

Curso de programación en Python

Contents

- ¿Qué es Python?
- Variables
- Colecciones
- Control de Flujo
- Funciones
- ¿Cómo surge la Orientación a Objetos?
- Clases
- Veamos un ejemplo de estos conceptos
- Integrar las clases a Python
- Iterador
- Gestión de errores
- Módulos
- Introducción a NumPy
- Conclusión
- Introducción a la Librería Pandas en Python

Este libro constituye los apuntes de clase para el curso de programación en Python.

El curso está diseñado para los estudiantes de **Laboratorio de Computación 4** de la carrera de **Tecnicatura en Programación** que se dicta en la **Facultad Regional Tucumán de la Universidad Tecnológica Nacional**.

El libro está organizado en capítulos que cubren los temas básicos de la programación en Python. Cada capítulo incluye ejemplos de código y ejercicios para que los estudiantes puedan practicar los conceptos aprendidos.

¿Qué es Python?

Python es un lenguaje de propósito general que fue creado por Guido van Rossum en 1991. Es un lenguaje de programación interpretado, interactivo y orientado a objetos. Python es un lenguaje de alto nivel, lo que significa que es fácil de leer y escribir.

Python es un lenguaje de programación versátil que se puede utilizar para desarrollar aplicaciones de escritorio, aplicaciones web, aplicaciones móviles, juegos, inteligencia artificial, análisis de datos, entre otros.

Python es un lenguaje de programación multiplataforma, lo que significa que se puede ejecutar en diferentes sistemas operativos como Windows, Linux y macOS.

Python es un lenguaje de programación de código abierto, lo que significa que es gratuito y que su código fuente está disponible para que cualquiera pueda modificarlo y mejorarlo.

Python es un lenguaje de programación popular que es utilizado por empresas como Google, Facebook, Instagram, Spotify, Netflix, Dropbox, entre otras.

Variables

Las variables son utilizadas para almacenar información en la memoria de un programa. En Python, una variable es un nombre que se refiere a un valor.

Los nombres de las variables en Python pueden contener letras, números y guiones bajos. Sin embargo, el nombre de una variable no puede comenzar con un número.

En Python, las variables son sensibles a mayúsculas y minúsculas. Esto significa que las variables `nombre`, `Nombre` y `NOMBRE` son consideradas diferentes.

Para asignar un valor a una variable en Python, se utiliza el operador de asignación `=`. Por ejemplo:

```
nombre = "Juan"  
edad = 30  
altura = 1.75
```

En el ejemplo anterior, se han creado tres variables: `nombre`, `edad` y `altura`. La variable `nombre` almacena un valor de tipo `str`, la variable `edad` almacena un valor de tipo `int` y la variable `altura` almacena un valor de tipo `float`.

En Python es necesario declarar una variable antes de usarla. Esto se hace asignándole un valor inicial. Por ejemplo:

```
nombre = "Juan"  
print(nombre)
```

En este caso, la variable `nombre` se declara y se le asigna un valor en la misma línea. Luego, se imprime el valor de la variable utilizando la función `print()`.

Las variables tienen un tipo de dato asociado que determina el tipo de valor que pueden almacenar. Algunos de los tipos de datos más comunes en Python son:

- `int`: para números enteros.
- `float`: para números de punto flotante.
- `str`: para cadenas de texto.
- `bool`: para valores booleanos (`True` o `False`).

Es importante tener en cuenta que en Python no es necesario especificar el tipo de dato de una variable al declararla, ya que el intérprete de Python infiere automáticamente el tipo de dato en función del valor asignado.

Reglas para nombrar variables

Al nombrar variables en Python, es importante seguir algunas reglas para garantizar que el código sea legible y fácil de entender. Algunas de las reglas más comunes son:

- Los nombres de las variables deben ser descriptivos y representativos del valor que almacenan.
- Los nombres de las variables deben comenzar con una letra o un guion bajo.
- Los nombres de las variables no pueden contener espacios en blanco.
- Los nombres de las variables no pueden ser palabras reservadas de Python, como `if`, `else`, `for`, `while`, etc.

- Los nombres de las variables son sensibles a mayúsculas y minúsculas.

Al seguir estas reglas, se facilita la lectura y comprensión del código, lo que es fundamental para el desarrollo de programas eficientes y mantenibles.

Tipos básicos: `int`

Los enteros (`int`) son un tipo de dato básico en Python que se utiliza para representar números enteros. Se pueden utilizar para realizar operaciones matemáticas simples, como sumas, restas, multiplicaciones y divisiones. Tienen la particularidad de que no tienen límite en su tamaño (bueno, siempre hay límites pero son muy grandes).

Se identifican porque comienzan con un dígito y no tienen punto decimal. Existen múltiples formas de describir un entero, pero más allá de la forma de ingresarlos, siempre se almacena como un número entero.

```
a = 100 # a es un entero en base 10 (decimal)
b = 0b10010101 # b es un entero en base 2 (binario)
c = 0o1234567 # c es un entero en base 8 (octal)
d = 0x1234567890ABCDEF # d es un entero en base 16 (hexadecimal)

print(a) # 100
print(b) # 149
print(c) # 342391
print(d) # 1311768467294899695
```

Sobre los enteros se pueden realizar las operaciones matemáticas básicas como suma, resta, multiplicación y división. Además, se pueden realizar operaciones más complejas como el cálculo de módulo, división entera y potenciación.

```

# Operaciones matemáticas básicas
a = 10
b = 3
c = a + b    #> 13          # Suma           __add__
d = a - b    #> 7           # Resta          __sub__
e = a * b    #> 30          # Multiplicación __mul__
f = a / b    #> 3.333333333  # División       __truediv__
g = a // b   #> 3           # División entera __floordiv__
h = a % b    #> 1           # Módulo (resto) __mod__
i = a ** b   #> 1000        # Potenciación  __pow__

# Comparaciones
j = a == b   #> False        # Igualdad      __eq__
k = a != b   #> True         # Desigualdad   __ne__
l = a < b    #> False        # Menor que     __lt__
m = a > b    #> True         # Mayor que     __gt__
n = a <= b   #> False        # Menor o igual que __le__
o = a >= b   #> True         # Mayor o igual que __ge__

a = 10_000    #> 10000      # Se pueden usar guiones bajos para separar los dígitos
b = 0b1_0000  #> 16          # Se pueden usar guiones bajos para separar los dígitos

print(2 ** 100)
# 1267650600228229401496703205376

# ¿Cuántos '0' tiene el 2 ** 1000?
print(str(2 ** 1000).count('0'))

```

En Python se pueden realizar operaciones aritméticas con los enteros sin preocuparse por el desbordamiento de los valores. Python maneja automáticamente el desbordamiento de los enteros, lo que significa que los enteros pueden crecer tanto como la memoria de la computadora lo permita.

Por último, se pueden realizar conversiones entre los diferentes tipos de datos numéricos en Python. Por ejemplo, se puede convertir un entero a un número de punto flotante utilizando la función `float()` o un número de punto flotante a un entero utilizando la función `int()`.

```

# Conversión de booleanos a enteros
int(True)    #> 1
int(False)   #> 0

# Conversión de flotantes a enteros
int(3.14159)  #> 3
int(10.9999)  #> 10  (trunca el decimal)

# Conversión de cadenas a enteros
int('10')      #> 10 (base 10)
int('10', 2)    #> 2  (base 2)
int('10', 8)    #> 8  (base 8)
int('10', 16)   #> 16 (base 16)
int('A', 16)    #> 10 (base 16)
int('xx', 16)   #> ValueError: invalid literal for int() with base 16: 'xx'

# Conversión segura de cadenas a enteros
a = "10"

# Verificando si la cadena es un número
if a.isnumeric():
    b = int(a)
    print(b)
else:
    print("No es un número")

# Usando try-except para manejar errores
try:
    b = int(a)
    print(b)
except ValueError:
    print("No es un número")

```

Tipos básicos: `float`

Los números de punto flotante (`float`) son un tipo de dato básico en Python que se utiliza para representar números reales. Se pueden utilizar para realizar operaciones matemáticas más complejas, como sumas, restas, multiplicaciones y divisiones con decimales.

Se identifican porque tienen un punto decimal o usan la notación científica. Existen múltiples formas de describir un número de punto flotante, pero más allá de la forma de ingresarlo, siempre se almacena como un número real.

```

a = 3.14159      # a es un número de punto flotante
b = 1.0e-3        # b es un número de punto flotante en notación científica
c = 1_000.0       # c es un número de punto flotante con guiones bajos para separar los componentes

d = .0            #> 0.0
e = 0.             #> 0.0
f = 1.             #> 1.0
g = 1e100         #> 1.0e+100

```

Existe la librería `math` que contiene funciones matemáticas que se pueden utilizar con los números de punto flotante.

```

import math

a = 3.14159
b = math.sin(a)          # Seno
c = math.cos(a)          # Coseno
d = math.tan(a)          # Tangente

e = math.asin(a)          # Arcoseno
f = math.acos(a)          # Arcocoseno
g = math.atan(a)          # Arcotangente

i = math.trunc(a)          # Trunca el número
j = math.floor(a)          # Redondea hacia abajo
k = math.ceil(a)           # Redondea hacia arriba

```

Tipos básicos: `bool`

Los valores booleanos (`bool`) son un tipo de dato básico en Python que se utiliza para representar valores de verdad. Pueden tener dos posibles valores: `True` (verdadero) o `False` (falso).

Los valores booleanos se utilizan en expresiones lógicas y de comparación para determinar si una condición es verdadera o falsa. Por ejemplo:

```

a = True
b = False

c = 10 > 5    #> True
d = 10 < 5    #> False
e = 10 == 5   #> False

```

En Python, los siguientes valores son considerados falsos en un contexto booleano: `False`, `None`, `0`, `0.0`, `''`, `[]`, `{}`, `()`. Todos los demás valores se consideran verdaderos.

Las expresiones lógicas y de comparación en Python devuelven un valor booleano que se puede almacenar en una variable o utilizar en una estructura de control, como un `if` o un `while`.

Se pueden realizar operaciones lógicas con los valores booleanos utilizando los operadores `and`, `or` y `not`. Por ejemplo:

```
a = True  
b = False  
  
c = a and b  #> False  
d = a or b   #> True  
e = not a    #> False
```

Las expresiones lógicas en Python se evalúan de izquierda a derecha y se detienen tan pronto como se determina el resultado, evitando evaluar innecesariamente el resto. Esto se llama cortocircuito y es útil para prevenir errores en expresiones complejas.

```
a = True  
  
b = a or 10/0 # True  
# 10/0 da un error de división por cero, pero como la expresión es cortocircuitada,  
# c = a and 10/0 # ZeroDivisionError: division by zero
```

Python realiza dos conversiones automáticas cuando trabaja con valores booleanos:

- Cuando se usan valores booleanos en operaciones aritméticas, `True` se convierte en `1` y `False` se convierte en `0`.
- Cuando se espera una expresión booleana, como en un `if`, un `while` o en una `and`, `or`, `not`, automáticamente se convierte en un valor booleano.

```

# Conversión de enteros a booleanos
# Cualquier número diferente de cero se convierte en True, el cero se convierte en False
bool(0)      # False
bool(1)      # True
bool(10)     # True

# Conversión de flotantes a booleanos
# Cualquier número diferente de cero se convierte en True, el cero se convierte en False
bool(0.0)    # False
bool(0.1)    # True
bool(10.0)   # True

# Conversión de cadenas a booleanos
# Cualquier cadena no vacía se convierte en True, la cadena vacía se convierte en False
bool('')     # False
bool('a')    # True
bool(' ')    # True

# Conversión de listas, diccionarios y tuplas a booleanos
# Cualquier secuencia no vacía se convierte en True, la secuencia vacía se convierte en False
bool([])     # False
bool([1])    # True
bool([0])    # True

bool({})     # False
bool({1})    # True

bool(()))   # False
bool((1,))  # True

```

Tipos básicos: str

Los `str` o cadenas en Python son un tipo de dato básico que se utiliza para representar texto. Se pueden utilizar para almacenar mensajes, nombres, direcciones, números de teléfono y cualquier otro tipo de información que se pueda representar como texto.

Las cadenas en Python se pueden definir utilizando comillas simples (`'`) o dobles (`"`). Por ejemplo:

```

a = 'Hola, mundo!'
b = "¡Hola, mundo!"
c = '''Hola, mundo!'''
d = """¡Hola, mundo!"""

```

También se pueden definir cadenas multilínea utilizando comillas triples simples () o dobles (). Por ejemplo:

```
a = '''Este es un ejemplo  
de una cadena multilínea'''  
  
b = """Este es otro ejemplo  
de una cadena multilínea"""  
  
print(a)  
#> Este es un ejemplo  
#> de una cadena multilínea
```

Existe un tipo especial de cadena que permite dar formato a los valores que se insertan en ella. Estas cadenas se conocen como cadenas de formato y se definen utilizando la letra antes de las comillas de apertura. Se conocen como f-strings o 'string interpolations'.

Por ejemplo:

```

nombre = "Juan"
apellido = "Perez"
mensaje = f'Hola, {nombre} {apellido}'
print(mensaje)
#> Hola, Juan Perez

a = 10
b = 20
print(f'{a} + {b} = {a + b}') # Con la f se evalúa la expresión dentro de las llaves
#> 10 + 20 = 30

print('{a} + {b} = {a + b}') # Sin la f no se evalúa la expresión
#> {a} + {b} = {a + b}

# Con ':' se pueden especificar el formato de los valores
print(f'{"Hola":>10}') # Alineado a la derecha en un campo de 10 caracteres
#>      Hola

print(f'{"Hola":<10}') # Alineado a la izquierda en un campo de 10 caracteres
#> Hola

print(f'{"Hola":^10}') # Alineado al centro en un campo de 10 caracteres
#>      Hola

print(f'{3.14159:.2f}') # Redondea a 2 decimales
#> 3.14

print(f'{1000:10.2f}') # Un campo de 10 caracteres con 2 decimales
#> 1000.00

print(f'{10:05d}') # Rellena con ceros a la izquierda en un campo de 5 caracteres
#> 00010

```

Los `str` en Python son inmutables, lo que significa que una vez que se crea una cadena, no se puede modificar. Sin embargo, se pueden realizar operaciones con las cadenas, como concatenarlas, dividirlas, reemplazar partes de ellas, etc.

```

a = 'Hola, '
b = 'mundo!'
c = a + b
print(c)
#> 'Hola, mundo!'

d = 'Hola, mundo!'.split(', ')
print(d)
#> ['Hola', 'mundo!']

e = 'Hola, mundo!'.replace('mundo', 'Python')
print(e)
#> 'Hola, Python!'

f = 'Hola, mundo!'[0:5]
print(f)
#> 'Hola'

g = 'Hola, mundo!'[::-1]
print(g)
#> '!odnum ,aloH'

h = 'Hola, mundo!'.upper()
print(h)
#> 'HOLA, MUNDO!'

i = 'Hola, mundo!'.lower()
print(i)
#> 'hola, mundo!'

j = 'Hola, mundo!'.title()
print(j)
#> 'Hola, Mundo!'

k = 'Hola' * 3 # Repite la cadena 3 veces
print(k)
#> 'HolaHolaHola'

l = 'Hola, que tengas un ' + 'muy ' * 3 + 'buen día!'
print(l)
#> 'Hola, que tengas un muy muy muy buen día!'

```

Los `str` son una secuencia de caracteres y pueden ser tratados como una lista de caracteres. Se puede acceder a los caracteres individuales de una cadena utilizando la notación de corchetes `[]` y el índice del carácter que se desea acceder. Por ejemplo:

```

a = 'Hola, mundo!'
print(a[0])    #> 'H'      # Primer carácter
print(a[1])    #> 'o'      # Segundo carácter
print(a[-1])   #> '!'     # Último carácter

for i in range(len(a)): # Recorre la cadena e imprime cada carácter
    print(a[i], end=' ')
#> H o l a ,   m u n d o !

l = list('Hola, mundo!') # Convierte la cadena en una lista de caracteres
print(l)
#> ['H', 'o', 'l', 'a', ',', ' ', 'm', 'u', 'n', 'd', 'o', '!']

print(a[6:8])
#> 'mu'

```

En Python, todos los tipos se pueden convertir a cadena, es decir, siempre se puede obtener una representación de un valor como una cadena. Para convertir un valor a una cadena se puede utilizar la función `str()`. Esta función está definida para todos los tipos de datos en Python. Para los tipos creados (clases), se puede proveer una función para convertir a cadena.

```

# Conversión de enteros a cadenas
a = str(10)    #> '10'
b = str(1000)  #> '1000'

# Conversión de flotantes a cadenas
c = str(3.14159) #> '3.14159'
d = str(10.9999) #> '10.9999'

# Conversión de booleanos a cadenas
e = str(True)   #> 'True'
f = str(False)  #> 'False'

# Conversión de listas, diccionarios y tuplas a cadenas
g = str([1, 2, 3])      #> '[1, 2, 3]'
h = str({'a': 1, 'b': 2}) #> "{'a': 1, 'b': 2}"
i = str((1, 2, 3))       #> '(1, 2, 3)'

```

Python realiza conversiones automáticas a cadena cuando es necesario. Por ejemplo, cuando se imprime un valor, se convierte a cadena automáticamente. Sin embargo, para concatenar una cadena con un valor que no es una cadena, es necesario convertirlo explícitamente o utilizar formatos.

```

print(10, 1 > 2, "Hola", [1, 2, 3])           # Llamada a str en forma implíc.
#> 10 False Hola [1, 2, 3]

print(str(10), str(1 > 2), "Hola", str([1, 2, 3])) # Llamada a str en forma explíc.
#> 10 False Hola [1, 2, 3]

print(f"{10} {1 > 2} Hola {[1, 2, 3]}")      # Llamada a str en forma implíc.
#> 10 False Hola [1, 2, 3]

```

Existe otra función para convertir un objeto en una cadena, `repr()`. Esta función devuelve una representación de cadena del objeto que es más cercana a la representación interna del objeto. Es útil para depurar y para mostrar objetos de forma más detallada. Se puede decir que `str()` es para mostrar al usuario y `repr()` es para mostrar al programador.

```

from datetime import datetime

# Crear un objeto datetime
now = datetime.now()

# Usar str() para convertir el objeto a una cadena (representación legible)
print(f'str: {str(now)}')
#> str: 2023-10-05 14:30:00.123456

# Usar repr() para convertir el objeto a una cadena (representación detallada)
print(f'repr: {repr(now)}')
#> repr: datetime.datetime(2023, 10, 5, 14, 30, 0, 123456)

print(f'{now}') # Llamada a str() en forma implícita
#> 2023-10-05 14:30:00.123456

```

En este ejemplo, `str(now)` devuelve una representación legible del objeto `datetime`, mientras que `repr(now)` devuelve una representación más detallada y precisa del objeto, que incluye información adicional sobre la clase y el módulo.

Tipos básicos: `None`

El valor especial `None` en Python representa la ausencia de un valor o un valor nulo. Es un tipo de dato único que se utiliza para indicar que una variable no tiene ningún valor asignado o que una función no devuelve ningún valor explícitamente.

El tipo de dato de `None` es `NoneType`. Solo existe una instancia de `None` en todo el programa, lo que significa que es un singleton. Se puede utilizar `None` en diversas situaciones, como inicializar

variables que aún no tienen un valor, o para indicar que una función no devuelve nada.

Por ejemplo:

```
a = None # Inicializar una variable sin valor
print(a) #> None

def saludo(nombre):
    print(f"Hola, {nombre}!")

b = saludo("Juan") # La función imprime un saludo pero no devuelve nada
print(b) #> None
```

En este ejemplo, la variable `a` se inicializa con `None`, indicando que no tiene un valor específico. La función `saludo` imprime un mensaje pero no devuelve ningún valor explícito, por lo que `b` se asigna el valor `None`.

Es importante destacar que `None` no es lo mismo que `False`, `0` o una cadena vacía `" "`. Aunque todos estos valores son considerados falsos en un contexto booleano, representan conceptos diferentes.

Para comprobar si una variable es `None`, se debe utilizar el operador `is` en lugar de `==`. Esto se debe a que `None` es un singleton, y se recomienda comparar identidades en lugar de valores.

Por ejemplo:

```
a = None

if a is None:
    print("a es None")
else:
    print("a no es None")
```

También se puede utilizar `is not` para comprobar que una variable no es `None`:

```
b = 10

if b is not None:
    print("b tiene un valor")
else:
    print("b es None")
```

Uso de `None` en funciones

En Python, si una función no tiene una sentencia `return` o la sentencia `return` no devuelve ningún valor, la función devuelve implícitamente `None`.

```
def sin_retorno():
    pass

resultado = sin_retorno()
print(resultado) #> None
```

En este ejemplo, la función `sin_retorno` no devuelve ningún valor, por lo que `resultado` es `None`.

Contexto booleano de `None`

En un contexto booleano, como en una condición `if`, el valor `None` se considera falso. Esto permite utilizar `None` para comprobar si una variable tiene un valor asignado o no.

```
a = None

if not a:
    print("a es falso o None")
else:
    print("a tiene un valor verdadero")
```

Sin embargo, es recomendable utilizar `is None` o `is not None` para verificar explícitamente si una variable es `None`, ya que otros valores como `0`, `False` o cadenas vacías también se consideran falsos.

Conversión de `None` a otros tipos

Intentar convertir `None` a otros tipos puede generar errores. Por ejemplo, convertir `None` a una cadena con `str()` es posible y devuelve la cadena `'None'`. Sin embargo, convertir `None` a un entero o a un número de punto flotante genera un error.

```
a = None

print(str(a)) #> 'None'

# int(a) # TypeError: int() argument must be a string, a bytes-like object or a number
# float(a) # TypeError: float() argument must be a string or a number, not 'NoneType'
```

Resumen

El valor `None` es un tipo especial en Python que representa la ausencia de valor. Es útil para inicializar variables sin asignarles un valor específico, indicar que una función no devuelve nada o para comprobar si una variable no tiene un valor asignado.

Colecciones

Python permite almacenar una **colección** de elementos en una sola variable. Las colecciones más comunes en Python son las **listas, tuplas, conjuntos y diccionarios**.

Listas `list`

Una **lista** es una colección ordenada de elementos que se pueden modificar. En Python, una lista se define utilizando corchetes `[]` y los elementos se separan por comas `,`.

Por ejemplo:

```
nombres = ["Juan", "María", "Carlos"]
edades = [30, 25, 35]
alturas = [1.75, 1.65, 1.80]
```

Creación de una lista

```
# Expresión literal de una lista
nombres = ["Juan", "María", "Carlos"]
#> ['Juan', 'María', 'Carlos']

# Creación de una lista con la función `list()`
nombres = list(["Juan", "María", "Carlos"])
#> ['Juan', 'María', 'Carlos']

# Creación de una lista vacía y agregar elementos
nombres = [] # Crear una lista vacía
nombres.append("Juan") # Agregar un elemento a la lista
nombres.append("María")
#> ['Juan', 'María']

# Otras formas de crear una lista vacía
vacía = [] # Lista vacía
vacía = list() # Lista vacía usando la función `list()`

# Copiar una lista
nueva = list(nombres)

# Crear una lista a partir de un diccionario (solo las claves)
d = list({'x': 1, 'y': 2})
#> ['x', 'y']

# Crear una lista a partir de una tupla
t = list((1, 2, 3))
#> [1, 2, 3]

# Crear una lista a partir de una secuencia de valores
digitos = list(range(10)) # Números del 0 al 9
#> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

vocales = list("aeiou") # Lista con las vocales
#> ['a', 'e', 'i', 'o', 'u']
```

Acceso a los elementos de una lista

Los elementos de una lista se acceden mediante un **índice**. El índice comienza en `0`. Si el índice es negativo, se cuenta desde el final de la lista.

```
nombres = ["Juan", "María", "Carlos"]

print(nombres[0])      # Acceder al primer elemento
#> Juan

print(nombres[-1])    # Acceder al último elemento
#> Carlos

# Modificar un elemento de la lista
nombres[0] = "Pedro"
print(nombres)
#> ['Pedro', 'María', 'Carlos']
```

Nota: En Python, asignar una lista a otra variable no crea una copia de la lista, sino que crea una referencia a la lista original. Por lo tanto, si se modifica la lista original, la lista referenciada también se modificará.

```

a = [1, 2, 3]
b = a
print(a, b)
#> [1, 2, 3] [1, 2, 3]

a[0] = 10
print(a, b) # `b` también se modifica
#> [10, 2, 3] [10, 2, 3]

b[1] = 20
print(a, b) # `a` también se modifica
#> [10, 20, 3] [10, 20, 3]

# Verificar si `a` y `b` son la misma lista
print(a is b)
#> True

# Para copiar una lista, usar el método `copy()`
a = [1, 2, 3]
b = a.copy()

print(a, b)
#> [1, 2, 3] [1, 2, 3]

a[0] = 10
print(a, b) # `b` no se modifica
#> [10, 2, 3] [1, 2, 3]

b[1] = 20
print(a, b) # `a` no se modifica
#> [10, 2, 3] [1, 20, 3]

# Otras formas de copiar una lista
b = list(a)           # Usando el constructor `list()`
b = a[:]              # Usando el operador slicing `[:]`
b = [*a]               # Usando el operador `*` para desempaquetar la lista
b = [x for x in a]    # Usando una comprensión de listas

```

Slicing de listas

El **slicing** es una técnica en Python que permite obtener una sublista de una lista. Esta funcionalidad se puede usar tanto para extraer una sublista como para modificarla. Se puede llamar explícitamente a la función `slice()` o se puede usar la notación `[:]`, siendo esta última la más común y fácil de usar.

```

lista = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(lista[slice(2, 7, 2)]) # Tomar del índice 2 al 7 (sin incluir) con saltos de
#> [30, 50, 70]
# slice(inicio, fin, salto) === lista[inicio:fin:salto]

# Si no se especifica el inicio, se toma desde el principio
print(lista[:7]) # Tomar desde el inicio hasta el índice 7 (sin incluir)
#> [10, 20, 30, 40, 50, 60, 70]

# Si no se especifica el fin, se toma hasta el final
print(lista[2:]) # Tomar desde el índice 2 hasta el final
#> [30, 40, 50, 60, 70, 80, 90]

# Si no se especifica el salto, se toma de uno en uno
print(lista[2:7]) # Tomar del índice 2 al 7 (sin incluir) de uno en uno
#> [30, 40, 50, 60, 70]

# Se pueden combinar todas las opciones
print(lista[2:7:2]) # Tomar del índice 2 al 7 (sin incluir) de dos en dos
#> [30, 50, 70]

# Si no se especifica el inicio, el fin y el salto, se toma toda la lista
print(lista[:]) # Tomar toda la lista
#> [10, 20, 30, 40, 50, 60, 70, 80, 90]

# Se pueden usar índices negativos
print(lista[-7:-2]) # Tomar del índice -7 al -2 (sin incluir)
#> [30, 40, 50, 60, 70]

# Copiar una lista
copia = lista[:]

# Invertir una lista
invertida = lista[::-1]

```

En todos estos casos, el slicing se usa para extraer valores cuando se utiliza en el lado derecho de una asignación. Pero también se puede usar para modificar una lista cuando se utiliza en el lado izquierdo de una asignación.

```
lista = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(lista)
#> [10, 20, 30, 40, 50, 60, 70, 80, 90]

lista[2:7] = [31, 41, 51, 61, 71] # Modificar del índice 2 al 7 (sin incluir)
print(lista)
#> [10, 20, 31, 41, 51, 61, 71, 80, 90]

lista[2:7:2] = [32, 52, 72] # Modificar del índice 2 al 7 (sin incluir) de dos en dos
print(lista)
#> [10, 20, 32, 41, 52, 61, 72, 80, 90]

# Se puede usar para borrar elementos
lista[2:7] = [] # Borrar del índice 2 al 7 (sin incluir)
print(lista)
#> [10, 20, 80, 90]

# Se puede usar para insertar elementos
lista[2:2] = [30, 40] # Insertar en el índice 2
print(lista)
#> [10, 20, 30, 40, 80, 90]
```

El **slicing** es una herramienta muy poderosa y versátil, y es muy común en Python, por lo que es importante conocerla y saber usarla. El operador `:` es la forma más común de utilizarlo, pero también se puede usar la función `slice()` explícitamente.

