# Solución al Ejercicio 7.2
set_python = set(estudiantes_python)
set_sql = set(estudiantes_sql)

# La intersección nos da los elementos comunes
comunes = set_python.intersection(set_sql)

print(f"Estudiantes en ambas clases: {comunes}")
```

---

8. Funciones

8.1. Definición y llamada de funciones

Teoría: Una **función** es un bloque de código organizado y reutilizable que se utiliza para realizar una única acción relacionada. Las funciones permiten dividir un programa grande en partes más pequeñas y manejables. Se definen con la palabra clave `def`.

Ejemplo práctico:

```
def saludar(nombre):
    """Esta función saluda a la persona que se le pasa como parámetro."""
    print(f"¡Hola, {nombre}! Bienvenido/a a la programación.")

# Llamada a la función
saludar("Martín")
saludar("Sofía")
```

Explicación línea por línea:

- `def saludar(nombre):` : Define la función llamada `saludar` que acepta un parámetro llamado `nombre`.
- `"""..."""` : Es un *docstring*, una documentación que explica qué hace la función.
- `saludar("Martín")` : Llama a la función, pasando `"Martín"` como argumento para el parámetro `nombre`.

8.2. Parámetros y argumentos (posicionales y por palabra clave)

Teoría:

- **Parámetro:** El nombre de la variable que se usa en la definición de la función (ej. `nombre` en `def saludar(nombre)`).
- **Argumento:** El valor real que se pasa a la función cuando se llama (ej. `"Martín"` en `saludar("Martín")`).

Los argumentos pueden pasarse por **posición** (el orden importa) o por **palabra clave** (el orden no importa, pero se especifica el nombre del parámetro).

Ejemplo práctico:

```
def describir_producto(nombre, precio, stock):
    print(f"Producto: {nombre}")
    print(f"Precio: ${precio}")
    print(f"Stock disponible: {stock} unidades")

# Argumentos posicionales (el orden es crucial)
describir_producto("Laptop", 1200, 50)

# Argumentos por palabra clave (el orden no importa)
describir_producto(stock=10, nombre="Mouse", precio=25)
```

8.3. Valores de retorno (`return`)

Teoría: La palabra clave `return` se utiliza para que una función devuelva un valor al código que la llamó. Si una función no tiene `return`, devuelve implícitamente `None`.

Ejemplo práctico:

```
def calcular_iva(monto, tasa=0.19):
    """Calcula el IVA de un monto dado."""
    iva = monto * tasa
    return iva

precio_neto = 1000
impuesto = calcular_iva(precio_neto)

print(f"El IVA a pagar es: ${impuesto}")
print(f"El precio total es: ${precio_neto + impuesto}")
```

8.4. Ámbito de las variables (local y global)

Teoría: El **ámbito** (o *scope*) de una variable define dónde puede ser accedida.

- **Variables Locales:** Definidas dentro de una función. Solo existen dentro de esa función.

- **Variables Globales:** Definidas fuera de cualquier función. Pueden ser accedidas desde cualquier parte del código.

Ejemplo práctico:

```
mensaje_global = "Soy visible en todas partes." # Variable Global

def mi_funcion():
    mensaje_local = "Solo existo aquí dentro." # Variable Local
    print(mensaje_global)
    # print(mensaje_local) # Esto funciona

mi_funcion()
# print(mensaje_local) # Esto causaría un error (NameError)
```

8.5. Funciones Lambda

Teoría: Una **función lambda** es una pequeña función anónima (sin nombre) que puede tomar cualquier número de argumentos, pero solo puede tener una expresión. Se definen con la palabra clave `lambda`. Son útiles para tareas sencillas donde no se justifica definir una función completa.

Sintaxis: `lambda argumentos: expresión`

Ejemplo práctico:

```
# Función normal
def duplicar(x):
    return x * 2

# Función lambda equivalente
duplicar_lambda = lambda x: x * 2

print(f"Resultado normal: {duplicar(5)}")
print(f"Resultado lambda: {duplicar_lambda(5)}")
```

Ejercicio 8.5: Crea una función que reciba una lista de números y devuelva una nueva lista con solo los números que son mayores a 10.

Solución 8.5:

```python
# Solución al Ejercicio 8.5
def filtrar_mayores_a_diez(lista_numeros):
    resultado = []
    for numero in lista_numeros:
        if numero > 10:
            resultado.append(numero)
    return resultado

numeros = [5, 12, 8, 20, 3, 15]
filtrados = filtrar_mayores_a_diez(numeros)
print(f"Números originales: {numeros}")
print(f"Números mayores a 10: {filtrados}")
```

---

9. Manejo de Errores y Excepciones

9.1. Tipos comunes de errores

Teoría: En programación, los errores se dividen en dos categorías principales:

- **Errores de Sintaxis:** El código no sigue las reglas del lenguaje (ej. olvidar dos puntos `:`). Python los detecta antes de la ejecución.
- **Excepciones (Errores en Tiempo de Ejecución):** El código es sintácticamente correcto, pero ocurre un problema durante la ejecución (ej. dividir por cero, intentar acceder a un índice inexistente en una lista).

Algunas excepciones comunes son: `ZeroDivisionError`, `NameError`, `TypeError`, `IndexError`, `KeyError`.

9.2. Bloques `try`, `except`, `else` y `finally`

Teoría: Para manejar las excepciones de forma elegante y evitar que el programa se detenga, usamos los bloques `try...except`.

- `try`: Contiene el código que podría generar una excepción.
- `except`: Contiene el código que se ejecuta si ocurre una excepción en el bloque `try`. Puedes especificar el tipo de excepción a capturar.
- `else`: (Opcional) Contiene el código que se ejecuta si el bloque `try` **no** generó ninguna excepción.

- `finally`: (Opcional) Contiene el código que se ejecuta **siempre**, haya ocurrido una excepción o no.

Ejemplo práctico:

```
try:
    num1 = int(input("Ingresa el numerador: "))
    num2 = int(input("Ingresa el denominador: "))
    resultado = num1 / num2

except ZeroDivisionError:
    print("Error: No se puede dividir por cero.")
except ValueError:
    print("Error: Debes ingresar un número entero válido.")
except Exception as e:
    print(f"Ocurrió un error inesperado: {e}")
else:
    print(f"El resultado de la división es: {resultado}")
finally:
    print("Fin del intento de división.")
```

Ejercicio 9.2: Escribe una función que solicite al usuario un índice y una lista. Usa `try...except` para manejar el error `IndexError` si el índice está fuera del rango de la lista.

Solución 9.2:

```python
# Solución al Ejercicio 9.2
def obtener_elemento_seguro(lista):
    try:
        indice_str = input("Ingresa un índice para la lista: ")
        indice = int(indice_str)

        elemento = lista[indice]
        print(f"El elemento en el índice {indice} es: {elemento}")

    except ValueError:
        print("Error: El índice debe ser un número entero.")
    except IndexError:
        print(f"Error: El índice {indice} está fuera del rango de la lista
(0 a {len(lista) - 1}).")

colores = ["rojo", "verde", "azul"]
obtener_elemento_seguro(colores)
```

---

10. Módulos y Paquetes

10.1. ¿Qué son los módulos y cómo importarlos?

Teoría: Un **módulo** es simplemente un archivo de Python (`.py`) que contiene funciones, clases y variables. Permite organizar el código y reutilizarlo. Para usar el código de un módulo en otro archivo, se utiliza la palabra clave `import`.

Formas de importar:

- `import modulo`: Importa todo el módulo. Se accede con `modulo.funcion()`.
- `import modulo as alias`: Importa con un nombre más corto. Se accede con `alias.funcion()`.
- `from modulo import funcion`: Importa solo una función específica. Se accede directamente con `funcion()`.

Ejemplo práctico (Módulo `math`):

```python
import math
from math import sqrt as raiz_cuadrada

# Usando el módulo completo
print(f"El valor de PI es: {math.pi}")

# Usando la función importada con alias
print(f"La raíz cuadrada de 16 es: {raiz_cuadrada(16)}")
```

10.2. Creación de módulos propios

Teoría: Cualquier archivo `.py` que crees puede ser un módulo. Si tienes un archivo llamado `calculadora.py` con una función `sumar()`, puedes importarlo en tu archivo principal.

Ejemplo práctico (Simulado): Asumiendo que existe un archivo `utilidades.py` con la función `saludar()`.

```python
# Contenido de utilidades.py (simulado)
# def saludar(nombre):
#     return f"Hola {nombre} desde el módulo utilidades."

# En tu archivo principal:
import utilidades

print(utilidades.saludar("Estudiante"))
```

10.3. Introducción a la gestión de paquetes con `pip`

Teoría: Un **paquete** es una colección de módulos. El ecosistema de Python tiene miles de paquetes creados por la comunidad. **`pip`** es el gestor de paquetes estándar de Python, que te permite instalar, desinstalar y gestionar librerías externas.

Comandos básicos de `pip` (se ejecutan en la terminal, no en Python):

- `pip install nombre_paquete`: Instala un paquete.
- `pip uninstall nombre_paquete`: Desinstala un paquete.
- `pip list`: Muestra todos los paquetes instalados.

Ejemplo práctico (Simulado): Para instalar la librería `requests` (que veremos en el Nivel Experto):

```
# Comando en la terminal
pip install requests
```

---

11. Manejo de Archivos

11.1. Apertura, lectura y escritura de archivos de texto (`.txt`)

Teoría: Para interactuar con archivos en el disco, usamos la función `open()`. Esta función toma el nombre del archivo y el **modo** de apertura.

Modo	Descripción
`'r'`	Lectura (Read). El archivo debe existir.
`'w'`	Escritura (Write). Crea el archivo si no existe, o lo sobrescribe si existe.
`'a'`	Añadir (Append). Crea el archivo si no existe, o añade contenido al final si existe.

Ejemplo práctico (Escritura y Lectura):

```
# Escritura (sobrescribe el archivo)
archivo = open("datos.txt", "w")
archivo.write("Línea uno de prueba.\n")
archivo.write("Línea dos de prueba.\n")
archivo.close() # ¡Importante cerrar el archivo!

# Lectura
archivo = open("datos.txt", "r")
contenido = archivo.read()
print("\nContenido del archivo:")
print(contenido)
archivo.close()
```

11.2. Uso del contexto `with open(...)`

Teoría: El uso de `open()` y `close()` puede ser propenso a errores (olvidar cerrar el archivo). El bloque `with open(...)` es la forma recomendada, ya que garantiza que el archivo se cierre automáticamente, incluso si ocurre un error.

Ejemplo práctico (Recomendado):

```
# Escritura con with
with open("registro.txt", "a") as archivo:
    archivo.write("Nuevo registro de datos.\n")

# Lectura línea por línea con with
print("\nContenido de registro.txt:")
with open("registro.txt", "r") as archivo:
    for linea in archivo:
        print(linea.strip()) # strip() elimina el salto de línea
```

Ejercicio 11.2: Escribe un programa que pida al usuario su nombre y luego lo guarde en un archivo llamado `nombres.txt`, añadiéndolo al final si el archivo ya existe.

Solución 11.2:

```
# Solución al Ejercicio 11.2
nombre_usuario = input("Ingresa tu nombre para el registro: ")

with open("nombres.txt", "a") as archivo:
    archivo.write(nombre_usuario + "\n")

