Introducción a la Programación con Python

Bibliografía Recomendada

- "Python Crash Course" por Eric Matthes
- "Automate the Boring Stuff with Python" por Al Sweigart
- "Think Python: How to Think Like a Computer Scientist" por Allen B. Downey
- "Python Cookbook" por David Beazley y Brian K. Jones
- Documentación oficial de Python (python.org)

UNIDAD 1: FUNDAMENTOS DEL PENSAMIENTO COMPUTACIONAL

SEMANA 1: INTRODUCCIÓN AL PENSAMIENTO COMPUTACIONAL

¿Qué es un algoritmo?

Un algoritmo es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema

¿Qué es la programación?

La programación es el proceso de crear un conjunto de instrucciones que le indican a una computadora cómo realizar una tarea específica. Es similar a escribir una receta detallada que una máquina puede seguir para lograr un resultado deseado. La programación implica:

Comunicación con máquinas: Utilizamos lenguajes de programación para comunicarnos con las computadoras.

Resolución de problemas: Identificamos problemas y creamos soluciones sistemáticas.

Traducción de ideas: Convertimos ideas y procesos del mundo real en instrucciones precisas que una computadora puede ejecutar.

Creatividad estructurada: Combinamos pensamiento creativo con reglas lógicas y sintácticas estrictas.

La programación no es solo escribir código; es una forma de pensar y resolver problemas de manera sistemática y lógica. Python, el lenguaje que aprenderemos en este curso, es conocido por su legibilidad y simplicidad, lo que lo convierte en una excelente elección para principiantes.

Resolución de problemas y pensamiento algorítmico

El pensamiento algorítmico es un enfoque sistemático para resolver problemas siguiendo pasos definidos. Este proceso incluye:

1. **Definición clara del problema**: Comprender exactamente qué necesita resolverse.
2. **Identificación de entradas y salidas**: Determinar qué información tenemos y qué resultado queremos obtener.
3. **Desarrollo de un plan**: Crear una estrategia paso a paso para transformar las entradas en las salidas deseadas.
4. **Implementación de la solución**: Ejecutar el plan de manera precisa.
5. **Verificación y refinamiento**: Comprobar que la solución funciona correctamente y mejorarla si es necesario.

Ejercicios Ejemplo

Resolver el problema de calcular el promedio de calificaciones.

- **Problema**: Calcular el promedio de calificaciones de un estudiante.
- **Entradas**: Lista de calificaciones (por ejemplo: 8, 9, 7, 6, 8).
- **Salida**: El promedio de las calificaciones.
- **Plan**:
    1. Sumar todas las calificaciones
    2. Contar el número total de calificaciones

3. Dividir la suma por el número total
- **Implementación**: (8 + 9 + 7 + 6 + 8) ÷ 5 = 7.6
- **Verificación**: El resultado es un número entre 0 y 10, lo que es razonable para un promedio de calificaciones.

Descomposición de problemas complejos

La descomposición es la habilidad de dividir un problema complejo en partes más pequeñas y manejables. Beneficios:

- Facilita abordar problemas que inicialmente parecen abrumadores
- Permite trabajar en componentes individuales de forma independiente
- Facilita la identificación de soluciones reutilizables
- Mejora la comprensión del problema en su totalidad

Crear una aplicación para una biblioteca.

En lugar de intentar programar toda la aplicación de una vez, podemos descomponerla en subsistemas:

- Sistema de gestión de libros (agregar, eliminar, editar información)
- Sistema de préstamos (registrar préstamos, devoluciones, multas)
- Sistema de usuarios (registro, información de miembros)
- Interfaz de búsqueda de libros
- Sistema de informes y estadísticas

Cada subsistema puede descomponerse aún más. Por ejemplo, el sistema de gestión de libros podría incluir:

- Función para agregar nuevos libros
- Función para eliminar libros
- Función para editar información de libros
- Función para consultar el estado de un libro

Reconocimiento de patrones

El reconocimiento de patrones es la capacidad de identificar similitudes o tendencias en diferentes problemas o datos. Esto nos permite:

- Reutilizar soluciones existentes
- Predecir comportamientos o resultados
- Clasificar y categorizar información
- Optimizar nuestras soluciones

Ejemplos de patrones en programación:

1. **Patrones de iteración**: Recorrer listas, procesar cada elemento de un conjunto de datos
2. **Patrones de acumulación**: Sumar valores, concatenar textos

3. **Patrones de filtrado**: Seleccionar elementos que cumplan ciertos criterios
4. **Patrones de transformación**: Convertir datos de un formato a otro

Caso práctico: Al analizar diferentes juegos de mesa, podemos identificar el patrón común de "turnos", donde cada jugador realiza acciones en un orden específico. Este patrón puede aplicarse en múltiples juegos y simulaciones.

Abstracción y modelado

La abstracción es el proceso de identificar los aspectos esenciales de un problema y eliminar los detalles innecesarios. El modelado utiliza la abstracción para crear representaciones simplificadas pero funcionales de sistemas complejos.

Niveles de abstracción:

- **Alta abstracción**: Descripción general del problema (ej. "Sistema de biblioteca")
- **Media abstracción**: Componentes principales (ej. "Sistema de préstamos")
- **Baja abstracción**: Detalles específicos (ej. "Cálculo de fecha de devolución")

Ejemplo: Modelar un estudiante en un sistema escolar.

En lugar de incluir toda la información posible sobre un estudiante (altura, peso, color favorito, etc.), nos enfocamos en lo relevante para el sistema escolar:

- Nombre
- Número de identificación
- Calificaciones
- Cursos inscritos
- Asistencia

La abstracción nos permite crear modelos mentales claros y manejar la complejidad de forma efectiva.

Ejercicios prácticos sin computadora (pensamiento computacional desconectado)

1. **Instrucciones para hacer un mate**:
    - Todos escriben instrucciones precisas para hacer un mate.
    - Otro estudiante debe seguir exactamente esas instrucciones.
    - Identificar ambigüedades y mejorar las instrucciones.
2. **Juego de patrones**:
    - Presentar secuencias numéricas con patrones (ej. 2, 4, 6, 8, ...)
    - Los estudiantes deben identificar el patrón y predecir los siguientes números.
    - Discutir cómo reconocer patrones ayuda en programación.
3. **Descomposición de tareas cotidianas**:
    - Seleccionar una actividad común (preparar una comida, organizar un evento)
    - Descomponer en pasos y sub-tareas específicas.
    - Organizar las tareas en un diagrama jerárquico.

SEMANA 2: ALGORITMOS Y LÓGICA

Conceptos básicos de algoritmos

Un algoritmo es un conjunto finito de instrucciones definidas, ordenadas y no ambiguas que permiten solucionar un problema o realizar una tarea específica. Características esenciales:

Finitud: Debe terminar después de un número finito de pasos.

Definición: Cada paso debe estar claramente definido y ser preciso.

Entrada: Puede tener cero o más entradas.

Salida: Debe producir al menos una salida.

Efectividad: Debe ser lo suficientemente básico para que una persona pueda realizarlo usando papel y lápiz.

Analogía: Una receta de cocina es similar a un algoritmo:

- Tiene ingredientes (entradas)
- Incluye pasos específicos a seguir
- Produce un resultado (la comida terminada)
- Tiene un final claro

Ejemplos cotidianos de algoritmos:

- Instrucciones para llegar a un destino
- Tutorial para armar un mueble
- Pasos para resolver un cubo de Rubik
- Receta de cocina

Representación de algoritmos: pseudocódigo y diagramas de flujo

Pseudocódigo

El pseudocódigo es una descripción informal de un algoritmo que utiliza una mezcla de lenguaje natural y convenciones de programación, pero sin seguir la sintaxis estricta de un lenguaje específico.

Ventajas:

- Fácil de entender para humanos
- Independiente del lenguaje de programación
- Permite enfocarse en la lógica sin preocuparse por la sintaxis

Ejemplo de pseudocódigo para calcular el promedio:

```
INICIO
  Declarar variables: suma, contador, promedio
  Asignar suma = 0
  Asignar contador = 0

  MIENTRAS haya más números para leer HACER
    Leer número
    Sumar número a suma
    Incrementar contador en 1
  FIN MIENTRAS

  SI contador > 0 ENTONCES
    Calcular promedio = suma / contador
    Mostrar promedio
  SINO
    Mostrar "No hay números para promediar"
  FIN SI
FIN
```

Diagramas de flujo

Los diagramas de flujo representan algoritmos gráficamente mediante símbolos conectados con flechas que indican el flujo de ejecución.

Símbolos comunes:

- **Óvalo**: Inicio/Fin
- **Rectángulo**: Proceso/Acción
- **Rombo**: Decisión
- **Paralelogramo**: Entrada/Salida
- **Flechas**: Dirección del flujo

Ventajas:

- Representación visual intuitiva
- Facilita la identificación de puntos de decisión y bucles
- Útil para comunicar algoritmos a personas no técnicas

Secuencia, selección y repetición

Estas son las tres estructuras fundamentales que conforman cualquier algoritmo:

1. Secuencia

Ejecución de instrucciones una después de otra, en orden. Es la estructura más simple.

Ejemplo (Pseudocódigo):

```
INICIO
  Leer lado
  Calcular área = lado * lado
  Mostrar área
FIN
```

2. Selección (Condicional)

Permite ejecutar diferentes acciones dependiendo de una condición. Las principales estructuras son:

- IF-THEN (Si-Entonces)
- IF-THEN-ELSE (Si-Entonces-Sino)
- SWITCH-CASE (Según-Caso)

Ejemplo (Pseudocódigo):

```
INICIO
  Leer edad

  SI edad >= 18 ENTONCES
    Mostrar "Eres mayor de edad"
  SINO
    Mostrar "Eres menor de edad"
  FIN SI
FIN
```

3. Repetición (Ciclos/Bucles)

Permite ejecutar una sección de código múltiples veces. Los principales tipos son:

- FOR (Para): Utilizado cuando conocemos el número de iteraciones.
- WHILE (Mientras): Ejecuta mientras una condición sea verdadera.
- DO-WHILE (Hacer-Mientras): Similar a WHILE, pero garantiza al menos una ejecución.

Ejemplo (Pseudocódigo):

```
INICIO
  Asignar suma = 0

  PARA i DESDE 1 HASTA 10 HACER
    suma = suma + i
  FIN PARA

  Mostrar "La suma de los primeros 10 números es:", suma
FIN
```

Análisis de algoritmos cotidianos

Analizar algoritmos nos permite entender cómo funcionan y cómo se pueden mejorar. Aspectos a considerar:

Corrección: ¿El algoritmo produce el resultado esperado?
Eficiencia: ¿Cuántos recursos (tiempo, memoria) utiliza?
Simplicidad: ¿Es fácil de entender y mantener?
Adaptabilidad: ¿Puede manejar diferentes tipos de entradas?

Ejemplo 1: Algoritmo para preparar café

1. Hervir agua
2. Colocar café en el filtro
3. Verter agua caliente sobre el café
4. Esperar a que el agua pase por el filtro
5. Servir en una taza

Análisis:

- **Corrección**: Produce café correctamente
- **Eficiencia**: Requiere varios pasos pero es razonablemente eficiente
- **Simplicidad**: Fácil de entender y seguir
- **Adaptabilidad**: Funciona con diferentes tipos/cantidades de café

Ejemplo 2: Búsqueda de un contacto en una lista telefónica ordenada alfabéticamente

1. Abrir la lista en la mitad
2. Verificar si el contacto está en esa página
3. Si es menor alfabéticamente, buscar en la primera mitad
4. Si es mayor alfabéticamente, buscar en la segunda mitad
5. Repetir hasta encontrar el contacto

Análisis:

- **Corrección**: Encuentra el contacto si existe
- **Eficiencia**: Muy eficiente (búsqueda binaria)
- **Simplicidad**: Conceptualmente simple pero requiere entender la búsqueda binaria
- **Adaptabilidad**: Funciona con listas de cualquier tamaño

Ejercicio práctico: Diseñar algoritmos para situaciones cotidianas

Ejercicio 1: Algoritmo para cambiar un neumático

Diseñar un algoritmo detallado que describa los pasos para cambiar un **neumático** pinchado. Incluir consideraciones de seguridad y herramientas necesarias.