Recorrer una lista

Para recorrer una lista, podemos usar un bucle `for` o un bucle `while`, utilizando la función `len()` para obtener la longitud de la lista.

```

nombres = ['Juan', 'María', 'Carlos']

# Con un bucle `while`
i = 0
while i < len(nombres):
    nombre = nombres[i]
    print(nombre, end=' ')
    i += 1
#> Juan María Carlos

print() # Para añadir una nueva línea

# Con un bucle `for` usando índices
for i in range(len(nombres)):
    nombre = nombres[i]
    print(nombre, end=' ')
#> Juan María Carlos

print()

# Usando un bucle `for` directamente con los elementos
for nombre in nombres:
    print(nombre, end=' ')
#> Juan María Carlos

print()

# Usando la función `enumerate()` para obtener el índice y el valor
for i, nombre in enumerate(nombres):
    print(i, nombre)
#> 0 Juan
#> 1 María
#> 2 Carlos

```

Operaciones con listas

Las listas poseen métodos para agregar, eliminar y modificar elementos. Algunas de las funciones más comunes son:

- **append()**: Agrega un elemento al final de la lista.
- **insert()**: Agrega un elemento en una posición específica.
- **pop()**: Elimina un elemento de la lista y lo retorna.
- **remove()**: Elimina un elemento de la lista por su valor.
- **index()**: Retorna el índice de un elemento.
- **count()**: Cuenta la cantidad de veces que un elemento aparece en la lista.

- `sort()`: Ordena la lista.
- `reverse()`: Invierte el orden de la lista.
- `copy()`: Crea una copia de la lista.

```
nombres = ["Juan", "María", "Carlos"]

# Agregar un elemento al final
nombres.append("Ana")
print(nombres)
#> ['Juan', 'María', 'Carlos', 'Ana']

# Insertar un elemento en una posición específica
nombres.insert(1, "Luis")
print(nombres)
#> ['Juan', 'Luis', 'María', 'Carlos', 'Ana']

# Eliminar el último elemento y retornarlo
ultimo = nombres.pop()
print(ultimo) # Ana
#> Ana
print(nombres)
#> ['Juan', 'Luis', 'María', 'Carlos']

# Eliminar un elemento por su valor
nombres.remove("Luis")
print(nombres)
#> ['Juan', 'María', 'Carlos']

# Obtener el índice de un elemento
indice = nombres.index("María")
print(indice)
#> 1

# Contar cuántas veces aparece un elemento
cantidad = nombres.count("Carlos")
print(cantidad)
#> 1

# Ordenar la lista
nombres.sort()
print(nombres)
#> ['Carlos', 'Juan', 'María']

# Invertir el orden de la lista
nombres.reverse()
print(nombres)
#> ['María', 'Juan', 'Carlos']

# Copiar la lista
nombres_copia = nombres.copy()
print(nombres_copia)
#> ['María', 'Juan', 'Carlos']
```

Diccionarios `dict`

Los **diccionarios** permiten mantener una colección **desordenada** de valores heterogéneos que se pueden acceder mediante una **clave**. En Python, un diccionario se define utilizando llaves `{}` y los pares clave-valor se separan por comas `,`.

Las claves pueden ser cualquier valor inmutable y hashable, como una cadena, un número o una tupla cuyos elementos también sean inmutables. Los valores pueden ser de cualquier tipo de dato.

Creación de un diccionario

```
# Expresión literal de un diccionario
persona = {"nombre": "Juan", "edad": 30}

# Creación de un diccionario con la función `dict()`
persona = dict(nombre="Juan", edad=30)

# Creación de un diccionario a partir de una lista de tuplas
persona = dict([("nombre", "Juan"), ("edad", 30)])

# Creación de un diccionario vacío y agregar elementos
persona = {} # Diccionario vacío
persona["nombre"] = "Juan" # Agregar un elemento
persona["edad"] = 30
```

Acceso a los elementos de un diccionario

Para acceder a un elemento de un diccionario usamos la **clave** entre corchetes `[]`. Si accedemos a una clave inexistente, se generará un error `KeyError`. Para evitar este error, podemos usar el método `get()` o verificar la existencia de la clave con el operador `in`.

```
persona = {'nombre': 'Juan', 'edad': 30}

# Acceder a un valor usando la clave
print(persona['nombre'])
#> Juan

# Usar el método `get()` para acceder a un valor
print(persona.get('nombre'))
#> Juan

# Intentar acceder a una clave inexistente con `get()`, retorna `None`
print(persona.get('apellido'))
#> None

# Intentar acceder a una clave inexistente con `get()`, retornando un valor por defecto
print(persona.get('apellido', '(anónimo)'))
#> (anónimo)

# Verificar si una clave existe usando el operador `in`
if 'apellido' in persona:
    print(persona['apellido'])
else:
    print('(anónimo)')
#> (anónimo)
```

Métodos adicionales de diccionarios

Además de los métodos básicos, los diccionarios en Python ofrecen una variedad de métodos útiles para manejar sus elementos:

- **keys()**: Retorna una vista de las claves del diccionario.
- **values()**: Retorna una vista de los valores del diccionario.
- **items()**: Retorna una vista de los pares clave-valor.
- **update()**: Actualiza el diccionario con los pares clave-valor de otro diccionario o iterable.
- **pop()**: Elimina el elemento con la clave especificada y lo retorna.
- **clear()**: Elimina todos los elementos del diccionario.

```

persona = {"nombre": "Juan", "edad": 30, "ciudad": "Madrid"}

# Obtener todas las claves
claves = persona.keys()
print(claves)
#> dict_keys(['nombre', 'edad', 'ciudad'])

# Obtener todos los valores
valores = persona.values()
print(valores)
#> dict_values(['Juan', 30, 'Madrid'])

# Obtener todos los pares clave-valor
items = persona.items()
print(items)
#> dict_items([('nombre', 'Juan'), ('edad', 30), ('ciudad', 'Madrid')])

# Actualizar el diccionario
persona.update({"edad": 31, "profesion": "Ingeniero"})
print(persona)
#> {'nombre': 'Juan', 'edad': 31, 'ciudad': 'Madrid', 'profesion': 'Ingeniero'}

# Eliminar un elemento con `pop()`
edad = persona.pop("edad")
print(edad)
#> 31
print(persona)
#> {'nombre': 'Juan', 'ciudad': 'Madrid', 'profesion': 'Ingeniero'}

# Eliminar todos los elementos
persona.clear()
print(persona)
#> {}

```

Conjuntos `set`

Los **conjuntos** son colecciones desordenadas de elementos únicos. Se utilizan para almacenar elementos sin duplicados y para realizar operaciones matemáticas como la unión, intersección y diferencia.

En Python, un conjunto se define utilizando llaves `{}` para conjuntos con elementos o la función `set()`.

Creación de un conjunto

```
# Usando llaves con elementos
frutas = {"manzana", "banana", "cereza"}

# Usando la función `set()`
numeros = set([1, 2, 3, 4, 5])

# Crear un conjunto vacío
vacío = set() # No se puede usar {} porque crea un diccionario vacío
```

Nota: Para crear un conjunto vacío se debe usar `set()`. Usar `{}` creará un diccionario vacío.

Operaciones con conjuntos

- **Agregar elementos:** `add()`
- **Eliminar elementos:** `remove()` o `discard()`
- **Unión:** `union()` o `|`
- **Intersección:** `intersection()` o `&`
- **Diferencia:** `difference()` o `-`
- **Diferencia simétrica:** `symmetric_difference()` o `^`

```

# Agregar un elemento
frutas.add("naranja")
print(frutas)
#> {'banana', 'manzana', 'naranja', 'cereza'}

# Eliminar un elemento
frutas.remove("banana")
print(frutas)
#> {'manzana', 'naranja', 'cereza'}

# Unión de conjuntos
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union = set1.union(set2)
print(union)
#> {1, 2, 3, 4, 5}

# Intersección de conjuntos
interseccion = set1.intersection(set2)
print(interseccion)
#> {3}

# Diferencia de conjuntos
diferencia = set1.difference(set2)
print(diferencia)
#> {1, 2}

# Diferencia simétrica
diff_simetrica = set1.symmetric_difference(set2)
print(diff_simetrica)
#> {1, 2, 4, 5}

```

Métodos adicionales de conjuntos

- **`issubset()`**: Verifica si un conjunto es subconjunto de otro.
- **`issuperset()`**: Verifica si un conjunto es superconjunto de otro.
- **`copy()`**: Crea una copia del conjunto.
- **`clear()`**: Elimina todos los elementos del conjunto.

```
set1 = {1, 2}
set2 = {1, 2, 3, 4, 5}

# Verificar subconjunto
print(set1.issubset(set2))
#> True

# Verificar superconjunto
print(set2.issuperset(set1))
#> True

# Copiar un conjunto
set3 = set1.copy()
print(set3)
#> {1, 2}

# Eliminar todos los elementos
set3.clear()
print(set3)
#> set()
```

Tuplas tuple

Las **tuplas** son colecciones ordenadas e **inmutables** de elementos. Son similares a las listas, pero una vez creadas, no pueden modificarse (no se pueden agregar, eliminar ni cambiar sus elementos).

En Python, una tupla se define utilizando paréntesis () o simplemente separando los elementos por comas.

Creación de una tupla

```
# Usando paréntesis
coordenadas = (10, 20)

# Sin usar paréntesis
colores = "rojo", "verde", "azul"

# Crear una tupla vacía
vacia = ()

# Tupla con un solo elemento (necesita una coma)
solo_un_elemento = (5,)
```

Acceso a los elementos de una tupla

Al igual que las listas, los elementos de una tupla se acceden mediante índices.

```
colores = ("rojo", "verde", "azul")

print(colores[0])
#> rojo

print(colores[-1])
#> azul

# Intentar modificar un elemento generará un error
# colores[0] = "amarillo" # TypeError: 'tuple' object does not support item assignment
```

Operaciones con tuplas

Aunque las tuplas son inmutables, se pueden realizar operaciones como concatenación y repetición.

```
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)

# Concatenación
concatenada = tupla1 + tupla2
print(concatenada)
#> (1, 2, 3, 4, 5, 6)

# Repetición
repetida = tupla1 * 2
print(repetida)
#> (1, 2, 3, 1, 2, 3)

# Verificar la existencia de un elemento
print(2 in tupla1)
#> True

# Obtener el índice de un elemento
print(tupla1.index(3))
#> 2

# Contar la cantidad de veces que aparece un elemento
print(tupla1.count(2))
#> 1
```

Métodos adicionales de tuplas

Las tuplas ofrecen métodos que facilitan su manipulación:

- `count()`: Cuenta cuántas veces aparece un elemento.
- `index()`: Retorna el índice de la primera aparición de un elemento.

```
tupla = (1, 2, 3, 2, 4, 2)

# Contar la cantidad de veces que aparece el número 2
print(tupla.count(2))
#> 3

# Obtener el índice de la primera aparición del número 2
print(tupla.index(2))
#> 1
```

Control de Flujo

El **control de flujo** en Python permite dirigir la ejecución de un programa en función de condiciones y repeticiones. Los elementos principales del control de flujo son las **condicionales**, los **bucles**, las **funciones**, la **recursividad** y el manejo de **excepciones**.

Condicionales

Las estructuras condicionales permiten ejecutar diferentes bloques de código según se cumplan o no ciertas condiciones. Las principales estructuras condicionales en Python son `if`, `elif` y `else`.

Sintaxis básica

```
if condición:  
    # Bloque de código si la condición es verdadera  
    ...  
elif otra_condición:  
    # Bloque de código si la otra condición es verdadera  
    ...  
else:  
    # Bloque de código si ninguna condición anterior es verdadera  
    ...
```

Ejemplos

Uso de `if` y `else`

```
edad = 20  
  
if edad >= 18:  
    print("Eres mayor de edad.")  
else:  
    print("Eres menor de edad.")  
#> Eres mayor de edad.
```

Uso de `if`, `elif` y `else`

```
nota = 85  
  
if nota >= 90:  
    print("Excelente")  
elif nota >= 70:  
    print("Aprobado")  
else:  
    print("Reprobado")  
#> Aprobado
```

Operadores de comparación y lógicos

Las condiciones suelen utilizar operadores de comparación (`==`, `!=`, `>`, `<`, `>=`, `<=`) y operadores lógicos (`and`, `or`, `not`).

```
a = 5
b = 10

if a < b and b < 15:
    print("a es menor que b y b es menor que 15.")
#> a es menor que b y b es menor que 15.
```

Anidación de condicionales

Es posible anidar estructuras condicionales dentro de otras para manejar múltiples condiciones.

```
edad = 25
tiene_permiso = True

if edad >= 18:
    if tiene_permiso:
        print("Puedes conducir.")
    else:
        print("Necesitas un permiso para conducir.")
else:
    print("Eres menor de edad y no puedes conducir.")
#> Puedes conducir.
```

Bucles

Los **bucles** permiten repetir un bloque de código múltiples veces. En Python, los bucles principales son `for` y `while`.

Bucle `for`

El bucle `for` se utiliza para iterar sobre una secuencia (como una lista, tupla, diccionario, conjunto o cadena).

Sintaxis básica

```
for elemento in secuencia:  
    # Bloque de código a ejecutar en cada iteración  
    ...
```

Ejemplos

```
frutas = ["manzana", "banana", "cereza"]  
  
for fruta in frutas:  
    print(fruta)  
#> manzana  
#> banana  
#> cereza
```

Bucle `while`

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

Sintaxis básica

```
while condición:  
    # Bloque de código a ejecutar mientras la condición sea verdadera  
    ...
```

Ejemplos

```
contador = 0

while contador < 5:
    print(f"Contador: {contador}")
    contador += 1
#> Contador: 0
#> Contador: 1
#> Contador: 2
#> Contador: 3
#> Contador: 4
```

Control de bucles

Python proporciona declaraciones para controlar el flujo dentro de los bucles:

- **break**: Sale completamente del bucle.
- **continue**: Omite el resto del bloque de código y pasa a la siguiente iteración.
- **else**: Se ejecuta después de que el bucle finaliza normalmente (no se ejecuta si el bucle se interrumpe con **break**).

Ejemplos

```

# Uso de break
for numero in range(10):
    if numero == 5:
        break
    print(numero)
#> 0
#> 1
#> 2
#> 3
#> 4

# Uso de continue
for numero in range(5):
    if numero == 2:
        continue
    print(numero)
#> 0
#> 1
#> 3
#> 4

# Uso de else
for numero in range(3):
    print(numero)
else:
    print("Bucle completado sin interrupciones.")
#> 0
#> 1
#> 2
#> Bucle completado sin interrupciones.

```

Nota: En el ejemplo del uso de `continue`, el número `2` se omite en la salida porque la sentencia `continue` hace que el bucle pase a la siguiente iteración cuando `numero == 2`.

Iteración

La **iteración** en Python se logra principalmente mediante los bucles `for` y `while`, como se describió anteriormente. Además, Python ofrece herramientas adicionales para manejar iteraciones de manera eficiente.

Función `range()`

La función `range()` genera una secuencia de números, que es comúnmente utilizada en los bucles `for`.

Sintaxis

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Ejemplos

```
# Del 0 al 4
for i in range(5):
    print(i)
#> 0
#> 1
#> 2
#> 3
#> 4

# Del 2 al 6
for i in range(2, 7):
    print(i)
#> 2
#> 3
#> 4
#> 5
#> 6

# Del 0 al 10 de 2 en 2
for i in range(0, 11, 2):
    print(i)
#> 0
#> 2
#> 4
#> 6
#> 8
#> 10
```

Comprehensiones de listas

Las **comprehensiones de listas** proporcionan una forma concisa de crear listas basadas en bucles.

Sintaxis básica

```
[nueva_expresión for elemento in secuencia if condición]
```

Ejemplos

```
# Crear una lista de cuadrados
cuadrados = [x**2 for x in range(5)]
print(cuadrados)
#> [0, 1, 4, 9, 16]

# Crear una lista de números pares
pares = [x for x in range(10) if x % 2 == 0]
print(pares)
#> [0, 2, 4, 6, 8]
```

Función `enumerate()`

La función `enumerate()` permite iterar sobre una secuencia y obtener el índice y el valor de cada elemento.

Ejemplo

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

for indice, fruta in enumerate(frutas):
    print(f"Índice {indice}: {fruta}")
#> Índice 0: manzana
#> Índice 1: banana
#> Índice 2: cereza
```

Funciones

Las **funciones** son bloques de código reutilizables que realizan una tarea específica. Permiten organizar el código, mejorar la legibilidad y facilitar el mantenimiento.

Definición de una función

Se define utilizando la palabra clave `def`, seguida del nombre de la función y paréntesis que pueden contener parámetros.

Sintaxis básica

```
def nombre_de_la_funcion(parámetros):
    # Bloque de código
    ...
    return valor
```

Ejemplos

```
# Función sin parámetros y sin valor de retorno
def saludar():
    print("¡Hola!")

saludar()
#> ¡Hola!

# Función con parámetros
def saludar(nombre):
    print(f"¡Hola, {nombre}!")

saludar("Ana")
#> ¡Hola, Ana!

# Función con valor de retorno
def sumar(a, b):
    return a + b

resultado = sumar(5, 3)
print(resultado)
#> 8
```

Parámetros y argumentos

Las funciones pueden tener **parámetros** (variables en la definición) y recibir **argumentos** (valores al llamar la función).

Tipos de parámetros

- **Posicionales:** Se asignan según el orden.
- **Por palabra clave:** Se asignan mediante el nombre del parámetro.
- **Predeterminados:** Tienen un valor por defecto si no se proporciona uno.
- **Variables:** Permiten pasar un número variable de argumentos (`*args` y `**kwargs`).

Ejemplos

```
# Parámetros posicionales
def multiplicar(a, b):
    return a * b

print(multiplicar(2, 3))
#> 6

# Parámetros por palabra clave
print(multiplicar(b=4, a=5))
#> 20

# Parámetros con valores predeterminados
def saludar(nombre, mensaje="¡Hola!"):
    print(f"{mensaje}, {nombre}.")

saludar("Carlos")
#> ¡Hola!, Carlos.
saludar("Carlos", "Buenos días")
#> Buenos días, Carlos.

# Parámetros variables (*args y **kwargs)
def funcion_variable(*args, **kwargs):
    print("Args:", args)
    print("Kwargs:", kwargs)

funcion_variable(1, 2, 3, nombre="Ana", edad=25)
#> Args: (1, 2, 3)
#> Kwargs: {'nombre': 'Ana', 'edad': 25}
```

Ámbito de las variables

Las variables definidas dentro de una función tienen un **ámbito local** y no son accesibles fuera de la función. Las variables definidas fuera de cualquier función tienen un **ámbito global**.

Ejemplo

```
x = 10 # Variable global

def imprimir_valor():
    y = 5 # Variable local
    print(f"x dentro de la función: {x}")
    print(f"y dentro de la función: {y}")

imprimir_valor()
#> x dentro de la función: 10
#> y dentro de la función: 5

print(x)
#> 10
# print(y) # Esto generaría un error: NameError: name 'y' is not defined
```

Funciones lambda

Las **funciones lambda** son funciones anónimas de una sola línea que se utilizan para operaciones simples.

Sintaxis básica

```
lambda parámetros: expresión
```

Ejemplos

```
# Función lambda para sumar dos números
sumar = lambda a, b: a + b
print(sumar(3, 4))
#> 7

# Uso de lambda con `map()`
numeros = [1, 2, 3, 4]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)
#> [1, 4, 9, 16]
```

Recursividad

La **recursividad** es una técnica donde una función se llama a sí misma para resolver subproblemas más pequeños del problema original. Es útil para problemas que pueden dividirse en subproblemas similares.

Ejemplo: Factorial

El factorial de un número n (denotado como $n!$) es el producto de todos los enteros positivos desde 1 hasta n .

Implementación recursiva

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
#> 120
```

Consideraciones

- **Caso base:** Es esencial definir un caso base para evitar llamadas infinitas.
- **Eficiencia:** Las soluciones recursivas pueden ser menos eficientes en términos de memoria y tiempo comparadas con las iterativas.
- **Límites de recursión:** Python tiene un límite en la profundidad de recursión (por defecto, alrededor de 1000 llamadas).

Ejemplo de límite de recursión

```
def contar_recursivo(n):
    print(n)
    return contar_recursivo(n + 1)

# contar_recursivo(1) # Esto generará un error: RecursionError: maximum recursion
```

Excepciones

Las **excepciones** son errores que ocurren durante la ejecución de un programa. Python proporciona mecanismos para manejar estos errores de manera controlada, evitando que el programa termine abruptamente.

Manejo básico de excepciones

Se utilizan las declaraciones `try`, `except`, `else` y `finally` para manejar excepciones.

Sintaxis básica

```
try:
    # Bloque de código que puede generar una excepción
    ...
except TipoDeExcepcion as e:
    # Bloque de código para manejar la excepción
    ...
else:
    # Bloque de código que se ejecuta si no ocurre ninguna excepción
    ...
finally:
    # Bloque de código que siempre se ejecuta, ocurra o no una excepción
    ...
```

Ejemplos

```
# Manejo de división por cero
try:
    resultado = 10 / 0
except ZeroDivisionError as e:
    print("Error: No se puede dividir por cero.")
#> Error: No se puede dividir por cero.
```

```
# Uso de else
try:
    numero = int(input("Ingresa un número: "))
except ValueError:
    print("Eso no es un número válido.")
else:
    print(f"Has ingresado el número {numero}.")
```

Múltiples excepciones

Se pueden manejar diferentes tipos de excepciones en un solo bloque `try`.

```
try:
    valor = int(input("Ingresa un número: "))
    resultado = 10 / valor
except ValueError:
    print("Eso no es un número válido.")
except ZeroDivisionError:
    print("No se puede dividir por cero.")
else:
    print(f"El resultado es {resultado}.")
```

Cláusula `finally`

La cláusula `finally` se ejecuta siempre, independientemente de si ocurrió una excepción o no. Es útil para liberar recursos, cerrar archivos, etc.

```
try:
    archivo = open("datos.txt", "r")
    contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no existe.")
else:
    print(contenido)
finally:
    archivo.close()
    print("Archivo cerrado.")
```

Lanzar excepciones

Es posible **lanzar** excepciones de manera intencional utilizando la declaración `raise`.

```
def dividir(a, b):
    if b == 0:
        raise ValueError("El divisor no puede ser cero.")
    return a / b

try:
    dividir(10, 0)
except ValueError as e:
    print(e)
#> El divisor no puede ser cero.
```

Creación de excepciones personalizadas

Se pueden crear excepciones personalizadas definiendo una clase que herede de `Exception`.

```
class EdadInvalidaError(Exception):
    """Excepción lanzada cuando la edad es inválida."""
    pass

def validar_edad(edad):
    if edad < 0:
        raise EdadInvalidaError("La edad no puede ser negativa.")
    print("Edad válida.")

try:
    validar_edad(-5)
except EdadInvalidaError as e:
    print(e)
#> La edad no puede ser negativa.
```

Funciones

En python, una función es un bloque de código que realiza una tarea específica. Las funciones permiten dividir un programa en bloques más pequeños y fáciles de entender. Además, las funciones permiten reutilizar código y evitar la repetición de código.

Un ejemplo de uso es:

```
nombre = "Juan"
sueldo = 5000
print("Informe de sueldo")
print(f" Nombre: {nombre}")
print(f" Sueldo: ${sueldo}")
print("")

nombre = "Ana"
sueldo = 6000
print("Informe de sueldo")
print(f" Nombre: {nombre}")
print(f" Sueldo: ${sueldo}")
print("")
```

Podemos simplificar el código anterior utilizando una función:

```
def imprimir_informe():
    print("Informe de sueldo")
    print(f" Nombre: {nombre}")
    print(f" Sueldo: ${sueldo}")
    print("")

nombre = "Juan"
sueldo = 5000
imprimir_informe()

nombre = "Ana"
sueldo = 6000
imprimir_informe()
```

Ahora la función nos permite interpretar el código de manera más sencilla, ahora con el nombre de la función es más fácil de entender que es lo que hace. A la vez permite reutilizar el código, si necesitamos imprimir otro informe de sueldo, solo necesitamos llamar a la función

`imprimir_informe()`.

Ahora bien, esta función depende de las variables `nombre` y `sueldo`, si cambiamos el nombre de la variable, la función no funcionará correctamente. Para solucionar este problema, podemos pasar los valores de las variables como argumentos a la función.

```

def imprimir_informe(nombre, sueldo):
    print("Informe de sueldo")
    print(f" Nombre: {nombre}")
    print(f" Sueldo: ${sueldo}")
    print("")

n = "Juan"
s = 5000
imprimir_informe(n, s)

imprimir_informe("Ana", 6000)
imprimir_informe("Pedro", 7000)

```

Con el paso de parametros podemos reutilizar la función para imprimir diferentes informes de sueldo. Ya no dependemos de las variables `nombre` y `sueldo`.

Sin embargo esta función todavía puede ser mejorada. La función 'imprimir_informe' le da formato a los datos y los imprime. Pero si necesitamos guardar el informe en un archivo, tendríamos que modificar la función. Para evitar esto, podemos dividir la función en dos funciones, una que le da formato a los datos y otra que imprime los datos.

```

def generar_informe(nombre, sueldo):
    return f"""
Informe de sueldo
- Nombre: {nombre}
- Sueldo: ${sueldo}"""

informe = generar_informe("Juan", 5000)
print(informe)

informe = generar_informe("Ana", 6000)
print(informe)

informe = generar_informe("Pedro", 7000)
with open("informe.txt", "w") as archivo:
    archivo.write(informe)

```

Pasaje de argumentos

Python es muy flexible en cuanto al pasaje de argumentos a las funciones. Podemos pasar argumentos por posición, por nombre, y podemos tener argumentos con valores por defecto.

Pasaje de argumentos por posición

```
def suma(a, b):
    return a + b

print(suma(2, 3)) # Parametros por posición
#> 5
```

El pasaje de parámetros implica una asignación de las variables a los parámetros de la función en el orden en que se pasan los argumentos. Es equivalente a la asignación simultánea `a = 2` y `b = 3`, o `a, b = 2, 3`.

Pasaje de argumentos por nombre

```
def par(x, y):
    print(f"x={x} y={y}")

par(2, 3) # Parametros por posición
#> x=2 y=3

par(y=3, x=2) # Parametros por nombre (el orden no importa)
#> x=2 y=3

par(2, y=3) # Parametros por posición y nombre
#> x=2 y=3

# par(x=2, 3) # Error de sintaxis. Los argumentos por nombre deben ir después de los
```

Argumentos con valores por defecto

En algunas ocasiones, es útil definir valores por defecto para los argumentos de una función. Esto permite llamar a la función sin pasar todos los argumentos. Hace que el uso sea fácil para los casos más comunes, pero permite la flexibilidad de cambiar los valores por defecto si es necesario.

```

def suma(a, b=0, c=0, d=0):
    return a + b

print(suma(2)) # Solo pasamos un argumento
#> 2

print(suma(2, 3)) # Pasamos dos argumentos
#> 5

print(suma(2, 3, 4)) # Pasamos tres argumentos
#> 5

print(suma(2, 3, 4, 5)) # Pasamos cuatro argumentos
#> 5

```

En este caso la función suma tiene 4 argumentos, pero solo uno es obligatorio, los otros tres tienen un valor por defecto de 0. Esto permite llamar a la función con un solo argumento, o con dos, tres o cuatro argumentos. Sin embargo, si se pasan más de cuatro argumentos, la función lanzará un error.