print(f"Nombre '{nombre_usuario}' guardado en nombres.txt.")
```

---

12. Evaluación Parcial del Nivel Intermedio

Instrucciones: Responde las siguientes preguntas y resuelve los ejercicios de codificación.

Cuestionario de Opción Múltiple

1. ¿Cuál es la principal diferencia entre una lista y un diccionario? a) Las listas son mutables y los diccionarios son inmutables. b) Las listas usan índices numéricos y los diccionarios usan claves. c) Los diccionarios permiten duplicados y las listas no. d) Las listas se definen con `{}` y los diccionarios con `[]`.

2. ¿Qué valor devuelve una función que no tiene una sentencia `return` explícita? a) Un error b) El último valor calculado c) `None` d) `0`

3. ¿Qué modo de apertura de archivo se utiliza para añadir contenido al final de un archivo existente sin borrar lo anterior? a) `'r'` b) `'w'` c) `'a'` d) `'x'`

Ejercicios de Codificación Cortos

1. **Contador de Palabras:** Escribe una función que reciba una cadena de texto y devuelva un diccionario donde las claves sean las palabras y los valores sean la cantidad de veces que aparece cada palabra. (Pista: usa el método `.split()` para separar las palabras).
2. **Conversor de Temperatura Seguro:** Escribe un programa que solicite una temperatura en Celsius. Usa `try...except` para manejar el error si el usuario ingresa un valor que no es un número. Si el valor es válido, convierte a Fahrenheit e imprime el resultado. (Fórmula: $F = C \times 9/5 + 32$).
3. **Módulo de Saludo:** Crea un archivo llamado `saludador.py` (simulado) con una función `saludo_formal(nombre)`. Luego, en tu archivo principal, impórtala y úsala.

Soluciones a la Evaluación

Cuestionario:

1. b) Las listas usan índices numéricos y los diccionarios usan claves.
2. c) `None`
3. c) `'a'`

Ejercicios de Codificación:

1. **Contador de Palabras:**

```python
def contar_palabras(texto):
    palabras = texto.lower().split()
    conteo = {}
    for palabra in palabras:
        # Usamos .get(clave, valor_por_defecto) para simplificar
        conteo[palabra] = conteo.get(palabra, 0) + 1
    return conteo


frase = "Python es genial y Python es fácil"
resultado_conteo = contar_palabras(frase)
print(f"Conteo de palabras: {resultado_conteo}")
```

2. **Conversor de Temperatura Seguro:**

```python
try:
    celsius_str = input("Ingresa la temperatura en Celsius: ")
    celsius = float(celsius_str)


    fahrenheit = celsius * (9/5) + 32
    print(f"{celsius}°C es igual a {fahrenheit}°F")


except ValueError:
    print("Error: La entrada no es un número válido.")
```

3. **Módulo de Saludo (Simulado):**

```python
# Simulación del archivo saludador.py
# def saludo_formal(nombre):
#     return f"Estimado/a {nombre}, le damos la bienvenida."

# En el archivo principal:
# import saludador
# print(saludador.saludo_formal("Dr. Pérez"))
```

MÓDULO 3: NIVEL AVANZADO - Programación Orientada a Objetos y Estructuras Complejas

¡Felicidades! Estás entrando en el nivel Avanzado. Aquí dominarás la **Programación Orientada a Objetos (POO)**, una metodología fundamental para escribir código escalable, mantenible y reutilizable. También exploraremos herramientas poderosas como los decoradores y las expresiones regulares.

13. Programación Orientada a Objetos (POO)

13.1. Clases y Objetos: Definición y atributos

Teoría: La POO es un paradigma de programación que utiliza **objetos** para modelar entidades del mundo real.

- Una **Clase** es un plano o plantilla para crear objetos. Define las propiedades (atributos) y los comportamientos (métodos) que tendrán los objetos de ese tipo.
- Un **Objeto** es una instancia específica de una clase.

Ejemplo práctico:

```
class Coche:
    # Atributo de clase (compartido por todos los objetos)
    ruedas = 4

    def __init__(self, marca, color):
        # Atributos de instancia (únicos para cada objeto)
        self.marca = marca
        self.color = color

# Creación de objetos (instanciación)
mi_coche = Coche("Toyota", "Rojo")
otro_coche = Coche("Ford", "Azul")

print(f"Mi coche es un {mi_coche.marca} de color {mi_coche.color}")
print(f"El otro coche tiene {otro_coche.ruedas} ruedas")
```

13.2. El constructor `__init__` y el método `self`

Teoría:

- El método especial `__init__` (conocido como **constructor**) se llama automáticamente cada vez que se crea un nuevo objeto de la clase. Se utiliza para inicializar los atributos del objeto.
- El parámetro `self` es una referencia al objeto que se está creando o al que se está llamando. Es el primer parámetro de todos los métodos de instancia y permite acceder a los atributos y otros métodos del objeto.

13.3. Métodos de instancia

Teoría: Los **métodos** son funciones definidas dentro de una clase que describen el comportamiento de los objetos.

Ejemplo práctico:

```
class Mascota:
    def __init__(self, nombre, especie):
        self.nombre = nombre
        self.especie = especie

    # Método de instancia
    def hacer_sonido(self):
        if self.especie == "perro":
            print(f"{self.nombre} dice: ¡Guau!")
        elif self.especie == "gato":
            print(f"{self.nombre} dice: ¡Miau!")
        else:
            print(f"{self.nombre} hace un sonido desconocido.")

mi_mascota = Mascota("Fido", "perro")
mi_mascota.hacer_sonido()
```

13.4. Encapsulamiento: Atributos privados y públicos

Teoría: El **encapsulamiento** es el principio de agrupar datos (atributos) y los métodos que operan sobre esos datos en una unidad (la clase), y restringir el acceso directo a algunos de los componentes del objeto.

En Python, por convención:

- **Público:** Atributos normales (ej. `self.nombre`). Accesibles desde cualquier lugar.
- **Protegido:** Atributos que comienzan con un guion bajo (ej. `self._saldo`). Sugiere que no deben ser modificados directamente fuera de la clase o sus subclases.
- **Privado:** Atributos que comienzan con doble guion bajo (ej. `self.__clave`). Python "deforma" el nombre para hacer el acceso directo más difícil (aunque no imposible).

Ejemplo práctico:

```
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self.__saldo = saldo_inicial # Atributo privado

    def depositar(self, monto):
        if monto > 0:
            self.__saldo += monto
            print(f"Depósito exitoso. Nuevo saldo: {self.__saldo}")

    def obtener_saldo(self):
        return self.__saldo

cuenta = CuentaBancaria(100)
cuenta.depositar(50)
# print(cuenta.__saldo) # Esto daría un error (AttributeError)
print(f"Saldo actual: {cuenta.obtener_saldo()}")
```

---

14. Pilares de la POO

14.1. Herencia: Clases padre e hijo, `super()`

Teoría: La **Herencia** permite que una clase (clase hija o subclase) adquiera los atributos y métodos de otra clase (clase padre o superclase). Esto promueve la reutilización de código.

- La clase hija se define como `class Hija(Padre):`.

- La función `super()` se utiliza para llamar a métodos de la clase padre, especialmente el constructor `__init__`.

Ejemplo práctico:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def presentarse(self):
        print(f"Soy {self.nombre} y tengo {self.edad} años.")

class Estudiante(Persona): # Estudiante hereda de Persona
    def __init__(self, nombre, edad, carrera):
        super().__init__(nombre, edad) # Llama al constructor de Persona
        self.carrera = carrera

    # Sobreescritura del método presentarse
    def presentarse(self):
        print(f"Soy {self.nombre}, estudio {self.carrera} y tengo
{self.edad} años.")

estudiante = Estudiante("Laura", 20, "Ingeniería")
estudiante.presentarse()
```

14.2. Polimorfismo

Teoría: El **Polimorfismo** (muchas formas) permite que objetos de diferentes clases respondan al mismo método de manera diferente. En el ejemplo anterior, tanto `Persona` como `Estudiante` tienen un método `presentarse()`, pero cada uno lo ejecuta de forma distinta.

14.3. Abstracción

Teoría: La **Abstracción** se centra en mostrar solo la información esencial al usuario y ocultar los detalles de implementación complejos. Cuando usas el método `depositar()` de la `CuentaBancaria`, no necesitas saber cómo se actualiza internamente el saldo; solo necesitas saber que el dinero se deposita.

Ejercicio 14.3: Crea una clase `Figura` con un método `area()` que devuelva 0. Luego, crea una clase `Rectangulo` que herede de `Figura` y sobrescriba el método `area()` para calcular el área real.

Solución 14.3:

```
# Solución al Ejercicio 14.3
class Figura:
    def area(self):
        return 0

class Rectangulo(Figura):
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def area(self):
        return self.ancho * self.alto

rect = Rectangulo(10, 5)
print(f"El área del rectángulo es: {rect.area()}")
```

---

15. Decoradores y Generadores

15.1. Introducción a los decoradores

Teoría: Un **decorador** es una función que toma otra función como argumento, le añade funcionalidad y la devuelve, sin modificar explícitamente el código de la función original. Se utilizan con el símbolo `@` antes de la definición de la función.

Ejemplo práctico (Simulador de tiempo de ejecución):

```python
import time

def medir_tiempo(func):
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = func(*args, **kwargs)
        fin = time.time()
        print(f"La función {func.__name__} tardó {fin - inicio:.4f}
segundos.")
        return resultado
    return wrapper

@medir_tiempo
def tarea_lenta(n):
    suma = 0
    for i in range(n):
        suma += i
    return suma

resultado = tarea_lenta(1000000)
print(f"Resultado de la tarea: {resultado}")
```

15.2. Comprensión de listas, diccionarios y conjuntos

Teoría: Las **comprensiones** (list, dict, set comprehensions) son una forma concisa y eficiente de crear colecciones a partir de otras colecciones.

Ejemplo práctico:

```python
numeros = [1, 2, 3, 4, 5]

# Comprensión de lista: crear una lista con el cuadrado de cada número
cuadrados = [n ** 2 for n in numeros]
print(f"Cuadrados: {cuadrados}") # [1, 4, 9, 16, 25]

# Comprensión de diccionario: crear un diccionario con {número: cuadrado}
diccionario_cuadrados = {n: n ** 2 for n in numeros if n % 2 == 0}
print(f"Diccionario de pares: {diccionario_cuadrados}") # {2: 4, 4: 16}
```

15.3. Generadores y la palabra clave `yield`

Teoría: Un **generador** es una función que devuelve un iterador. En lugar de usar `return`, usa la palabra clave `yield`. La principal ventaja es que los generadores no almacenan todos los resultados en la memoria a la vez, sino que los generan "sobre la marcha" (uno a la vez), lo que los hace muy eficientes para trabajar con grandes conjuntos de datos.

Ejemplo práctico:

```
def generador_pares(limite):
    n = 0
    while n <= limite:
        yield n # Pausa la función y devuelve n
        n += 2

# Usar el generador
for par in generador_pares(10):
    print(par)
```

---

16. Expresiones Regulares (`re`)

16.1. Sintaxis básica de expresiones regulares

Teoría: Las **Expresiones Regulares (Regex)** son secuencias de caracteres que forman un patrón de búsqueda. Se utilizan para encontrar y manipular texto basado en patrones complejos.

Símbolo	Descripción	Ejemplo
`.`	Cualquier carácter (excepto salto de línea)	`a.c` (coincide con `abc`, `aXc`)
`*`	Cero o más repeticiones del anterior	`a*` (coincide con `, a, aa, aaa`)
`+`	Una o más repeticiones del anterior	`a+` (coincide con `a`, `aa`, `aaa`)
`?`	Cero o una repetición del anterior	`a?` (coincide con `, a`)
`[abc]`	Cualquiera de los caracteres dentro de los corchetes	`[aeiou]` (coincide con cualquier vocal)
`[a-z]`	Rango de caracteres	`[0-9]` (coincide con cualquier dígito)
`\d`	Coincide con cualquier dígito (equivalente a `[0-9]`)	`\d{4}` (coincide con 4 dígitos)
`^`	Inicio de la cadena	`^Hola` (coincide si la cadena empieza con "Hola")
`$`	Fin de la cadena	`Mundo$` (coincide si la cadena termina con "Mundo")

16.2. Uso del módulo `re` para búsqueda y sustitución

Teoría: El módulo estándar de Python para trabajar con expresiones regulares es `re`.

Funciones clave:

- `re.search(patron, texto)`: Busca la primera coincidencia del patrón en el texto.
- `re.findall(patron, texto)`: Devuelve una lista de todas las coincidencias.
- `re.sub(patron, reemplazo, texto)`: Reemplaza todas las coincidencias del patrón.