Ejercicio 2: Sistema de préstamo bibliotecario

Crear un algoritmo que gestione el préstamo de un libro, incluyendo verificación de disponibilidad, registro del préstamo y cálculo de fecha de devolución.

Ejercicio 3: Cálculo de calificación final

Diseñar un algoritmo que calcule la calificación final de un estudiante basado en diferentes componentes (exámenes, tareas, participación) con distintos porcentajes.

Metodología de trabajo:

1. Trabajar en grupos de 2-3 estudiantes
2. Representar cada algoritmo tanto en pseudocódigo como en diagrama de flujo
3. Analizar cada algoritmo según corrección, eficiencia, simplicidad y adaptabilidad
4. Presentar al grupo y discutir posibles mejoras

UNIDAD 2: INTRODUCCIÓN A PYTHON

SEMANA 3: PRIMEROS PASOS CON PYTHON

Estructura básica de un programa Python

Todo programa en Python sigue una estructura general que facilita su lectura y ejecución. A diferencia de otros lenguajes de programación, Python utiliza la indentación (espacios al inicio de una línea) para definir bloques de código, lo que contribuye a su legibilidad.

Componentes básicos de un programa Python:

1. **Comentarios**: Líneas que comienzan con # y no son ejecutadas por el intérprete. Sirven para documentar el código.

```
# Esto es un comentario de una línea

"""
Esto es un comentario
de múltiples líneas (docstring)
"""
```

2. **Importación de módulos**: Permite utilizar funcionalidades adicionales disponibles en la biblioteca estándar de Python o en módulos externos.

```
import math  # Importa el módulo completo
from datetime import datetime  # Importa una clase específica
```

3. **Definición de variables**: Asignación de valores a identificadores.

```
nombre = "Ana"
edad = 25
```

4. **Instrucciones y expresiones**: Operaciones que el programa ejecuta.

```
resultado = 10 + 5
print("El resultado es:", resultado)
```

5. **Funciones**: Bloques de código reutilizables.

```
def saludar(nombre):
    return "Hola, " + nombre + "!"
```

6. **Estructura principal**: Muchos programas incluyen un bloque que se ejecuta solo cuando el archivo se ejecuta directamente.

```
if __name__ == "__main__":
    print("Este programa se está ejecutando directamente")
```

Ejemplo de un programa Python simple:

```
# Programa para calcular el área de un círculo
import math

def calcular_area(radio):
    """Calcula el área de un círculo dado su radio"""
    return math.pi * radio ** 2

# Programa principal
if __name__ == "__main__":
```

```
r = 5
area = calcular_area(r)
print(f"El área de un círculo con radio {r} es {area:.2f}")
```

Variables y tipos de datos básicos (números, cadenas)

Las variables en Python son nombres que hacen referencia a valores almacenados en la memoria del computador. Python es un lenguaje de tipado dinámico, lo que significa que no es necesario declarar el tipo de una variable; el intérprete lo infiere automáticamente.

Nombres de variables:

- Deben comenzar con una letra o guion bajo (_)
- Pueden contener letras, números y guiones bajos
- Son sensibles a mayúsculas/minúsculas (case-sensitive)
- No pueden ser palabras reservadas (como `if`, `for`, `while`, etc.)

```
nombre = "Juan"  # Correcto
_contador = 0  # Correcto
1er_valor = 10  # Incorrecto (comienza con número)
for = "bucle"  # Incorrecto (palabra reservada)
```

Tipos de datos numéricos:

1. **Enteros (int)**: Números sin parte decimal.

```
edad = 25
temperatura_bajo_cero = -10
```

2. **Flotantes (float)**: Números con parte decimal.

```
estatura = 1.75
pi_aproximado = 3.14159
notacion_cientifica = 1.5e3  # 1.5 × 10³ = 1500.0
```

3. **Complejos (complex)**: Números con parte real y parte imaginaria.

```
z = 3 + 4j  # Donde j representa la unidad imaginaria
```

Cadenas de texto (str):

Secuencias de caracteres delimitadas por comillas simples (`'`) o dobles (`"`). Para cadenas multilínea se utilizan triples comillas (`'''` o `"""`).

```
nombre = "María"
apellido = 'González'
direccion = """Calle Principal 123,
Ciudad Ejemplo,
País"""
```

Operaciones con cadenas:

1. **Concatenación**: Unir cadenas mediante el operador +.

```
nombre_completo = nombre + " " + apellido  # "María González"
```

2. **Repetición**: Repetir una cadena mediante el operador *.

```
repeticion = "Na" * 4 + " Batman!"  # "NaNaNaNa Batman!"
```

3. **Indexación**: Acceder a caracteres individuales (los índices comienzan en 0).

```
primera_letra = nombre[0]  # "M"
ultima_letra = nombre[-1]  # "a"
```

4. **Rebanado (slicing)**: Extraer subcadenas.

```
subcadena = nombre[1:3]  # "ar"
```

5. **Métodos comunes de cadenas**:

```
saludo = "hola mundo"
saludo_mayusculas = saludo.upper()  # "HOLA MUNDO"
saludo_capitalizado = saludo.capitalize()  # "Hola mundo"
posicion = saludo.find("mundo")  # 5
reemplazo = saludo.replace("mundo", "Python")  # "hola Python"
```

Conversión entre tipos de datos:

Python permite convertir explícitamente entre diferentes tipos de datos usando funciones constructoras.

```
# Conversión de str a int
numero_texto = "100"
numero = int(numero_texto)  # 100

# Conversión de int a str
cantidad = 50
mensaje = "Tienes " + str(cantidad) + " mensajes."

# Conversión de str a float
```

```
decimal_texto = "3.14"
pi = float(decimal_texto)
```

Operadores aritméticos y lógicos

Operadores aritméticos:

Operador	Descripción	Ejemplo	Resultado
`+`	Suma	`5 + 3`	`8`
`-`	Resta	`5 - 3`	`2`
`*`	Multiplicación	`5 * 3`	`15`
`/`	División	`5 / 3`	`1.6666...`
`//`	División entera	`5 // 3`	`1`
`%`	Módulo (residuo)	`5 % 3`	`2`
`**`	Potencia	`5 ** 3`	`125`

Operadores de asignación:

Operador	Ejemplo	Equivalente
`=`	`x = 5`	`x = 5`
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 3`	`x = x * 3`
`/=`	`x /= 3`	`x = x / 3`
`//=`	`x //= 3`	`x = x // 3`
`%=`	`x %= 3`	`x = x % 3`
`**=`	`x **= 3`	`x = x ** 3`

Operadores lógicos:

Operador	Descripción	Ejemplo	Resultado
`and`	Verdadero si ambos operandos son verdaderos	`True and False`	`False`
`or`	Verdadero si al menos un operando es verdadero	`True or False`	`True`
`not`	Invierte el valor de verdad	`not True`	`False`

Orden de precedencia:

Los operadores se evalúan en el siguiente orden (de mayor a menor precedencia):

1. Paréntesis `()`
2. Potencia `**`
3. Positivo/negativo unario `+x`, `-x`
4. Multiplicación, división, módulo `*`, `/`, `//`, `%`
5. Suma, resta `+`, `-`
6. Operadores de comparación `<`, `<=`, `>`, `>=`, `==`, `!=`
7. `not`
8. `and`
9. `or`

```
resultado = 2 + 3 * 4  # Primero 3 * 4 = 12, luego 2 + 12 = 14
resultado_parentesis = (2 + 3) * 4  # Primero (2 + 3) = 5, luego 5 * 4 = 20

# Operadores lógicos
x = 5
y = 10
condicion = x > 0 and y < 20  # True
```

Entrada y salida básica (input(), print())

Función print():

La función `print()` muestra información en la consola.

```
print("Hola, mundo!")  # Imprime: Hola, mundo!

# Imprimir múltiples valores separados por espacios
nombre = "Ana"
edad = 25
print("Nombre:", nombre, "Edad:", edad)  # Imprime: Nombre: Ana Edad: 25

# Cambiar el separador
```

```
print("Python", "es", "genial", sep="-")  # Imprime: Python-es-genial

# Cambiar el final de línea
print("Línea 1", end=" || ")
print("Línea 2")  # Imprime: Línea 1 || Línea 2

# Formateo de cadenas con f-strings (Python 3.6+)
altura = 1.75
peso = 68.5
print(f"Altura: {altura}m, Peso: {peso}kg")  # Imprime: Altura: 1.75m, Peso: 68.5kg

# Formateo con especificadores de formato
pi = 3.14159265359
print(f"Valor de pi: {pi:.2f}")  # Imprime: Valor de pi: 3.14
```

Función input():

La función `input()` permite obtener datos ingresados por el usuario a través del teclado. Siempre devuelve una cadena de texto.

```
nombre = input("Ingrese su nombre: ")
print(f"Hola, {nombre}!")

# Para obtener valores numéricos, es necesario convertir la entrada
edad_texto = input("Ingrese su edad: ")
edad = int(edad_texto)  # Convierte la cadena a entero

# También se puede hacer en una sola línea
peso = float(input("Ingrese su peso en kg: "))
```

Ejemplo completo de entrada/salida:

```
# Programa para calcular el índice de masa corporal (IMC)
print("Calculadora de IMC")
print("-----------------")

nombre = input("Ingrese su nombre: ")
peso = float(input("Ingrese su peso en kg: "))
altura = float(input("Ingrese su altura en metros: "))
imc = peso / (altura ** 2)
print("-----------------")
print(f"Nombre: {nombre}")
print(f"IMC: {imc:.2f}")
if imc < 18.5:
    categoria = "bajo peso"
elif imc < 25:
    categoria = "peso normal"
```

```
elif imc < 30:
    categoria = "sobrepeso"
else:
    categoria = "obesidad"

print(f"Categoría: {categoria}")
```

Práctica: Calculadora simple

Realizar un programa en Python que implemente una calculadora simple capaz de realizar operaciones básicas: suma, resta, multiplicación y división.

Ejercicios adicionales:

1. Modificar la calculadora para que el usuario pueda realizar múltiples operaciones sin tener que reiniciar el programa.
2. Agregar nuevas operaciones como potencia, raíz cuadrada y módulo.
3. Implementar manejo de errores para entradas no numéricas.
4. Agregar una opción para borrar la memoria o usar el resultado anterior en una nueva operación.

SEMANA 4: ESTRUCTURAS DE CONTROL

Estructuras condicionales (if, elif, else)

Las estructuras condicionales permiten que un programa tome decisiones y ejecute diferentes bloques de código dependiendo de si ciertas condiciones se cumplen o no.

Sentencia if:

La estructura más básica de control condicional. Ejecuta un bloque de código solo si la condición es verdadera.

```
edad = 18
if edad >= 18:
    print("Eres mayor de edad")
```

Sentencia if-else:

Permite especificar un bloque de código alternativo que se ejecutará si la condición es falsa.

```
edad = 16
if edad >= 18:
```

```
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

Sentencia if-elif-else:

Permite evaluar múltiples condiciones en secuencia. Se ejecuta el bloque de código correspondiente a la primera condición verdadera.

```
calificacion = 85
if calificacion >= 90:
    print("A - Excelente")
elif calificacion >= 80:
    print("B - Muy bien")
elif calificacion >= 70:
    print("C - Bien")
elif calificacion >= 60:
    print("D - Suficiente")
else:
    print("F - Reprobado")
```

Condiciones anidadas:

Se pueden incluir estructuras condicionales dentro de otras.

```
edad = 25
tiene_licencia = True

if edad >= 18:
    if tiene_licencia:
        print("Puede conducir")
    else:
        print("Es mayor de edad pero no puede conducir sin licencia")
else:
    print("No puede conducir por ser menor de edad")
```

Expresiones condicionales (operador ternario):

Permiten asignar valores a variables basados en una condición, en una sola línea.