Si quisieramos hacer más flexible la función podríamos intentar que acepte un número variable de argumentos. Para ello podríamos recibir una lista de valores en lugar de argumentos separados.

Si bien esto es una solución hace más difícil el uso original de la función, ya que ahora se necesita pasar una lista de valores en lugar de argumentos separados.

```

def suma(lista):
    suma = 0
    for valor in lista:
        suma += valor
    return suma

print(suma([2])) # Solo pasamos un argumento
#> 2
print(suma([1, 2, 3, 4, 5, 6])) # Pasamos varios argumentos
#> 21

```

Esto es posible y es válido pero no es la mejor solución. Python nos permite definir funciones que acepten un número variable de argumentos. Para ello podemos usar el operador `*` para indicar que la función acepta un número variable de argumentos.

```
def suma(*args):
    suma = 0
    for valor in args:
        suma += valor
    return suma

print(suma(2)) # Solo pasamos un argumento
#> 2
print(suma(1, 2, 3, 4, 5, 6)) # Pasamos varios argumentos
#> 21
```

En este caso, la función `suma` acepta un número variable de argumentos. Los argumentos se almacenan en una tupla llamada `args`. La función recorre la tupla y suma los valores. Ahora podemos llamar a la función con un solo argumento, o con varios argumentos.

Pero si tenemos los datos inicialmente en una lista, no podríamos pasar la lista directamente a la función. Para solucionar este problema, podemos usar el operador `*` para desempaquetar la lista y pasar los valores como argumentos.

```
valores = [1, 2, 3, 4, 5, 6]
print(suma(*valores)) # Pasamos varios argumentos
#> 21
```

El operador `*` desempaquetá la lista cuando se pasa como argumento a la función. Y empaqueta los valores en una tupla cuando se recibe en la función.

Este concepto de empaquetar y desempaquetar se puede usar en una forma más general combinando argumentos fijos, argumentos por nombre, argumentos con valores por defecto y argumentos variables.

En primer lugar pueden definirse los argumentos fijos, luego los argumentos con valores por defecto, luego los argumentos variables y finalmente los argumentos por nombre. El orden es importante, los argumentos por nombre deben ir después de los argumentos por posición.

Existe un operador `**` que permite pasar un número variable de argumentos por nombre. Los argumentos se almacenan en un diccionario llamado `kwargs`.

```

def parametros(a, b, *args, c=0, d=0, **kwargs):
    print(f"a={a} b={b} args={args} c={c} d={d} kwargs={kwargs}")

parametros(1, 2) # Solo pasamos dos argumentos
#> a=1 b=2 args=() c=0 d=0 kwargs={}

parametros(1, 2, 3, 4, 5, c=6, d=7) # Pasamos varios argumentos y argumentos por nombre
#> a=1 b=2 args=(3, 4, 5) c=6 d=7 kwargs={}

parametros(1, 2, 3, 4, 5, c=6, d=7, e=8, f=9) # Pasamos varios argumentos y argumentos por nombre
#> a=1 b=2 args=(3, 4, 5) c=6 d=7 kwargs={'e': 8, 'f': 9}

```

En este caso, la función `parametros` acepta un número variable de argumentos por posición, argumentos con valores por defecto, y un número variable de argumentos por nombre. Los argumentos por posición se almacenan en una tupla llamada `args`, y los argumentos por nombre se almacenan en un diccionario llamado `kwargs`.

Así como `*` desempaquetá una lista, `**` desempaquetá un diccionario. Por lo tanto, si tenemos un diccionario con los argumentos por nombre, podemos desempaquetar el diccionario y pasar los argumentos a la función.

```

valores = {"c": 6, "d": 7, "e": 8, "f": 9}
parametros(1, 2, 3, 4, 5, **valores) # Pasamos varios argumentos y argumentos por nombre

def informer(nombre, sueldo):
    print(f"Nombre: {nombre} Sueldo: ${sueldo}")

empleado = {"nombre": "Juan", "sueldo": 5000}
informer(**empleado)

# También funciona al revés
def listar(**kwargs):
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")

listar(nombre="Juan", sueldo=5000)
#> nombre: Juan
#> sueldo: 5000

listar(nombre="Ana", sueldo=6000, edad=30)
#> nombre: Ana
#> sueldo: 6000
#> edad: 30

```

En este caso, el diccionario `valores` contiene los argumentos por nombre. Al pasar el diccionario a la función `parametros`, los argumentos se desempaquetan y se pasan a la función.

Retorno de valores

Cuando una función termina de ejecutarse, puede devolver un valor. Para devolver un valor, se utiliza la instrucción `return`. Si no se especifica un valor de retorno, la función devuelve `None`.

```
# Una funcion
def suma(a, b):
    return a + b

resultado = suma(2, 3)

# Un procedimiento no devuelve nada
def imprimir_suma(a, b):
    print(f"{a} + {b} = {a + b}")

# Pero si queremos podemos usarla como una funcion

resultado = imprimir_suma(2, 3)
print(resultado)
#> None
```

En este caso, la función `suma` devuelve la suma de los argumentos `a` y `b`. La función `imprimir_suma` no devuelve nada, por lo tanto, la variable `resultado` contiene `None`.

Las funciones pueden retornar cualquier tipo de valor, incluso colecciones o funciones.

Una forma muy comoda de retornar multiples valores es retornar una tupla. Las tuplas tienen la ventaja de que pueden contener cualquier tipo de valor, y pueden ser desempaquetadas facilmente. Incluso no es necesaria el uso de los parentesis para crear una tupla, simplemente separar los valores con comas.

```

def minmax(lista):
    minimo = maximo = lista[0]
    for x in lista:
        if x < minimo: minimo = x
        if x > maximo: maximo = x
    return (minimo, maximo)

valores = [1, 2, 3, 4, 5]
(minimo, maximo) = minmax(valores)
print(f"Minimo: {minimo} Maximo: {maximo}")

# Tambien se puede hacer sin parentesis
def divmod(a, b):
    return a // b, a % b

cociente, resto = divmod(10, 3)
print(f"Cociente: {cociente} Resto: {resto}")

```

En este caso, la función `minmax` devuelve una tupla con el mínimo y el máximo de una lista. La función `divmod` devuelve una tupla con el cociente y el resto de una división.

Funciones como objetos

En Python, las funciones son objetos de primera clase. Esto significa que las funciones pueden ser asignadas a variables, almacenadas en estructuras de datos, pasadas como argumentos a otras funciones, y retornadas como valores de otras funciones.

```

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

saludo = saludar
print(saludo("Juan"))
#> Hola Juan

```

Cuando definimos una función en realidad estamos definiendo una variable que contiene la función. Por lo tanto, podemos asignar la función a otra variable y llamar a la función a través de la nueva variable.

Si se usa la variable con parentesis, se llama a la función, si se usa sin parentesis, se accede a la función.

```

def suma(a, b): return a + b
def producto(a, b): return a * b

# Aplica la función a cada elemento de la lista en forma acumulativa
def reducir(funcion, lista):
    resultado = lista[0]
    for x in lista[1:]:
        resultado = funcion(resultado, x)
    return resultado

valores = [1, 2, 3, 4, 5]
print(reducir(suma, valores))
#> 15

print(reducir(producto, valores))
#> 120

nombres = ["Juan", "Ana", "Pedro"]
print(reducir(lambda a, b: a + " " + b, nombres))
#> Juan Ana Pedro

print(sorted(nombres))
#> ['Ana', 'Juan', 'Pedro']
def longitud(nombre):
    return len(nombre)

print(sorted(nombres, key = longitud))
#> ['Ana', 'Juan', 'Pedro']

print(sorted(nombres, key = lambda nombre: len(nombre)))
#> ['Ana', 'Juan', 'Pedro']

print(sorted(nombres, key = len))
#> ['Ana', 'Juan', 'Pedro']

```

En este caso, la función `reducir` recibe una función y una lista, y aplica la función a cada elemento de la lista en forma acumulativa. La función `sorted` recibe una lista y un argumento `key` que indica la función que se debe aplicar a cada elemento de la lista antes de ordenarla.

Funciones anónimas

En Python, las funciones anónimas son funciones que no tienen nombre. Se definen con la palabra clave `lambda`, seguida de los argumentos y el cuerpo de la función. Las funciones anónimas son útiles para definir funciones simples en una sola línea.

```
suma = lambda a, b: a + b
print(suma(2, 3))

producto = lambda a, b: a * b
print(producto(2, 3))
```

La ventaja de las funciones anónimas es que son concisas y se pueden definir en una sola línea. La desventaja es que son menos legibles que las funciones normales, y no pueden contener múltiples instrucciones.

Las funciones anónimas son útiles cuando se necesita una función temporal, o cuando se necesita una función simple para pasar como argumento a otra función.

```
valores = [1, 2, 3, 4, 5]
print(reducir(lambda a, b: a + b, valores))
#> 15

print(reducir(lambda a, b: a * b, valores))
#> 120

nombres = ["Juan", "Ana", "Pedro"]
print(reducir(lambda a, b: a + " " + b, nombres))
#> Juan Ana Pedro

print(sorted(nombres, key = lambda nombre: len(nombre)))
#> ['Ana', 'Juan', 'Pedro']
```

Parametros por referencia

En python toda asignación de variables es por referencia. Esto significa que cuando se pasa una variable a una función, se pasa la referencia a la variable, no una copia de la variable. Por lo tanto, si se modifica la variable dentro de la función, se modifica la variable original.

```
def duplicar(lista):
    for i in range(len(lista)):
        lista[i] *= 2

valores = [1, 2, 3, 4, 5]
duplicar(valores)
print(valores)
#> [2, 4, 6, 8, 10]
```

En este caso, la función `duplicar` recibe una lista y multiplica cada elemento por 2. Como la lista se pasa por referencia, la lista original se modifica.

Si se necesita modificar una variable dentro de una función sin modificar la variable original, se puede hacer una copia de la variable antes de modificarla.

```
def duplicar(lista):
    lista = lista.copy()
    for i in range(len(lista)):
        lista[i] *= 2
    return lista

valores = [1, 2, 3, 4, 5]
valores_duplicados = duplicar(valores)
print(valores)
#> [1, 2, 3, 4, 5]
print(valores_duplicados)
#> [2, 4, 6, 8, 10]
```

Funciones anidadas

En Python, las funciones pueden definirse dentro de otras funciones. Las funciones anidadas pueden acceder a las variables de la función externa, y pueden devolverse como valores de la función externa.

```
def operacion(operador):
    def suma(a, b):
        return a + b

    def resta(a, b):
        return a - b

    if operador == "+":
        return suma
    elif operador == "-":
        return resta

funcion = operacion("+")
print(funcion(2, 3))
#> 5

funcion = operacion("-")
print(funcion(2, 3))
#> -1
```

Funciones recursivas

En Python, una función puede llamarse a sí misma. Esto se conoce como recursión. La recursión es útil para resolver problemas que pueden dividirse en subproblemas más pequeños.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))

def suma(lista):
    if len(lista) == 0:
        return 0
    else:
        return lista[0] + suma(lista[1:])

valores = [1, 2, 3, 4, 5]
print(suma(valores))
#> 15

# Busqueda lineal
def buscar(lista, valor):
    if len(lista) == 0:
        return False
    elif lista[0] == valor:
        return True
    else:
        return buscar(lista[1:], valor)
```

En este caso, la función `factorial` calcula el factorial de un número utilizando recursión.

En la función `suma` y `buscar` se usa la recursión para recorrer una lista. Esto reemplaza el uso de un ciclo `for` para recorrer la lista.

Una curiosidad... es 1930, el matemático Kurt Gödel demostró que cualquier función computable puede ser calculada mediante recursión. Esto se conoce como la tesis de Church-Turing.

Funciones generadoras

En Python, una función generadora es una función que devuelve un generador. Un generador es un tipo especial de iterador que permite recorrer una secuencia de valores de uno en uno.

```
def contador(n):
    for i in range(n):
        yield i

for i in contador(5):
    print(i)
```

Las funciones generadoras usan `yield` en lugar de `return` para devolver valores. Cuando se llama a una función generadora, se obtiene un generador. Un generador es un iterador que permite recorrer los valores de uno en uno.

Cuando encuentra un `yield`, la función generadora se detiene y devuelve el valor. Cuando se llama al generador nuevamente, la función generadora se reanuda en el `yield` y continúa la ejecución.

```
def tres():
    yield 1
    yield 2
    yield 3

for i in tres():
    print(i, end=" ")
#> 1 2 3

def cuenta_regresiva(n):
    while n > 0:
        yield n
        n -= 1
    yield "Despegue!"
for i in contar(5):
    print(i, end=" ")
#> 5 4 3 2 1 Despegue!
```

En este caso, la función `contador` es una función generadora que devuelve un generador. El generador permite recorrer los valores de 0 a `n-1`. La función `tres` es una función generadora que devuelve un generador que permite recorrer los valores 1, 2 y 3. La función

`cuenta_regresiva` es una función generadora que devuelve un generador que permite recorrer los valores de `n` a 1, y finalmente el valor "Despegue!".

Funciones de orden superior

En Python, una función de orden superior es una función que recibe una o más funciones como argumentos, o devuelve una función como resultado.

```
def aplicar(funcion, valor):
    return funcion(valor)

def cuadrado(x):
    return x ** 2

def cubo(x):
    return x ** 3

print(aplicar(cuadrado, 2))
#> 4

print(aplicar(cubo, 2))

def multiplicador(n):
    def funcion(x):
        return x * n
    return funcion

duplicar = multiplicador(2)
print(duplicar(3))
#> 6

triplicar = multiplicador(3)
print(triplicar(3))
#> 9
```

En este caso, la función `aplicar` es una función de orden superior que recibe una función y un valor, y aplica la función al valor.

La función `multiplicador` es una función de orden superior que recibe un número y devuelve una función que multiplica un valor por ese número.

Con la función `multiplicador` se puede crear una función que multiplica por 2 y otra que multiplica por 3.

Decoradores

En Python, un decorador es una función que recibe una función como argumento, y devuelve otra función. Los decoradores se utilizan para modificar o extender el comportamiento de una función sin modificar su código.

```
def decorador(funcion):
    def nueva_funcion(*args, **kwargs):
        print(f"Antes de llamar a {funcion.__name__}")
        resultado = funcion(*args, **kwargs)
        print(f"Despues de llamar a {funcion.__name__}")
        return resultado
    return nueva_funcion

@decorador
def saludar(nombre):
    return f"Hola {nombre}"

print(saludar("Juan"))
#> Antes de llamar a saludar
#> Hola Juan
#> Despues de llamar a saludar
```

¿Cómo surge la Orientación a Objetos?

Supongamos que tenemos que resolver el siguiente problema:

Revisar si una expresión aritmética tiene los paréntesis balanceados.

Este es un problema que ya vimos, pero lo complicamos un poco más: digamos que la expresión aritmética puede tener paréntesis, corchetes y llaves.

Revisar si una expresión aritmética tiene los paréntesis, corchetes y llaves balanceados.

A diferencia del primer problema, donde bastaba con contar la cantidad de paréntesis de apertura y cierre, ahora debemos verificar que los paréntesis, corchetes y llaves estén correctamente balanceados y anidados.

Para ello, debemos recordar cuál fue el último paréntesis, corchete o llave que abrimos y verificar que el cierre sea el correspondiente.

Para realizar esta función, debemos ir almacenando las **aperturas** y verificar que los **cierres** sean los correspondientes. La última **apertura** que se almacenó debe ser la primera en cerrarse.

```
def parentesis_balanceados(expresion):
    pila = [None] * len(expresion) # Crear un array con el tamaño de la expresión
    cantidad = 0 # Inicializar el contador de elementos

    apertura = {'(': ')', '{': '}', '[': ']'}
    cierre = {')': '(', '}': '{', ']': '['}

    for char in expresion:
        if char in apertura:
            pila[cantidad] = char # Agregar el paréntesis de apertura a la pila
            cantidad += 1 # Incrementar el contador
        elif char in cierre:
            if cantidad == 0 or pila[cantidad - 1] != cierre[char]:
                return False # La pila está vacía o no coincide el tipo de paréntesis
            cantidad -= 1 # Decrementar el contador

    return cantidad == 0 # La pila debe estar vacía si los paréntesis están balanceados
```

Este es un ejemplo de una función que verifica si los paréntesis, corchetes y llaves están balanceados.

Este enfoque funciona bien, pero en el código se mezclan los datos y la lógica. Una solución alternativa sería usar una estructura de datos abstracta llamada **pila**. La pila define tres funciones básicas y luego se puede usar para resolver el problema.

```
pila = [None] * 10 # Crear una pila con capacidad para 10 elementos
cantidad = 0 # Inicializar el contador de elementos en la pila

def push(elemento):
    global cantidad
    pila[cantidad] = elemento
    cantidad += 1

def pop():
    global cantidad
    cantidad -= 1
    return pila[cantidad]

def is_empty():
    return cantidad == 0
```

Con esta estructura de datos, se puede reescribir la función anterior de la siguiente manera:

```
def parentesis_balanceados(expresion):
    apertura = {'(': ')', '{': '}', '[': ']'}
    cierre   = {')': '(', '}': '{', ']': '['}

    for char in expresion:
        if char in apertura:
            push(char)
        elif char in cierre:
            if is_empty() or pop() != cierre[char]:
                return False

    return is_empty()
```

Esta solución es más clara y fácil de entender. La pila es una estructura de datos que se puede reutilizar en otros problemas.

La pila es un ejemplo de una estructura de datos abstracta. Una estructura de datos abstracta define un conjunto de operaciones y propiedades, pero no especifica cómo se implementan esas operaciones y propiedades.

Ahora no nos interesa cómo se implementa la pila, solo nos interesa que podemos agregar elementos, quitar elementos y verificar si está vacía.

El problema con esta implementación es que la pila es global y si se llama a la función `parentesis_balanceados` varias veces, la pila se va a mezclar entre las llamadas.

Una solución sería encapsular la pila en métodos que se encarguen de gestionar el estado interno.

```

def Stack():      # Crear una pila vacía (constructor)
    return {'cantidad': 0, 'pila': [None] * 10} # Usamos un diccionario para encapsular la pila

def push(pila, elemento):
    pila['pila'][pila['cantidad']] = elemento
    pila['cantidad'] += 1

def pop(pila):
    pila['cantidad'] -= 1
    return pila['pila'][pila['cantidad']]

def is_empty(pila):
    return pila['cantidad'] == 0

def parentesis_balanceados(expresion):
    pila = Stack()

    apertura = {'(': ')', '{': '}', '[': ']'}
    cierre   = {')': '(', '}': '{', ']': '['}

    for char in expresion:
        if char in apertura:
            push(pila, char)
        elif char in cierre:
            if is_empty(pila) or pop(pila) != cierre[char]:
                return False

    return is_empty(pila)

```

Ahora la pila está encapsulada en un diccionario y se pasa como argumento a las funciones que la manipulan.

Esto nos permite tener varias pilas independientes y no mezclar los datos entre llamadas a la función `parentesis_balanceados`.

Podríamos usar varias pilas para resolver problemas diferentes.

```

a = Stack() # Creo una 'instancia' de la pila
b = Stack() # Creo otra 'instancia' de la pila

push(a, 10) # Agrego elementos a la pila 'a'
push(b, 20) # Agrego elementos a la pila 'b'
push(a, 30) # Agrego otro elemento a la pila 'a'

while not is_empty(a): # Muevo los elementos de la pila 'a' a la pila 'b'
    push(b, pop(a))

while not is_empty(b): # Imprimo los elementos de la pila 'b'
    print(pop(b))

#> 30
#> 10
#> 20

```

Una de las ventajas de este enfoque es que el programador no tiene que pensar en cómo fue implementada la pila, solo tiene que saber cómo usarla.

Por ejemplo, si aprovechamos que en Python disponemos de listas y sus métodos `append` y `pop`, podemos reescribir la pila de la siguiente manera:

```

def Stack():      # Crear una pila vacía (constructor)
    return []     # Usamos una lista para encapsular la pila

def push(pila, elemento):
    pila.append(elemento)

def pop(pila):
    return pila.pop()

def is_empty(pila):
    return len(pila) == 0

```

Esta implementación es más simple y utiliza las listas de Python para almacenar los elementos de la pila. No sería tan eficiente como la implementación anterior, pero es más simple y fácil de entender.

Este es un ejemplo de cómo surge la programación orientada a objetos. Se comienza a **encapsular** los datos y la lógica en estructuras de datos y funciones que operan sobre ellas.

Sin embargo, todavía tenemos un problema con esta implementación. Supongamos que tenemos que implementar una cola, que es una estructura de datos similar a una pila, pero en la que el

primer elemento que se agrega es el primero en salir.

```
def Queue():      # Crear una cola vacía (constructor)
    return []     # Usamos una lista para encapsular la cola

def enqueue(cola, elemento):
    cola.append(elemento)

def dequeue(cola):
    return cola.pop(0)

def is_empty(cola):
    return len(cola) == 0
```

Esta implementación es simple y reutilizable, pero tenemos un problema: ambas estructuras tienen un método que se llama `is_empty`, y si queremos usar ambas al mismo tiempo, vamos a tener un conflicto. En este caso puntual no es un problema grave porque si vemos la implementación de `is_empty`, es la misma para ambas estructuras.

Pero eso rompe el **encapsulamiento** de los datos y la lógica. Si queremos cambiar la implementación de `is_empty` en una de las estructuras, vamos a tener que cambiarla en todas las estructuras.

Una solución sería encapsular las estructuras en clases y definir los métodos como métodos de instancia.

Podríamos llamar a la función `is_empty` de otra manera para evitar el conflicto.

```
def is_empty_queue(cola):
    return len(cola) == 0

def is_empty_stack(pila):
    return len(pila) == 0
```

Pero esto no es una solución elegante. Antes usábamos la función `is_empty` para saber si una estructura estaba vacía, ahora tenemos que recordar qué función usar para cada estructura.

Lo que queremos en realidad es que cada estructura tenga su propio método `is_empty` y que se llame de la misma manera. Es lo que se llama **polimorfismo**: que cada objeto se comporte de la misma manera pero adecue su conducta al tipo de datos que le corresponda.

Para esto surgen las clases y los objetos. Una clase es una plantilla que define las propiedades y los métodos de un objeto. Un objeto es una instancia de una clase.

```
class Stack:  
    def __init__(self):      # Crear una pila vacía (constructor)  
        self.lista = []       # Usamos una lista para encapsular la pila  
  
    def push(self, elemento):  
        self.lista.append(elemento)  
  
    def pop(self):  
        return self.lista.pop()  
  
    def is_empty(self):  
        return len(self.lista) == 0  
  
class Queue:  
    def __init__(self):      # Crear una cola vacía (constructor)  
        self.lista = []       # Usamos una lista para encapsular la cola  
  
    def enqueue(self, elemento):  
        self.lista.append(elemento)  
  
    def dequeue(self):  
        return self.lista.pop(0)  
  
    def is_empty(self):  
        return len(self.lista) == 0
```

Al definir las clases de esta manera, estamos creando funciones que se aplican a los datos de la clase. Estas funciones se llaman **métodos** y se definen dentro de la clase.

Aunque no es habitual, podemos usar las funciones directamente en la forma tradicional, solo que prefijadas por el nombre de la clase.

```
a = Stack()  
  
Stack.push(a, 10)  
Stack.push(a, 20)  
while not Stack.is_empty(a):  
    print(Stack.pop(a))  
  
b = Queue()  
Queue.enqueue(b, 10)  
Queue.enqueue(b, 20)  
while not Queue.is_empty(b):  
    print(Queue.dequeue(b))
```

Ahora no hay conflicto de nombres porque cada instancia de la clase tiene sus propios métodos.

Sin embargo, por más que esta forma de usar las clases sea válida, no es la más común. Lo más común es usar la notación de punto `.` para acceder a los métodos de una clase.

```
a = Stack()
a.push(10)
a.push(20)
while not a.is_empty():
    print(a.pop())

b = Queue()
b.enqueue(10)
b.enqueue(20)
while not b.is_empty():
    print(b.dequeue())
```

Ahora el código es más claro y fácil de entender. Se puede ver que `a` es una pila y `b` es una cola, y se pueden usar de la misma manera.

En resumen, la programación orientada a objetos es un paradigma de programación que se basa en el concepto de objetos, que son instancias de clases. Las clases definen las propiedades y los métodos de los objetos y permiten encapsular los datos y la lógica en una estructura coherente y reutilizable.

Observemos algunos elementos de la sintaxis de Python que nos permiten trabajar con clases y objetos.

Método `__init__`

El primero es que se usa un nombre especial para inicializar los valores de la clase: `__init__`. Este método se llama **constructor** y se utiliza para inicializar los valores de los objetos cuando se crean.

Si vamos muy al detalle, en realidad no es un constructor, sino un inicializador. En Python, los objetos se crean primero y luego se inicializan con el método `__init__`, pero a efectos prácticos, se puede considerar un constructor.

Existen muchos **métodos especiales** que se pueden definir en una clase para modificar el comportamiento de los objetos. Por ejemplo, el método `__str__` se utiliza para definir la representación en forma de cadena de un objeto. Estos métodos los usa Python para realizar

operaciones internas y no es necesario llamarlos directamente. Veremos algunos de estos métodos más adelante.

Parámetro `self`

El segundo elemento es que cada **método** recibe un primer parámetro que se llama `self`. Este parámetro se utiliza para referenciar al objeto que se está manipulando. Es una convención en Python utilizar `self` como nombre de este parámetro, pero se puede utilizar cualquier nombre.

Acceso a Propiedades y Métodos

El tercer elemento es que se puede acceder a las propiedades y los métodos de un objeto utilizando la notación de punto `.`. Por ejemplo, si tenemos un objeto `a` de la clase `Stack`, podemos llamar a los métodos `a.push()`, `a.pop()` y `a.is_empty()`.

Definición de Propiedades

El cuarto elemento es que se pueden definir propiedades de un objeto utilizando la palabra clave `self`. Por ejemplo, si queremos definir una propiedad `lista` en la clase `Stack`, esta propiedad se crea al asignarle un valor y puede ser accedida y modificada utilizando la notación de punto `.`.

Python permite acceder a las propiedades de un objeto desde fuera de la clase. Si bien esto es posible, no es recomendable ya que se pierde el encapsulamiento de los datos y la lógica. Es mejor definir métodos para acceder y modificar las propiedades de un objeto.

```
a = Stack()
print(a.lista) # Output: []
a.push(10)
a.push(20)
print(a.lista) # Output: [10, 20]

a.lista.clear() # Se puede acceder a la propiedad lista y modificarla directamente
print(a.is_empty()) # Output: True
```

El quinto elemento es que se pueden definir **métodos estáticos y de clase** en una clase. Un método estático es un método que no recibe el parámetro `self` y se puede llamar sin crear una

instancia de la clase. Un método de clase es un método que recibe el parámetro `cls` en lugar de `self` y se puede llamar utilizando la clase en lugar de un objeto.