Ejemplo práctico:

```python
import re

texto = "Mi correo es ana@ejemplo.com y el de Juan es juan@dominio.net"
patron_correo = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"

# Encontrar todos los correos
correos_encontrados = re.findall(patron_correo, texto)
print(f"Correos encontrados: {correos_encontrados}")

# Sustituir correos
texto_anonimizado = re.sub(patron_correo, "[CORREO OCULTO]", texto)
print(f"Texto anonimizado: {texto_anonimizado}")
```

Ejercicio 16.2: Usa el módulo `re` para encontrar todos los números de teléfono de 8 dígitos (ej. 12345678) en la siguiente cadena de texto.

```python
texto_tel = "Llama al 12345678 o al 98765432. El número corto es 123."
```

Solución 16.2:

```python
# Solución al Ejercicio 16.2
import re

texto_tel = "Llama al 12345678 o al 98765432. El número corto es 123."
patron_telefono = r"\d{8}" # Coincide con exactamente 8 dígitos

telefonos = re.findall(patron_telefono, texto_tel)
print(f"Teléfonos de 8 dígitos encontrados: {telefonos}")
```

---

17. Evaluación Parcial del Nivel Avanzado

Instrucciones: Responde las siguientes preguntas y resuelve los ejercicios de codificación.

Cuestionario de Opción Múltiple

1. En POO, ¿cuál es el método especial que se llama automáticamente al crear un objeto? a) `__new__` b) `__call__` c) `__init__` d) `__str__`

2. ¿Cuál es la principal ventaja de usar un generador en lugar de una función que devuelve una lista completa? a) Es más rápido en todos los casos. b) Consume menos memoria al generar valores "sobre la marcha". c) Permite la herencia de clases. d) No requiere la palabra clave `return`.

3. ¿Qué hace el operador `+` en una expresión regular? a) Coincide con cero o más repeticiones. b) Coincide con una o más repeticiones. c) Coincide con el final de la cadena. d) Coincide con cualquier carácter.

Ejercicio de Diseño de Clases

Diseño de un Sistema de Gestión de Biblioteca: Diseña un sistema de clases para una biblioteca que cumpla con los siguientes requisitos:

1. Una clase base `Material` con atributos `titulo` y `autor`.
2. Una clase `Libro` que herede de `Material` y añada el atributo `isbn`.
3. Una clase `Revista` que herede de `Material` y añada el atributo `numero_revista`.
4. Implementa un método `mostrar_info()` en cada clase para demostrar el polimorfismo, donde cada uno imprima la información específica de su tipo.

Soluciones a la Evaluación

Cuestionario:

1. c) `__init__`
2. b) Consume menos memoria al generar valores "sobre la marcha".
3. b) Coincide con una o más repeticiones.

Ejercicio de Diseño de Clases:

```python
# Solución al Ejercicio de Diseño de Clases
class Material:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def mostrar_info(self):
        print(f"Título: {self.titulo}, Autor: {self.autor}")

class Libro(Material):
    def __init__(self, titulo, autor, isbn):
        super().__init__(titulo, autor)
        self.isbn = isbn

    def mostrar_info(self):
        # Sobreescribe el método de la clase padre
        print("--- LIBRO ---")
        super().mostrar_info() # Llama al método del padre para reutilizar
código
        print(f"ISBN: {self.isbn}")

class Revista(Material):
    def __init__(self, titulo, autor, numero_revista):
        super().__init__(titulo, autor)
        self.numero_revista = numero_revista

    def mostrar_info(self):
        # Sobreescribe el método de la clase padre
        print("--- REVISTA ---")
        super().mostrar_info()
        print(f"Número: {self.numero_revista}")

# Prueba de Polimorfismo
item1 = Libro("Cien Años de Soledad", "García Márquez", "978-0307474728")
item2 = Revista("National Geographic", "Varios", 500)

item1.mostrar_info()
print("-" * 20)
item2.mostrar_info()
```

MÓDULO 4: NIVEL EXPERTO - Aplicaciones Reales y Librerías Esenciales

¡Has llegado al nivel Experto! En este módulo, verás cómo Python se aplica en el mundo real, utilizando las librerías más populares para tareas de automatización, análisis de datos, visualización y desarrollo de aplicaciones.

18. Automatización y Sistema Operativo (`os`, `datetime`)

18.1. Interacción con el sistema de archivos (`os`, `pathlib`)

Teoría: El módulo estándar `os` (Operating System) proporciona funciones para interactuar con el sistema operativo, como crear carpetas, renombrar archivos o ejecutar comandos. El módulo `pathlib` ofrece una alternativa más moderna y orientada a objetos para manejar rutas de archivos.

Ejemplo práctico (Creación de carpetas):

```python
import os
from pathlib import Path

# 1. Usando el módulo os
try:
    os.mkdir("reportes_os")
    print("Carpeta 'reportes_os' creada con os.")
except FileExistsError:
    print("Carpeta 'reportes_os' ya existe.")

# 2. Usando el módulo pathlib (más moderno)
ruta = Path("reportes_pathlib")
try:
    ruta.mkdir()
    print(f"Carpeta '{ruta}' creada con pathlib.")
except FileExistsError:
    print(f"Carpeta '{ruta}' ya existe.")
```

Explicación línea por línea:

- `import os` y `from pathlib import Path`: Importan los módulos necesarios.

- `os.mkdir(...)`: Intenta crear un directorio.
- `try...except FileExistsError`: Maneja el error si la carpeta ya existe.
- `ruta = Path(...)`: Crea un objeto `Path` que representa la ruta.
- `ruta.mkdir()`: Crea el directorio usando el método del objeto `Path`.

18.2. Manejo de fechas y tiempo (`datetime`)

Teoría: El módulo `datetime` permite manipular fechas y horas de forma sencilla y precisa. Es fundamental para registrar eventos, calcular duraciones o programar tareas.

Ejemplo práctico (Calcular edad):

```
from datetime import date, timedelta

hoy = date.today()
fecha_nacimiento = date(1995, 10, 25)

# Cálculo de la diferencia
diferencia = hoy - fecha_nacimiento
edad_en_dias = diferencia.days

print(f"Hoy es: {hoy}")
print(f"Días vividos: {edad_en_dias}")

# Sumar 10 días a la fecha actual
fecha_futura = hoy + timedelta(days=10)
print(f"Fecha dentro de 10 días: {fecha_futura}")
```

Ejercicio 18.2: Escribe un programa que solicite al usuario una fecha en formato `AAAA-MM-DD` y calcule cuántos días faltan para esa fecha (asumiendo que es en el futuro).