```
edad = 20
estatus = "Mayor de edad" if edad >= 18 else "Menor de edad"
```

Operadores de comparación y operadores lógicos

Operadores de comparación:

Operador	Descripción	Ejemplo	Resultado
`==`	Igual a	`5 == 5`	`True`
`!=`	Diferente de	`5 != 3`	`True`
`>`	Mayor que	`5 > 3`	`True`
`<`	Menor que	`5 < 3`	`False`
`>=`	Mayor o igual que	`5 >= 5`	`True`
`<=`	Menor o igual que	`5 <= 3`	`False`

Operadores lógicos:

Operador	Descripción	Ejemplo	Resultado
`and`	Verdadero si ambas condiciones son verdaderas	`(5 > 3) and (10 < 20)`	`True`
`or`	Verdadero si al menos una condición es verdadera	`(5 < 3) or (10 < 20)`	`True`
`not`	Invierte el valor de verdad de una condición	`not (5 < 3)`	`True`

Ejemplos combinados:

```
# Verificar si un número está en un rango
numero = 15
en_rango = numero >= 10 and numero <= 20
print(f"¿{numero} está entre 10 y 20? {en_rango}")  # True

# Verificar múltiples condiciones
dia = "sábado"
es_fin_de_semana = dia == "sábado" or dia == "domingo"
print(f"¿{dia} es fin de semana? {es_fin_de_semana}")  # True

# Negar una condición
tiene_descuento = False
no_tiene_descuento = not tiene_descuento
print(f"¿No tiene descuento? {no_tiene_descuento}")  # True
```

Evaluación de cortocircuito:

Python utiliza evaluación de cortocircuito para los operadores lógicos, lo que significa:

- Para `and`: si el primer operando es `False`, el segundo no se evalúa.
- Para `or`: si el primer operando es `True`, el segundo no se evalúa.

```
# Ejemplo de cortocircuito con and
x = 5
y = 0
# La segunda condición no se evalúa porque la primera es False
if x > 10 and y > 0:
    print("Ambas condiciones son verdaderas")
else:
    print("Al menos una condición es falsa")

# Verificación segura para evitar división por cero
divisor = 0
# La segunda condición solo se evalúa si la primera es True
if divisor != 0 and 10 / divisor > 2:
    print("El resultado es mayor que 2")
else:
    print("No se puede dividir o el resultado no es mayor que 2")
```

Estructuras repetitivas (for, while)

Las estructuras repetitivas o bucles permiten ejecutar un bloque de código múltiples veces.

Bucle for:

El bucle `for` en Python itera sobre una secuencia (como una lista, tupla, diccionario, conjunto o cadena) o cualquier objeto iterable.

```
# Iteración sobre una lista
frutas = ["manzana", "banana", "cereza"]
for fruta in frutas:
    print(fruta)

# Iteración sobre un rango de números
for i in range(5):  # 0, 1, 2, 3, 4
    print(i)

# Rango con inicio, fin y paso
for i in range(2, 10, 2):  # 2, 4, 6, 8
    print(i)

# Iteración sobre una cadena
for letra in "Python":
    print(letra)

# Iteración con índice usando enumerate
```

```
for indice, fruta in enumerate(frutas):
    print(f"Índice {indice}: {fruta}")

# Iteración sobre múltiples secuencias a la vez con zip
nombres = ["Ana", "Carlos", "Elena"]
edades = [25, 30, 22]
for nombre, edad in zip(nombres, edades):
    print(f"{nombre} tiene {edad} años")
```

Bucle while:

El bucle `while` ejecuta un bloque de código mientras una condición sea verdadera.

```
# Contador simple
contador = 0
while contador < 5:
    print(contador)
    contador += 1

# Bucle con condición basada en entrada de usuario
respuesta = ""
while respuesta.lower() != "salir":
    respuesta = input("Escribe 'salir' para terminar: ")
    print(f"Has escrito: {respuesta}")

# Bucle infinito con break
contador = 0
while True:
    print(contador)
    contador += 1
    if contador >= 5:
        break
```

Bucles anidados:

Se pueden incluir bucles dentro de otros bucles.

```
# Generar una tabla de multiplicar
for i in range(1, 5):
    for j in range(1, 5):
        print(f"{i} x {j} = {i*j}")
    print("-" * 15)  # Separador entre tablas

# Patrón de asteriscos
for i in range(1, 6):
    print("*" * i)
```

Control de bucles (break, continue)

Python ofrece dos instrucciones principales para controlar el flujo de ejecución dentro de los bucles.

Instrucción break:

Termina el bucle actual y continúa con la ejecución después del bucle.

```
# Encontrar el primer número divisible por 7
for i in range(1, 100):
    if i % 7 == 0:
        print(f"El primer número divisible por 7 es: {i}")
        break  # Sale del bucle después de encontrar el primer número

# Salir de un bucle while cuando se cumple una condición
contador = 0
while True:  # Bucle infinito
    contador += 1
    if contador > 10:
        break  # Sale del bucle cuando el contador supera 10
print(f"El contador llegó a {contador}")
```

Instrucción continue:

Salta el resto del código dentro del bucle actual y pasa a la siguiente iteración.

```
# Imprimir solo los números impares
for i in range(10):
    if i % 2 == 0:
        continue  # Salta los números pares
    print(i)

# Omitir múltiplos de 3
i = 0
while i < 20:
    i += 1
    if i % 3 == 0:
        continue
    print(i)
```

Instrucción else en bucles:

Python permite utilizar una cláusula `else` con bucles `for` y `while`. El bloque `else` se ejecuta cuando el bucle termina normalmente (sin ser interrumpido por `break`).

```
# Verificar si un número es primo
```

```
numero = 17
for i in range(2, numero):
    if numero % i == 0:
        print(f"{numero} no es primo - divisible por {i}")
        break
else:
    print(f"{numero} es primo")

# Uso de else con while
contador = 0
while contador < 5:
    print(contador)
    contador += 1
else:
    print("El bucle while ha terminado normalmente")
```

Práctica: Juego de adivinanza de números

Presentar un juego completo de adivinanza de números en Python, donde el programa elige un número aleatorio y el usuario debe adivinarlo con la menor cantidad de intentos posible.

Conceptos aplicados en el juego:

1. **Generación de números aleatorios** con el módulo `random`
2. **Estructuras condicionales** para verificar si el número es correcto
3. **Bucle while** para permitir múltiples intentos
4. **Break** para salir del bucle cuando el usuario adivina correctamente
5. **Manejo de excepciones** para validar la entrada del usuario
6. **Recursión** para permitir jugar nuevamente
7. **Formateo de cadenas** para mensajes informativos

Ejercicios adicionales:

1. Modificar el juego para que se adapte dinámicamente a la habilidad del jugador, reduciendo o ampliando el rango.
2. Implementar un sistema de puntaje basado en la cantidad de intentos necesarios.
3. Agregar diferentes niveles de dificultad con rangos más amplios y menos intentos.
4. Crear un modo de dos jugadores donde cada uno trate de adivinar el número del otro.

SEMANA 5: ESTRUCTURAS DE DATOS BÁSICAS

Listas y operaciones con listas

Las listas son colecciones ordenadas y modificables que permiten elementos duplicados. Son una de las estructuras de datos más utilizadas en Python.

Creación de listas:

```
# Lista vacía
lista_vacia = []

# Lista con elementos
frutas = ["manzana", "banana", "cereza"]
numeros = [1, 2, 3, 4, 5]
mixta = [1, "dos", 3.0, True]

# Lista mediante constructor list()
numeros = list(range(1, 6))  # [1, 2, 3, 4, 5]
caracteres = list("Python")  # ['P', 'y', 't', 'h', 'o', 'n']

# Lista por comprensión
cuadrados = [x**2 for x in range(1, 6)]  # [1, 4, 9, 16, 25]
pares = [x for x in range(1, 11) if x % 2 == 0]  # [2, 4, 6, 8, 10]
```

Acceso a elementos:

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

# Acceso por índice (cero-indexado)
primera_fruta = frutas[0]  # "manzana"
ultima_fruta = frutas[-1]  # "kiwi"

# Rebanado (slicing)
dos_primeras = frutas[0:2]  # ["manzana", "banana"]
del_medio = frutas[1:4]  # ["banana", "cereza", "naranja"]
todas_menos_primera = frutas[1:]  # ["banana", "cereza", "naranja", "kiwi"]
todas_menos_ultima = frutas[:-1]  # ["manzana", "banana", "cereza", "naranja"]
copia_lista = frutas[:]  # Copia completa de la lista

# Rebanado con paso/saltos
cada_dos = frutas[::2]  # ["manzana", "cereza", "kiwi"]
reversa = frutas[::-1]  # ["kiwi", "naranja", "cereza", "banana", "manzana"]
```

Nota: Probar en la consola con distintos valores en las posiciones

Modificación de listas:

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

# Cambiar un elemento
frutas[1] = "fresa"  # ["manzana", "fresa", "cereza"]

# Añadir elementos
```

```
frutas.append("naranja")  # Añade al final: ["manzana", "fresa", "cereza", "naranja"]
frutas.insert(1, "kiwi")  # Inserta en posición: ["manzana", "kiwi", "fresa", "cereza", "naranja"]
frutas.extend(["uva", "pera"])  # Extiende con otra lista: ["manzana", "kiwi", "fresa", "cereza", "naranja", "uva", "pera"]

# Eliminar elementos
frutas.remove("fresa")  # Elimina la primera ocurrencia: ["manzana", "kiwi", "cereza", "naranja", "uva", "pera"]
eliminado = frutas.pop(2)  # Elimina y devuelve el elemento en posición 2: ["manzana", "kiwi", "naranja", "uva", "pera"], eliminado = "cereza"
del frutas[1]  # Elimina sin devolver: ["manzana", "naranja", "uva", "pera
```

Unidad 3: Funciones y Modularidad

Semana 6: Fundamentos de funciones

¿Qué es una función?

Una función es un bloque de código reutilizable diseñado para realizar una tarea específica. Imagina una función como una "mini-máquina" que:

- Recibe datos (o no)
- Procesa esos datos siguiendo instrucciones específicas
- Devuelve un resultado (o no)

Las funciones nos permiten organizar nuestro código de manera ordenada y evitar repetir las mismas instrucciones muchas veces.

¿Por qué usar funciones?

- **Reutilización**: Escribes el código una vez y lo usas muchas veces.
- **Organización**: Divide problemas grandes en tareas más pequeñas y manejables.
- **Mantenimiento**: Es más fácil encontrar y corregir errores en bloques pequeños de código.
- **Legibilidad**: El código es más fácil de entender cuando está organizado en funciones con nombres descriptivos.

Analogía de la vida real: Una función es como una receta de cocina. Tiene un nombre (ej. "Tortilla de patatas"), necesita ingredientes (los parámetros), sigue pasos específicos (las instrucciones) y produce un resultado (el plato terminado).

Estructura básica de una función:

```
def nombre_de_la_funcion():
    # Instrucciones que se ejecutarán cuando llames a la función
    print("¡Hola desde mi primera función!")
```

Llamando a una función:

```
nombre_de_la_funcion()  # Esto ejecuta la función
```

Parámetros y argumentos

Parámetros: Son las variables que aparecen en la definición de la función. Actúan como "espacios reservados" para recibir información.

Argumentos: Son los valores reales que pasamos a la función cuando la llamamos.

Ejemplo:

```
def saludar(nombre):  # 'nombre' es un parámetro
    print(f"¡Hola, {nombre}!")

saludar("Ana")  # "Ana" es un argumento
saludar("Carlos")  # "Carlos" es un argumento
```

Tipos de parámetros:

Parámetros posicionales: Se asignan según el orden.

```
def describir_persona(nombre, edad, ciudad):
    print(f"{nombre} tiene {edad} años y vive en {ciudad}.")

describir_persona("María", 25, "Madrid")
```

Parámetros con valores predeterminados: Tienen un valor por defecto si no se especifica.