```
class Stack:  
    @staticmethod  
    def is_empty_list(lista):  
        return len(lista) == 0  
  
    @classmethod  
    def from_list(cls, lista):  
        stack = cls()  
        stack.lista = lista  
        return stack  
  
a = Stack.from_list([10, 20])  
print(Stack.is_empty_list(a.lista)) # Output: False
```

Este tipo de métodos se usan para aprovechar que los nombres están definidos dentro de la clase y de esta manera no entran en conflicto con otros nombres definidos fuera de la clase. La distinción entre un método de clase y un método estático es útil y se puede utilizar según la necesidad. Pero la idea es que se pueden llamar sin crear una instancia de la clase.

Nótese que en el método `from_list` se crea una instancia de la clase `Stack` y se inicializa la propiedad `lista` con el valor pasado como argumento. Este es un ejemplo de un método de clase que se utiliza para crear instancias de la clase.

Decoradores en Clases

En ambos casos se usa un **decorador** para indicarle a Python el comportamiento especial del método. Los decoradores son una característica avanzada de Python que permite modificar el comportamiento de una función o método.

Otro decorador que se puede utilizar en una clase es `@property`. Este decorador se utiliza para definir propiedades de solo lectura en una clase. Por ejemplo, en lugar de usar un método para averiguar si una pila está vacía, podríamos usar una propiedad de solo lectura.

```

class Stack:
    def __init__(self):
        self.lista = []

    def push(self, elemento):
        self.lista.append(elemento)

    def pop(self):
        return self.lista.pop()

    @property
    def empty(self):
        return len(self.lista) == 0

a = Stack()
a.push(10)
a.push(20)
while not a.empty:
    print(a.pop())

```

En este caso, la propiedad `empty` se define utilizando el decorador `@property` y se puede acceder a ella como si fuera un atributo de la clase. Sin embargo, no se puede modificar la propiedad `empty` directamente, ya que es de solo lectura.

Este concepto de propiedades de solo lectura es útil para definir propiedades que se calculan a partir de otras propiedades o métodos de la clase, pero también se puede usar para proteger propiedades de la clase de modificaciones no deseadas.

Protección de Propiedades con Getters y Setters

Veamos un ejemplo de cómo usar propiedades para proteger el acceso a las propiedades de una clase.

Supongamos que tenemos que modelar un producto en una tienda y queremos asegurarnos de que el precio del producto sea siempre mayor que cero. Podríamos guardar el precio en una variable interna `_precio` y acceder al mismo a través de un método `get_precio` y modificarlo a través de un método `set_precio`. De esta manera, podemos controlar que el precio sea siempre mayor que cero.

```

class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.set_precio(precio)

    def get_precio(self):
        return self._precio

    def set_precio(self, precio):
        if precio <= 0:
            raise ValueError("El precio debe ser mayor que cero.")
        self._precio = precio

    def __str__(self):
        return f"{self.nombre}: ${self.get_precio()}"


producto = Producto("Laptop", 1000)
print(producto)          # Output: Laptop: $1000
print(producto.get_precio()) # Output: 1000

producto.set_precio(1500)      # Modifica el precio del producto
print(producto)          # Output: Laptop: $1500
print(producto.get_precio()) # Output: 1500

# Si intentamos modificar el precio a un valor inválido, se lanza una excepción
# producto.set_precio(-1000) # ValueError: El precio debe ser mayor que cero.

```

Esto soluciona el problema del acceso y modificación de las propiedades de la clase. Ahora el precio del producto es una propiedad protegida que solo se puede modificar a través del método `set_precio`. En realidad, se sigue pudiendo acceder a la propiedad `_precio` directamente, pero se desaconseja hacerlo. Python no tiene un mecanismo para proteger las propiedades de una clase de modificaciones no deseadas, pero usa la convención de que las propiedades que comienzan con un guion bajo `_` son privadas y no deben ser accedidas directamente.

Este patrón de diseño se llama **getter** y **setter** y se utiliza para controlar el acceso y la modificación de las propiedades de una clase. En Python, se puede utilizar el decorador `@property` para definir un getter y el decorador `@<nombre_propiedad>.setter` para definir un setter.

```

class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    @property
    def precio(self):
        return self._precio

    @precio.setter
    def precio(self, precio):
        if precio <= 0:
            raise ValueError("El precio debe ser mayor que cero.")
        self._precio = precio

    def __str__(self):
        return f"{self.nombre}: ${self.precio}"

producto = Producto("Laptop", 1000)
print(producto)          # Output: Laptop: $1000
print(producto.precio)   # Output: 1000

producto.precio = 1500   # Modifica el precio del producto
print(producto)          # Output: Laptop: $1500
print(producto.precio)   # Output: 1500

# Si intentamos modificar el precio a un valor inválido, se lanza una excepción
# producto.precio = -1000 # ValueError: El precio debe ser mayor que cero.

```

Recuerda que en Python, las propiedades de una clase se pueden acceder y modificar directamente, pero es una buena práctica utilizar métodos getter y setter para controlar el acceso y la modificación de las propiedades de una clase. Mantenemos la elegancia del código y, a la vez, podemos proteger el acceso a las propiedades de la clase manteniendo el encapsulamiento.

Estos son algunos de los elementos básicos de la programación orientada a objetos en Python. La programación orientada a objetos es un paradigma de programación poderoso que permite modelar el mundo real de una manera más natural y reutilizable.

En los próximos capítulos, veremos cómo utilizar la programación orientada a objetos para resolver problemas más complejos y cómo aprovechar las características avanzadas de Python para crear aplicaciones más eficientes y organizadas.

Repaso de las Ideas Principales

- La programación orientada a objetos es un paradigma de programación que se basa en el concepto de **objetos**, que son instancias de **clases**.
- Una **clase** es una plantilla que define las propiedades y los métodos de un objeto.
- Un **objeto** es una instancia de una clase y encapsula los datos y la lógica en una estructura coherente y reutilizable.
- Los objetos se crean utilizando la palabra clave `class` seguida del nombre de la clase y dos puntos `:`.
- Los **métodos** de una clase se definen utilizando la palabra clave `def` dentro de la clase y reciben un parámetro `self` que se refiere al objeto que se está manipulando.
- Las **propiedades** de un objeto se definen utilizando la palabra clave `self` y se pueden acceder y modificar utilizando la notación de punto `.`.
- Los **métodos estáticos y de clase** se definen utilizando los decoradores `@staticmethod` y `@classmethod`, respectivamente.
- El decorador `@property` se utiliza para definir propiedades de solo lectura en una clase. Con `@<nombre_propiedad>.setter` se define un setter para la propiedad.
- Los **métodos getter y setter** se utilizan para controlar el acceso y la modificación de las propiedades de una clase.
- La programación orientada a objetos es un paradigma poderoso que permite modelar el mundo real de una manera más natural y reutilizable.

Clases

En Python, una clase es un modelo que define las propiedades y comportamientos de un objeto. Las clases permiten crear objetos que comparten las mismas características y comportamientos.

Definición de clases

En Python, una clase se define utilizando la palabra clave `class`, seguida del nombre de la clase y dos puntos `:`. Por ejemplo:

```
class Persona:  
    pass
```

En el ejemplo anterior, se ha definido una clase llamada `Persona` que no tiene ninguna propiedad ni comportamiento definido. La palabra clave `pass` se utiliza en Python para indicar que no hay ninguna instrucción en el bloque de código.

Creación de objetos

Un objeto es una instancia de una clase. Para crear un objeto en Python, se utiliza el nombre de la clase seguido de paréntesis `()`. Por ejemplo:

```
juan = Persona()
```

En el ejemplo anterior, se ha creado un objeto de la clase `Persona` y se ha asignado a la variable `juan`.

Propiedades de los objetos

Las propiedades de un objeto son variables que almacenan información específica de cada objeto. Para definir las propiedades de un objeto en Python, se utilizan los métodos especiales `__init__` y `self`. Por ejemplo:

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

En el ejemplo anterior, se ha definido un método `__init__` que recibe dos parámetros `nombre` y `edad` y asigna estos valores a las propiedades `nombre` y `edad` del objeto utilizando la palabra clave `self`.

Métodos de los objetos

Los métodos de un objeto son funciones que definen el comportamiento de un objeto. Para definir un método en Python, se utiliza la palabra clave `def` dentro de la clase. Por ejemplo:

```
class Persona:  
    def __init__(self, nombre, edad):  
        self.nombre = nombre  
        self.edad = edad  
  
    def saludar(self):  
        print(f"Hola, me llamo {self.nombre} y tengo {self.edad} años.")
```

En el ejemplo anterior, se ha definido un método `saludar` que imprime un saludo con el nombre y la edad del objeto.

Uso de objetos

Para utilizar un objeto en Python, se llama a sus métodos y se accede a sus propiedades utilizando la notación de punto `.`. Por ejemplo:

```
juan = Persona("Juan", 30)  
juan.saludar() # Output: Hola, me llamo Juan y tengo 30 años.
```

En el ejemplo anterior, se ha creado un objeto `juan` de la clase `Persona` con el nombre `"Juan"` y la edad `30`, y se ha llamado al método `saludar` del objeto `juan`.

Las clases y los objetos son conceptos fundamentales en la programación orientada a objetos y permiten modelar el mundo real de una manera más eficiente y organizada.

Herencia

La herencia es un mecanismo que permite crear una nueva clase a partir de una clase existente. La nueva clase hereda las propiedades y métodos de la clase existente y puede añadir nuevas propiedades y métodos o modificar los existentes.

En Python, la herencia se define colocando el nombre de la clase base entre paréntesis después del nombre de la clase derivada. Por ejemplo:

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

    def estudiar(self):
        print(f"Estoy estudiando la carrera de {self.carrera}.")
```

En el ejemplo anterior, se ha definido una clase `Estudiante` que hereda de la clase `Persona` y añade una nueva propiedad `carrera` y un nuevo método `estudiar`.

Polimorfismo

El polimorfismo es un concepto que permite que un objeto pueda comportarse de diferentes maneras según el contexto en el que se utilice. En Python, el polimorfismo se logra mediante el uso de métodos con el mismo nombre en diferentes clases.

Por ejemplo, si se define un método `saludar` en la clase `Persona` y en la clase `Estudiante`, cada clase puede tener su propia implementación del método `saludar`.

```
class Persona:
    def saludar(self):
        print("Hola, soy una persona.")

class Estudiante(Persona):
    def saludar(self):
        print("Hola, soy un estudiante.")
```

En el ejemplo anterior, se ha definido un método `saludar` en las clases `Persona` y `Estudiante` con implementaciones diferentes.

El polimorfismo permite que un objeto pueda ser tratado como un objeto de su clase base o de una clase derivada, lo que facilita la reutilización del código y la creación de programas más flexibles y escalables.

Encapsulamiento

El encapsulamiento es un concepto que permite ocultar los detalles de implementación de un objeto y exponer solo la interfaz pública. En Python, el encapsulamiento se logra utilizando métodos y propiedades privadas.

Para definir una propiedad o método privado en Python, se utiliza un guion bajo `_` al principio del nombre. Luego se utiliza el decorador `@property` para definir un método getter y el decorador `@nombre.setter` para definir un método setter.

Veamos un ejemplo de estos conceptos

Supongamos que queremos modelar un producto en una tienda. Podríamos crear una clase `Producto` con las propiedades `nombre`, `precio` y `cantidad`, y un método `total` que calcule el precio total del producto.

```

class Producto:
    def __init__(self, nombre, precio, cantidad):
        self._nombre = nombre
        self._precio = precio
        self._cantidad = cantidad

    @property
    def nombre(self):
        return self._nombre

    @property
    def precio(self):
        return self._precio

    @property
    def cantidad(self):
        return self._cantidad

    @property
    def total(self):
        return self._precio * self._cantidad

    def __str__(self):
        return f"{self._nombre} - ${self._precio:.2f} * {self._cantidad}u = ${self._precio * self._cantidad:.2f}"

producto = Producto("Leche", 2.50, 3)
print(producto) # Output: Leche - $2.50 * 3u = $7.50

print(producto.nombre) # Output: Leche
print(producto.precio) # Output: 2.50
print(producto.cantidad) # Output: 3
print(producto.total) # Output: 7.50

```

Este caso creamos lo que se conoce como un objeto inmutable, ya que no se pueden modificar sus propiedades una vez creado. El encapsulamiento permite proteger los datos de un objeto y garantizar que solo se puedan acceder y modificar a través de métodos específicos.

Si quisieramos modificar el precio y la cantidad de un producto pero que la misma no permita valores invalidos, podríamos hacerlo de la siguiente forma:

```

class Producto:
    def __init__(self, nombre, precio, cantidad):
        self._nombre = nombre
        self._precio = precio
        self._cantidad = cantidad

    @property
    def nombre(self):
        return self._nombre

    @property
    def precio(self):
        return self._precio

    @precio.setter # Utilizamos el decorador setter para modificar el precio en forma segura
    def precio(self, nuevo_precio):
        if nuevo_precio < 0: # Validamos que el precio no sea negativo
            raise ValueError("El precio no puede ser negativo.")

        self._precio = nuevo_precio

    @property
    def cantidad(self):
        return self._cantidad

    @cantidad.setter # Utilizamos el decorador setter para modificar la cantidad en forma segura
    def cantidad(self, nueva_cantidad):
        if nueva_cantidad < 0: # Validamos que la cantidad no sea negativa
            raise ValueError("La cantidad no puede ser negativa.")

        self._cantidad = nueva_cantidad

    @property
    def total(self):
        return self._precio * self._cantidad

    def __str__(self):
        return f"{self._nombre} - ${self._precio:.2f} * {self._cantidad}u = ${self._precio * self._cantidad:.2f}"

```

En este caso, hemos agregado los decoradores `@precio.setter` y `@cantidad.setter` para modificar el precio y la cantidad de un producto de forma segura. Si se intenta asignar un valor negativo al precio o a la cantidad, se lanzará una excepción `ValueError` indicando que el valor no es válido.

Ahora supongamos que queremos modelar un producto que tenga un descuento. Podríamos crear una clase `ProductoDescuento` que herede de la clase `Producto` y añada una propiedad `descuento` y un método `total_descuento` que calcule el precio total con descuento.

```

class ProductoDescuento(Producto):                      # Heredamos de la clase Producto
    def __init__(self, nombre, precio, cantidad, descuento):
        super().__init__(nombre, precio, cantidad) # Llamamos al constructor de la clase base
        self._descuento = descuento

    @property
    def descuento(self):
        return self._descuento

    @descuento.setter # Utilizamos el decorador setter para modificar el descuento
    def descuento(self, nuevo_descuento):
        if nuevo_descuento < 0 or nuevo_descuento > 100:
            raise ValueError("El descuento debe estar entre 0 y 100.")

        self._descuento = nuevo_descuento

    @property      # Calculamos el total con descuento
    def total_descuento(self):
        return super().total * (1 - self._descuento / 100) # super() nos permite acceder a los métodos de la clase base

    @property
    def total(self):      # Sobreescribimos el método total para incluir el descuento
        return super().total - self.total_descuento # super() nos permite acceder a los métodos de la clase base

    def __str__(self):
        return f"{self.nombre:10} - ${self.precio:.2f} * {self.cantidad}u = ${super().total - self.total_descuento:.2f}"

producto_descuento = ProductoDescuento("Leche", 2.50, 3, 10)
print(producto_descuento)      # Output: Leche      - $2.50 * 3u = $7.50 - 10% = $6.75
print(producto_descuento.total) # Output: 6.75
print(producto_descuento.total_descuento) # Output: 0.75

```

En este caso, hemos creado una clase `ProductoDescuento` que hereda de la clase `Producto` y añade una nueva propiedad `descuento` y un nuevo método `total_descuento` que calcula el precio total con descuento. Además, hemos sobrescrito el método `total` para incluir el descuento en el cálculo del precio total.

El encapsulamiento, la herencia, el polimorfismo y la composición son conceptos fundamentales de la programación orientada a objetos que permiten crear programas más flexibles, escalables y fáciles de mantener. Estos conceptos nos permiten modelar el mundo real de una manera más eficiente y organizada, y nos ayudan a reutilizar código y evitar la repetición de código.

El encapsulamiento nos permite proteger los datos de un objeto y garantizar que solo se puedan acceder y modificar a través de métodos específicos.

La herencia nos permite crear una nueva clase a partir de una clase existente y heredar las propiedades y métodos de la clase base.

El polimorfismo nos permite que un objeto pueda comportarse de diferentes maneras según el contexto en el que se utilice.

Ahora supongamos que queremos modelar una tienda que tiene una colección de productos. Podríamos crear una clase **Tienda** que tenga una lista de productos y métodos para agregar, eliminar y mostrar los productos de la tienda.

```
class Tienda:
    def __init__(self):
        self._productos = []

    def agregar(self, producto):
        self._productos.append(producto)

    def eliminar(self, producto):
        self._productos.remove(producto)

    def mostrar(self):
        print("Productos en la tienda:")
        for producto in self._productos:
            print(" ", producto)

tienda = Tienda()
tienda.agregar(Producto("Leche", 2.50, 3))
tienda.agregar(ProductoDescuento("Pan", 1.50, 5, 20))
tienda.mostrar()
# Output:
# Productos en la tienda:
#   Leche      - $2.50 * 3u = $7.50
#   Pan        - $1.50 * 5u = $7.50 - 20% = $6.00
```

En este caso, hemos creado una clase **Tienda** que tiene una lista de productos y métodos para agregar, eliminar y mostrar los productos de la tienda. Hemos agregado un producto de tipo **Producto** y un producto de tipo **ProductoDescuento** a la tienda y hemos mostrado los productos en la tienda.

La composición nos permite combinar objetos de diferentes clases para crear objetos más complejos y completos. En este caso, hemos combinado objetos de las clases **Producto** y **ProductoDescuento** en la clase **Tienda** para modelar una tienda que tiene una colección de productos.

Decimos que la tienda esta compuesta por productos porque usa objetos de la clase Producto y ProductoDescuento para modelar su comportamiento. La composición nos permite crear objetos más complejos y completos combinando objetos más simples y reutilizables.

Una observación importante es que gracias a la herencia y al polimorfismo, podemos tratar a los objetos de las clases `Producto` y `ProductoDescuento` de la misma manera en la clase `Tienda`, ya que ambas clases comparten la misma interfaz pública. Esto nos permite reutilizar el código y crear programas más flexibles y escalables.

Esto nos permitia agregar productos de diferentes tipos a la tienda y mostrarlos de la misma manera, independientemente de si son productos normales o productos con descuento. Esto nos permite reutilizar el código y crear programas más flexibles y escalables.

Agregemos una nueva oferta, esta fue creada despues que se creo la tienda pero podemos usarla en la tienda si problema. Por ejemplo agregemos 2x1 como oferta.

```
class Producto2x1(ProductoOferta):
    def __init__(self, nombre, precio, cantidad):
        super().__init__(nombre, precio, cantidad, 100) # 100% de descuento en la compra
        self._cantidad_minima = 2 # Cantidad minima para aplicar la oferta

    @property
    def descuento(self):
        if self._cantidad < self._cantidad_minima:
            return 0

        return self._descuento * (self._cantidad // self._cantidad_minima)

    @descuento.setter
    def descuento(self, nuevo_descuento):
        raise ValueError("No se puede modificar el descuento de un producto 2x1.")
```

Ahora podemos usar la tienda para agregar productos de tipo `Producto2x1` y mostrarlos de la misma manera que los otros productos.

```
tienda.agregar(Producto2x1("Galletas", 1.00, 3))
tienda.mostrar()

# Output:
# Productos en la tienda:
# Leche      - $2.50 * 3u = $7.50
# Pan        - $1.50 * 5u = $7.50 - 20% = $6.00
# Galletas   - $1.00 * 3u = $3.00 - 1.00 = $2.00
```

En este caso, hemos creado una clase `Producto2x1` que hereda de la clase `ProductoOferta` y añade una nueva propiedad `cantidad_minima` que indica la cantidad mínima de unidades para aplicar la oferta 2x1. Hemos sobreescrito el método `descuento` para calcular el descuento de la oferta 2x1 y el método `total` para incluir el descuento en el cálculo del precio total.

Integrar las clases a Python

Todo en Python son objetos, los tipos de datos básicos son objetos, las funciones son objetos, los módulos son objetos y las clases son objetos.

Un aspecto muy especial de Python es que nos permite crear tipos de datos personalizados y que los mismos se comporten exactamente igual que los tipos de datos básicos. Para lograrlos se utilizan métodos especiales que permiten definir el comportamiento de los objetos de una clase. Estos métodos nos ayudan a usar todo el potencial de las clases en Python.

Existen métodos especiales para convertir un objeto, realizar operaciones matemáticas o de comparación, o crear colecciones entre otros.

Todos los métodos especiales se identifican por tener dos guiones bajos al principio y al final del nombre del método. Por ejemplo, el método `__init__` es un método especial que se utiliza para inicializar un objeto cuando se crea una instancia de una clase. Estos métodos son usados internamente por Python y no se llaman directamente en el código aunque se pueden llamar si es necesario. Una excepción a esto es el método `__new__`, el cual si bien se llama internamente por Python, también se puede llamar directamente cuando usamos herencia.

Podemos considerar que Python en realidad define una marco de trabajo para la programación orientada a objetos, esto es que además de las clases instrumenta un conjunto de métodos para que las clases personalizadas se comporten como los tipos de datos básicos.

Metodos especiales

Metodo `__init__`

El metodo `__init__` es un metodo especial que se utiliza para inicializar un objeto cuando se crea una instancia de una clase. Este metodo se llama automaticamente cuando se crea un objeto de una clase y se utiliza para inicializar los atributos de la clase.

Por ejemplo, supongamos que queremos crear una clase `Persona` que tenga los atributos `nombre`, `edad` y `altura`. Podemos definir la clase `Persona` de la siguiente manera:

```
class Persona:  
    def __init__(self, nombre, edad, altura):  
        self.nombre = nombre  
        self.edad = edad  
        self.altura = altura  
  
p = Persona("Juan", 30, 1.75)
```

En este caso el metodo `__init__` recibe cuatro parametros: `self`, `nombre`, `edad` y `altura`. El parametro `self` hace referencia al objeto actual y se utiliza para acceder a los atributos y metodos de la clase. Los parametros `nombre`, `edad` y `altura` se utilizan para inicializar los atributos `nombre`, `edad` y `altura` del objeto.

Metodo `__str__`

El metodo `__str__` es un metodo especial que se utiliza para devolver una representación en forma de cadena de un objeto. Este metodo se llama automaticamente cuando se utiliza la funcion `str()` o `print()` con un objeto de una clase.

Por ejemplo, supongamos que queremos crear una clase `Persona` que tenga los atributos `nombre`, `edad` y `altura`. Podemos definir la clase `Persona` de la siguiente manera:

```

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

    def __str__(self):
        return f"Nombre: {self.nombre}, Edad: {self.edad}, Altura: {self.altura}"

p = Persona("Juan", 30, 1.75)

print(p) # Output: Nombre: Juan, Edad: 30, Altura: 1.75

```

En este caso, el metodo `__str__` devuelve una cadena con los atributos `nombre`, `edad` y `altura` del objeto. Cuando se utiliza la funcion `print()` con el objeto `p`, se llama automaticamente al metodo `__str__` y se imprime la cadena devuelta por el metodo.

Los mismos sucede cuando se utiliza la funcion `str()` con el objeto `p`. Por ejemplo:

```

s = str(p)
print(s) # Output: Nombre: Juan, Edad: 30, Altura: 1.75

```

Internamente la funcion `str()` llama al metodo `__str__` del objeto `p` y devuelve la cadena devuelta por el metodo. Esto permite utilizar el 'polimorfismo' en Python, es decir, que un objeto pueda comportarse de diferentes maneras dependiendo del contexto en el que se utilice.

La funcion `str()` no sabe como convertir un objeto en un representación de cadena, pero si el objeto tiene un metodo `__str__` definido, entonces la funcion `str()` llama a ese metodo para obtener la representación en forma de cadena del objeto.

Es decir, internamente Python delega la responsabilidad de convertir un objeto en una cadena al metodo `__str__` del objeto si este metodo esta definido.

La funcion `print` intermanete llama a la funcion `str()` para obtener la representación en forma de cadena del objeto y luego imprime esa cadena en la consola. La funcion `str()` a su vez llama al metodo `__str__` del objeto si este metodo esta definido.

De esta manera las funciones creadas antes de la definición de la clase `Persona` no necesitan ser modificadas para imprimir un objeto de la clase `Persona`.

Si vamos mas al detalle en realidad str() es el constructor de la clase str. Esto lo usamos como si fuera una funcion (y lo es porque en Python las clases son llamables como funciones) pero en realidad es un constructor de la clase str.

```
a = str(5)
print(a) # Output: '5'

b = str(2>1)
print(b) # Output: True

c = str([1, 2, 3]) # Output: '[1, 2, 3]'

print(5, 2>1, [1, 2, 3])      # Implicitamente se llama a str() para convertir los argu
# Output: 5 True [1, 2, 3]

print(str(5), str(2>1), str([1, 2, 3])) # Llamamos explicitamente a str() para conv
# Output: 5 True [1, 2, 3]
```

Lo interesante es que Python es un lenguaje fuertemente tipado, pero a la vez es un lenguaje dinámico. Es fuertemente tipado porque siempre respeta las operaciones definidas para los tipos de datos, pero es dinámico porque permite que los objetos se comporten de diferentes maneras dependiendo del contexto en el que se utilicen.

Esto lo logra a través de los métodos especiales que permiten definir el comportamiento de los objetos de una clase.

Metodo __repr__

El metodo __repr__ es un metodo especial que se utiliza para devolver una representación en forma de cadena de un objeto. Este metodo se llama automaticamente cuando se utiliza la función repr() con un objeto de una clase. Es similar al metodo __str__ pero se utiliza para obtener una representación más detallada del objeto. En cambio, el metodo __str__ se utiliza para obtener una representación más amigable del objeto.

Por ejemplo, supongamos que queremos crear una clase Persona que tenga los atributos nombre, edad y altura. Podemos definir la clase Persona de la siguiente manera:

```

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

    def __str__(self):
        return f"Nombre: {self.nombre}, Edad: {self.edad}, Altura: {self.altura}"

    def __repr__(self):
        return f"Persona({self.nombre!r}, {self.edad!r}, {self.altura!r})"

a = Persona("Juan", 30, 1.75)
print(a)          # Output: Nombre: Juan, Edad: 30, Altura: 1.75

print(repr(a))  # Output: Persona('Juan', 30, 1.75)

# f-string con !s y !r para llamar a str() y repr() respectivamente

# por defecto usa !s si no se especifica
print(f'{a}')   # Output: Nombre: Juan, Edad: 30, Altura: 1.75

# Equivalente a...
print(f'{a!s}') # Output: Nombre: Juan, Edad: 30, Altura: 1.75

# Pero si especificamos !r usa repr()
print(f'{a!r}') # Output: Persona('Juan', 30, 1.75)

```

En resumen se usa `str()` para obtener una representación amigable del objeto y `repr()` para obtener una representación más detallada del objeto.

Metodo `_bool_`

Otro ejemplo de este comportamiento es el método `_bool_` que se utiliza para devolver un valor booleano de un objeto. Este método se llama automáticamente cuando se utiliza el operador `bool()` con un objeto de una clase.

Al comienzo vimos que en Python se considera `False` a `False`, a `0`, a `""`, a `[]`, a `{}`, a `set()` y a `None`; todo lo demás se considera `True`.