Solución 18.2:

```python
# Solución al Ejercicio 18.2
from datetime import date, datetime

fecha_str = input("Ingresa una fecha futura (AAAA-MM-DD): ")

try:
    fecha_objetivo = datetime.strptime(fecha_str, "%Y-%m-%d").date()
    hoy = date.today()

    diferencia = fecha_objetivo - hoy

    if diferencia.days > 0:
        print(f"Faltan {diferencia.days} días para la fecha
{fecha_objetivo}.")
    else:
        print("La fecha ingresada ya pasó o es hoy.")

except ValueError:
    print("Error: Formato de fecha incorrecto. Usa AAAA-MM-DD.")
```

---

19. Análisis de Datos con Pandas y NumPy

19.1. Introducción a `NumPy` y arrays

Teoría: NumPy (Numerical Python) es la librería fundamental para la computación científica en Python. Proporciona el objeto `ndarray` (array N-dimensional), que es mucho más rápido y eficiente que las listas de Python para operaciones matemáticas con grandes volúmenes de datos.

Ejemplo práctico:

```python
import numpy as np

# Creación de un array
array_a = np.array([1, 2, 3, 4, 5])
array_b = np.array([10, 20, 30, 40, 50])

# Operaciones vectorizadas (se aplican a cada elemento)
suma = array_a + array_b
multiplicacion = array_a * 2

print(f"Array A: {array_a}")
print(f"Suma de A y B: {suma}")
print(f"Multiplicación de A por 2: {multiplicacion}")
```

19.2. Introducción a `Pandas` y DataFrames

Teoría: Pandas es la librería más popular para el análisis y manipulación de datos. Introduce dos estructuras de datos clave:

- **Series:** Un array unidimensional etiquetado (similar a una columna de una hoja de cálculo).
- **DataFrame:** Una estructura de datos bidimensional etiquetada con columnas de tipos potencialmente diferentes (similar a una hoja de cálculo o tabla SQL).

Ejemplo práctico (Creación de un DataFrame):

```python
import pandas as pd

datos = {
    'Nombre': ['Ana', 'Beto', 'Carla', 'David'],
    'Edad': [25, 30, 22, 40],
    'Ciudad': ['Lima', 'Bogotá', 'Santiago', 'Lima']
}

df = pd.DataFrame(datos)

print("DataFrame completo:")
print(df)

print("\nSolo la columna 'Edad':")
print(df['Edad'])

print("\nFiltrar por Ciudad 'Lima':")
print(df[df['Ciudad'] == 'Lima'])
```

19.3. Carga y exploración básica de datos (CSV, Excel)

Teoría: Pandas facilita la lectura de datos desde diversos formatos, siendo el CSV (Comma Separated Values) el más común.

Ejemplo práctico (Simulado de lectura de CSV): Asumiendo que existe un archivo `ventas.csv`.

```python
# df_ventas = pd.read_csv('ventas.csv')
# print(df_ventas.head()) # Muestra las primeras 5 filas

# Simulación de exploración
print("\nEstadísticas descriptivas de las edades:")
print(df['Edad'].describe())
```

Ejercicio 19.3: Usando el DataFrame `df` del ejemplo anterior, calcula la edad promedio de todas las personas.

Solución 19.3:

```python
# Solución al Ejercicio 19.3
# Se utiliza el método .mean() de la Serie 'Edad'
edad_promedio = df['Edad'].mean()
print(f"La edad promedio es: {edad_promedio:.2f} años.")
```

---

20. Visualización de Datos con Matplotlib

20.1. Creación de gráficos básicos (líneas, barras, dispersión)

Teoría: Matplotlib es la librería de visualización más popular de Python. Permite crear gráficos estáticos, animados e interactivos. La interfaz más común es `pyplot`, que imita la forma de trabajar de MATLAB.

Ejemplo práctico (Gráfico de barras):

```python
import matplotlib.pyplot as plt
import pandas as pd

# Datos de ejemplo
productos = ['A', 'B', 'C', 'D']
ventas = [150, 300, 100, 450]

# Crear el gráfico de barras
plt.figure(figsize=(8, 5)) # Define el tamaño de la figura
plt.bar(productos, ventas, color='skyblue')

# Añadir etiquetas y título
plt.title('Ventas por Producto')
plt.xlabel('Producto')
plt.ylabel('Ventas (USD)')

# Guardar la figura (se simula la ejecución y guardado)
# plt.savefig('ventas_por_producto.png')
# plt.show() # Muestra el gráfico (no se ejecuta en este entorno)

print("Gráfico de barras generado (simulado).")
```

20.2. Personalización de gráficos

Teoría: Matplotlib ofrece un control granular sobre cada elemento del gráfico (colores, fuentes, leyendas, rejillas, etc.).

Ejercicio 20.2: Crea un gráfico de línea simple que muestre la evolución de la temperatura diaria durante una semana.

Solución 20.2:

```
# Solución al Ejercicio 20.2
# import matplotlib.pyplot as plt # Ya importado arriba

dias = ['Lun', 'Mar', 'Mié', 'Jue', 'Vie', 'Sáb', 'Dom']
temperaturas = [20, 22, 21, 25, 24, 26, 23]

plt.figure(figsize=(7, 4))
plt.plot(dias, temperaturas, marker='o', linestyle='--', color='red')
plt.title('Temperatura Semanal')
plt.xlabel('Día')
plt.ylabel('Temperatura (°C)')
plt.grid(True)
# plt.show()
print("Gráfico de línea generado (simulado).")
```

---

21. Desarrollo Web y APIs (`requests`)

21.1. Introducción a la comunicación HTTP

Teoría: La mayoría de las aplicaciones web se comunican a través del protocolo **HTTP** (Hypertext Transfer Protocol). Una **API** (Application Programming Interface) es un conjunto de reglas que permite que diferentes programas de software se comuniquen entre sí. Python es excelente para consumir APIs.

21.2. Consumo de APIs con el módulo `requests` (GET, POST)

Teoría: La librería `requests` simplifica enormemente la realización de solicitudes HTTP. Es la herramienta estándar para interactuar con APIs externas.