```
def saludar(nombre, mensaje="¡Hola!"):
    print(f"{mensaje} {nombre}")

saludar("Luis")  # Usa el mensaje predeterminado
saludar("Luis", "¡Bienvenido!")  # Usa el mensaje personalizado
```

Parámetros con nombre: Se especifican por su nombre, no por posición.

```
describir_persona(ciudad="Barcelona", nombre="Pedro", edad=30)
```

Retorno de valores

Las funciones pueden devolver valores que luego podemos usar en nuestro programa.

Uso de `return`:

```
def sumar(a, b):
    resultado = a + b
    return resultado  # Devuelve el valor

total = sumar(5, 3)  # total = 8
print(total)
```

Características del retorno:

- Una función puede retornar cualquier tipo de dato (números, texto, listas, etc.)
- Cuando se ejecuta `return`, la función termina inmediatamente
- Si no hay instrucción `return`, la función devuelve `None` automáticamente
- Se pueden retornar múltiples valores (como una tupla)

Ejemplo de retorno múltiple:

```
def calcular_estadisticas(numeros):
    suma = sum(numeros)
    promedio = suma / len(numeros)
    maximo = max(numeros)
    minimo = min(numeros)
    return suma, promedio, maximo, minimo

datos = [10, 5, 8, 12, 7]
suma, promedio, maximo, minimo = calcular_estadisticas(datos)
```

Documentación de funciones

La documentación ayuda a otros (y a ti mismo) a entender para qué sirve una función y cómo usarla.

Docstrings: Son cadenas de texto entre triple comillas que describen la función.

```
def calcular_area_rectangulo(base, altura):
    """
    Calcula el área de un rectángulo.

    Parámetros:
        base (float): Longitud de la base del rectángulo.
```

```
    altura (float): Altura del rectángulo.

  Retorna:
    float: El área del rectángulo.
  """
  return base * altura
```

Elementos de una buena documentación:

- Descripción breve de lo que hace la función
- Explicación de los parámetros (tipo y propósito)
- Descripción del valor de retorno
- Ejemplos de uso (opcional)
- Notas o advertencias (opcional)

Ámbito de variables (local y global)

El ámbito (scope) determina dónde es accesible una variable en el código.

Variables locales:

- Definidas dentro de una función
- Solo accesibles dentro de esa función
- Se crean al llamar a la función y se destruyen cuando termina

```
def calcular_doble(numero):
  resultado = numero * 2  # 'resultado' es una variable local
  return resultado

# print(resultado)  # Error: 'resultado' no existe fuera de la función
```

Variables globales:

- Definidas fuera de cualquier función
- Accesibles desde cualquier parte del programa
- Deben declararse explícitamente como globales si se quieren modificar dentro de una función

```
contador = 0  # Variable global

def incrementar_contador():
  global contador  # Indica que usaremos la variable global
  contador += 1

incrementar_contador()
print(contador)  # Muestra 1
```

Recomendación: Limita el uso de variables globales ya que pueden hacer que el código sea más difícil de entender y mantener.

Práctica semana 6: Calculadora multifunción

Ahora vamos a aplicar estos conceptos creando una calculadora que puede realizar diferentes operaciones.

Actividades propuestas:

1. **Ejercicio básico**: Crea una función que convierta temperaturas entre Celsius y Fahrenheit.

2. **Ejercicio intermedio**: Crea una función que determine si un número es primo.

3. **Ejercicio avanzado**: Amplía la calculadora para incluir operaciones como potencia, raíz cuadrada y cálculo de porcentajes.

4. **Desafío**: Crea un programa que use funciones para gestionar una lista de tareas (añadir, eliminar, marcar como completada, mostrar todas).

Conceptos clave a recordar:

- Las funciones son bloques de código reutilizables
- Los parámetros son "variables de entrada" para las funciones
- El valor de retorno es lo que una función "devuelve" tras ejecutarse
- Las variables dentro de una función son locales por defecto
- Una buena documentación hace que tu código sea más fácil de entender y usar
- Las funciones bien diseñadas realizan una sola tarea específica

Unidad 3: Funciones y Modularidad

Semana 7: Funciones avanzadas

1. Funciones con argumentos por defecto

Los argumentos por defecto nos permiten especificar valores predeterminados para los parámetros, haciendo que sean opcionales al llamar a la función.

Sintaxis y uso básico:

```
def saludar(nombre, mensaje="Hola"):
    """Saluda a una persona con un mensaje personalizable."""
    print(f"{mensaje}, {nombre}!")
```

```
# Llamadas posibles:
saludar("Ana")          # Usa el mensaje por defecto: "Hola, Ana!"
saludar("Juan", "Buenos días")  # Mensaje personalizado: "Buenos días, Juan!"
```

Reglas importantes:

- Los parámetros con valores por defecto deben colocarse después de los parámetros sin valores por defecto.
- El valor por defecto se evalúa solo una vez, cuando se define la función.

2. Argumentos posicionales y nominales

Python permite pasar argumentos a funciones de dos maneras: por posición o por nombre.

Argumentos posicionales: Se asignan según el orden en que aparecen.

```
def describir_mascota(animal, nombre, edad):
    """Describe una mascota."""
    return f"{nombre} es un {animal} de {edad} años."

# Los argumentos se asignan según su posición
print(describir_mascota("perro", "Toby", 5))  # "Toby es un perro de 5 años."
```

Argumentos nominales: Se asignan explícitamente por nombre, independientemente del orden.

```
# Los argumentos se asignan por nombre
print(describir_mascota(nombre="Luna", edad=3, animal="gato"))  # "Luna es un gato de 3 años."
```

Combinando ambos tipos:

```
# Primero posicionales, luego nominales
print(describir_mascota("hámster", nombre="Pipo", edad=1))
```

3. Funciones con número variable de argumentos (*args, **kwargs)

A veces necesitamos funciones que acepten un número variable de argumentos.

`*args`**:** Permite pasar un número variable de argumentos posicionales.

```
def sumar(*numeros):
    """Suma cualquier cantidad de números."""
```

```
    resultado = 0
    for num in numeros:
        resultado += num
    return resultado

# Podemos pasar cualquier cantidad de argumentos:
print(sumar(1, 2))              # 3
print(sumar(1, 2, 3, 4, 5))     # 15
print(sumar(10))                # 10
print(sumar())                  # 0
```

`**kwargs`: Permite pasar un número variable de argumentos con nombre (recibidos como un diccionario).

```
def crear_estudiante(**datos):
    """Crea un diccionario con datos de estudiante."""
    print("Datos del estudiante:")
    for clave, valor in datos.items():
        print(f"- {clave}: {valor}")
    return datos

# Ejemplos de uso:
crear_estudiante(nombre="Ana", edad=22, carrera="Informática")
crear_estudiante(nombre="Luis", promedio=8.5, semestre=3, beca=True)
```

Combinando todos los tipos de argumentos:

```
def funcion_completa(param1, param2, *args, param3="valor_por_defecto", **kwargs):
    """Función que muestra el uso de todos los tipos de argumentos."""
    print(f"Parámetros obligatorios: {param1}, {param2}")
    print(f"Argumentos variables (args): {args}")
    print(f"Parámetro con valor por defecto: {param3}")
    print(f"Argumentos con nombre variables (kwargs): {kwargs}")

funcion_completa(10, 20, 30, 40, 50, param3="personalizado", x=100, y=200)
```

4. Funciones lambda

Las funciones lambda son pequeñas funciones anónimas (sin nombre) definidas con una sintaxis concisa.

Sintaxis básica:

```
# Estructura: lambda parámetros: expresión
```

Comparación con funciones normales:

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

# Equivalente con lambda
duplicar_lambda = lambda x: x * 2

print(duplicar(5))        # 10
print(duplicar_lambda(5)) # 10
```

Características de las lambdas:

- Limitadas a una sola expresión
- No pueden contener instrucciones o anotaciones
- No necesitan instrucción `return` (implícito)
- Útiles como funciones de un solo uso

Usos comunes:

1. Con funciones de orden superior como `map`, `filter` y `sorted`:

```
# Usando map con lambda para duplicar cada elemento
numeros = [1, 2, 3, 4, 5]
duplicados = list(map(lambda x: x * 2, numeros))
print(duplicados)  # [2, 4, 6, 8, 10]

# Usando filter con lambda para filtrar números pares
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # [2, 4]

# Usando sorted con lambda para ordenar tuplas por el segundo elemento
datos = [(1, 'c'), (3, 'a'), (2, 'b')]
ordenados = sorted(datos, key=lambda x: x[1])
print(ordenados)  # [(3, 'a'), (2, 'b'), (1, 'c')]
```

2. Para funciones simples de un solo uso:

```
# Calculadora simple con lambdas
operaciones = {
    'suma': lambda a, b: a + b,
    'resta': lambda a, b: a - b,
    'multiplicacion': lambda a, b: a * b,
    'division': lambda a, b: a / b if b != 0 else "Error: División por cero"
}
```

```
print(operaciones['suma'](5, 3))  # 8
print(operaciones['division'](10, 2))  # 5.0
```

5. Recursividad básica

La recursividad es una técnica donde una función se llama a sí misma para resolver un problema.

Componentes de una función recursiva:

1. **Caso base:** Condición que detiene la recursión
2. **Caso recursivo:** Llamada a la misma función con un problema más pequeño

Ejemplo: Factorial

```
def factorial(n):
    """Calcula el factorial de un número n."""
    # Caso base
    if n == 0 or n == 1:
        return 1
    # Caso recursivo
    else:
        return n * factorial(n - 1)

print(factorial(5))  # 5 * 4 * 3 * 2 * 1 = 120
```

Traza de la ejecución para factorial(5):

1. `factorial(5)` → 5 * `factorial(4)`
2. `factorial(4)` → 4 * `factorial(3)`
3. `factorial(3)` → 3 * `factorial(2)`
4. `factorial(2)` → 2 * `factorial(1)`
5. `factorial(1)` → 1 (caso base)
6. Retroceso: 2 * 1 = 2
7. Retroceso: 3 * 2 = 6
8. Retroceso: 4 * 6 = 24
9. Retroceso: 5 * 24 = 120

Ejemplo: Secuencia Fibonacci

```
def fibonacci(n):
    """
    Calcula el n-ésimo número de la secuencia Fibonacci.
    La secuencia comienza con 0, 1, 1, 2, 3, 5, 8, ...
```

```
    """
    # Casos base
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Caso recursivo
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Primeros 10 números de Fibonacci
for i in range(10):
    print(f"fibonacci({i}) = {fibonacci(i)}")
```

Ventajas y desventajas de la recursividad:

Ventajas:

- Soluciones elegantes y concisas para ciertos problemas
- Código más legible para algoritmos inherentemente recursivos
- Simplifica problemas complejos dividiéndolos en casos más simples

Desventajas:

- Mayor uso de memoria (cada llamada recursiva usa espacio en la pila de llamadas)
- Potencial de desbordamiento de pila con recursiones profundas
- Generalmente más lento que soluciones iterativas

Práctica semana 7:

Procesamiento de datos con funciones Crear un conjunto de funciones para procesar una lista de estudiantes y sus calificaciones. Funciones a definir:

- calcular promedio de un alumno tendra 5 notas, las notas van de 0 a 100
- asignar una letra segun el promedio (A: 90-100, B: 80-89, C: 70-79, D: 60-69, F: 0-59)
- Generar reporte total: de los 10 alumnos, indicar el nombre del mejor promedio, cuantos aprobados, y reprobados.

Actividades propuestas:

1. **Ejercicio básico**: Crea una función lambda que convierta grados Celsius a Fahrenheit (F = C * 9/5 + 32).

2. **Ejercicio intermedio**: Implementa la función `fibonacci()` de manera iterativa (sin recursividad) y compara su rendimiento con la versión recursiva.

3. **Ejercicio avanzado**: Modifica el sistema de procesamiento de estudiantes para incluir:

    - Estadísticas por materia (asumiendo que cada calificación corresponde a una materia)
    - Gráfico de distribución de calificaciones
    - Predicción de rendimiento futuro basado en tendencia actual
4. **Desafío**: Crea una función recursiva que calcule la suma de los dígitos de un número. Por ejemplo, para 12345 debe devolver 1+2+3+4+5 = 15. Pista: puedes usar división y módulo para obtener los dígitos.