Esto puede parecer que existen valores “mágicos”, valores que tienen un valor especial en Python. Pero en realidad no es así, en Python no existen valores mágicos, sino que existen métodos especiales que permiten definir el comportamiento de los objetos de una clase.

Cuando Python quiere saber si un `int` se puede convertir a `bool` usa la función `__bool__` de la clase `int`, cuando hacemos `bool('hola')` Python llama a la función `__bool__` de la clase `str` y cuando hacemos `bool([1, 2, 3])` Python llama a la función `__bool__` de la clase `list`.

Este proceso de convercion lo hace automaticamente cuando se usa una variable en el contexto de una condición por ejemplo en un `if`, en un `while` o incluso en `and` o `or`.

```
if 10:                      # Convierte (implicitamente) 10 a bool
    print('10 es True')
else:
    print('10 es False')

# Output: 10 es True

#Equivale a
if bool(10):                  # Convierte (explicitamente) 10 a bool
    print('10 es True')
else:
    print('10 es False')

# Output: 10 es True

# que a la vez equivale a
if int.__bool__(10):          # Convierte (explicitamente usando __bool__) 10 a bool
    print('10 es True')
else:
    print('10 es False')

# Output: 10 es True
```

Volvamos a nuestra clase `Producto` y veamos cómo se comporta en un contexto de condición.

```

class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def __str__(self):
        return f"{self.nombre} - ${self.precio}"

    def __bool__(self):
        return self.precio > 0

p = Producto("Laptop", 1000)
if p:
    print('Lo podemos vender')
else:
    print('No lo podemos vender')

# Output: Lo podemos vender

p = Producto("Mouse", 0)
if p:
    print('Lo podemos vender')
else:
    print('No lo podemos vender')

# Output: No lo podemos vender

```

En este caso, el método `__bool__` devuelve `True` si el precio del producto es mayor que cero y `False` en caso contrario. Cuando se utiliza el objeto `p` en un contexto de condición, se llama automáticamente al método `__bool__` y se evalúa si el producto se puede vender o no.

En la tienda, podríamos tener una lista de productos y querer saber si podemos vender todos los productos de la lista. Podemos hacer esto de la siguiente manera:

```

class Tienda:
    ...
    def mostrar(self):
        for p in self.productos:
            if p:
                print(p)
    ...

```

En este caso, el método `mostrar` recorre la lista de productos de la tienda y utiliza cada producto en un contexto de condición para determinar si se puede vender o no. Si el producto se puede

vender, se imprime el producto en la consola.

De esta manera, podemos utilizar el polimorfismo en Python para que los objetos de una clase se comporten de diferentes maneras dependiendo del contexto en el que se utilicen.

Metodo len

El método len es un método especial que se utiliza para devolver la longitud de un objeto. Este método se llama automáticamente cuando se utiliza la función len() con un objeto de una clase.

A diferencia de otros lenguajes, para averiguar la longitud de una lista en Python se usa la función len() en lugar de 'a.length' o 'a.size()' como en otros lenguajes.

Esto es así porque la función len() llama al método len del objeto a para obtener la longitud del objeto. Si el método len no está definido para el objeto a, se produce un error.

Sigamos con el caso de la clase Tienda, si quisiéramos saber cuántos productos hay en la tienda podríamos hacer lo siguiente:

```
class Tienda:  
    #...  
    def __len__(self):  
        return len(self.productos)  
    #...  
  
t = Tienda()  
t.agregar(Producto("Laptop", 1000))  
t.agregar(Producto("Mouse", 0))  
  
print(f'Hay {len(t)} productos en la tienda')  
# Output: Hay 2 productos en la tienda
```

En este caso, el método len devuelve la longitud de la lista de productos de la tienda.

Cuando se utiliza la función len() con el objeto t, se llama automáticamente al método len y se obtiene la cantidad de productos en la tienda. Se podría haber hecho lo mismo con la función len(t.productos) pero de esta manera se encapsula la lógica de la longitud de la lista de productos en la clase Tienda.

Si vamos más al detalle, podemos ver que Python usa la función `len()` para determinar si un objeto es vacío o no. Por ejemplo, si hacemos `if len(t):` Python llama al método `__len__` de la clase `Tienda` para determinar si la tienda tiene productos o no.

Si observamos bien, veremos que cuando convertíamos a `bool()`, las cadenas, las listas, los diccionarios, las tuplas y los conjuntos se consideraban `False` si estaban vacíos y `True` si tenían elementos. Esto es porque Python llama al método `__len__` de la clase correspondiente para determinar si el objeto está vacío o no cuando no encuentra un método `__bool__` definido.

Por ejemplo en la clase `Tienda` podríamos usarla en un contexto de condición para saber si la tienda tiene productos o no.

```
t = Tienda()
t.agregar(Producto("Laptop", 0))
t.agregar(Producto("Mouse", 0))

if t:
    print(f'La tienda tiene productos {len(t)} productos disponibles')
else:
    print('La tienda no tiene productos para vender')
```

Metodo `__getitem__`

El método `__getitem__` es un método especial que se utiliza para obtener un elemento de un objeto mediante su índice. Este método se llama automáticamente cuando se utiliza la indexación con corchetes `[]` en un objeto de una clase.

Por ejemplo, si queremos listar los `Productos` de la `Tienda` podríamos aprovechar que sabemos que se implementó con una lista de productos y recorrerlo así:

```

t = Tienda()
t.agregar(Producto("Laptop", 1000))
t.agregar(Producto("Mouse", 0))

for i in range(len(t.productos)):
    p = t.productos[i]
    print(p)

# o incluso
for p in t.productos:
    print(p)

```

Pero este enfoque rompe el encapsulamiento de la clase `Tienda` ya que estamos accediendo directamente a la lista de productos. Podemos mejorar esto implementando el método `__getitem__` en la clase `Tienda` de la siguiente manera:

```

class Tienda :
    ...
    def __len__(self):
        return len(self.productos)

    def __getitem__(self, i):
        return self.productos[i]
    ...

# Ahora podemos listar los productos de la tienda de la siguiente manera:
t = Tienda()
t.agregar(Producto("Laptop", 1000))
t.agregar(Producto("Mouse", 0))

for i in range(len(t)):
    p = t[i]
    print(p)

# o incluso
for p in t:
    print(p)

```

Esto es así porque Python llama al método `__getitem__` de la clase `Tienda` para obtener el elemento de la lista de productos en la posición `i`. De esta manera, podemos utilizar la indexación con corchetes `[]` en un objeto de la clase `Tienda` para obtener un producto de la tienda.

Es decir podemos usar `Tienda` como si fuera una lista de productos y tratarla como tal. Esto es un ejemplo de polimorfismo en Python, es decir, que un objeto pueda comportarse de diferentes

maneras dependiendo del contexto en el que se utilice.

Esta es una de las razones por las que Python es un lenguaje tan poderoso y flexible, ya que nos permite definir el comportamiento de los objetos de una clase de acuerdo a nuestras necesidades.

Esto tiene implicancias muy interesantes, por ejemplo:

```
t = Tienda()
t.agregar(Producto("Laptop", 1000))
t.agregar(Producto("Mouse", 0))
t.agregar(Producto("Teclado", 500))

print(t[0])      # Output: Laptop - $1000
print(t[1])      # Output: Mouse - $0
print(t[-1])     # Output: Teclado - $500

parte = t[1:]    # usamos slicing para obtener una parte de la lista de productos
print(parte)     # Output: [Mouse - $0, Teclado - $500]

parte = t[:2]    #
print(parte)     # Output: [Laptop - $1000, Mouse - $0]

sorted_tienda = sorted(t)  # Usamos la función sorted() para ordenar los productos
                           # por precios
```

Metodo __setitem__

El método __setitem__ es un método especial que se utiliza para asignar un valor a un elemento de un objeto mediante su índice. Este método se llama automáticamente cuando se utiliza la indexación con corchetes `[]` en un objeto de una clase en el lado izquierdo de una asignación.

Por ejemplo, si queremos modificar un `Producto` de la `Tienda` podríamos aprovechar que sabemos que se implementó con una lista de productos y modificarlo así:

```

t = Tienda()
t.agregar(Producto("Laptop", 1000))

t.productos[0] = Producto("Mouse", 0)

for p in t.productos:
    print(p)

```

Pero este enfoque rompe el encapsulamiento de la clase `Tienda` ya que estamos accediendo directamente a la lista de productos. Podemos mejorar esto implementando el método `__setitem__` en la clase `Tienda` de la siguiente manera:

```

class Tienda :
    ...
    def __setitem__(self, i, p):
        self.productos[i] = p
    ...

# Ahora podemos modificar un producto de la tienda de la siguiente manera:
t = Tienda()
t.agregar(Producto("Laptop", 1000))

t[0] = Producto("Mouse", 0)

for p in t:
    print(p)

```

Esto es así porque Python llama al método `__setitem__` de la clase `Tienda` para asignar un producto a la lista de productos en la posición `i`. De esta manera, podemos utilizar la indexación con corchetes `[]` en un objeto de la clase `Tienda` para modificar un producto de la tienda.

Una observación interesante, la clase `str` en realidad es una lista de caracteres, por lo que podemos usar indexación y slicing en una cadena de texto. En dicha clase se implementó `__len__` y `__getitem__` pero no `__setitem__`. Esto es porque las cadenas de texto son inmutables, es decir, no se pueden modificar una vez creadas.

Otros métodos de colección:

- `__contains__`: Se utiliza para determinar si un objeto contiene un elemento. Este método se llama automáticamente cuando se utiliza el operador `in` con un objeto de una clase.

- `__delitem__`: Se utiliza para eliminar un elemento de un objeto mediante su índice. Este método se llama automáticamente cuando se utiliza la instrucción `del` con un objeto de una clase.
- `__iter__`: Se utiliza para devolver un iterador de un objeto. Este método se llama automáticamente cuando se utiliza la función `iter()` con un objeto de una clase.

Metodo `__add__`, `__sub__`, `__mul__`, `__truediv__`, etc.

Estos son metodos que se usan para realizar operaciones matematicas con objetos de una clase. Cuando se usa un operador matematico con un objeto de una clase, Python llama al metodo correspondiente para realizar la operación.

Para `+` se llama a `__add__`, para `-` se llama a `__sub__`, para `*` se llama a `__mul__`, para `/` se llama a `__truediv__` etc.

Es por esta razon que podemos sumar dos cadenas de texto, dos listas o dos enteros. Python llama al metodo `__add__` de la clase `str`, `list` o `int` respectivamente para realizar la suma.

Esta es una caracetistica que hace muy expresivo al lenguaje. Por ejemplo, estas son las implementaciones de los metodos `__add__` y `__mul__` en la clase `str`:

```

class str:
    #...
    def __add__(self, other):
        if not isinstance(other, str):
            return NotImplemented
        return self.join([self, other])

    def __mul__(self, other):
        if not isinstance(other, int):
            return NotImplemented
        return self.join([self]*other)
    #...

a = ('Hola ' + 'Mundo! ') * 3
print(a) # Output: Hola Mundo! Hola Mundo! Hola Mundo!

# Incluso podemos hacer algo como esto
b = ('Hola, ' + 'muy ' * 3 + 'buenos dias!' + '\n') * 3

# usa los parentesis para separar las operaciones y hacerlo mas legible, ejecuta pr
print(b)
# Output:
# Hola, muy muy muy buenos dias!
# Hola, muy muy muy buenos dias!
# Hola, muy muy muy buenos dias!

```

En el caso de str el `+` concatena las cadenas y el `*` repite la cadena.

Podriamos hacer un ejemplo similar con la clase `Producto` para sumar cantidades y incrementar los precios.

```

class Producto:
    def __init__(self, nombre, precio, cantidad):
        self.nombre = nombre
        self.precio = precio
        self.cantidad = cantidad

    def __add__(self, other):
        if not isinstance(other, int):
            return NotImplemented
        return Producto(self.nombre, self.precio, self.cantidad + other.cantidad)

    def __mul__(self, aumento):
        if not isinstance(other, int, float):
            return NotImplemented
        return Producto(self.nombre, self.precio * (1 + aumento / 100), self.cantidad)

    def __str__(self):
        return f"{self.nombre} - ${self.precio} - {self.cantidad} unidades"

    def __repr__(self):
        return f"Producto('{self.nombre}', {self.precio}, {self.cantidad})"

    def __bool__(self):
        return self.precio > 0 and self.cantidad > 0

p = Producto("Laptop", 1000, 10)
print(repr(p)) # Output: Producto('Laptop', 1000, 10)

p = p + 10      # Agrega 10 unidades
print(repr(p)) # Output: Producto('Laptop', 1000, 20)

p += 10         # Incrementa el precio un 10%
print(repr(p)) # Output: Producto('Laptop', 1100, 30)

p *= 10         # Incrementa el precio un 10%
print(repr(p)) # Output: Producto('Laptop', 1210, 30)

```

Observe que en este caso cuando se suma o multiplica un `Producto` con un entero, se devuelve un nuevo `Producto` con la cantidad o el precio modificado. De esta manera, podemos utilizar los operadores matemáticos con objetos de la clase `Producto` de manera similar a como lo hacemos con los tipos de datos básicos.

Esto es lo que se llama una clase inmutable, es decir, una clase cuyos objetos no se pueden modificar una vez creados. En este caso, los objetos de la clase `Producto` son inmutables porque

no se pueden modificar una vez creados. De esta manera están implementadas las clase int, float, str, tuple, etc.

Metodo `__eq__`, `__lt__`, `__gt__`, `__le__`, `__ge__`, `__ne__`

Estos son metodos que se usan para realizar comparaciones con objetos de una clase. Cuando se usa un operador de comparación con un objeto de una clase, Python llama al metodo correspondiente para realizar la comparación.

- `__eq__` se llama cuando se usa el operador `==` para comparar dos objetos.
- `__lt__` se llama cuando se usa el operador `<` para comparar dos objetos.
- `__gt__` se llama cuando se usa el operador `>` para comparar dos objetos.
- `__le__` se llama cuando se usa el operador `<=` para comparar dos objetos.
- `__ge__` se llama cuando se usa el operador `>=` para comparar dos objetos.
- `__ne__` se llama cuando se usa el operador `!=` para comparar dos objetos.

Por ejemplo, si queremos comparar dos `Productos` por su precio podríamos hacer lo siguiente:

```
class Producto:  
    #...  
    def __eq__(self, other):  
        return self.precio == other.precio  
  
    def __lt__(self, other):  
        return self.precio < other.precio  
    #...  
  
p1 = Producto("Laptop", 1000)  
p2 = Producto("Mouse", 500)  
  
if p1 == p2:  
    print('Los productos tienen el mismo precio')  
elif p1 < p2:  
    print('El producto 1 es más barato que el producto 2')  
else:  
    print('El producto 1 es más caro que el producto 2')
```

En este caso, el método `__eq__` devuelve `True` si el precio del producto 1 es igual al precio del producto 2 y `False` en caso contrario. El método `__lt__` devuelve `True` si el precio del producto 1 es menor que el precio del producto 2 y `False` en caso contrario.

De esta manera, podemos utilizar los operadores de comparación con objetos de la clase **Producto** de manera similar a como lo hacemos con los tipos de datos básicos.

Metodo `__call__`

El método `__call__` es un método especial que se utiliza para llamar a un objeto como si fuera una función. Este método se llama automáticamente cuando se utiliza el objeto con paréntesis `()`.

Por ejemplo, si queremos calcular el precio total de un producto podríamos hacer lo siguiente:

```
class Producto:  
    #...  
    def __call__(self, cantidad):  
        return self.precio * cantidad  
    #...  
  
p = Producto("Laptop", 1000)  
total = p(10)  
print(f'El precio total es ${total}')
```

En este caso, el método `__call__` recibe un parámetro `cantidad` y devuelve el precio total del producto. Cuando se utiliza el objeto `p` con paréntesis `()`, se llama automáticamente al método `__call__` y se calcula el precio total del producto.

De esta manera, podemos utilizar los objetos de la clase **Producto** como si fueran funciones para calcular el precio total del producto.

Metodo `__iter__` y `__next__`

Vimos que con el método `__len__` y `__getitem__` podemos hacer que un objeto de una clase se comporte como una lista. Pero no siempre se puede hacer esto, por ejemplo, si queremos recorrer los números pares de 0 a 10 no podemos hacerlo con una lista.

En este caso, podemos implementar el método `__iter__` y `__next__` para crear un iterador que nos permita recorrer los números pares de 0 a 10.

Supongamos que queremos hacer un contador (similar a `range(1, n)`) es puede implementar de la siguiente manera:

```

class Contador:
    def __init__(self, n):
        self.n = n
        self.i = 0

    def __next__(self):
        if self.i >= self.n:
            raise StopIteration # Se lanza una excepción para indicar que se ha alcanzado el final de la secuencia
        self.i += 1          # Incrementa el contador
        return self.i

    def __iter__(self): # Devuelve el objeto iterador para el bucle for
        return self

c = Contador(10)
print(next(c)) # Output: 1
print(next(c)) # Output: 2
print(next(c)) # Output: 3
while True:
    try:
        print(next(c))
    except StopIteration:
        break
# Output: 4 5 6 7 8 9 10
for i in Contador(10):
    print(i, end=' ')
# Output: 1 2 3 4 5 6 7 8 9 10

```

En este caso, el método `__next__` devuelve el siguiente número par en la secuencia de 0 a 10. Cuando se utiliza el objeto `c` en un contexto de iteración, se llama automáticamente al método `__iter__` y `__next__` para recorrer los números pares de 0 a 10.

De esta manera, podemos utilizar los objetos de la clase `Contador` como si fueran iteradores para recorrer los números pares de 0 a 10.

Iterador

En muchas circunstancias, necesitamos recorrer los elementos de una colección, como una lista, tupla, conjunto o diccionario, para realizar alguna operación con cada uno de ellos. Para esto, Python proporciona una estructura llamada **iterador** que nos permite recorrer los elementos de una colección de forma secuencial.

Un **iterador** es un objeto que implementa el protocolo de iteración, lo que significa que puede ser utilizado en un bucle `for` para recorrer los elementos de una colección. Los iteradores en Python se utilizan para recorrer secuencialmente los elementos de una colección sin tener que conocer la estructura interna de la misma.

Creación de un iterador

En Python, un iterador se crea utilizando la función `iter()` sobre una colección, como una lista, tupla, conjunto o diccionario. Por ejemplo:

```
nombres = ["Juan", "María", "Carlos"]
iterador = iter(nombres)
```

En el ejemplo anterior, se ha creado un iterador a partir de una lista de nombres utilizando la función `iter()`. El iterador `iterador` se puede utilizar para recorrer secuencialmente los elementos de la lista `nombres`.

Recorrido de un iterador

Para recorrer los elementos de un iterador en Python, se utiliza un bucle `for`. Por ejemplo:

```
nombres = ["Juan", "María", "Carlos"]
iterador = iter(nombres)

for nombre in iterador:
    print(nombre)
```

En el ejemplo anterior, se ha recorrido el iterador `iterador` utilizando un bucle `for` e imprimiendo cada uno de los nombres de la lista `nombres`.

Función `next()`

En Python, la función `next()` se utiliza para obtener el siguiente elemento de un iterador. Por ejemplo:

```
nombres = ["Juan", "María", "Carlos"]
iterador = iter(nombres)

print(next(iterador)) # Output: Juan
print(next(iterador)) # Output: María
print(next(iterador)) # Output: Carlos
# print(next(iterador)) # Esto causaría un error: StopIteration
```

En el ejemplo anterior, se ha utilizado la función `next()` para obtener secuencialmente cada uno de los nombres de la lista `nombres` a través del iterador `iterador`. Intentar llamar a `next(iterador)` una cuarta vez causaría una excepción `StopIteration`, ya que no hay más elementos en la lista.

Iteradores en bucles `for`

En Python, los bucles `for` utilizan iteradores para recorrer los elementos de una colección. Por ejemplo:

```
nombres = ["Juan", "María", "Carlos"]

for nombre in nombres:
    print(nombre)
```

Un iterador muy usado es `range()`, que genera una secuencia de números enteros. Por ejemplo:

```
for i in range(5):
    print(i)
```

La sentencia `for` busca si el objeto pasado tiene el método `__iter__`. Si no lo tiene, intenta acceder a los elementos mediante el método `__getitem__`, comenzando por el índice `0` hasta que se produzca una excepción `IndexError`. No utiliza el método `__len__`.

La ventaja de los iteradores es que permiten recorrer colecciones de elementos de manera eficiente, sin tener que cargar todos los elementos en memoria al mismo tiempo. Esto es especialmente útil cuando se trabaja con colecciones grandes o infinitas.

Python utiliza intensamente los iteradores. Por ejemplo:

```

l = list("abracadabra")
print(l)
# Output: ['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']
# list usa un iterador para recorrer la cadena

s = set("abracadabra")
print(s)
# Output: {'a', 'b', 'r', 'c', 'd'}
# set usa un iterador para recorrer la cadena

for i, vocal in enumerate("aeiou"):
    print(i, vocal)
# Output:
# 0 a
# 1 e
# 2 i
# 3 o
# 4 u

# enumerate usa un iterador para recorrer la cadena (agrega un índice a cada elemento)

d = dict(enumerate("aeiou"))
print(d)
# Output: {0: 'a', 1: 'e', 2: 'i', 3: 'o', 4: 'u'}

# dict usa un iterador para recorrer la cadena (agrega un índice a cada elemento)

l = list(set("abracadabra"))
print(l)
# Output: ['a', 'b', 'r', 'c', 'd']
# set usa un iterador para recorrer la cadena, y list usa un iterador para recorrer

c = sum((1, 2))
print(c)
# Output: 3
# sum usa un iterador para recorrer los elementos de la tupla (1, 2)

c = sum([1, 2, 4, 7])
print(c)
# Output: 14

# sum usa un iterador para recorrer los elementos de la lista [1, 2]

c = sum({1, 2, 3, 1, 2, 3, 1, 3})
print(c)
# Output: 6

# sum usa un iterador para recorrer los elementos del conjunto {1, 2, 3} (eliminados)

c = sum(range(1, 100, 2))
print(c)
# Output: 2500
# sum usa un iterador para recorrer los elementos del rango (1, 100, 2) (1, 3, 5, . .

```

```

m = max("aeiou")
print(m)
# Output: u

m = max([1, 2, 5, 4, 3])
print(m)
# Output: 5

cuadrados = map(lambda x: x**2, range(10))
print(list(cuadrados))
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# map usa un iterador para recorrer los elementos del rango (0, 10) y aplicar la función

def par(x):
    return x % 2 == 0

def impar(x):
    return x % 2 != 0

def doble(x):
    return x * 2

pares = all(map(par, range(0, 20)))
# Output: False (¿son todos pares?)

l = list(map(doble, filter(par, range(0, 20)))))

# 1. Filtrar todos los pares del rango (0, 20)
# 2. Aplicar la función doble a cada elemento
# 3. Convertir el resultado en una lista

print(l)
# Output: [0, 4, 8, 12, 16, 20, 24, 28, 32, 36]

```

Los iteradores, como cualquier objeto en Python, pueden ser pasados como argumentos a una función, devueltos por una función, asignados a una variable, etc.

```

def doble(iterador):
    return map(lambda x: x * 2, iterador) # map toma un iterador y devuelve otro igual

for a in doble(range(10)):
    print(a, end=" ")
# Output: 0 2 4 6 8 10 12 14 16 18

print(list(doble(range(10))))
# Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

a, b, *c = doble(range(7))
print(a, b, c)
# Output: 0 2 [4, 6, 8, 10, 12]