Métodos HTTP comunes:

- **GET:** Solicita datos de un recurso específico (el más común para obtener información).
- **POST:** Envía datos para crear o actualizar un recurso.

Ejemplo práctico (Solicitud GET a una API pública):

```
import requests

# URL de una API pública de prueba (JSONPlaceholder)
url = "https://jsonplaceholder.typicode.com/posts/1"

try:
    # Realizar la solicitud GET
    respuesta = requests.get(url)

    # Verificar si la solicitud fue exitosa (código 200)
    if respuesta.status_code == 200:
        datos = respuesta.json() # Convierte la respuesta JSON a un
diccionario de Python
        print("\nDatos del Post 1:")
        print(f"Título: {datos['title']}")
        print(f"Cuerpo: {datos['body'][:50]}...")
    else:
        print(f"Error al obtener datos. Código de estado:
{respuesta.status_code}")

except requests.exceptions.RequestException as e:
    print(f"Error de conexión: {e}")
```

21.3. Manejo de datos JSON

Teoría: JSON (JavaScript Object Notation) es el formato de intercambio de datos más común en las APIs web. La función `respuesta.json()` de `requests` convierte automáticamente el texto JSON recibido en un objeto de Python (normalmente un diccionario o una lista).

Ejercicio 21.3: Realiza una solicitud GET a la API de prueba `https://jsonplaceholder.typicode.com/users/5` y extrae el nombre de usuario (`username`) y el correo electrónico (`email`) del usuario con ID 5.

Solución 21.3:

```python
# Solución al Ejercicio 21.3
# import requests # Ya importado arriba

url_usuario = "https://jsonplaceholder.typicode.com/users/5"
respuesta_usuario = requests.get(url_usuario)

if respuesta_usuario.status_code == 200:
    datos_usuario = respuesta_usuario.json()

    username = datos_usuario['username']
    email = datos_usuario['email']

    print(f"\nUsuario ID 5:")
    print(f"Nombre de Usuario: {username}")
    print(f"Correo Electrónico: {email}")
else:
    print("Error al obtener datos del usuario.")
```

---

22. Introducción a la Inteligencia Artificial (IA)

22.1. Conceptos básicos de Machine Learning

Teoría: El **Machine Learning (ML)** es una rama de la IA que permite a los sistemas aprender de los datos, identificar patrones y tomar decisiones con mínima intervención humana.

Tipos de ML:

- **Aprendizaje Supervisado:** El modelo se entrena con datos etiquetados (ej. predecir el precio de una casa a partir de sus características).
- **Aprendizaje No Supervisado:** El modelo busca patrones en datos sin etiquetar (ej. agrupar clientes por comportamiento).
- **Aprendizaje por Refuerzo:** El modelo aprende a tomar decisiones a través de prueba y error en un entorno.

22.2. Ejemplo práctico con `scikit-learn` (ej. regresión lineal simple)

Teoría: Scikit-learn es la librería más popular de Python para Machine Learning. Proporciona herramientas sencillas y eficientes para tareas de clasificación, regresión,

clustering y reducción de dimensionalidad.

Ejemplo práctico (Regresión Lineal Simple):

```
import numpy as np
from sklearn.linear_model import LinearRegression

# 1. Preparar los datos (simulados)
# X: Horas de estudio (variable independiente)
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
# Y: Nota obtenida (variable dependiente)
Y = np.array([5, 7, 9, 11, 13])

# 2. Crear y entrenar el modelo
modelo = LinearRegression()
modelo.fit(X, Y)

# 3. Realizar una predicción
horas_a_predecir = np.array([[6]])
prediccion = modelo.predict(horas_a_predecir)

print(f"\nPredicción de nota para 6 horas de estudio: {prediccion[0]:.2f}")
print(f"Coeficiente (pendiente): {modelo.coef_[0]:.2f}")
```

Explicación línea por línea:

- `reshape(-1, 1)`: Transforma el array `X` en un formato de columna, requerido por `scikit-learn`.
- `LinearRegression()`: Crea una instancia del modelo de regresión lineal.
- `modelo.fit(X, Y)`: Entrena el modelo con los datos de entrada (`X`) y los resultados esperados (`Y`).
- `modelo.predict(...)`: Utiliza el modelo entrenado para predecir un nuevo valor.

Ejercicio 22.2: Usando el modelo entrenado, predice la nota que se obtendría con 7.5 horas de estudio.

Solución 22.2:

```python
# Solución al Ejercicio 22.2
# import numpy as np # Ya importado arriba
# from sklearn.linear_model import LinearRegression # Ya importado arriba

horas_a_predecir_2 = np.array([[7.5]])
prediccion_2 = modelo.predict(horas_a_predecir_2)

print(f"Predicción de nota para 7.5 horas de estudio:
{prediccion_2[0]:.2f}")
```

---

23. Proyecto Final Integrador

23.1. Descripción del proyecto: Analizador de Datos de Clima y Generador de Reporte

Objetivo: Aplicar los conocimientos de POO, manejo de APIs, análisis de datos con Pandas y manejo de archivos para crear una herramienta que obtenga datos de clima de una API pública, los procese y genere un informe final.

Requisitos del Proyecto:

1. **Clase `ClimaAPI` (POO y `requests`):**
    - Debe ser una clase que maneje la conexión con una API de clima (se puede simular o usar una gratuita como OpenWeatherMap, aunque para este ejercicio se simulará la respuesta para evitar dependencias de claves API).
    - Debe tener un método `obtener_datos(ciudad)` que devuelva datos simulados de temperatura, humedad y velocidad del viento para una ciudad.
    - Debe manejar posibles errores de conexión (`try...except`).
2. **Función `procesar_datos` (`pandas`):**
    - Debe recibir los datos de la API (como una lista de diccionarios).
    - Debe crear un `DataFrame` de Pandas.
    - Debe calcular la temperatura promedio, la humedad máxima y la velocidad del viento mínima.