Conceptos clave a recordar:

- Los argumentos por defecto permiten hacer parámetros opcionales
- Con `*args` y `**kwargs` puedes crear funciones flexibles que acepten diferentes números de argumentos
- Las funciones lambda son útiles para operaciones simples y de un solo uso
- La recursividad es potente para ciertos problemas, pero tiene limitaciones de rendimiento
- Las funciones bien diseñadas pueden componerse para crear sistemas complejos
- El uso adecuado de diferentes tipos de argumentos mejora la usabilidad de tus funciones

Semana 8: Módulos y Paquetes en Python

1. Importación de Módulos

Un módulo en Python es simplemente un archivo con código Python que puede ser importado y utilizado en otros programas. Los módulos nos permiten organizar nuestro código de manera lógica y reutilizable.

Formas de importar módulos:

Importación completa

```
import math
resultado = math.sqrt(16)  # Resultado: 4.0
```

Importación de elementos específicos

```
from math import sqrt, pi
resultado = sqrt(16)  # Ya no necesitamos escribir math.sqrt
circunferencia = 2 * pi * 5  # Circunferencia de un círculo de radio 5
```

Importación con alias

```
import math as m
resultado = m.sqrt(16)  # Usamos el alias 'm' en lugar de 'math'
```

Importación de todos los elementos (no recomendada)

```
from math import *
resultado = sqrt(16)  # Funciona, pero puede causar conflictos de nombres
```

2. Biblioteca Estándar de Python

Python viene con una amplia biblioteca estándar que ofrece módulos para diversas tareas. Algunos de los módulos más útiles incluyen:

`math`: Funciones matemáticas

```
import math
print(math.sqrt(16))  # Raíz cuadrada
print(math.cos(math.pi))  # Coseno de pi (-1.0)
```

`random`: Generación de números aleatorios

```
import random
print(random.randint(1, 10))  # Número entero aleatorio entre 1 y 10
print(random.choice(['manzana', 'naranja', 'plátano']))  # Elemento aleatorio de una lista
```

`datetime`: Manejo de fechas y horas

```
from datetime import datetime, timedelta
ahora = datetime.now()
print(ahora)
futuro = ahora + timedelta(days=7)  # 7 días en el futuro
print(futuro)
```

`os`: Interacción con el sistema operativo

```
import os
print(os.getcwd())  # Obtener directorio de trabajo actual
archivos = os.listdir('.')  # Listar archivos en el directorio actual
```

`json`: Procesamiento de datos JSON

```
import json
datos = {'nombre': 'Ana', 'edad': 25}
json_str = json.dumps(datos)  # Convertir a string JSON
print(json_str)
datos_recuperados = json.loads(json_str)  # Convertir de string JSON a diccionario
```

3. Creación de Módulos Propios

Crear nuestros propios módulos es muy sencillo. Simplemente escribimos funciones, clases o variables en un archivo .py y luego lo importamos.

Ejemplo: Creación de un módulo de utilidades

Archivo: `utilidades.py`

```
def saludar(nombre):
    return f"Hola, {nombre}!"

def sumar(a, b):
    return a + b

PI = 3.14159

class Calculadora:
    def __init__(self):
        self.resultado = 0

    def sumar(self, numero):
        self.resultado += numero
        return self.resultado
```

Para usar este módulo en otro archivo:

```
import utilidades

print(utilidades.saludar("María"))  # Hola, María!
print(utilidades.sumar(5, 3))  # 8
print(utilidades.PI)  # 3.14159

calc = utilidades.Calculadora()
print(calc.sumar(10))  # 10
print(calc.sumar(5))  # 15
```

El bloque `if __name__ == "__main__":`

Este bloque es útil para incluir código que se ejecutará solo cuando el archivo se ejecuta directamente (no cuando se importa como módulo):

```
# En utilidades.py
def funcion_util():
    return "Soy útil"

if __name__ == "__main__":
    # Este código solo se ejecuta si el archivo se ejecuta directamente
    # ej: python utilidades.py
    print("Prueba de utilidades.py")
    print(funcion_util())
```

4. Organización de Código en Paquetes

Un paquete es una carpeta que contiene múltiples módulos relacionados. Para crear un paquete, debemos crear una carpeta con un archivo especial llamado `__init__.py`.

Estructura básica de un paquete

```
mi_paquete/
    __init__.py
    modulo1.py
    modulo2.py
    subpaquete/
        __init__.py
        modulo3.py
```

El archivo `__init__.py` puede estar vacío, pero debe existir para que Python reconozca la carpeta como un paquete.

Importación desde paquetes

```
# Importar un módulo completo del paquete
import mi_paquete.modulo1

# Importar una función específica
from mi_paquete.modulo2 import función_específica

# Importar desde un subpaquete
from mi_paquete.subpaquete.modulo3 import otra_función
```

Archivo `__init__.py`

Este archivo puede usarse para:

1. Inicializar el paquete
2. Definir qué se importa cuando se usa `from paquete import *`
3. Simplificar las importaciones

Ejemplo de `__init__.py`:

```
# Definir qué se importa con from mi_paquete import *
__all__ = ['modulo1', 'modulo2']

# Importar funciones comunes para facilitar su acceso
from mi_paquete.modulo1 import funcion_principal
from mi_paquete.modulo2 import otra_funcion
```

5. Práctica: Sistema de Registro Simple

Vamos a desarrollar un sistema de registro simple de usuarios, en el archivo de usuarios vamos a definir las siguientes funciones, imprimir datos del usuario como nombre, email, activo, validar email, al menos debe tener un '@'. Mostrar menu, el cual el usuario podra elegir distintas opciones, listar usuarios. Luego definir app.py, en el cual iniciará el programa.

Ejecutando la aplicación:

Para ejecutar esta aplicación, nos situamos en el directorio de app.py

python -m app.app

Este ejemplo demuestra:

1. Organización en paquetes y módulos
2. Separación de responsabilidades
3. Importaciones entre módulos
4. Uso de funciones de la biblioteca estándar
5. Aplicación del patrón `if __name__ == "__main__"`

Ejercicios propuestos:

1. Añadir un módulo `validacion.py` con funciones para validar datos de usuario
2. Implementar la función de editar información de usuario

3. Avanzado/opcional. Añadir un módulo para manejar exportación a formatos CSV, Excel
4. Crear un subpaquete para manejar estadísticas de uso del sistema

Unidad 4: Manejo de Archivos y Excepciones en Python

Semana 9: Trabajo con archivos

1. Operaciones básicas: apertura, lectura, escritura y cierre

El manejo de archivos es una parte fundamental de cualquier lenguaje de programación, y Python ofrece una manera sencilla y potente de trabajar con ellos. Las operaciones básicas incluyen abrir, leer, escribir y cerrar archivos.

Utilizando la sentencia `with` (recomendado)

La forma más segura de trabajar con archivos es utilizando la sentencia `with`, que garantiza que el archivo se cierre correctamente, incluso si ocurre un error durante el procesamiento:

```
# Lectura de un archivo
with open('archivo.txt', 'r') as archivo:
    contenido = archivo.read()
    print(contenido)
# El archivo se cierra automáticamente al salir del bloque with

# Escritura en un archivo
with open('nuevo_archivo.txt', 'w') as archivo:
    archivo.write('Hola mundo\n')
    archivo.write('Este es un archivo de texto creado con Python')
```

Método tradicional (no recomendado para uso general)

```
# Apertura
archivo = open('archivo.txt', 'r')
```

```
# Lectura
contenido = archivo.read()
print(contenido)

# Cierre (importante para liberar recursos)
archivo.close()
```

Métodos de lectura

Python ofrece varios métodos para leer archivos:

```
with open('archivo.txt', 'r') as f:
    # Leer todo el contenido como una cadena
    contenido_completo = f.read()

    # Volver al inicio del archivo
    f.seek(0)

    # Leer una línea
    primera_linea = f.readline()

    # Volver al inicio
    f.seek(0)

    # Leer todas las líneas en una lista
    lineas = f.readlines()

    # Volver al inicio
    f.seek(0)

    # Iterar por cada línea (eficiente para archivos grandes)
    for linea in f:
        print(linea.strip())  # strip() elimina espacios en blanco y saltos de línea
```

Métodos de escritura

```
with open('nuevo.txt', 'w') as f:
    # Escribir una cadena
    f.write('Primera línea\n')

    # Escribir varias líneas
    lineas = ['Segunda línea\n', 'Tercera línea\n', 'Cuarta línea\n']
    f.writelines(lineas)
```

2. Modos de apertura de archivos

Al abrir un archivo con la función `open()`, debemos especificar el modo de apertura:

Modo	Descripción
`'r'`	Lectura (default). El archivo debe existir.
`'w'`	Escritura. Si el archivo existe, se sobrescribe. Si no existe, se crea.
`'a'`	Append (añadir). Escribe al final del archivo sin sobrescribir. Si no existe, se crea.
`'x'`	Creación exclusiva. Falla si el archivo ya existe.
`'b'`	Modo binario (se añade a otros modos, por ejemplo `'rb'` o `'wb'`).
`'t'`	Modo texto (default, se añade a otros modos, por ejemplo `'rt'` o `'wt'`).
`'+'`	Actualización (lectura y escritura, se añade a otros modos, por ejemplo `'r+'` o `'w+'`).

Ejemplos:

```
# Lectura de texto (modo por defecto)
with open('archivo.txt', 'r') as f:
    contenido = f.read()

# Escritura de texto (sobrescribe si existe)
with open('archivo.txt', 'w') as f:
    f.write('Nuevo contenido')

# Añadir al final del archivo
with open('archivo.txt', 'a') as f:
    f.write('\nUna línea más')

# Lectura y escritura
with open('archivo.txt', 'r+') as f:
    contenido = f.read()
    f.write('\nAgregado al final')

# Archivos binarios (para imágenes, documentos, etc.)
with open('imagen.jpg', 'rb') as f:
    datos_binarios = f.read()
```

3. Manipulación de archivos de texto, CSV y Excel

Archivos de texto plano

Para archivos de texto, utilizamos los métodos que ya hemos visto:

```
# Leer un archivo de texto
with open('datos.txt', 'r', encoding='utf-8') as f:
    contenido = f.read()

# Escribir en un archivo de texto
with open('salida.txt', 'w', encoding='utf-8') as f:
    f.write('Este texto incluye caracteres especiales: áéíóúñ')
```

Nota: El parámetro `encoding` es importante para trabajar correctamente con caracteres especiales. UTF-8 es generalmente una buena elección.