```

Incluso se puede combinar varios iteradores en uno solo:

```

par = zip('aeiou', range(1, 11, 2))
print(list(par))
# Output: [('a', 1), ('e', 3), ('i', 5), ('o', 7), ('u', 9)]

numerar = zip(range(1000), 'abracadabra') # Zip combina dos iteradores en uno solo
print(list(numerar))
# Output: [(0, 'a'), (1, 'b'), (2, 'r'), (3, 'a'), (4, 'c'), (5, 'a'), (6, 'd'), (7, 'a'), (8, 'b'), (9, 'r'), (10, 'a'), (11, 'c'), (12, 'a'), (13, 'd'), (14, 'a'), (15, 'b'), (16, 'r'), (17, 'a'), (18, 'c'), (19, 'a'), (20, 'd'), (21, 'a'), (22, 'b'), (23, 'r'), (24, 'a'), (25, 'c'), (26, 'a'), (27, 'd'), (28, 'a'), (29, 'b'), (30, 'r'), (31, 'a'), (32, 'c'), (33, 'a'), (34, 'd'), (35, 'a'), (36, 'b'), (37, 'r'), (38, 'a'), (39, 'c'), (40, 'a'), (41, 'd'), (42, 'a'), (43, 'b'), (44, 'r'), (45, 'a'), (46, 'c'), (47, 'a'), (48, 'd'), (49, 'a'), (50, 'b'), (51, 'r'), (52, 'a'), (53, 'c'), (54, 'a'), (55, 'd'), (56, 'a'), (57, 'b'), (58, 'r'), (59, 'a'), (60, 'c'), (61, 'a'), (62, 'd'), (63, 'a'), (64, 'b'), (65, 'r'), (66, 'a'), (67, 'c'), (68, 'a'), (69, 'd'), (70, 'a'), (71, 'b'), (72, 'r'), (73, 'a'), (74, 'c'), (75, 'a'), (76, 'd'), (77, 'a'), (78, 'b'), (79, 'r'), (80, 'a'), (81, 'c'), (82, 'a'), (83, 'd'), (84, 'a'), (85, 'b'), (86, 'r'), (87, 'a'), (88, 'c'), (89, 'a'), (90, 'd'), (91, 'a'), (92, 'b'), (93, 'r'), (94, 'a'), (95, 'c'), (96, 'a'), (97, 'd'), (98, 'a'), (99, 'b'), (100, 'r'), (101, 'a'), (102, 'c'), (103, 'a'), (104, 'd'), (105, 'a'), (106, 'b'), (107, 'r'), (108, 'a'), (109, 'c'), (110, 'a'), (111, 'd'), (112, 'a'), (113, 'b'), (114, 'r'), (115, 'a'), (116, 'c'), (117, 'a'), (118, 'd'), (119, 'a'), (120, 'b'), (121, 'r'), (122, 'a'), (123, 'c'), (124, 'a'), (125, 'd'), (126, 'a'), (127, 'b'), (128, 'r'), (129, 'a'), (130, 'c'), (131, 'a'), (132, 'd'), (133, 'a'), (134, 'b'), (135, 'r'), (136, 'a'), (137, 'c'), (138, 'a'), (139, 'd'), (140, 'a'), (141, 'b'), (142, 'r'), (143, 'a'), (144, 'c'), (145, 'a'), (146, 'd'), (147, 'a'), (148, 'b'), (149, 'r'), (150, 'a'), (151, 'c'), (152, 'a'), (153, 'd'), (154, 'a'), (155, 'b'), (156, 'r'), (157, 'a'), (158, 'c'), (159, 'a'), (160, 'd'), (161, 'a'), (162, 'b'), (163, 'r'), (164, 'a'), (165, 'c'), (166, 'a'), (167, 'd'), (168, 'a'), (169, 'b'), (170, 'r'), (171, 'a'), (172, 'c'), (173, 'a'), (174, 'd'), (175, 'a'), (176, 'b'), (177, 'r'), (178, 'a'), (179, 'c'), (180, 'a'), (181, 'd'), (182, 'a'), (183, 'b'), (184, 'r'), (185, 'a'), (186, 'c'), (187, 'a'), (188, 'd'), (189, 'a'), (190, 'b'), (191, 'r'), (192, 'a'), (193, 'c'), (194, 'a'), (195, 'd'), (196, 'a'), (197, 'b'), (198, 'r'), (199, 'a'), (200, 'c'), (201, 'a'), (202, 'd'), (203, 'a'), (204, 'b'), (205, 'r'), (206, 'a'), (207, 'c'), (208, 'a'), (209, 'd'), (210, 'a'), (211, 'b'), (212, 'r'), (213, 'a'), (214, 'c'), (215, 'a'), (216, 'd'), (217, 'a'), (218, 'b'), (219, 'r'), (220, 'a'), (221, 'c'), (222, 'a'), (223, 'd'), (224, 'a'), (225, 'b'), (226, 'r'), (227, 'a'), (228, 'c'), (229, 'a'), (230, 'd'), (231, 'a'), (232, 'b'), (233, 'r'), (234, 'a'), (235, 'c'), (236, 'a'), (237, 'd'), (238, 'a'), (239, 'b'), (240, 'r'), (241, 'a'), (242, 'c'), (243, 'a'), (244, 'd'), (245, 'a'), (246, 'b'), (247, 'r'), (248, 'a'), (249, 'c'), (250, 'a'), (251, 'd'), (252, 'a'), (253, 'b'), (254, 'r'), (255, 'a'), (256, 'c'), (257, 'a'), (258, 'd'), (259, 'a'), (260, 'b'), (261, 'r'), (262, 'a'), (263, 'c'), (264, 'a'), (265, 'd'), (266, 'a'), (267, 'b'), (268, 'r'), (269, 'a'), (270, 'c'), (271, 'a'), (272, 'd'), (273, 'a'), (274, 'b'), (275, 'r'), (276, 'a'), (277, 'c'), (278, 'a'), (279, 'd'), (280, 'a'), (281, 'b'), (282, 'r'), (283, 'a'), (284, 'c'), (285, 'a'), (286, 'd'), (287, 'a'), (288, 'b'), (289, 'r'), (290, 'a'), (291, 'c'), (292, 'a'), (293, 'd'), (294, 'a'), (295, 'b'), (296, 'r'), (297, 'a'), (298, 'c'), (299, 'a'), (300, 'd'), (301, 'a'), (302, 'b'), (303, 'r'), (304, 'a'), (305, 'c'), (306, 'a'), (307, 'd'), (308, 'a'), (309, 'b'), (310, 'r'), (311, 'a'), (312, 'c'), (313, 'a'), (314, 'd'), (315, 'a'), (316, 'b'), (317, 'r'), (318, 'a'), (319, 'c'), (320, 'a'), (321, 'd'), (322, 'a'), (323, 'b'), (324, 'r'), (325, 'a'), (326, 'c'), (327, 'a'), (328, 'd'), (329, 'a'), (330, 'b'), (331, 'r'), (332, 'a'), (333, 'c'), (334, 'a'), (335, 'd'), (336, 'a'), (337, 'b'), (338, 'r'), (339, 'a'), (340, 'c'), (341, 'a'), (342, 'd'), (343, 'a'), (344, 'b'), (345, 'r'), (346, 'a'), (347, 'c'), (348, 'a'), (349, 'd'), (350, 'a'), (351, 'b'), (352, 'r'), (353, 'a'), (354, 'c'), (355, 'a'), (356, 'd'), (357, 'a'), (358, 'b'), (359, 'r'), (360, 'a'), (361, 'c'), (362, 'a'), (363, 'd'), (364, 'a'), (365, 'b'), (366, 'r'), (367, 'a'), (368, 'c'), (369, 'a'), (370, 'd'), (371, 'a'), (372, 'b'), (373, 'r'), (374, 'a'), (375, 'c'), (376, 'a'), (377, 'd'), (378, 'a'), (379, 'b'), (380, 'r'), (381, 'a'), (382, 'c'), (383, 'a'), (384, 'd'), (385, 'a'), (386, 'b'), (387, 'r'), (388, 'a'), (389, 'c'), (390, 'a'), (391, 'd'), (392, 'a'), (393, 'b'), (394, 'r'), (395, 'a'), (396, 'c'), (397, 'a'), (398, 'd'), (399, 'a'), (400, 'b'), (401, 'r'), (402, 'a'), (403, 'c'), (404, 'a'), (405, 'd'), (406, 'a'), (407, 'b'), (408, 'r'), (409, 'a'), (410, 'c'), (411, 'a'), (412, 'd'), (413, 'a'), (414, 'b'), (415, 'r'), (416, 'a'), (417, 'c'), (418, 'a'), (419, 'd'), (420, 'a'), (421, 'b'), (422, 'r'), (423, 'a'), (424, 'c'), (425, 'a'), (426, 'd'), (427, 'a'), (428, 'b'), (429, 'r'), (430, 'a'), (431, 'c'), (432, 'a'), (433, 'd'), (434, 'a'), (435, 'b'), (436, 'r'), (437, 'a'), (438, 'c'), (439, 'a'), (440, 'd'), (441, 'a'), (442, 'b'), (443, 'r'), (444, 'a'), (445, 'c'), (446, 'a'), (447, 'd'), (448, 'a'), (449, 'b'), (450, 'r'), (451, 'a'), (452, 'c'), (453, 'a'), (454, 'd'), (455, 'a'), (456, 'b'), (457, 'r'), (458, 'a'), (459, 'c'), (460, 'a'), (461, 'd'), (462, 'a'), (463, 'b'), (464, 'r'), (465, 'a'), (466, 'c'), (467, 'a'), (468, 'd'), (469, 'a'), (470, 'b'), (471, 'r'), (472, 'a'), (473, 'c'), (474, 'a'), (475, 'd'), (476, 'a'), (477, 'b'), (478, 'r'), (479, 'a'), (480, 'c'), (481, 'a'), (482, 'd'), (483, 'a'), (484, 'b'), (485, 'r'), (486, 'a'), (487, 'c'), (488, 'a'), (489, 'd'), (490, 'a'), (491, 'b'), (492, 'r'), (493, 'a'), (494, 'c'), (495, 'a'), (496, 'd'), (497, 'a'), (498, 'b'), (499, 'r'), (500, 'a'), (501, 'c'), (502, 'a'), (503, 'd'), (504, 'a'), (505, 'b'), (506, 'r'), (507, 'a'), (508, 'c'), (509, 'a'), (510, 'd'), (511, 'a'), (512, 'b'), (513, 'r'), (514, 'a'), (515, 'c'), (516, 'a'), (517, 'd'), (518, 'a'), (519, 'b'), (520, 'r'), (521, 'a'), (522, 'c'), (523, 'a'), (524, 'd'), (525, 'a'), (526, 'b'), (527, 'r'), (528, 'a'), (529, 'c'), (530, 'a'), (531, 'd'), (532, 'a'), (533, 'b'), (534, 'r'), (535, 'a'), (536, 'c'), (537, 'a'), (538, 'd'), (539, 'a'), (540, 'b'), (541, 'r'), (542, 'a'), (543, 'c'), (544, 'a'), (545, 'd'), (546, 'a'), (547, 'b'), (548, 'r'), (549, 'a'), (550, 'c'), (551, 'a'), (552, 'd'), (553, 'a'), (554, 'b'), (555, 'r'), (556, 'a'), (557, 'c'), (558, 'a'), (559, 'd'), (560, 'a'), (561, 'b'), (562, 'r'), (563, 'a'), (564, 'c'), (565, 'a'), (566, 'd'), (567, 'a'), (568, 'b'), (569, 'r'), (570, 'a'), (571, 'c'), (572, 'a'), (573, 'd'), (574, 'a'), (575, 'b'), (576, 'r'), (577, 'a'), (578, 'c'), (579, 'a'), (580, 'd'), (581, 'a'), (582, 'b'), (583, 'r'), (584, 'a'), (585, 'c'), (586, 'a'), (587, 'd'), (588, 'a'), (589, 'b'), (590, 'r'), (591, 'a'), (592, 'c'), (593, 'a'), (594, 'd'), (595, 'a'), (596, 'b'), (597, 'r'), (598, 'a'), (599, 'c'), (599, 'd'), (600, 'a'), (601, 'b'), (602, 'r'), (603, 'a'), (604, 'c'), (605, 'a'), (606, 'd'), (607, 'a'), (608, 'b'), (609, 'r'), (610, 'a'), (611, 'c'), (612, 'a'), (613, 'd'), (614, 'a'), (615, 'b'), (616, 'r'), (617, 'a'), (618, 'c'), (619, 'a'), (620, 'd'), (621, 'a'), (622, 'b'), (623, 'r'), (624, 'a'), (625, 'c'), (626, 'a'), (627, 'd'), (628, 'a'), (629, 'b'), (630, 'r'), (631, 'a'), (632, 'c'), (633, 'a'), (634, 'd'), (635, 'a'), (636, 'b'), (637, 'r'), (638, 'a'), (639, 'c'), (640, 'a'), (641, 'd'), (642, 'a'), (643, 'b'), (644, 'r'), (645, 'a'), (646, 'c'), (647, 'a'), (648, 'd'), (649, 'a'), (650, 'b'), (651, 'r'), (652, 'a'), (653, 'c'), (654, 'a'), (655, 'd'), (656, 'a'), (657, 'b'), (658, 'r'), (659, 'a'), (660, 'c'), (661, 'a'), (662, 'd'), (663, 'a'), (664, 'b'), (665, 'r'), (666, 'a'), (667, 'c'), (668, 'a'), (669, 'd'), (670, 'a'), (671, 'b'), (672, 'r'), (673, 'a'), (674, 'c'), (675, 'a'), (676, 'd'), (677, 'a'), (678, 'b'), (679, 'r'), (680, 'a'), (681, 'c'), (682, 'a'), (683, 'd'), (684, 'a'), (685, 'b'), (686, 'r'), (687, 'a'), (688, 'c'), (689, 'a'), (690, 'd'), (691, 'a'), (692, 'b'), (693, 'r'), (694, 'a'), (695, 'c'), (696, 'a'), (697, 'd'), (698, 'a'), (699, 'b'), (700, 'r'), (701, 'a'), (702, 'c'), (703, 'a'), (704, 'd'), (705, 'a'), (706, 'b'), (707, 'r'), (708, 'a'), (709, 'c'), (7010, 'a'), (7011, 'd'), (7012, 'a'), (7013, 'b'), (7014, 'r'), (7015, 'a'), (7016, 'c'), (7017, 'a'), (7018, 'd'), (7019, 'a'), (7020, 'b'), (7021, 'r'), (7022, 'a'), (7023, 'c'), (7024, 'a'), (7025, 'd'), (7026, 'a'), (7027, 'b'), (7028, 'r'), (7029, 'a'), (7030, 'c'), (7031, 'a'), (7032, 'd'), (7033, 'a'), (7034, 'b'), (7035, 'r'), (7036, 'a'), (7037, 'c'), (7038, 'a'), (7039, 'd'), (7040, 'a'), (7041, 'b'), (7042, 'r'), (7043, 'a'), (7044, 'c'), (7045, 'a'), (7046, 'd'), (7047, 'a'), (7048, 'b'), (7049, 'r'), (7050, 'a'), (7051, 'c'), (7052, 'a'), (7053, 'd'), (7054, 'a'), (7055, 'b'), (7056, 'r'), (7057, 'a'), (7058, 'c'), (7059, 'a'), (7060, 'd'), (7061, 'a'), (7062, 'b'), (7063, 'r'), (7064, 'a'), (7065, 'c'), (7066, 'a'), (7067, 'd'), (7068, 'a'), (7069, 'b'), (7070, 'r'), (7071, 'a'), (7072, 'c'), (7073, 'a'), (7074, 'd'), (7075, 'a'), (7076, 'b'), (7077, 'r'), (7078, 'a'), (7079, 'c'), (7080, 'a'), (7081, 'd'), (7082, 'a'), (7083, 'b'), (7084, 'r'), (7085, 'a'), (7086, 'c'), (7087, 'a'), (7088, 'd'), (7089, 'a'), (7090, 'b'), (7091, 'r'), (7092, 'a'), (7093, 'c'), (7094, 'a'), (7095, 'd'), (7096, 'a'), (7097, 'b'), (7098, 'r'), (7099, 'a'), (7100, 'c'), (7101, 'a'), (7102, 'd'), (7103, 'a'), (7104, 'b'), (7105, 'r'), (7106, 'a'), (7107, 'c'), (7108, 'a'), (7109, 'd'), (7110, 'a'), (7111, 'b'), (7112, 'r'), (7113, 'a'), (7114, 'c'), (7115, 'a'), (7116, 'd'), (7117, 'a'), (7118, 'b'), (7119, 'r'), (7120, 'a'), (7121, 'c'), (7122, 'a'), (7123, 'd'), (7124, 'a'), (7125, 'b'), (7126, 'r'), (7127, 'a'), (7128, 'c'), (7129, 'a'), (7130, 'd'), (7131, 'a'), (7132, 'b'), (7133, 'r'), (7134, 'a'), (7135, 'c'), (7136, 'a'), (7137, 'd'), (7138, 'a'), (7139, 'b'), (7140, 'r'), (7141, 'a'), (7142, 'c'), (7143, 'a'), (7144, 'd'), (7145, 'a'), (7146, 'b'), (7147, 'r'), (7148, 'a'), (7149, 'c'), (7150, 'a'), (7151, 'd'), (7152, 'a'), (7153, 'b'), (7154, 'r'), (7155, 'a'), (7156, 'c'), (7157, 'a'), (7158, 'd'), (7159, 'a'), (7160, 'b'), (7161, 'r'), (7162, 'a'), (7163, 'c'), (7164, 'a'), (7165, 'd'), (7166, 'a'), (7167, 'b'), (7168, 'r'), (7169, 'a'), (7170, 'c'), (7171, 'a'), (7172, 'd'), (7173, 'a'), (7174, 'b'), (7175, 'r'), (7176, 'a'), (7177, 'c'), (7178, 'a'), (7179, 'd'), (7180, 'a'), (7181, 'b'), (7182, 'r'), (7183, 'a'), (7184, 'c'), (7185, 'a'), (7186, 'd'), (7187, 'a'), (7188, 'b'), (7189, 'r'), (7190, 'a'), (7191, 'c'), (7192, 'a'), (7193, 'd'), (7194, 'a'), (7195, 'b'), (7196, 'r'), (7197, 'a'), (7198, 'c'), (7199, 'a'), (7200, 'd'), (7201, 'a'), (7202, 'b'), (7203, 'r'), (7204, 'a'), (7205, 'c'), (7206, 'a'), (7207, 'd'), (7208, 'a'), (7209, 'b'), (7210, 'r'), (7211, 'a'), (7212, 'c'), (7213, 'a'), (7214, 'd'), (7215, 'a'), (7216, 'b'), (7217, 'r'), (7218, 'a'), (7219, 'c'), (7220, 'a'), (7221, 'd'), (7222, 'a'), (7223, 'b'), (7224, 'r'), (7225, 'a'), (7226, 'c'), (7227, 'a'), (7228, 'd'), (7229, 'a'), (7230, 'b'), (7231, 'r'), (7232, 'a'), (7233, 'c'), (7234, 'a'), (7235, 'd'), (7236, 'a'), (7237, 'b'), (7238, 'r'), (7239, 'a'), (7240, 'c'), (7241, 'a'), (7242, 'd'), (7243, 'a'), (7244, 'b'), (7245, 'r'), (7246, 'a'), (7247, 'c'), (7248, 'a'), (7249, 'd'), (7250, 'a'), (7251, 'b'), (7252, 'r'), (7253, 'a'), (7254, 'c'), (7255, 'a'), (7256, 'd'), (7257, 'a'), (7258, 'b'), (7259, 'r'), (7260, 'a'), (7261, 'c'), (7262, 'a'), (7263, 'd'), (7264, 'a'), (7265, 'b'), (7266, 'r'), (7267, 'a'), (7268, 'c'), (7269, 'a'), (7270, 'd'), (7271, 'a'), (7272, 'b'), (7273, 'r'), (7274, 'a'), (7275, 'c'), (7276, 'a'), (7277, 'd'), (7278, 'a'), (7279, 'b'), (7280, 'r'), (7281, 'a'), (7282, 'c'), (7283, 'a'), (7284, 'd'), (7285, 'a'), (7286, 'b'), (7287, 'r'), (7288, 'a'), (7289, 'c'), (7290, 'a'), (7291, 'd'), (7292, 'a'), (7293, 'b'), (7294, 'r'), (7295, 'a'), (7296, 'c'), (7297, 'a'), (7298, 'd'), (7299, 'a'), (7300, 'b'), (7301, 'r'), (7302, 'a'), (7303, 'c'), (7304, 'a'), (7305, 'd'), (7306, 'a'), (7307, 'b'), (7308, 'r'), (7309, 'a'), (7310, 'c'), (7311, 'a'), (7312, 'd'), (7313, 'a'), (7314, 'b'), (7315, 'r'), (7316, 'a'), (7317, 'c'), (7318, 'a'), (7319, 'd'), (7320, 'a'), (7321, 'b'), (7322, 'r'), (7323, 'a'), (7324, 'c'), (7325, 'a'), (7326, 'd'), (7327, 'a'), (7328, 'b'), (7329, 'r'), (7330, 'a'), (7331, 'c'), (7332, 'a'), (7333, 'd'), (7334, 'a'), (7335, 'b'), (7336, 'r'), (7337, 'a'), (7338, 'c'), (7339, 'a'), (7340, 'd'), (7341, 'a'), (7342, 'b'), (7343, 'r'), (7344, 'a'), (7345, 'c'), (7346, 'a'), (7347, 'd'), (7348, 'a'), (7349, 'b'), (7350, 'r'), (7351, 'a'), (7352, 'c'), (7353, 'a'), (7354, 'd'), (7355, 'a'), (7356, 'b'), (7357, 'r'), (7358, 'a'), (7359, 'c'), (7360, 'a'), (7361, 'd'), (7362, 'a'), (7363, 'b'), (7364, 'r'), (7365, 'a'), (7366, 'c'), (7367, 'a'), (7368, 'd'), (7369, 'a'), (7370, 'b'), (7371, 'r'), (7372, 'a'), (7373, 'c'), (7374, 'a'), (7375, 'd'), (7376, 'a'), (7377, 'b'), (7378, 'r'), (7379, 'a'), (7380, 'c'), (7381, 'a'), (7382, 'd'), (7383, 'a'), (7384, 'b'), (7385, 'r'), (7386, 'a'), (7387, 'c'), (7388, 'a'), (7389, 'd'), (7390, 'a'), (7391, 'b'), (7392, 'r'), (7393, 'a'), (7394, 'c'), (7395, 'a'), (7396, 'd'), (7397, 'a'), (7398, 'b'), (7399, 'r'), (7399, 'd'), (7400, 'a'), (7401, 'b'), (7402, 'r'), (7403, 'a'), (7404, 'c'), (7405, 'a'), (7406, 'd'), (7407, 'a'), (7408, 'b'), (7409, 'r'), (7410, 'a'), (7411, 'c'), (7412, 'a'), (7413, 'd'), (7414, 'a'), (7415, 'b'), (7416, 'r'), (7417, 'a'), (7418, 'c'), (7419, 'a'), (7420, 'd'), (7421, 'a'), (7422, 'b'), (7423, 'r'), (7424, 'a'), (7425, 'c'), (7426, 'a'), (7427, 'd'), (7428, 'a'), (7429, 'b'), (7430, 'r'), (7431, 'a'), (7432, 'c'), (7433, 'a'), (7434, 'd'), (7435, 'a'), (7436, 'b'), (7437, 'r'), (7438, 'a'), (7439, 'c'), (7440, 'a'), (7441, 'd'), (7442, 'a'), (7443, 'b'), (7444, 'r'), (7445, 'a'), (7446, 'c'), (7447, 'a'), (7448, 'd'), (7449, 'a'), (7450, 'b'), (7451, 'r'), (7452, 'a'), (7453, 'c'), (7454, 'a'), (7455, 'd'), (7456, 'a'), (7457, 'b'), (7458, 'r'), (7459, 'a'), (7460, 'c'), (7461, 'a'), (7462, 'd'), (7463, 'a'), (7464, 'b'), (7465, 'r'), (7466, 'a'), (7467, 'c'), (7468, 'a'), (7469, 'd'), (7470, 'a'), (7471, 'b'), (7472, 'r'), (7473, 'a'), (7474, 'c'), (7475, 'a'), (7476, 'd'), (7477, 'a'), (7478, 'b'), (7479, 'r'), (7479, 'd'), (7480, 'a'), (7481, 'b'), (7482, 'r'), (7483, 'a'), (7484, 'c'), (7485, 'a'), (7486, 'd'), (7487, 'a'), (7488, 'b'), (7489, 'r'), (7490, 'a'), (7491, 'c'), (7492, 'a'), (7493, 'd'), (7494, 'a'), (7495, 'b'), (7496, 'r'), (7497, 'a'), (7498, 'c'), (7499, 'a'), (7500, 'd'), (7501, 'a'), (7502, 'b'), (7503, 'r'), (7504, 'a'), (7505, 'c'), (7506, 'a'), (7507, 'd'), (7508, 'a'), (7509, 'b'), (7510, 'r'), (7511, 'a'), (7512, 'c'), (7513, 'a'), (7514, 'd'), (7515, 'a'), (7516, 'b'), (7517, 'r'), (7518, 'a'), (7519, 'c'), (7520, 'a'), (7521, 'd'), (7522, 'a'), (7523, 'b'), (7524, 'r'), (7525, 'a'), (7526, 'c'), (7527, 'a'), (7528, 'd'), (7529, 'a'), (7530, 'b'), (7531, 'r'), (7532, 'a'), (7533, 'c'), (7534, 'a'), (7535, 'd'), (7536, 'a'), (7537, 'b'), (7538, 'r'), (7539, 'a'), (7540, 'c'), (7541, 'a'), (7542, 'd'), (7543, 'a'), (7544, 'b'), (7545, 'r'), (7546, 'a'), (7547, 'c'), (7548, 'a'), (7549, 'd'), (7550, 'a'), (7551, 'b'), (7552, 'r'), (7553, 'a'), (7554, 'c'), (7555, 'a'), (7556, 'd'), (7557, 'a'), (7558, 'b'), (7559, 'r'), (7559, 'd'), (7560, 'a'), (7561, 'b'), (7562, 'r'), (7563, 'a'), (7564, 'c'), (7565, 'a'), (7566, 'd'), (7567, 'a'), (7568, 'b'), (7569, 'r'), (7570, 'a'), (7571, 'c'), (7572, 'a'), (7573, 'd'), (7574, 'a'), (7575, 'b'), (7576, 'r'), (7577, 'a'), (7578, 'c'), (7579, 'a'), (7580, 'd'), (7581, 'a'), (7582, 'b'), (7583, 'r'), (7584, 'a'), (7585, 'c'), (7586, 'a'), (7587, 'd'), (7588, 'a'), (7589, 'b'), (7590, 'r'), (7591, 'a'), (7592, 'c'), (7593, 'a'), (7594, 'd'), (7595, 'a'), (7596, 'b'), (7597, 'r'), (7598, 'a'), (7599, 'c'), (7599, 'd'), (7600, 'a'), (7601, 'b'), (7602, 'r'), (7603, 'a'), (7604, 'c'), (7605, 'a'), (7606, 'd'), (7607, 'a'), (7608, 'b'), (7609, 'r'), (7610, 'a'), (76
```

```

from itertools import *

for i in count(10, 2):
    print(i, end=" ")
    if i > 20: break
# Output: 10 12 14 16 18 20 22

for i in cycle("aeiou"):
    print(i, end=" ")
    if i == 'u': break
# Output: a e i o u

for i in repeat("Hola", 3):
    print(i, end=" ")
# Output: Hola Hola Hola

for i in chain("aeiou", range(10)):
    print(i, end=" ")
# Output: a e i o u 0 1 2 3 4 5 6 7 8 9

for i in combinations("abcd", 2):
    print(i, end=" ")
# Output: ('a', 'b') ('a', 'c') ('a', 'd') ('b', 'c') ('b', 'd') ('c', 'd')

for i in accumulate(range(10)):
    print(i, end=" ")
# Output: 0 1 3 6 10 15 21 28 36 45

for i in product("abc", "123"):
    print(i, end=" ")
# Output: ('a', '1') ('a', '2') ('a', '3') ('b', '1') ('b', '2') ('b', '3') ('c', '1') ('c', '2') ('c', '3')

# Agrupar las vocales y las consonantes de una cadena

def es_vocal(x):
    return x in "aeiou"

for k, g in groupby(sorted("hola mundo"), es_vocal):
    print(k, list(g))
# Output:
# False [' ', 'a', 'c', 'd', 'h', 'l', 'm', 'n', 'o', 'r', 'u']
# True ['e', 'i', 'o', 'u']

```

Algunas funciones de la librería `itertools`:

- **`islice`**: Genera una secuencia de elementos de un iterador.
- **`takewhile`**: Genera una secuencia de elementos mientras se cumpla una condición.
- **`dropwhile`**: Genera una secuencia de elementos después de que se cumpla una condición.
- **`filterfalse`**: Genera una secuencia de elementos que no cumplen una condición.

- **compress**: Genera una secuencia de elementos seleccionados por una máscara.
- **count**: Genera una secuencia de números enteros.
- **cycle**: Genera una secuencia cíclica de elementos.
- **repeat**: Genera una secuencia de elementos repetidos.
- **chain**: Concatena varias secuencias en una sola.
- **combinations**: Genera todas las combinaciones posibles de una secuencia.
- **accumulate**: Genera una secuencia acumulada de elementos.
- **product**: Genera el producto cartesiano de varias secuencias.
- **permutations**: Genera todas las permutaciones posibles de una secuencia.
- **groupby**: Agrupa elementos consecutivos de una secuencia.

Generadores

Existe una forma muy simple de generar iteradores en Python, utilizando funciones generadoras. Una función generadora es una función que contiene la palabra clave `yield` en lugar de `return`. Cuando en una función se encuentra con la palabra clave `yield`, retorna un valor y se suspende la ejecución de la función. Cuando se llama a la función nuevamente, la ejecución se reanuda en el punto donde se suspendió.

```
# Genera tres valores: 1, 2, 3
def tres():
    yield 1
    yield 2
    yield 3

for i in tres():
    print(i, end=" ")
# Output: 1 2 3

# Genera los números del 1 al n
def contar(n):
    for i in range(n):
        yield i+1

for i in contar(5):
    print(i, end=" ")
# Output: 1 2 3 4 5

# También se puede usar `next()` para obtener los valores de un generador de uno en
c = contar(3)
print(next(c)) # 1
print(next(c)) # 2
print(next(c)) # 3
# print(next(c)) # Esto causaría un error: StopIteration
```

La maravilla de los generadores es que trabajan en forma perezosa (lazy), es decir, no generan todos los valores de una vez, sino que los generan a medida que se necesitan. Esto es especialmente útil cuando se trabaja con colecciones grandes o infinitas, ya que no es necesario cargar todos los elementos en memoria al mismo tiempo.