3. **Función `generar_reporte` (Manejo de Archivos):**
    - Debe recibir los resultados del procesamiento.
    - Debe escribir un archivo de texto (`reporte_clima.txt`) con el siguiente formato:

```
REPORTE DE CLIMA - [Fecha Actual]
---------------------------------
Ciudad Analizada: [Nombre de la Ciudad]
Temperatura Promedio: [Valor] °C
Humedad Máxima: [Valor] %
Velocidad del Viento Mínima: [Valor] km/h
```

4. **Función Principal (`main`):**
    - Debe orquestar la ejecución: crear la instancia de la clase, obtener los datos, procesarlos y generar el reporte.

23.2. Criterios de Evaluación

Criterio	Ponderación	Descripción
Estructura POO	30%	Correcta implementación de la clase `ClimaAPI` con atributos y métodos.
Manejo de Datos	30%	Uso correcto de `pandas` para la creación del DataFrame y el cálculo de estadísticas.
Manejo de Archivos	20%	Generación correcta del archivo de reporte con el formato especificado.
Manejo de Errores	10%	Inclusión de bloques `try...except` para manejo de errores de API o de datos.
Legibilidad	10%	Código limpio, comentado y siguiendo las buenas prácticas de Python (PEP 8).

APÉNDICE

A.1. Glosario de términos

Término	Definición	Módulo/Nivel
Variable	Un nombre simbólico que hace referencia a un valor almacenado en la memoria.	Básico
Mutable	Tipo de dato cuyo valor puede ser modificado después de su creación (ej. `list`, `dict`).	Básico/Intermedio
Inmutable	Tipo de dato cuyo valor no puede ser modificado después de su creación (ej. `tuple`, `str`).	Básico/Intermedio
Función	Bloque de código organizado y reutilizable que realiza una tarea específica.	Intermedio
Excepción	Un evento que interrumpe el flujo normal de un programa (error en tiempo de ejecución).	Intermedio
Clase	Plantilla o plano para crear objetos, definiendo sus atributos y métodos.	Avanzado
Objeto	Una instancia específica de una clase.	Avanzado
Herencia	Mecanismo de POO donde una clase adquiere propiedades de otra.	Avanzado
Decorador	Función que modifica o mejora otra función sin cambiar su código fuente.	Avanzado
DataFrame	Estructura de datos bidimensional etiquetada de la librería Pandas.	Experto
API	Interfaz de Programación de Aplicaciones; permite la comunicación entre sistemas.	Experto

A.2. Recursos adicionales y comunidad Python

- **Documentación Oficial de Python:** El recurso más completo y actualizado.
  - https://docs.python.org/es/3/

- **PEP 8 - Guía de Estilo de Código Python:** Estándares de formato para escribir código legible.
    - https://www.python.org/dev/peps/pep-0008/
- **Comunidades en Español:**
    - **Python España:** https://www.python.es/
    - **PyCon Latam:** https://pycon.lat/
- **Librerías Clave:**
    - **Pandas:** https://pandas.pydata.org/
    - **NumPy:** https://numpy.org/
    - **Requests:** https://requests.readthedocs.io/en/latest/

A.3. Soluciones a los ejercicios propuestos

(Nota: Las soluciones a los ejercicios de cada sección y las evaluaciones parciales se encuentran inmediatamente después de cada ejercicio o al final de cada módulo, respectivamente, a lo largo del manual.)

SOLUCIÓN AL PROYECTO FINAL INTEGRADOR

```
import requests
import pandas as pd
from datetime import date
import random

# 1. Clase ClimaAPI (Simulada)
class ClimaAPI:
    """Clase para simular la obtención de datos de clima de una API."""

    def __init__(self, ciudad):
        self.ciudad = ciudad
        self.datos_simulados = [
            {"temp": 25.5, "humedad": 70, "viento": 15},
            {"temp": 28.0, "humedad": 65, "viento": 10},
            {"temp": 22.1, "humedad": 75, "viento": 20},
            {"temp": 26.3, "humedad": 68, "viento": 12},
        ]

    def obtener_datos(self):
        """Simula la llamada a la API y devuelve los datos."""
        print(f"-> Obteniendo datos simulados para {self.ciudad}...")
        try:
            # Simulación de un posible error de conexión
            if random.random() < 0.1: # 10% de probabilidad de error
                raise requests.exceptions.RequestException("Error de
conexión simulado.")

            return self.datos_simulados

        except requests.exceptions.RequestException as e:
            print(f"ERROR DE API: {e}")
            return None

# 2. Función procesar_datos
def procesar_datos(datos):
    """Procesa los datos de clima usando Pandas."""
    if not datos:
        return None

    df = pd.DataFrame(datos)

    # Cálculo de estadísticas
```

```
    temp_promedio = df['temp'].mean()
    humedad_maxima = df['humedad'].max()
    viento_minimo = df['viento'].min()

    return {
        "temp_promedio": temp_promedio,
        "humedad_maxima": humedad_maxima,
        "viento_minimo": viento_minimo
    }

# 3. Función generar_reporte
def generar_reporte(ciudad, resultados):
    """Genera un archivo de reporte de clima."""
    if not resultados:
        print("No se pudo generar el reporte debido a la falta de datos.")
        return

    nombre_archivo = "reporte_clima.txt"
    fecha_actual = date.today().strftime("%Y-%m-%d")

    contenido = f"""
REPORTE DE CLIMA - {fecha_actual}
---------------------------------
Ciudad Analizada: {ciudad}
Temperatura Promedio: {resultados['temp_promedio']:.2f} °C
Humedad Máxima: {resultados['humedad_maxima']} %
Velocidad del Viento Mínima: {resultados['viento_minimo']} km/h
"""

    with open(nombre_archivo, "w") as f:
        f.write(contenido.strip())

    print(f"\nReporte generado exitosamente en '{nombre_archivo}'")

# 4. Función Principal (main)
def main():
    """Función principal que orquesta el proyecto."""
    ciudad_a_analizar = "Buenos Aires"

    # 1. Obtener datos
    api = ClimaAPI(ciudad_a_analizar)
    datos_clima = api.obtener_datos()

    # 2. Procesar datos
    resultados_procesados = procesar_datos(datos_clima)
```

```
    # 3. Generar reporte
    generar_reporte(ciudad_a_analizar, resultados_procesados)

if __name__ == "__main__":
    main()
```