Archivos CSV (Comma-Separated Values)

Python incluye un módulo `csv` para trabajar con este formato:

```
import csv

# Lectura de un archivo CSV
with open('datos.csv', 'r', newline='', encoding='utf-8') as archivo_csv:
    lector = csv.reader(archivo_csv)
    # Leer encabezados
    encabezados = next(lector)
    print(f"Columnas: {encabezados}")

    # Leer datos
    for fila in lector:
        print(fila)  # fila es una lista con los valores de cada columna

# Escritura en un archivo CSV
with open('nuevo.csv', 'w', newline='', encoding='utf-8') as archivo_csv:
    escritor = csv.writer(archivo_csv)

    # Escribir encabezados
    escritor.writerow(['Nombre', 'Edad', 'Ciudad'])

    # Escribir datos
    escritor.writerow(['Ana', 25, 'Madrid'])
    escritor.writerow(['Juan', 30, 'Barcelona'])

    # Escribir múltiples filas
    datos = [
        ['María', 28, 'Valencia'],
        ['Pedro', 35, 'Sevilla'],
        ['Luisa', 22, 'Bilbao']
    ]
```

```
    escritor.writerows(datos)
```

Trabajando con diccionarios en CSV

A menudo es más conveniente trabajar con diccionarios para acceder a campos por nombre:

```
import csv

# Lectura con DictReader
with open('datos.csv', 'r', newline='', encoding='utf-8') as archivo_csv:
    lector = csv.DictReader(archivo_csv)

    for fila in lector:
        print(f"Nombre: {fila['Nombre']}, Edad: {fila['Edad']}")

# Escritura con DictWriter
with open('personas.csv', 'w', newline='', encoding='utf-8') as archivo_csv:
    campos = ['Nombre', 'Edad', 'Ciudad']
    escritor = csv.DictWriter(archivo_csv, fieldnames=campos)

    # Escribir encabezados
    escritor.writeheader()

    # Escribir filas individuales
    escritor.writerow({'Nombre': 'Ana', 'Edad': 25, 'Ciudad': 'Madrid'})

    # Escribir múltiples filas
    personas = [
        {'Nombre': 'Juan', 'Edad': 30, 'Ciudad': 'Barcelona'},
        {'Nombre': 'María', 'Edad': 28, 'Ciudad': 'Valencia'}
    ]
    escritor.writerows(personas)
```

Archivos Excel con pandas

Para trabajar con archivos Excel, la biblioteca más popular es `pandas`, que debe instalarse previamente (`pip install pandas openpyxl`):

```
import pandas as pd

# Leer un archivo Excel
df = pd.read_excel('datos.xlsx', sheet_name='Hoja1')
print(df.head())  # Mostrar primeras 5 filas

# Manipular datos
```

```
df['Nueva_Columna'] = df['Columna1'] + df['Columna2']

# Escribir a un nuevo archivo Excel
df.to_excel('resultados.xlsx', sheet_name='Resultados', index=False)
```

Otras operaciones comunes con pandas

```
import pandas as pd

# Leer un CSV con pandas
df = pd.read_csv('datos.csv')

# Filtrar datos
personas_mayores = df[df['Edad'] > 30]

# Agrupar y resumir datos
por_ciudad = df.groupby('Ciudad').agg({
    'Edad': ['mean', 'min', 'max', 'count']
})

# Guardar en diferentes formatos
df.to_csv('salida.csv', index=False)
df.to_excel('salida.xlsx', index=False)
```

4. Navegación en directorios con os y pathlib

Python ofrece dos módulos principales para trabajar con rutas y directorios: el módulo tradicional `os` y el más moderno y orientado a objetos `pathlib` (Python 3.4+).

Usando el módulo os

```
import os

# Obtener directorio actual
directorio_actual = os.getcwd()
print(f"Estás en: {directorio_actual}")

# Listar archivos y carpetas
contenido = os.listdir('.')
print(f"Contenido del directorio: {contenido}")

# Comprobar si una ruta existe
if os.path.exists('archivo.txt'):
    print("El archivo existe")

    # Comprobar si es un archivo o directorio
    if os.path.isfile('archivo.txt'):
        print("Es un archivo")
```

```
    elif os.path.isdir('archivo.txt'):
        print("Es un directorio")

# Crear un directorio
os.mkdir('nueva_carpeta')  # Crea un solo directorio
os.makedirs('ruta/a/nueva/carpeta', exist_ok=True)  # Crea directorios anidados

# Unir rutas de manera compatible con el sistema operativo
ruta_completa = os.path.join('carpeta', 'subcarpeta', 'archivo.txt')

# Obtener información sobre una ruta
nombre_base = os.path.basename(ruta_completa)  # 'archivo.txt'
directorio = os.path.dirname(ruta_completa)    # 'carpeta/subcarpeta'
nombre, extension = os.path.splitext(nombre_base)  # ('archivo', '.txt')

# Tamaño de un archivo en bytes
if os.path.isfile('documento.txt'):
    tamaño = os.path.getsize('documento.txt')
    print(f"Tamaño: {tamaño} bytes")

# Eliminar
os.remove('archivo_a_eliminar.txt')  # Elimina un archivo
os.rmdir('carpeta_vacia')  # Elimina un directorio vacío
```

Usando pathlib (recomendado para código nuevo)

`pathlib` ofrece una forma más intuitiva y orientada a objetos de trabajar con rutas:

```
from pathlib import Path

# Directorio actual
directorio_actual = Path.cwd()
print(f"Estás en: {directorio_actual}")

# Crear una ruta
ruta = Path('carpeta') / 'subcarpeta' / 'archivo.txt'
print(ruta)  # Muestra la ruta completa

# Comprobar si existe
if ruta.exists():
    print("La ruta existe")

# Propiedades y métodos útiles
print(ruta.name)       # 'archivo.txt'
print(ruta.stem)       # 'archivo'
print(ruta.suffix)     # '.txt'
print(ruta.parent)     # La carpeta que contiene el archivo
```

```
print(ruta.absolute()) # Ruta absoluta completa

# Listar contenidos de un directorio
carpeta = Path('.')
for archivo in carpeta.iterdir():
    print(archivo)

# Buscar archivos con un patrón
for archivo_py in carpeta.glob('*.py'):
    print(f"Archivo Python: {archivo_py}")

# Busqueda recursiva
for archivo_txt in carpeta.rglob('*.txt'):
    print(f"Archivo de texto: {archivo_txt}")

# Crear directorios
nueva_carpeta = Path('nueva_carpeta')
nueva_carpeta.mkdir(exist_ok=True)

carpeta_anidada = Path('ruta/a/nueva/carpeta')
carpeta_anidada.mkdir(parents=True, exist_ok=True)  # Crea los padres si no existen

# Leer/escribir archivos
archivo = Path('datos.txt')
if archivo.exists():
    # Leer
    contenido = archivo.read_text(encoding='utf-8')
    print(contenido)

    # Leer líneas
    lineas = archivo.read_text(encoding='utf-8').splitlines()

# Escribir a un archivo
archivo.write_text('Contenido nuevo', encoding='utf-8')

# Manipular archivos binarios
datos = Path('imagen.jpg').read_bytes()  # Lee como binario
Path('copia.jpg').write_bytes(datos)     # Escribe como binario

# Eliminar
archivo_temp = Path('temporal.txt')
if archivo_temp.exists():
    archivo_temp.unlink()  # Elimina el archivo
```

5. Práctica: Creación de un sistema simple de registro en archivo

Este sistema trabaja con un archivo csv llamado registros.csv, el cual permite:

- Listar todos los registros
- Agregar nuevos registros
- Buscar por ID o nombre
- Eliminar registros

Semana 10: Manejo de excepciones

1. Conceptos de errores y excepciones

En Python, existen dos tipos de errores principales:

Errores de sintaxis (SyntaxError): Ocurren cuando el código no sigue las reglas gramaticales de Python.

```
 # Error de sintaxis (falta dos puntos)
if x > 5
    print("x es mayor que 5")
```

1.

Excepciones: Ocurren durante la ejecución cuando Python no puede realizar una operación específica.

```
 # Excepción: división por cero
resultado = 10 / 0  # ZeroDivisionError

# Excepción: índice fuera de rango
lista = [1, 2, 3]
elemento = lista[10]  # IndexError

# Excepción: tipo incorrecto
numero = int("abc")  # ValueError
```

2.

Algunas excepciones comunes en Python:

Excepción	Descripción
`ZeroDivisionError`	División por cero

`TypeError`	Operación con tipos incompatibles
`ValueError`	Operación con valor inapropiado
`NameError`	Variable no definida
`IndexError`	Índice fuera de rango
`KeyError`	Clave no existente en un diccionario
`FileNotFoundError`	Archivo no encontrado
`IOError`	Error de entrada/salida
`ImportError`	Error al importar un módulo
`AttributeError`	Atributo o método inexistente

2. Estructuras try-except

La estructura básica para manejar excepciones en Python es:

```
try:
    # Código que puede generar una excepción
    valor = int(input("Ingresa un número: "))
    resultado = 10 / valor
    print(f"El resultado es: {resultado}")
except:
    # Código que se ejecuta si ocurre cualquier excepción
    print("Ocurrió un error")
```

Sin embargo, este enfoque general (capturar cualquier excepción) no es una buena práctica. Es mejor especificar qué excepciones queremos capturar.

3. Captura de excepciones específicas

Es recomendable capturar solo las excepciones que esperamos, para no ocultar errores inesperados:

```
try:
    valor = int(input("Ingresa un número: "))
    resultado = 10 / valor
    print(f"El resultado es: {resultado}")
except ValueError:
    print("Error: Debes ingresar un número válido")
except ZeroDivisionError:
```

```
    print("Error: No puedes dividir por cero")
```

También podemos capturar múltiples excepciones en un solo bloque:

```
try:
    # Código que puede generar una excepción
    archivo = open("datos.txt", "r")
    contenido = archivo.read()
    archivo.close()
except (FileNotFoundError, IOError) as error:
    print(f"Error al abrir el archivo: {error}")
```

El `as error` captura la instancia de la excepción, permitiéndonos acceder a su mensaje de error.

4. Bloque finally y su importancia

El bloque `finally` se ejecuta siempre, independientemente de si ocurrió una excepción o no:

```
try:
    archivo = open("archivo.txt", "r")
    contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no existe")
finally:
    # Este código siempre se ejecuta
    print("Fin del proceso")
    # Cerrar el archivo si está abierto
    if 'archivo' in locals() and not archivo.closed:
        archivo.close()
```

El bloque `finally` es especialmente útil para:

- Liberar recursos (cerrar archivos, conexiones de red, etc.)
- Realizar limpieza final
- Registrar que una operación ha finalizado (exitosamente o no)

5. Bloque else en excepciones

Python también permite un bloque `else` que se ejecuta solo si no ocurrió ninguna excepción:

```
try:
    numero = int(input("Ingresa un número: "))
```

```
except ValueError:
    print("Eso no es un número válido")
else:
    # Este código solo se ejecuta si no hubo excepciones
    print(f"El doble de {numero} es {numero * 2}")
finally:
    print("Proceso completado")
```

6. Lanzamiento de excepciones propias (raise)

Podemos lanzar excepciones deliberadamente usando la palabra clave `raise`:

```
def dividir(a, b):
    if b == 0:
        raise ZeroDivisionError("No se puede dividir por cero")
    return a / b

# También podemos lanzar nuestras propias excepciones
def validar_edad(edad):
    if edad < 0:
        raise ValueError("La edad no puede ser negativa")
    if edad > 120:
        raise ValueError("Edad poco probable")
    return edad
```

Depuración básica

La depuración es el proceso de encontrar y corregir errores en el código. Python ofrece varias herramientas para esto:

Usando print para depurar

La forma más básica de depurar es agregar declaraciones `print`:

```
def calcular_area(radio):
    print(f"Calculando área para radio: {radio}")
    if radio < 0:
        print(f"Error: radio negativo ({radio})")
        return None
    area = 3.14159 * radio * radio
    print(f"Área calculada: {area}")
    return area
```

Usando logging. Opcional

El módulo `logging` es mejor que `print` para depuración en aplicaciones reales:

```
import logging

# Configuración básica
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    filename='app.log'  # Opcional: guardar en archivo
)

def dividir(a, b):
    logging.debug(f"Dividiendo {a} / {b}")
    if b == 0:
        logging.error("División por cero")
        raise ZeroDivisionError("No puedes dividir por cero")
    resultado = a / b
    logging.info(f"Resultado: {resultado}")
    return resultado
```

Usando el depurador integrado (pdb). Opcional

Python incluye un depurador interactivo llamado `pdb`:

```
import pdb

def funcion_compleja(a, b, c):
    resultado = a + b
    pdb.set_trace()  # El programa se detiene aquí y entra en modo depuración
    resultado = resultado * c
    return resultado
```

Cuando se ejecuta este código, Python se detendrá en `set_trace()` y permitirá explorar variables, ejecutar código paso a paso, etc.