Por ejemplo, el siguiente generador produce los números primos de forma indefinida:

```

def primos():
    n = 2
    while True:
        for i in range(2, n):
            if n % i == 0:
                break # Si es divisible por alguno, no es primo
        else:
            yield n # Si no es divisible por ninguno, es primo
        n += 1

p = primos()

# Se los puede traer uno a uno con `next()`
print(next(p)) # 2
print(next(p)) # 3
print(next(p)) # 5
print(next(p)) # 7
print(next(p)) # 11
# print(next(p)) # Continuar según se necesite

from itertools import *

def menor100(x):
    return x < 100

for i in takewhile(menor100, primos()): # Tome los primos mientras sean menores a 100
    print(i, end=" ")
# Output: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

# o más breve aún
print(list(takewhile(menor100, primos()))) # Tome los primos mientras sean menores a 100
# Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89]

print(list(islice(primos(), 2, 20, 3))) # Dame una lista de los primos desde el índice 2 hasta el 20, saltando de 3 en 3
# Output: [5, 11, 17, 23, 31, 41, 47, 59, 67, 79, 89]

print(list(zip(range(10), primos()))) # Empareja los primos con los primeros 10 números
# Output: [(0, 2), (1, 3), (2, 5), (3, 7), (4, 11), (5, 13), (6, 17), (7, 19), (8, 23), (9, 29)]

```

Se pueden realizar operaciones muy interesantes como concatenar generadores, filtrarlos, etc., usando `yield from`:

```

# Forma tradicional de concatenar dos generadores
def concatenar(a, b):
    for i in a:
        yield i
    for i in b:
        yield i

print(list(concatenar(range(3), range(5, 7))))
# Output: [0, 1, 2, 5, 6]

# Forma moderna de concatenar dos generadores
def concatenar(a, b):
    yield from a
    yield from b

print(list(concatenar(range(3), range(5, 7))))
# Output: [0, 1, 2, 5, 6]

# Forma tradicional de filtrar un generador
def filtrar(a, condicion):
    for i in a:
        if condicion(i):
            yield i

print(list(filtrar(range(10), lambda x: x % 2 == 0)))
# Output: [0, 2, 4, 6, 8]

```

Por último, hay una forma muy interesante de generar generadores usando una **expresión generadora**. Las expresiones generadoras son similares a las listas por comprensión, pero en lugar de devolver una lista, devuelven un generador. Por ejemplo:

```

# Dame los cuadrados de los números entre 0 y 100 que sean múltiplos de 32
g = (x**2 for x in range(100) if x % 32 == 0)
print(list(g))
# Output: [0, 1024, 4096, 9216]

# Se puede usar una expresión generadora en cualquier lugar donde se espera un gene

print(sum(x**2 for x in range(10)))
# Output: 285

def filtrar(a, condicion):
    return (i for i in a if condicion(i))

print(list(filtrar(range(10), lambda x: x % 2 == 0)))
# Output: [0, 2, 4, 6, 8]

```

Gestión de errores

En Python, un error es una situación excepcional que interrumpe la ejecución normal de un programa. Los errores pueden ser de diferentes tipos, como errores de sintaxis, errores de tiempo de ejecución y errores de lógica.

En este capítulo, aprenderemos a manejar los errores en Python utilizando las sentencias `try`, `except` y `finally`.

Errores de sintaxis

Los errores de sintaxis ocurren cuando el intérprete de Python no puede entender el código debido a una estructura incorrecta. Por ejemplo:

```
# Error de sintaxis
print("Hola, Mundo!")
```

Errores de lógica

Los errores de lógica ocurren cuando un programa no produce el resultado esperado debido a un error en el diseño o la implementación del código. Por ejemplo:

```
# Error de lógica
def sumar(a, b):
    return a - b

resultado = sumar(5, 3)

print(resultado) # Output: 2
```

En el ejemplo anterior, se ha cometido un error de lógica al restar los números `a` y `b` en lugar de sumarlos. En el ejemplo anterior, se ha cometido un error de sintaxis al no cerrar las comillas en la cadena de texto.

Errores de tiempo de ejecución

Los errores de tiempo de ejecución ocurren durante la ejecución de un programa y pueden ser causados por diferentes situaciones, como divisiones por cero, acceso a índices fuera de rango, entre otros. Por ejemplo:

```
# Error de tiempo de ejecución
resultado = 10 / 0
```

En el ejemplo anterior, se ha cometido un error de tiempo de ejecución al intentar dividir un número por cero.

Manejo de errores

Para manejar los errores en Python, podemos utilizar las sentencias `try`, `except` y `finally`. La sentencia `try` se utiliza para probar un bloque de código en busca de errores, la sentencia `except` se utiliza para manejar los errores que se producen en el bloque `try`, y la sentencia `finally` se utiliza para ejecutar un bloque de código independientemente de si se ha producido un error o no.

Por ejemplo:

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("Error: División por cero.")
finally:
    print("El programa ha finalizado.")
```

En el ejemplo anterior, se ha utilizado la sentencia `try` para intentar dividir un número por cero. Como se produce un error de división por cero, se ejecuta el bloque `except` que imprime un mensaje de error. Finalmente, se ejecuta el bloque `finally` que imprime un mensaje de finalización del programa.

Generación de excepciones

En Python, también es posible generar excepciones manualmente utilizando la sentencia `raise`.

Por ejemplo:

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

try:
    resultado = dividir(10, 0)
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

En el ejemplo anterior, se ha definido una función `dividir` que genera una excepción de división por cero si el segundo argumento es igual a cero. Luego, se ha utilizado la sentencia `try` para llamar a la función `dividir` con los argumentos `10` y `0`. Como se produce un error de división por cero, se ejecuta el bloque `except` que imprime un mensaje de error.

Un comportamiento interesante de la sentencia `raise` es que si se produce dentro de una función, la excepción se propaga hacia arriba en la pila de llamadas hasta que se maneja con una sentencia `try` o termina la ejecución del programa.

Excepciones personalizadas

En Python, también es posible crear excepciones personalizadas para manejar situaciones específicas en un programa. Para crear una excepción personalizada, se debe definir una clase que herede de la clase `Exception`. Por ejemplo:

```
class MiError(Exception):
    pass

try:
    raise MiError("Este es un mensaje de error personalizado.")
except MiError as e:
    print(f"Error: {e}")
```

En el ejemplo anterior, se ha creado una excepción personalizada llamada `MiError` que hereda de la clase `Exception`. Luego, se ha utilizado la sentencia `raise` para lanzar la excepción personalizada con un mensaje de error personalizado. Finalmente, se ha utilizado la sentencia `except` para manejar la excepción personalizada e imprimir el mensaje de error.

Módulos

En Python, los **módulos** son una de las herramientas más poderosas y versátiles que permiten organizar y reutilizar el código de manera eficiente. Un módulo es simplemente un archivo que contiene definiciones y declaraciones de Python, como funciones, clases y variables. Los módulos ayudan a mantener el código limpio, modular y fácil de mantener.

Utilidad de los Módulos

Los módulos proporcionan varias ventajas clave:

- **Reutilización de Código:** Permiten reutilizar funciones, clases y variables en diferentes programas sin necesidad de reescribir el código.
- **Organización:** Ayudan a organizar el código en archivos separados basados en su funcionalidad, lo que mejora la legibilidad y el mantenimiento.
- **Namespace:** Evitan conflictos de nombres al encapsular definiciones dentro de su propio espacio de nombres.
- **Colaboración:** Facilitan el trabajo en equipo, permitiendo que diferentes desarrolladores trabajen en diferentes módulos de un proyecto.

Creación de un Módulo

Crear un módulo en Python es sencillo. Solo necesitas crear un archivo con la extensión `.py` y definir las funciones, clases o variables que deseas incluir.

Ejemplo de Creación de un Módulo

Supongamos que queremos crear un módulo llamado `matematicas.py` que contiene funciones para operaciones matemáticas básicas.

```
# matematicas.py

def suma(a, b):
    """Devuelve la suma de dos números."""
    return a + b

def resta(a, b):
    """Devuelve la resta de dos números."""
    return a - b

def multiplicacion(a, b):
    """Devuelve la multiplicación de dos números."""
    return a * b

def division(a, b):
    """Devuelve la división de dos números. Lanza un error si el divisor es cero."""
    if b == 0:
        raise ValueError("El divisor no puede ser cero.")
    return a / b
```

En este ejemplo, hemos creado un módulo `matematicas.py` que define cuatro funciones básicas: `suma`, `resta`, `multiplicacion` y `division`.

Uso de Módulos

Una vez creado un módulo, puedes utilizar sus funciones, clases y variables en otros archivos de Python mediante la instrucción `import`.

Importar Todo el Módulo

Puedes importar todo el módulo y acceder a sus miembros utilizando la sintaxis `modulo.miembro`.

```
# usar_matematicas.py

import matematicas

resultado_suma = matematicas.suma(5, 3)
print("Suma:", resultado_suma) # Output: Suma: 8

resultado_resta = matematicas.resta(10, 4)
print("Resta:", resultado_resta) # Output: Resta: 6
```

Importar Funciones Específicas

También puedes importar funciones específicas del módulo utilizando la sintaxis `from modulo import miembro`.

```
# usar_matematicas_especificas.py

from matematicas import multiplicacion, division

resultado_multiplicacion = multiplicacion(7, 6)
print("Multiplicación:", resultado_multiplicacion) # Output: Multiplicación: 42

resultado_division = division(20, 4)
print("División:", resultado_division) # Output: División: 5.0
```

Importar con Alias

Para evitar conflictos de nombres o simplificar el acceso, puedes asignar un alias al módulo o a sus miembros.

```
# usar_matematicas_alias.py

import matematicas as mat

resultado = mat.suma(2, 3)
print("Resultado:", resultado) # Output: Resultado: 5

from matematicas import resta as r
print("Resta:", r(10, 5)) # Output: Resta: 5
```

Construcción de Paquetes

Un **paquete** en Python es una forma de organizar múltiples módulos en una estructura de directorios jerárquica. Los paquetes facilitan la gestión de grandes proyectos dividiendo el código en módulos más pequeños y manejables.

Estructura de un Paquete

Para crear un paquete, sigue estos pasos:

1. **Crear un Directorio:** Este directorio representará el paquete.
2. **Agregar un Archivo `__init__.py`:** Este archivo puede estar vacío o contener código de inicialización del paquete. Su presencia indica a Python que el directorio debe ser tratado como un paquete.
3. **Agregar Módulos:** Dentro del directorio, agrega archivos `.py` que serán los módulos del paquete.

Ejemplo de Creación de un Paquete

Supongamos que queremos crear un paquete llamado `utilidades` que contiene módulos para matemáticas y cadenas de texto.

Estructura del Paquete

```
utilidades/
└── __init__.py
└── matematicas.py
└── cadenas.py
```

Contenido de `matematicas.py`

```
# utilidades/matematicas.py

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

def resta(a, b):
    return a - b
```

Contenido de `cadenas.py`

```
# utilidades/cadenas.py

def mayusculas(texto):
    """Convierte el texto a mayúsculas."""
    return texto.upper()

def minusculas(texto):
    """Convierte el texto a minúsculas."""
    return texto.lower()
```

Contenido de `__init__.py`

```
# utilidades/__init__.py

from .matematicas import suma, resta
from .cadenas import mayusculas, minusculas

__all__ = ['suma', 'resta', 'mayusculas', 'minusculas']
```

En este ejemplo, el archivo `__init__.py` importa las funciones de los módulos `matematicas.py` y `cadenas.py` y define la lista `__all__` para controlar qué se exporta cuando se utiliza `from utilidades import *`.

Uso del Paquete

Una vez creado el paquete, puedes utilizarlo en otros archivos de Python de la siguiente manera:

```
# usar_utilidades.py

from utilidades import suma, mayusculas

resultado = suma(10, 5)
print("Suma:", resultado) # Output: Suma: 15

texto = "hola mundo"
texto_mayusculas = mayusculas(texto)
print("Texto en mayúsculas:", texto_mayusculas) # Output: Texto en mayúsculas: HOLA MUNDO
```

Ejemplos Prácticos

Creación y Uso de un Módulo

1. Crear el Módulo

Crea un archivo llamado `saludos.py` con el siguiente contenido:

```
# saludos.py

def saludar(nombre):
    """Imprime un saludo personalizado."""
    print(f"Hola, {nombre}!")

def despedir(nombre):
    """Imprime una despedida personalizada."""
    print(f"Adiós, {nombre}!")
```

2. Usar el Módulo

Crea otro archivo llamado `uso_saludos.py` para utilizar el módulo `saludos`.

```
# uso_saludos.py

import saludos

saludos.saludar("Ana")      # Output: Hola, Ana!
saludos.despedir("Ana")     # Output: Adiós, Ana!
```

Creación y Uso de un Paquete

1. Crear la Estructura del Paquete

```
mi_paquete/
  __init__.py
  matematicas.py
  utilidades_cadenas.py
```

2. Contenido de `matematicas.py`

```
# mi_paquete/matematicas.py

def multiplicar(a, b):
    return a * b

def dividir(a, b):
    if b == 0:
        raise ValueError("El divisor no puede ser cero.")
    return a / b
```

3. Contenido de `utilidades_cadenas.py`

```
# mi_paquete/utilidades_cadenas.py

def capitalizar(texto):
    """Capitaliza la primera letra de cada palabra."""
    return texto.title()

def invertir(texto):
    """Invierte el texto."""
    return texto[::-1]
```

4. Contenido de `__init__.py`

```
# mi_paquete/__init__.py

from .matematicas import multiplicar, dividir
from .utilidades_cadenas import capitalizar, invertir

__all__ = ['multiplicar', 'dividir', 'capitalizar', 'invertir']
```

5. Usar el Paquete

Crea un archivo llamado `uso_paquete.py` para utilizar el paquete `mi_paquete`.

```
# uso_paquete.py

from mi_paquete import multiplicar, capitalizar

resultado = multiplicar(4, 5)
print("Multiplicación:", resultado) # Output: Multiplicación: 20

texto = "hola mundo"
texto_capitalizado = capitalizar(texto)
print("Texto capitalizado:", texto_capitalizado) # Output: Texto capitalizado:
```

Uso de `__init__.py`

El archivo `__init__.py` es esencial para que Python reconozca un directorio como un paquete. Puede contener código de inicialización para el paquete o simplemente estar vacío. Además, es útil para definir qué componentes del paquete estarán disponibles cuando se importe utilizando `from paquete import *`.

Ejemplo de `__init__.py` con Código de Inicialización

```
# mi_paquete/__init__.py

print("Paquete 'mi_paquete' cargado.")

from .matematicas import multiplicar, dividir
from .utilidades_cadenas import capitalizar, invertir

__all__ = ['multiplicar', 'dividir', 'capitalizar', 'invertir']
```

Al importar el paquete, se ejecutará el código dentro de `__init__.py`.

```
# uso_paquete.py

import mi_paquete
# Output: Paquete 'mi_paquete' cargado.

from mi_paquete import multiplicar

resultado = multiplicar(2, 3)
print("Resultado:", resultado) # Output: Resultado: 6
```

Ejemplos de Paquetes

Paquete Estándar de Python: `math`

Python incluye una amplia variedad de módulos estándar que forman parte de la biblioteca estándar. Uno de ellos es el módulo `math`, que proporciona funciones matemáticas avanzadas.

```
# usar_math.py

import math

print("Valor de pi:", math.pi)          # Output: Valor de pi: 3.141592653589793
print("Raíz cuadrada de 16:", math.sqrt(16))  # Output: Raíz cuadrada de 16: 4.0
print("Seno de 90 grados:", math.sin(math.radians(90))) # Output: Seno de 90 grados: 1.0
```

Paquete de Terceros: `requests`

`requests` es un paquete de terceros ampliamente utilizado para realizar solicitudes HTTP de manera sencilla.

1. Instalación del Paquete

```
pip install requests
```

2. Uso del Paquete

```
# usar_requests.py

import requests

respuesta = requests.get("https://api.github.com")
print("Código de estado:", respuesta.status_code)  # Output: Código de estado: 200
print("Contenido:", respuesta.json())
```

Conclusión

Los módulos y paquetes son fundamentales para escribir código Python limpio, organizado y reutilizable. Al dividir el código en módulos, puedes gestionar mejor proyectos grandes, facilitar la colaboración y mantener una estructura lógica. Además, la capacidad de construir paquetes permite agrupar módulos relacionados, lo que mejora aún más la modularidad y escalabilidad de tus proyectos.

Resumen

1. Utilidad de los Módulos:

- Reutilización de código.
- Organización del código.
- Evitar conflictos de nombres.
- Facilitar la colaboración en equipo.

2. Creación de un Módulo:

- Crear un archivo `.py` con definiciones de funciones, clases y variables.

3. Uso de Módulos:

- Importar el módulo completo.
- Importar funciones o clases específicas.
- Utilizar alias para módulos o miembros.

4. Construcción de Paquetes:

- Crear un directorio con un archivo `__init__.py`.
- Agregar módulos al paquete.
- Importar y utilizar componentes del paquete.

5. Ejemplos Prácticos:

- Creación y uso de módulos individuales.
- Creación y uso de paquetes.
- Uso de paquetes estándar y de terceros.

Estos conceptos y prácticas te permitirán aprovechar al máximo las capacidades de Python para desarrollar aplicaciones robustas y mantenibles.

Introducción a NumPy

NumPy es una **librería** fundamental para el cálculo numérico en Python. Proporciona soporte para crear y manipular **arrays** y **matrices** multidimensionales de manera eficiente, además de una amplia colección de funciones matemáticas para operar con estos objetos.

Instalación y Importación

Para instalar NumPy, puedes utilizar `pip`:

```
pip install numpy
```

Una vez instalada, se importa comúnmente con el alias `np`:

```
import numpy as np
```

Utilidad de NumPy

NumPy es esencial para aplicaciones que requieren operaciones matemáticas y estadísticas de alto rendimiento, como análisis de datos, aprendizaje automático y procesamiento de imágenes. Sus arrays son más eficientes en términos de memoria y velocidad que las listas de Python para operaciones numéricas.

Creación de Arrays y Matrices

A partir de listas

Puedes crear un **array** unidimensional a partir de una lista:

```
lista = [1, 2, 3, 4, 5]
array = np.array(lista)
print(array)
#> [1 2 3 4 5]
```

Para crear una **matriz** (array bidimensional), utilizas una lista de listas:

```
lista_de_listas = [[1, 2], [3, 4]]
matriz = np.array(lista_de_listas)
print(matriz)
#>
# [[1 2]
#  [3 4]]
```

Arrays Multidimensionales

Con ceros, unos o valores arbitrarios

- **Array de ceros:**

```
ceros = np.zeros((2, 3))
print(ceros)
#>
# [[0. 0. 0.]
#  [0. 0. 0.]]
```

- **Array de unos:**

```
unos = np.ones((3, 2))
print(unos)
#>
# [[1. 1.]
#  [1. 1.]
#  [1. 1.]]
```

- **Array con un valor arbitrario:**

```
valor_arbitrario = np.full((2, 2), 7)
print(valor_arbitrario)
#>
# [[7 7]
#  [7 7]]
```

Matrices con valores aleatorios

- **Valores aleatorios entre 0 y 1:**

```
aleatorios = np.random.random((2, 2))
print(aleatorios)
#>
# [[0.5488135  0.71518937]
#  [0.60276338  0.54488318]]
```

- Valores aleatorios enteros:

```
aleatorios_enteros = np.random.randint(1, 10, size=(2, 3))
print(aleatorios_enteros)
#>
# [[3 7 9]
#  [2 7 8]]
```

Operaciones Numéricas con Matrices

Operaciones entre Matrices

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

suma = a + b
resta = a - b
producto = a * b # Producto elemento a elemento
producto_matricial = np.dot(a, b)

print("Suma:\n", suma)
#>
# [[ 6  8]
#  [10 12]]
print("Resta:\n", resta)
#>
# [[-4 -4]
#  [-4 -4]]

print("Producto elemento a elemento:\n", producto)
#>
# [[ 5 12]
#  [21 32]]
print("Producto matricial:\n", producto_matricial)
#>
# [[19 22]
#  [43 50]]
```

Operaciones con Valores Numéricos

```
escalar = 2
multiplicacion = a * escalar
division = a / escalar

print("Multiplicación por escalar:\n", multiplicacion)
#>
# [[2 4]
# [6 8]]

print("División por escalar:\n", division)
#>
# [[0.5 1. ]
# [1.5 2. ]]
```

Funciones Estadísticas y Procesamiento

```
matriz = np.array([[1, 2, 3], [4, 5, 6]])

print("Máximo:", matriz.max())
#> Máximo: 6
print("Mínimo:", matriz.min())
#> Mínimo: 1
print("Media:", matriz.mean())
#> Media: 3.5
print("Desviación estándar:", matriz.std())
#> Desviación estándar: 1.707825127659933

print("Suma por columnas:", matriz.sum(axis=0))
#> Suma por columnas: [5 7 9]
print("Suma por filas:", matriz.sum(axis=1))
#> Suma por filas: [ 6 15]
```

Acceso y Manipulación de Elementos

Acceder a Elementos y Slices

```
array = np.array([1, 2, 3, 4, 5])

print("Elemento en posición 2:", array[2])
#> Elemento en posición 2: 3
print("Slice de posición 1 a 3:", array[1:4])
#> Slice de posición 1 a 3: [2 3 4]

matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(matriz)
#>
# [[1 2 3]
# [4 5 6]
# [7 8 9]]

print("Elemento en posición (1, 2):", matriz[1, 2])
#> Elemento en posición (1, 2): 6

print("Slice de filas 0 y 1, columnas 1 y 2:\n", matriz[:2, 1:])
#> Slice de filas 0 y 1, columnas 1 y 2:
#> [[2 3]
#> [5 6]]

print("Esquina inferior derecha:\n", matriz[-2:, -2:])
#> Esquina inferior derecha:
#> [[5 6]
#> [8 9]]
```

Cambiar Elementos y Slices

Cuando usamo un slice del lado izquierdo de una asignacion podemos modificar los valores de los elementos seleccionados.

```
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("Matriz original:\n", matriz)
#> Matriz original:
# [[1 2 3]
# [4 5 6]
# [7 8 9]]

# Modificar un elemento
matriz[1, 2] = 100
print("Matriz modificada:\n", matriz)
#> Matriz modificada:
# [[ 1   2   3]
# [ 4   5 100]
# [ 7   8   9]]

# Modificar una fila
matriz[0] = [10, 20, 30]
print("Matriz con la primera fila modificada:\n", matriz)
#> Matriz con la primera fila modificada:
# [[10 20 30]
# [ 4  5 100]
# [ 7  8  9]]

# Modificar una columna
matriz[:, 1] = [40, 50, 60]
print("Matriz con la segunda columna modificada:\n", matriz)
#> Matriz con la segunda columna modificada:
# [[10 40 30]
# [ 4 50 100]
# [ 7 60  9]]

# Modificando todos juntos
matriz[:, 1] = 200
print("Matriz con la segunda columna modificada:\n", matriz)
#> Matriz con la segunda columna modificada:
# [[ 10 200 30]
# [ 4 200 100]
# [ 7 200  9]]

# Modificando un slice
matriz[1:, 1:] = 300
print("Matriz con la esquina inferior derecha modificada:\n", matriz)
#> Matriz con la esquina inferior derecha modificada:
# [[ 10 200 30]
# [ 4 300 300]
# [ 7 300 300]]
```

Cambio de forma de una matriz (Reshape)

Los datos en una matriz se almacenan en forma secuencial pero se acceden con índices de dos dimensiones. Podemos cambiar la forma de una matriz sin cambiar los datos que contiene.

```

### Reshape

original = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print("Array original:\n", original)
#> Array original:
#> [1 2 3 4 5 6 7 8]
print("Dimensiones del array original:", original.shape)
#> Dimensiones del array original: (8,)

reacomodado = original.reshape(2, 4)

print("Array reacomodado:\n", reacomodado)
#> Array reacomodado:
#> [[1 2 3 4]
#>  [5 6 7 8]]
print("Dimensiones del array reacomodado:", reacomodado.shape)
#> Dimensiones del array reacomodado: (2, 4)

reasignado = original.reshape(4, 2)
print("Array reasignado:\n", reasignado)
#> Array reasignado:
#> [[1 2]
#>  [3 4]
#>  [5 6]
#>  [7 8]]


d1 = original.reshape(8)
print("Array 1D:\n", d1)
#> Array 1D:
#> [1 2 3 4 5 6 7 8]

d2 = original.reshape(2, 4)
print("Array 2D:\n", d2)
#> Array 2D:
#> [[1 2 3 4]
#>  [5 6 7 8]]


d3 = original.reshape(2, 2, 2)
print("Array 3D:\n", d3)
#> Array 3D:
#> [[[1 2]
#>   [3 4]]
#>
#>  [[5 6]
#>   [7 8]]]


d4 = original.reshape(2,2,2,1)
print("Array 4D:\n", d4)
#> Array 4D:
#> [[[[1
#>    [2]]]
#>
```

```
#>   [[3]
#>     [4]]]
#>
#> [[[5]
#>   [6]]
#>
#> [[7]
#>   [8]]]]
```

Modificar Matrices

```
matriz = np.array([[1, 2], [3, 4]])
matriz[0, 1] = 20
print("Matriz modificada:\n", matriz)
```

Filtrado con Condiciones

Crear Matrices con Condiciones

```
array = np.arange(10).reshape(2, 5)
print("Array original:\n", array)
#> Array original:
#> [[0 1 2 3 4]
#>  [5 6 7 8 9]]

condicion = array > 3 * array < 8
print("Condición array > 3 y < 8:\n", condicion)
#> Condición array > 3 y < 8:
#> [[False False False False  True]
#>  [ True  True  True False False]]

array[condicion] = 0
print("Array modificado:\n", array)
#> Array modificado:
#> [[0 1 2 3 0]
#>  [0 0 0 8 9]]

print(a[condicion])
#> [4 5 6 7]

pares = array % 2 == 0
print("Elementos pares:", pares)
#> Elementos pares:
#> [[ True False  True False  True]

print(array[pares])
#> [0 2 0 0 8]
```

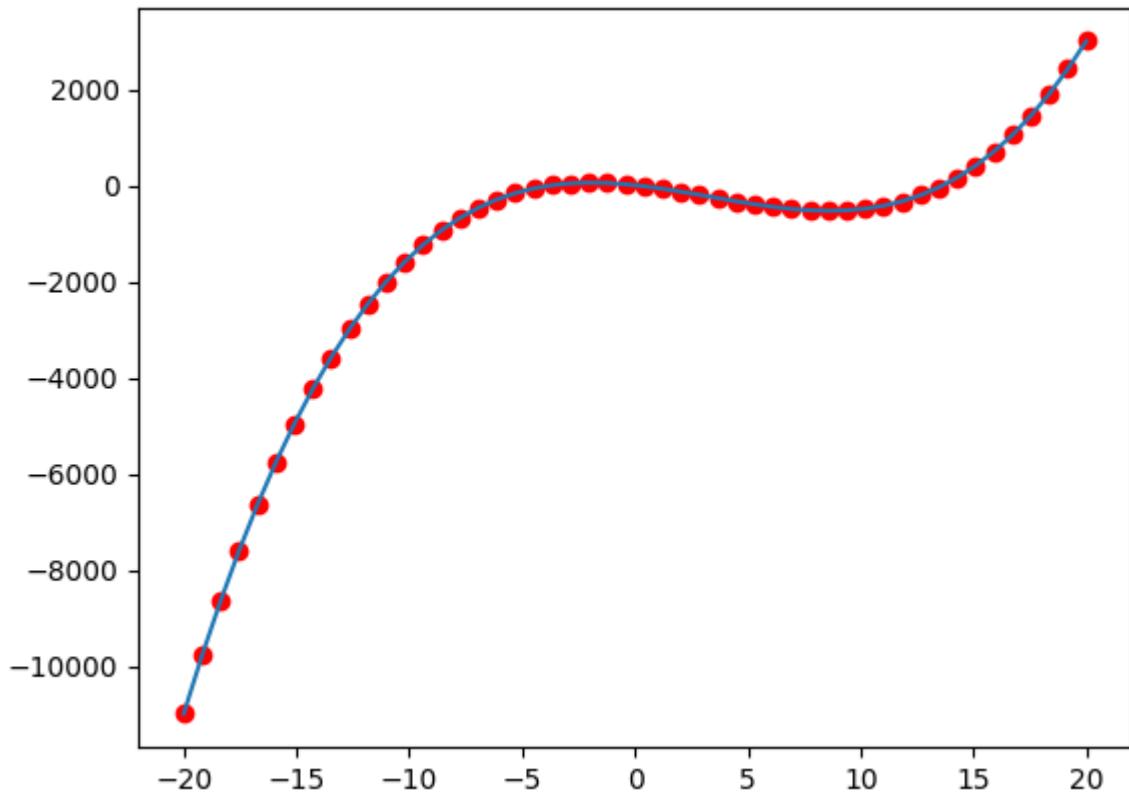
Graficar Funciones con NumPy

Generar y Graficar una Función Cuadrática

```
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 100)
y = x**2

plt.plot(x, y)
plt.title("Función Cuadrática")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Ajuste de Polinomios

Generar Datos y Agregar Ruido

```
np.random.seed(0)
x = np.linspace(-10, 10, 50)
y = x**2 + np.random.normal(0, 10, size=x.size)

plt.scatter(x, y, label="Datos con ruido")
```

Usar `polyfit` para Ajuste

```
Σ```
## Procesamiento de Imágenes
### Cargar una Imagen PNG en un Array
```python
from matplotlib.image import imread

imagen = imread('imagen.png')
print("Dimensiones de la imagen:", imagen.shape)
#> Dimensiones de la imagen: (512, 512, 4)
```

## Graficar los Cuatro Canales

```
fig, axs = plt.subplots(1, 4, figsize=(15, 5))

canales = ['Rojo', 'Verde', 'Azul', 'Alpha']
for i in range(4):
 axs[i].imshow(imagen[:, :, i], cmap='gray')
 axs[i].set_title(f'Canal {canales[i]}')
 axs[i].axis('off')

plt.show()
```

# Modificar los Canales

## Modificar el Canal Rojo

```
import numpy as np

alto, ancho, _ = imagen.shape
centro_alto = alto // 2
centro_ancho = ancho // 2
tamaño_cuadrado = 50

imagen_modificada = imagen.copy()
imagen_modificada[centro_alto - tamaño_cuadrado:centro_alto + tamaño_cuadrado,
 centro_ancho - tamaño_cuadrado:centro_ancho + tamaño_cuadrado, 0]
```

## Modificar el Canal Verde

```
imagen_modificada[0:tamaño_cuadrado * 2, centro_ancho - tamaño_cuadrado:centro_ancho + tamaño_cuadrado, 1] = 0
```

## Graficar los Canales Modificados

```
fig, axs = plt.subplots(1, 4, figsize=(15, 5))

for i in range(4):
 axs[i].imshow(imagen_modificada[:, :, i], cmap='gray')
 axs[i].set_title(f'Canal {canales[i]} Modificado')
 axs[i].axis('off')

plt.show()
```

# Conclusión

NumPy es una herramienta poderosa para el manejo eficiente de datos numéricos en Python. Su integración con otras librerías como Matplotlib amplía sus capacidades para análisis y visualización de datos, convirtiéndolo en un recurso indispensable para científicos de datos y desarrolladores.