Comandos útiles en pdb:

- `n`: ejecutar la siguiente línea
- `c`: continuar hasta el siguiente breakpoint
- `s`: entrar en una función llamada
- `p variable`: imprimir el valor de una variable
- `q`: salir del depurador

9. Práctica: Mejora del sistema de registro con manejo robusto de errores

Vamos a mejorar nuestro sistema de registro del ejercicio anterior, añadiendo manejo de excepciones

Características del sistema de registro mejorado:

1. **Manejo robusto de errores de archivo**:
    - Captura errores específicos (archivo no encontrado, permisos, etc.)
    - Proporciona mensajes de error informativos
2. **Uso del patrón de contexto (`with`)**:
    - Implementa los métodos `__enter__` y `__exit__` para usar con la declaración `with`
    - Garantiza que los recursos se liberen correctamente
3. **Jerarquía de excepciones**:
    - Maneja diferentes tipos de excepciones con diferentes estrategias
    - Propaga errores críticos que no puede manejar
4. **Registro de errores**:
    - El sistema puede registrar sus propios errores para facilitar la depuración
    - No se detiene ante errores no críticos

Este sistema de registro es más robusto porque:

- Continúa funcionando incluso cuando ocurren ciertos errores
- Proporciona información detallada sobre los problemas
- Garantiza que los recursos se liberen adecuadamente
- Es extensible para manejar diferentes tipos de errores

Unidad 5: Programación Orientada a Objetos en Python

Semana 11: Fundamentos de POO

Introducción a la Programación Orientada a Objetos

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que utiliza "objetos" para diseñar aplicaciones y programas de computadora. Este enfoque de programación se centra en los objetos que los programadores quieren manipular, en lugar de centrarse en la lógica necesaria para manipularlos.

Un **paradigma de programación** es un estilo o manera particular de programar, una forma de conceptualizar y estructurar el código de un programa.

Principales paradigmas de programación:

- **Programación imperativa**: Se enfoca en describir cómo un programa opera, mediante una secuencia de comandos que cambian el estado del programa.
- **Programación funcional**: Trata la computación como la evaluación de funciones matemáticas y evita el cambio de estado y datos mutables.
- **Programación orientada a objetos**: Organiza el código en entidades llamadas objetos que tienen propiedades y comportamientos.

Ventajas de la POO:

- **Modularidad**: El código está organizado en unidades lógicas (objetos).
- **Reutilización**: A través de la herencia y composición, se puede reutilizar código.
- **Mantenibilidad**: Es más fácil de mantener y ampliar código orientado a objetos bien diseñado.
- **Escalabilidad**: Facilita la gestión de proyectos grandes.

Python es un lenguaje que soporta múltiples paradigmas, pero tiene un fuerte enfoque en la POO, haciendo que sea intuitivo y poderoso.

Clases y Objetos

En POO, las **clases** y los **objetos** son conceptos fundamentales:

- **Clase**: Es un plano o plantilla para crear objetos. Define un conjunto de atributos y métodos que caracterizan cualquier objeto de ese tipo.
- **Objeto**: Es una instancia de una clase, una entidad concreta que se crea a partir de la plantilla definida por la clase.

Sintaxis para definir una clase en Python:

```
class NombreDeClase:
    # Atributos y métodos de la clase
    pass
```

Ejemplo de una clase `Persona`:

```
class Persona:
    """
    Esta clase representa a una persona con nombre y edad.
    """
```

```
    pass
```

Atributos y Métodos

Los componentes principales de una clase son los atributos y métodos:

- **Atributos**: Son las características o propiedades que describen a un objeto (como variables dentro de la clase).
- **Métodos**: Son las funciones dentro de la clase que definen los comportamientos de los objetos.

Ejemplo:

```
class Persona:
    # Atributo de clase - compartido por todas las instancias
    especie = "Homo sapiens"

    def __init__(self, nombre, edad):
        # Atributos de instancia - específicos para cada objeto
        self.nombre = nombre
        self.edad = edad

    # Método - define el comportamiento
    def saludar(self):
        return f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años."

    def cumplir_anos(self):
        self.edad += 1
        return f"{self.nombre} ahora tiene {self.edad} años."
```

Constructores (init)

El método `__init__` es el **constructor** de la clase. Se llama automáticamente cuando se crea una nueva instancia de la clase. Es utilizado para inicializar los atributos del objeto.

- `self` es una referencia al objeto que se está creando.
- Los parámetros adicionales son los valores que se pasan al crear el objeto.

Ejemplo:

```
class Libro:
    def __init__(self, titulo, autor, paginas):
        self.titulo = titulo
        self.autor = autor
        self.paginas = paginas
        self.actual_pagina = 0
```

```
        self.esta_abierto = False
```

Instanciación de Objetos

La **instanciación** es el proceso de crear un objeto a partir de una clase. En Python, esto se hace llamando a la clase como si fuera una función.

Ejemplo:

```
# Crear un objeto Persona
persona1 = Persona("Ana", 30)
persona2 = Persona("Carlos", 25)

# Usar métodos del objeto
print(persona1.saludar())  # Output: Hola, mi nombre es Ana y tengo 30 años.
print(persona2.cumplir_anos())  # Output: Carlos ahora tiene 26 años.

# Acceder a atributos
print(persona1.nombre)  # Output: Ana
print(Persona.especie)  # Output: Homo sapiens - accediendo al atributo de clase
```

Práctica: Modelado de objetos simples

Implementar un sistema de biblioteca con las clases biblioteca y libros. El sistema permitirá al usuario buscar libro por autor, por libros disponibles, listar los libros prestados,

Semana 12: Herencia y Polimorfismo

Concepto de Herencia

La **herencia** es un mecanismo fundamental en POO que permite que una clase (llamada clase derivada o subclase) adquiera las propiedades y métodos de otra clase (llamada clase base o superclase). Este concepto promueve la reutilización de código y establece una relación jerárquica entre clases.

Terminología:

- **Clase base/padre/superclase**: La clase de la que se hereda.
- **Clase derivada/hija/subclase**: La clase que hereda de otra.

Herencia Simple en Python

En Python, la herencia se implementa colocando el nombre de la clase base entre paréntesis después del nombre de la clase derivada:

```
class ClaseBase:
```

```
    # Atributos y métodos de la clase base
    pass

class ClaseDerivada(ClaseBase):
    # Atributos y métodos de la clase derivada
    # Además, hereda los atributos y métodos de ClaseBase
    pass
```

Ejemplo:

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

    def hacer_sonido(self):
        return "Sonido genérico de animal"

    def descripcion(self):
        return f"{self.nombre} es un animal de {self.edad} años."

class Perro(Animal):
    def __init__(self, nombre, edad, raza):
        # Llamar al constructor de la clase base
        super().__init__(nombre, edad)
        self.raza = raza

    def hacer_sonido(self):
        return "¡Guau, guau!"

    def descripcion(self):
        return f"{self.nombre} es un perro de raza {self.raza} y tiene {self.edad} años."

# Crear instancias
animal_generico = Animal("Criatura", 5)
mi_perro = Perro("Max", 3, "Labrador")

print(animal_generico.hacer_sonido())  # Output: Sonido genérico de animal
print(mi_perro.hacer_sonido())  # Output: ¡Guau, guau!

print(animal_generico.descripcion())  # Output: Criatura es un animal de 5 años.
print(mi_perro.descripcion())  # Output: Max es un perro de raza Labrador y tiene 3 años.
```

En este ejemplo:

- `Animal` es la clase base que define atributos y comportamientos comunes para todos los animales.
- `Perro` es una clase derivada que hereda de `Animal` y añade su propia especialización.
- `super().__init__(nombre, edad)` llama al constructor de la clase base para inicializar los atributos heredados.

Sobrescritura de Métodos

La **sobrescritura de métodos** ocurre cuando una clase derivada proporciona una implementación específica para un método que ya está definido en la clase base. Esto permite a la clase derivada cambiar el comportamiento de los métodos heredados.

En el ejemplo anterior:

- `hacer_sonido()` y `descripcion()` son métodos sobrescritos en la clase `Perro`.
- Cada animal tiene una forma específica de hacer sonido, por lo que la clase derivada proporciona su propia implementación.

Métodos y Atributos de Clase vs Instancia

En Python, es importante distinguir entre métodos/atributos de clase y métodos/atributos de instancia:

Atributos de Clase vs Instancia:

- **Atributos de clase**: Pertenecen a la clase en sí y son compartidos por todas las instancias. Se definen fuera de cualquier método.
- **Atributos de instancia**: Pertenecen a cada objeto individualmente. Se definen dentro del método `__init__` o en otros métodos.

Métodos de Clase vs Instancia:

- **Métodos de instancia**: Operan sobre una instancia específica y tienen acceso a `self`.
- **Métodos de clase**: Operan sobre la clase en general, no sobre una instancia específica. Se definen con el decorador `@classmethod` y reciben `cls` como primer parámetro.
- **Métodos estáticos**: No operan ni sobre la clase ni sobre una instancia. Se definen con el decorador `@staticmethod` y no reciben un primer parámetro especial.

Ejemplo:

```
class Estudiante:
    # Atributo de clase
    escuela = "Escuela Secundaria Python"
    contador_estudiantes = 0
```

```
    def __init__(self, nombre, edad):
        # Atributos de instancia
        self.nombre = nombre
        self.edad = edad
        self.calificaciones = []
        # Modificar atributo de clase
        Estudiante.contador_estudiantes += 1

    # Método de instancia
    def agregar_calificacion(self, calificacion):
        self.calificaciones.append(calificacion)

    def promedio(self):
        if not self.calificaciones:
            return 0
        return sum(self.calificaciones) / len(self.calificaciones)

    # Método de clase
    @classmethod
    def cambiar_escuela(cls, nueva_escuela):
        cls.escuela = nueva_escuela

    @classmethod
    def crear_desde_ano_nacimiento(cls, nombre, ano_nacimiento):
        from datetime import datetime
        edad = datetime.now().year - ano_nacimiento
        return cls(nombre, edad)

    # Método estático
    @staticmethod
    def es_mayor_de_edad(edad):
        return edad >= 18

# Uso de atributos y métodos de clase vs instancia
estudiante1 = Estudiante("Ana", 16)
estudiante2 = Estudiante("Carlos", 17)

# Acceso a atributos de clase
print(Estudiante.escuela)  # Output: Escuela Secundaria Python
print(estudiante1.escuela)  # También accesible desde instancias

# Modificación de atributos de clase mediante método de clase
Estudiante.cambiar_escuela("Preparatoria Avanzada Python")
print(estudiante2.escuela)  # Output: Preparatoria Avanzada Python

# Uso de método de clase como constructor alternativo
estudiante3 = Estudiante.crear_desde_ano_nacimiento("Elena", 2005)
print(f"{estudiante3.nombre} tiene {estudiante3.edad} años.")
```

```
# Uso de método estático
print(Estudiante.es_mayor_de_edad(19))  # Output: True
print(estudiante1.es_mayor_de_edad(estudiante1.edad))  # Output: False

# Atributo de clase que cuenta instancias
print(f"Total de estudiantes: {Estudiante.contador_estudiantes}")  # Output: 3
```

Polimorfismo

El **polimorfismo** es un concepto de POO que permite que objetos de diferentes clases respondan al mismo método o función. La palabra "polimorfismo" significa "muchas formas" y describe la capacidad de un objeto para tomar diferentes formas o comportarse de manera diferente según el contexto.

Tipos de polimorfismo en Python:

1. **Polimorfismo de método**: Ocurre cuando un método en la clase base es sobrescrito por clases derivadas.

```
def hacer_sonido_animal(animal):
    return animal.hacer_sonido()

# Usando las clases Animal y Perro definidas anteriormente
animal = Animal("Criatura", 5)
perro = Perro("Max", 3, "Labrador")

print(hacer_sonido_animal(animal))  # Output: Sonido genérico de animal
print(hacer_sonido_animal(perro))   # Output: ¡Guau, guau!
```

2. **Polimorfismo de operador**: Python permite que las clases definan cómo se comportan con operadores mediante métodos especiales.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, otro_vector):
        return Vector(self.x + otro_vector.x, self.y + otro_vector.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2  # Usa el método __add__
```

```
print(v3)  # Output: Vector(7, 10)
```

3. **Duck Typing**: Un concepto relacionado con el polimorfismo en Python. La idea es "si camina como un pato y suena como un pato, entonces debe ser un pato". En programación, esto significa que lo que importa es el comportamiento de un objeto (sus métodos y propiedades), no su tipo o clase.

```
class Pato:
    def hablar(self):
        return "¡Cuack!"

    def nadar(self):
        return "El pato nada"

class Persona:
    def hablar(self):
        return "Hola"

    def nadar(self):
        return "La persona nada estilo libre"

def hacer_hablar(entidad):
    return entidad.hablar()

def hacer_nadar(entidad):
    return entidad.nadar()

pato = Pato()
persona = Persona()

print(hacer_hablar(pato))    # Output: ¡Cuack!
print(hacer_hablar(persona)) # Output: Hola

print(hacer_nadar(pato))     # Output: El pato nada
print(hacer_nadar(persona))  # Output: La persona nada estilo libre
```

Práctica: Jerarquía de clases para un sistema escolar

Implementemos un sistema escolar con una jerarquía de clases que demuestre herencia y polimorfismo:

Crear clase Base es Persona(nombre, edad, id), clase estudiante(grado, promedio, cursos), profesor(área, materias)
Crear clase Escuela con los atributos nombre, dirección, profesores, alumnos

Semana 13: Encapsulamiento y Abstracción

Encapsulamiento en Python

El **encapsulamiento** es uno de los principios fundamentales de la POO. Se refiere a la restricción del acceso directo a algunos componentes de un objeto, y es una forma de prevenir el acceso accidental a los datos de un componente.

En Python, el encapsulamiento se implementa mediante convenciones de nomenclatura, ya que no tiene modificadores de acceso explícitos como `private`, `protected` y `public` que se encuentran en otros lenguajes como Java o C++.

Convenciones de Nombrado para Encapsulamiento

Python utiliza las siguientes convenciones para indicar el nivel de acceso:

Atributos y métodos públicos: No tienen prefijo especial.

```
 self.nombre = nombre  # Atributo público

def metodo_publico(self):
    # Método público
    pass
```

Atributos y métodos protegidos: Se prefijan con un guion bajo (`_`). Esto es solo una convención, no impide el acceso.

```
 self._edad = edad  # Atributo protegido

def _metodo_protegido(self):
    # Método protegido
    pass
```

Atributos y métodos privados: Se prefijan con doble guion bajo (`__`). Python renombra estos internamente para dificultar el acceso accidental (pero no lo impide).

```
 self.__numero_seguridad_social = nss  # Atributo privado

def __metodo_privado(self):
    # Método privado
    pass
```

Propiedades (@property, getters y setters)

Python proporciona un mecanismo más elegante para controlar el acceso a los atributos: las **propiedades**. Estas permiten definir métodos especiales que se ejecutan cuando se accede, modifica o elimina un atributo.

La principal ventaja de usar propiedades es que permiten cambiar la implementación interna de una clase sin afectar el código que utiliza la clase.

Decorador @property

El decorador `@property` convierte un método en un atributo de solo lectura:

```
class Persona:
    def __init__(self, nombre, fecha_nacimiento):
        self.nombre = nombre
        self._fecha_nacimiento = fecha_nacimiento  # YYYY-MM-DD

    @property
    def edad(self):
        import datetime
        today = datetime.date.today()
        birth_date = datetime.datetime.strptime(self._fecha_nacimiento, "%Y-%m-%d").date()
        return today.year - birth_date.year - ((today.month, today.day) < (birth_date.month,
birth_date.day))
```

Con esto, podemos acceder a `edad` como si fuera un atributo:

```
persona = Persona("Ana", "1990-05-15")
print(persona.edad)  # Calcula y devuelve la edad
```

Getters y Setters

Podemos ampliar las propiedades con getters, setters y deleters:

```
class Temperatura:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        # Getter
        return self._celsius

    @celsius.setter
    def celsius(self, valor):
        # Setter
        if valor < -273.15:
            raise ValueError("La temperatura no puede ser menor que -273.15°C")
```

```
        self._celsius = valor

    @property
    def fahrenheit(self):
        # Getter
        return (self.celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, valor):
        # Setter
        self.celsius = (valor - 32) * 5/9

# Uso de la clase Temperatura
temp = Temperatura(25)
print(f"Temperatura: {temp.celsius}°C / {temp.fahrenheit}°F")

# Cambiar temperatura en Celsius
temp.celsius = 30
print(f"Temperatura: {temp.celsius}°C / {temp.fahrenheit}°F")

# Cambiar temperatura en Fahrenheit
temp.fahrenheit = 68
print(f"Temperatura: {temp.celsius}°C / {temp.fahrenheit}°F")

# Intentar establecer una temperatura inválida
try:
    temp.celsius = -300
except ValueError as e:
    print(f"Error: {e}")
```

Métodos Especiales (str, repr, etc.)

Python tiene **métodos mágicos** (o dunder methods, por "double underscore") que permiten definir comportamientos específicos para operadores y funciones integradas.

Algunos de los métodos mágicos más comunes:

1. **str**: Define el comportamiento para la función `str()` y `print()`.
2. **repr**: Define el comportamiento para la función `repr()`, que debe devolver una representación del objeto que idealmente podría usarse para recrearlo.
3. **len**: Define el comportamiento para la función `len()`.
4. **eq**, **lt**, etc.: Definen el comportamiento para operadores de comparación (`==`, `<`, etc.).
5. **add**, **sub**, etc.: Definen el comportamiento para operadores aritméticos (`+`, `-`, etc.).

Ejemplo:

class Punto:

```
def __init__(self, x, y):
    self.x = x
    self.y = y

def __str__(self):
    """Devuelve una representación legible del objeto."""
    return f"Punto({self.x},
```

Semana 14: Patrones de Diseño Básicos en Python

Introducción a los Patrones de Diseño

Los patrones de diseño son soluciones típicas a problemas comunes en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en el código.

¿Qué son los patrones de diseño?

Los patrones de diseño representan las mejores prácticas utilizadas por desarrolladores experimentados. Son soluciones probadas a problemas comunes que han evolucionado con el tiempo.

Beneficios de los patrones de diseño:

- **Reutilización de código**: Proporcionan soluciones probadas que se pueden adaptar a diferentes situaciones.
- **Comunicación**: Establecen un vocabulario común para discutir arquitecturas.
- **Escalabilidad**: Facilitan la expansión del código sin grandes cambios.
- **Mantenimiento**: Hacen que el código sea más predecible y fácil de mantener.

Clasificación de patrones de diseño:

1. **Patrones creacionales**: Se ocupan de los mecanismos de creación de objetos.
    - Singleton, Factory, Builder, Prototype, Abstract Factory
2. **Patrones estructurales**: Tratan con la composición de clases u objetos.
    - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
3. **Patrones de comportamiento**: Se ocupan de la comunicación entre objetos.
    - Observer, Strategy, Command, State, Iterator, Mediator, Template Method

Patrón Singleton

Propósito

Garantizar que una clase tenga una única instancia y proporcionar un punto de acceso global a ella.

Cuándo usarlo

- Cuando debe haber exactamente una instancia de una clase
- Cuando se necesita un acceso controlado a un recurso único
- Ejemplos: configuraciones, conexiones a bases de datos, loggers

Implementación en Python

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
            # Inicialización que solo debe ocurrir una vez
            cls._instance.value = "Soy una instancia única"
        return cls._instance

# Uso
s1 = Singleton()
s2 = Singleton()

print(s1 is s2)  # True - son el mismo objeto
print(s1.value)  # "Soy una instancia única"
```

Usando un decorador (forma más pythónica)

```
def singleton(cls):
    instances = {}

    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class Logger:
    def log(self, msg):
```

```
        print(f"LOG: {msg}")

# Uso
logger1 = Logger()
logger2 = Logger()
print(logger1 is logger2)  # True
```

Pros y contras

Ventajas:

- Control sobre el acceso a recursos compartidos
- Evita duplicación de instancias
- Acceso global

Desventajas:

- Viola el principio de responsabilidad única
- Puede dificultar las pruebas unitarias
- Implementación thread-safe puede ser compleja

Patrón Factory

Propósito

Definir una interfaz para crear objetos, pero permitir a las subclases decidir qué clase instanciar.

Cuándo usarlo

- Cuando la creación de un objeto implica lógica compleja
- Cuando se necesita desacoplar la lógica de creación de objetos del uso de esos objetos
- Para crear familias de objetos relacionados

Implementación en Python

```
from abc import ABC, abstractmethod

# Producto abstracto
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# Productos concretos
class Dog(Animal):
```

```
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Factory
class AnimalFactory:
    def create_animal(self, animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            raise ValueError(f"Animal type {animal_type} not supported")

# Uso
factory = AnimalFactory()
dog = factory.create_animal("dog")
cat = factory.create_animal("cat")

print(dog.speak())  # "Woof!"
print(cat.speak())  # "Meow!"
```

Pros y contras

Ventajas:

- Promueve el acoplamiento débil
- Facilita la adición de nuevos tipos de objetos
- Centraliza la lógica de creación de objetos

Desventajas:

- Puede introducir muchas subclases, aumentando la complejidad
- En aplicaciones más simples, puede ser innecesariamente complejo

Patrón Observer

Propósito

Definir una dependencia uno-a-muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

Cuándo usarlo

- Cuando un cambio en un objeto debe reflejarse en otros objetos
- Para diseñar un sistema de publicación/suscripción
- Para implementar comunicación entre objetos de manera desacoplada

Implementación en Python

```
from abc import ABC, abstractmethod

# Interfaz Observer
class Observer(ABC):
    @abstractmethod
    def update(self, message):
        pass

# Observers concretos
class EmailNotifier(Observer):
    def update(self, message):
        print(f"Enviando email: {message}")

class SMSNotifier(Observer):
    def update(self, message):
        print(f"Enviando SMS: {message}")

# Subject
class NotificationSystem:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

# Uso
notification_system = NotificationSystem()

email_notifier = EmailNotifier()
sms_notifier = SMSNotifier()

notification_system.attach(email_notifier)
notification_system.attach(sms_notifier)
```

```
notification_system.notify("¡Sistema actualizado!")
# Salida:
# Enviando email: ¡Sistema actualizado!
# Enviando SMS: ¡Sistema actualizado!

notification_system.detach(sms_notifier)
notification_system.notify("Solo correo esta vez")
# Salida:
# Enviando email: Solo correo esta vez
```

Pros y contras

Ventajas:

- Desacoplamiento entre objetos que interactúan entre sí
- Relaciones dinámicas que pueden cambiar en tiempo de ejecución
- Comunicación broadcast a múltiples objetos

Desventajas:

- Los observadores son notificados en orden no especificado
- Posibles fugas de memoria si los observadores no se desvinculan
- Posible sobrecarga de actualizaciones (cascadas de notificaciones)

Conclusiones

Los patrones de diseño son herramientas poderosas que facilitan la creación de código limpio, mantenible y escalable. Sin embargo, deben utilizarse con criterio:

1. **No forzar patrones**: Usar un patrón cuando realmente resuelve un problema, no por el simple hecho de usarlo.

2. **Entender el problema primero**: Los patrones son soluciones a problemas, por lo que es crucial entender el problema antes de aplicar un patrón.

3. **Combinar patrones**: En aplicaciones reales, es común combinar varios patrones para resolver problemas complejos.

4. **Simplicidad**: A veces, una solución simple es mejor que un patrón complejo. El principio KISS (Keep It Simple, Stupid) siempre debe considerarse.

Los patrones estudiados en esta unidad son solo la punta del iceberg. Hay muchos más patrones que pueden ser útiles según el contexto y las necesidades específicas de tu aplicación.

Ejercicios propuestos

1. Implementa un registrador de eventos (logger) utilizando el patrón Singleton.

2. Crea un sistema de creación de documentos (PDF, HTML, Word) usando el patrón Factory.

3. Desarrolla un sistema de notificaciones (email, SMS, push) utilizando el patrón Observer.

4. Combina los tres patrones en un sistema de administración de tareas donde:

    - Singleton: Gestor de tareas central
    - Factory: Creación de diferentes tipos de tareas (urgente, normal, baja prioridad)
    - Observer: Notificación a diferentes departamentos cuando las tareas cambian de estado