# Introducción a la Librería Pandas en Python

**Pandas** es una de las librerías más populares de Python para el análisis de datos. Ofrece estructuras de datos y funciones de alto nivel que facilitan la manipulación y análisis de grandes cantidades de información. Pandas facilita leer, modificar, analizar y almacenar conjuntos de datos de una manera eficiente.

Pandas está construido sobre **NumPy**, lo cual significa que hereda la eficiencia y las capacidades matemáticas que caracterizan a NumPy. La librería se organiza alrededor de dos estructuras principales:

1. **Series**: Son arreglos unidimensionales, similares a listas o arreglos de NumPy, pero con la capacidad adicional de contar con un índice que facilita el acceso y la manipulación de datos.
2. **DataFrames**: Son estructuras bidimensionales (similares a una hoja de cálculo), que contienen tanto filas como columnas, y permiten un manejo más complejo y estructurado de los datos.

En este apunte, comenzaremos con **Series**, aprendiendo qué son, cómo crearlas y cómo realizar operaciones básicas con ellas.

## ¿Qué es una Serie en Pandas?

Una **Serie** es una estructura de datos unidimensional que puede almacenar datos de cualquier tipo (números, texto, booleanos, etc.). Cada elemento en una Serie tiene un **índice**, que sirve para identificar de forma única cada elemento dentro de ella. Esto es útil cuando se trabaja con datos reales, como el precio de una acción en ciertos días o las temperaturas diarias de una ciudad.

## Creación de Series

Podemos crear una Serie a partir de diferentes tipos de datos, como listas, diccionarios o incluso arreglos de NumPy. Para ello, primero debemos importar la librería Pandas:

```
import pandas as pd
```

Veamos algunas formas de crear una Serie:

## Creación a partir de una lista

Podemos pasar una lista directamente a la función `pd.Series()`:

```
import pandas as pd

Crear una Serie a partir de una lista de números
datos = [10, 20, 30, 40, 50]
serie_numerica = pd.Series(datos)
print(serie_numerica)
```

**Salida:**

```
0 10
1 20
2 30
3 40
4 50
dtype: int64
```

En este caso, la Serie tiene un índice numérico por defecto que comienza en 0.

## Creación a partir de un diccionario

Podemos crear una Serie a partir de un diccionario, donde las claves se convierten en los **índices** y los valores se convierten en los elementos de la Serie:

```
Crear una Serie a partir de un diccionario
estudiantes = {'Juan': 85, 'Ana': 90, 'Pedro': 78}
serie_estudiantes = pd.Series(estudiantes)
print(serie_estudiantes)
```

**Salida:**

```
Juan 85
Ana 90
Pedro 78
dtype: int64
```

En este caso, los nombres de los estudiantes son los índices de la Serie.

## Índices en una Serie

Los **índices** permiten acceder a los elementos de una Serie de forma sencilla. Podemos definir nuestros propios índices al crear la Serie:

```
Crear una Serie con índices personalizados
ciudades = ['Buenos Aires', 'Córdoba', 'Rosario']
poblacion = [2890000, 1391000, 948000]
serie_ciudades = pd.Series(poblacion, index=ciudades)
print(serie_ciudades)
```

**Salida:**

```
Buenos Aires 2890000
Córdoba 1391000
Rosario 948000
dtype: int64
```

Aquí, cada ciudad es un índice que está asociado con su población.

## Acceso a elementos en una Serie

Podemos acceder a los elementos de una Serie utilizando su **índice** o su **posición**:

- **Por índice:**

```
Acceder al valor de Córdoba
print(serie_ciudades['Córdoba'])
```

**Salida:**

```
1391000
```

- **Por posición** (similar a una lista o arreglo):

```
Acceder al segundo elemento (posición 1)
print(serie_ciudades[1])
```

**Salida:**

```
1391000
```

También podemos acceder a un **subconjunto** de elementos utilizando una lista de índices:

```
Acceder a los valores de Buenos Aires y Rosario
print(serie_ciudades[['Buenos Aires', 'Rosario']])
```

**Salida:**

```
Buenos Aires 2890000
Rosario 948000
dtype: int64
```

## Slicing en Series

Podemos utilizar **slicing** para acceder a un rango de elementos en una Serie, ya sea por **posición** o por **índice**:

- **Slicing por posición:** Al igual que en una lista, podemos usar el operador de dos puntos `:` para acceder a un rango de posiciones.

```
Acceder a los elementos desde la posición 0 hasta la 1 (sin incluir la posición 2)
slice_posicion = serie_ciudades[0:2]
print(slice_posicion)
```

**Salida:**

```
Buenos Aires 2890000
Córdoba 1391000
dtype: int64
```

- **Slicing por índice:** Podemos también utilizar valores de índices personalizados para hacer slicing.

```
Acceder a los elementos desde 'Buenos Aires' hasta 'Córdoba'
slice_indices = serie_ciudades['Buenos Aires':'Córdoba']
print(slice_indices)
```

**Salida:**

```
Buenos Aires 2890000
Córdoba 1391000
dtype: int64
```

Es importante notar que, a diferencia del slicing por posición, el slicing por índice **incluye** el índice final especificado.

## Operaciones Básicas con Series

Las Series soportan operaciones similares a las de los arreglos de NumPy. Veamos algunas operaciones básicas:

### Operaciones aritméticas

Podemos realizar operaciones aritméticas directamente sobre las Series:

```
Incrementar la población en un 10%
poblacion_incrementada = serie_ciudades * 1.10
print(poblacion_incrementada)
```

**Salida:**

```
Buenos Aires 3179000.0
Córdoba 1530100.0
Rosario 1042800.0
dtype: float64
```

## Aplicación de funciones

Podemos aplicar funciones como `sum()`, `mean()`, etc. para obtener información útil sobre los datos:

```
Sumar todas las poblaciones
total_poblacion = serie_ciudades.sum()
print(f"Población total: {total_poblacion}")
```

**Salida:**

```
Población total: 5229000
```

Estas operaciones nos permiten realizar análisis de datos de manera eficiente y sencilla.

## Métodos Avanzados para Manipular Series

`Series.apply`

El método `apply()` permite aplicar una función sobre los valores de una Serie. Es muy útil cuando se necesita transformar o manipular los datos de una manera personalizada.

```
Aplicar una función lambda para aumentar cada valor en un 5%
aumento = serie_ciudades.apply(lambda x: x * 1.05)
print(aumento)
```

**Salida:**

```
Buenos Aires 3034500.0
Córdoba 1460550.0
Rosario 995400.0
dtype: float64
```

`Series.agg`

Los métodos `agg()` y `aggregate()` permiten realizar una o varias operaciones de agregación sobre una Serie. Podemos pasar una función o una lista de funciones.

```
Calcular la suma y la media de la Serie
resultado = serie_ciudades.agg(['sum', 'mean'])
print(resultado)
```

**Salida:**

```
sum 5229000.0
mean 1743000.0
dtype: float64
```

## Series.transform

El método `transform()` permite aplicar una función sobre la Serie y devolver una Serie del mismo tamaño. Es útil cuando se necesita transformar los datos, pero manteniendo la misma estructura.

```
Transformar cada valor a su logaritmo natural
import numpy as np
transformada = serie_ciudades.transform(np.log)
print(transformada)
```

**Salida:**

```
Buenos Aires 14.878101
Córdoba 14.146659
Rosario 13.764979
dtype: float64
```

## Series.map

El método `map()` permite aplicar una función o un diccionario de mapeo a los valores de una Serie. Es útil para transformar los valores según reglas definidas.

```
Mapear los valores a categorías
mapa_poblacion = {2890000: 'Alta', 1391000: 'Media', 948000: 'Baja'}
categorias = serie_ciudades.map(mapa_poblacion)
print(categorias)
```

**Salida:**

```
Buenos Aires Alta
Córdoba Media
Rosario Baja
dtype: object
```

## Series.groupby

El método `groupby()` permite agrupar los datos de una Serie según uno o más criterios y aplicar operaciones de agregación sobre cada grupo. Es muy poderoso cuando se trabaja con datos categóricos.

```
import pandas as pd

productos = df.Series([100, 200, 150, 300, 250, 400], index=['A', 'B', 'C', 'A', 'B', 'C'])
total_productos = productos.groupby(productos.index).sum()
print(total_productos)
```

### Salida:

```
A 400
B 450
C 550
dtype: int64
```

Incluso se puede realizar varios agrupamientos en una sola línea:

```
promedio_productos = productos.groupby(productos.index).agg(['sum', 'mean'])
print(promedio_productos)
```

### Salida:

```
 sum mean
A 400 200.0
B 450 225.0
C 550 275.0
```

o mas explicito:

```
promedio_productos = productos.groupby(productos.index).agg({
 'total': 'sum',
 'promedio': 'mean',
 'cantidad': 'count'})
print(promedio_productos)
```

**Salida:**

	total	promedio	cantidad
A	400	200.0	2
B	450	225.0	2
C	550	275.0	2

## Extracción de Elementos que Cumplen una Condición

Podemos extraer todos los elementos de una Serie que cumplan con una condición utilizando operaciones lógicas. Por ejemplo, si queremos extraer las ciudades con más de 1,000,000 de habitantes:

```
Extraer ciudades con más de 1,000,000 de habitantes
ciudades_mayores = serie_ciudades[serie_ciudades > 1000000]
print(ciudades_mayores)
```

**Salida:**

```
Buenos Aires 2890000
Córdoba 1391000
dtype: int64
```

Veamos paso a paso cómo se realiza esta operación:

`series_ciudades > 1000000`

Este código genera una Serie de valores booleanos, donde cada valor indica si la condición se cumple o no:

```
print(serie_ciudades > 1000000)
```

## Salida:

```
Buenos Aires True
Córdoba True
Rosario False
dtype: bool
```

`serie_ciudades[serie_ciudades > 1000000]`

Finalmente usamos la nueva serie generada para filtrar la serie original. Cuando una serie recibe una serie de valores booleanos, solo mantiene los valores que corresponden a `True`.

Se puede generar condiciones compuesta con los operadores `&` (and) y `|` (or).

```
Extraer ciudades con más de 1,000,000 de habitantes y menos de 2,000,000
ciudades_mayores = serie_ciudades[(serie_ciudades > 1000000) & (serie_ciudades < 2000000)
print(ciudades_mayores)
```

## Salida:

```
Córdoba 1391000
dtype: int64
```

## Resumen

- Una **Serie** es una estructura de datos unidimensional con un índice que facilita el acceso a los elementos.
- Podemos crear Series a partir de listas, diccionarios o arreglos de NumPy, y personalizar sus índices.
- Podemos acceder a los elementos de una Serie mediante su índice o su posición.
- Podemos utilizar **slicing** para acceder a un rango de elementos, ya sea por posición o por índice.
- Las Series soportan operaciones aritméticas y funciones estadísticas para análisis de datos.

# Que es un DataFrame en Pandas?

Un DataFrame es una estructura de datos bidimensional, es decir, los datos se alinean de manera tabular en filas y columnas. Los datos pueden ser cualquier tipo de datos (numéricos, de cadena, de fecha, etc.). Un DataFrame tiene etiquetas de fila y columna, como las etiquetas de eje en una hoja de cálculo de Excel. Podemos realizar operaciones aritméticas en filas y columnas y podemos aplicar funciones matemáticas y estadísticas en los datos.

## Crear un DataFrame

Para crear un DataFrame, primero necesitamos importar la biblioteca Pandas. Luego, podemos crear un DataFrame pasando un diccionario de listas a la función DataFrame de Pandas. Cada clave del diccionario se convierte en una columna y cada lista en los valores de la columna. Si las listas no son del mismo tamaño, Pandas rellenará los valores faltantes con NaN.

```
import pandas as pd
data = {'Nombre': ['Juan', 'Ana', 'Pedro', 'María'],
 'Edad': [25, 30, 35, 40],
 'Ciudad': ['Madrid', 'Barcelona', 'Sevilla', 'Valencia']}
df = pd.DataFrame(data)
print(df)
```

Salida:

	Nombre	Edad	Ciudad
0	Juan	25	Madrid
1	Ana	30	Barcelona
2	Pedro	35	Sevilla
3	María	40	Valencia

Creamos un DataFrame con tres columnas: Nombre, Edad y Ciudad. Las claves del diccionario se convierten en los nombres de las columnas y las listas en los valores de las columnas.

Las columnas son en realidad Series, si le pasamos una lista la convierte automáticamente en una Serie.

```
nombre = pd.Series(['Juan', 'Ana', 'Pedro', 'María'], index=[0, 1, 2, 3])
edad = pd.Series([25, 30, 35, 40], index=[3, 2, 1, 0])

df = pd.DataFrame({'Nombre': nombre, 'Edad': edad})
print(df)
```

**Salida:**

	Nombre	Edad
0	María	40
1	Pedro	35
2	Ana	30
3	Juan	25

Nótese que las etiquetas de índice de las Series se han combinado para formar el índice del DataFrame. Si las etiquetas de índice no coinciden, Pandas rellenará los valores faltantes con NaN.

## Seleccionar columnas

Podemos seleccionar una columna de un DataFrame pasando el nombre de la columna entre corchetes. Esto devolverá una Serie.

```
print(df['Nombre'])
```

**Salida:**

0	Maria
1	Pedro
2	Ana
3	Juan

Name: Nombre, dtype: object

En este caso devuelve una serie con los valores de la columna 'Nombre' y el mismo índice del DataFrame.

Podemos seleccionar varias columnas pasando una lista de nombres de columnas.

```
print(df[['Nombre', 'Edad']])
```

**Salida:**

	Nombre	Edad
0	María	40
1	Pedro	35
2	Ana	30
3	Juan	25

En este caso devuelve un DataFrame con las columnas 'Nombre' y 'Edad' y el mismo índice del DataFrame.

## Seleccionar filas

Podemos seleccionar filas de un DataFrame usando el método `loc[]`. Podemos seleccionar una fila pasando la etiqueta de índice de la fila.

```
print(df.loc[2])
```

**Salida:**

```
Nombre Ana
Edad 30
Name: 2, dtype: object
```

En este caso va a generar una nueva serie donde el índice es el nombre de la columna y el valor es el valor de la fila.

Podemos seleccionar varias filas pasando una lista de etiquetas de índice de fila.

```
print(df.loc[[1, 3]])
```

**Salida:**

```
Nombre Edad
1 Pedro 35
3 Juan 25
```

En este caso va a generar un nuevo DataFrame con las filas 1 y 3.

## Seleccionar filas y columnas

Podemos seleccionar filas y columnas de un DataFrame pasando la etiqueta de índice de fila y el nombre de la columna al método `loc[]`.

```
print(df.loc[2, 'Nombre'])
```

**Salida:**

```
Ana
```

En este caso va a devolver el valor de la fila 2 y la columna 'Nombre'.

Podemos seleccionar varias filas y columnas pasando listas de etiquetas de índice de fila y nombres de columnas.

```
print(df.loc[[1, 3], ['Nombre', 'Edad']])
```

**Salida:**

```
Nombre Edad
1 Pedro 35
3 Juan 25
```

En este caso va a devolver un nuevo DataFrame con las filas 1 y 3 y las columnas 'Nombre' y 'Edad'.

# Seleccionar filas y columnas por índice

Podemos seleccionar filas y columnas de un DataFrame por índice en lugar de etiquetas de índice usando el método `iloc[]`.

```
print(df.iloc[2, 0])
```

**Salida:**

```
Ana
```

En este caso va a devolver el valor de la fila 2 y la columna 0.

Los DataFrame tambien tienen un indice que se puede acceder con el atributo `index`.

```
print(df.index)
```

**Salida:**

```
Int64Index([0, 1, 2, 3], dtype='int64')
```

Por defecto, si no le pasamos ninguno, el índice es un rango de números enteros de 0 a n-1, donde n es el número de filas en el DataFrame.

Le podemos asignar un índice a un DataFrame al crearlo con la propiedad `indice =` o bien pasando una lista de etiquetas de índice al atributo `index`.

```
df.index = ['a', 'b', 'c', 'd']
print(df)
```

**Salida:**

```
Nombre Edad
a María 40
b Pedro 35
c Ana 30
d Juan 25
```

En este caso le asignamos un índice de letras a las filas del DataFrame.

También podemos convertir una columna en el índice del DataFrame con el método `set_index()`.

```
df.set_index('Nombre', inplace=True)
print(df)
```

**Salida:**

```
Edad
Nombre
María 40
Pedro 35
Ana 30
Juan 25
```

En este caso convertimos la columna 'Nombre' en el índice del DataFrame.

Si queremos volver a la situación anterior, podemos usar el método `reset_index()`.

```
df.reset_index(inplace=True)
print(df)
```

**Salida:**

```
Nombre Edad
0 María 40
1 Pedro 35
2 Ana 30
3 Juan 25
```

En este caso volvemos a tener un índice numérico.

En general las operaciones sobre los dataframes generan una copia del dataframe original, si queremos modificar el dataframe original debemos usar el parámetro `inplace=True`.

## Filtrar datos

Podemos filtrar datos en un DataFrame usando condiciones booleanas. Por ejemplo, podemos seleccionar filas donde la edad sea mayor que 30.

```
print(df[df['Edad'] > 30])
```

**Salida:**

	Nombre	Edad
0	Maria	40
1	Pedro	35

Esto funciona porque `df['Edad'] > 30` devuelve una Serie de valores booleanos que se pueden usar para seleccionar filas en el DataFrame.

Como la serie tiene el mismo índice que el DataFrame, Pandas selecciona las filas donde el valor de la serie es `True`.

Al igual que con las series podemos combinar condiciones booleanas usando los operadores `&` (y), `|` (o) y `~` (no).

```
print(df[(df['Edad'] > 30) & (df['Edad'] < 40)])
```

**Salida:**

	Nombre	Edad
1	Pedro	35

En este caso selecciona las filas donde la edad es mayor que 30 y menor que 40.

# Modificar datos

Podemos modificar los datos en un DataFrame asignando nuevos valores a las celdas. Por ejemplo, podemos cambiar la edad de Juan a 50.

```
df.loc[3, 'Edad'] = 50
print(df)
```

**Salida:**

	Nombre	Edad
0	Maria	40
1	Pedro	35
2	Ana	30
3	Juan	50

En este caso cambia el valor de la fila 3 y la columna 'Edad' a 50.

# Eliminar datos

Podemos eliminar filas de un DataFrame usando el método `drop()`. Por ejemplo, podemos eliminar la fila 2.

```
df.drop(2, inplace=True)
print(df)
```

**Salida:**

	Nombre	Edad
0	Maria	40
1	Pedro	35
3	Juan	50

En este caso elimina la fila 2 del DataFrame.

También podemos eliminar columnas pasando el nombre de la columna al método `drop()` y el parámetro `axis=1`.

```
df.drop('Edad', axis=1, inplace=True)
print(df)
```

**Salida:**

```
Nombre
0 María
1 Pedro
2 Ana
3 Juan
```

`axis=1` indica que queremos eliminar una columna, si usamos `axis=0` eliminamos una fila.

## Agregar datos

Podemos agregar filas a un DataFrame usando el método `append()`. Por ejemplo, podemos agregar una nueva fila con el nombre 'Luis' y la edad 45.

```
df = df.append({'Nombre': 'Luis', 'Edad': 45}, ignore_index=True)
print(df)
```

**Salida:**

```
Nombre Edad
0 María 40
1 Pedro 35
2 Ana 30
3 Juan 50
4 Luis 45
```

En este caso agrega una nueva fila al DataFrame.

También podemos agregar columnas a un DataFrame asignando una Serie a una nueva columna.

```
df['Ciudad'] = ['Madrid', 'Barcelona', 'Sevilla', 'Valencia', 'Bilbao']

print(df)
```

**Salida:**

	Nombre	Edad	Ciudad
0	María	40	Madrid
1	Pedro	35	Barcelona
2	Ana	30	Sevilla
3	Juan	50	Valencia
4	Luis	45	Bilbao

En este caso agrega una nueva columna al DataFrame.

Las Series son en realidad array de NumPy, por lo que podemos realizar operaciones aritméticas en ellas.

```
df['Edad'] = df['Edad'] + 5

print(df)
```

**Salida:**

	Nombre	Edad	Ciudad
0	María	45	Madrid
1	Pedro	40	Barcelona
2	Ana	35	Sevilla
3	Juan	55	Valencia
4	Luis	50	Bilbao

En este caso suma 5 a la columna 'Edad'.

En consecuencia también podemos realizar operaciones aritméticas en columnas.

```
print(df['Edad'].mean())
```

**Salida:**

45.0

En este caso calcula la media de la columna 'Edad'.

También tenemos un método `describe()` que nos da un resumen estadístico de las columnas numéricas.

```
print(df.describe())
```

**Salida:**

	Edad
count	5.000000
mean	45.000000
std	8.366600
min	35.000000
25%	40.000000
50%	45.000000
75%	50.000000
max	55.000000

```
print(type(df.describe()))
```

**Salida:**

```
<class 'pandas.core.frame.DataFrame'>
```

En este caso devuelve un DataFrame con el resumen estadístico de las columnas numéricas.

```
print(df['Ciudad'].value_counts())
```

**Salida:**

```
Valencia 1
Sevilla 1
Madrid 1
Barcelona 1
Bilbao 1
Name: Ciudad, dtype: int64
```

En este caso devuelve una serie con el número de veces que se repite cada valor en la columna 'Ciudad'.

## Ordenar datos

Podemos ordenar un DataFrame por una columna usando el método `sort_values()`. Por ejemplo, podemos ordenar el DataFrame por la columna 'Edad'.

```
print(df.sort_values('Edad'))
```

**Salida:**

	Nombre	Edad	Ciudad
2	Ana	35	Sevilla
1	Pedro	40	Barcelona
0	Maria	45	Madrid
4	Luis	50	Bilbao
3	Juan	55	Valencia

En este caso ordena el DataFrame por la columna 'Edad'.

Podemos ordenar en orden descendente pasando el parámetro `ascending=False` y podemos ordenar por varias columnas pasando una lista de nombres de columnas.

```
print(df.sort_values(['Ciudad', 'Edad'], ascending=[True, False]))
```

**Salida:**

	Nombre	Edad	Ciudad
4	Luis	50	Bilbao
1	Pedro	40	Barcelona
0	Maria	45	Madrid
2	Ana	35	Sevilla
3	Juan	55	Valencia

En este caso ordena el DataFrame por la columna 'Ciudad' en orden ascendente y por la columna 'Edad' en orden descendente.

## Agrupar datos

```
datos_productos = {
 'Producto': ['A', 'B', 'A', 'C', 'B', 'A'],
 'Precio': [100, 150, 120, 200, 150, 100],
 'Cantidad': [1, 2, 3, 1, 2, 1]
}
productos = pd.DataFrame(datos_productos)

print(productos)
```

**Salida:**

	Producto	Precio	Cantidad
0	A	100	1
1	B	150	2
2	A	120	3
3	C	200	1
4	B	150	2
5	A	100	1

Agrupar por producto y sumarizar la cantidad

```
grupo_productos = productos.groupby('Producto')['Cantidad'].sum()
print(grupo_productos)
```

**Salida:**

```

Producto
A 5
B 4
C 1
Name: Cantidad, dtype: int64

```

Agrupar por producto, sumar los precios y las cantidades, y calcular el precio promedio

```

gp = productos.groupby('Producto').agg({
 'Precio' : 'sum',
 'Cantidad': 'sum'
})
gp['Precio Promedio'] = gp['Precio'] / gp['Cantidad']
print(gp)

```

**Salida:**

Producto	Precio	Cantidad	Precio Promedio
A	220	5	60.0
B	300	4	75.0
C	200	1	200.0

Agrupar por producto y calcular el precio total, la cantidad total y el precio promedio

```

```python
gp = productos.groupby('Producto').agg(
    Precio_Total=('Precio', 'sum'),
    Cantidad_Total=('Cantidad', 'sum'),
    Precio_Promedio=('Precio',
                     lambda x: x.sum() / productos.loc[x.index, 'Cantidad'].count())
)

print(gp)

```

Salida:

Producto	Precio_Total	Cantidad_Total	Precio_Promedio
A	220	2	110.0
B	300	2	150.0
C	200	1	200.0

Esta es la operación más compleja que podemos realizar con un grupo así que analicemos el código paso a paso:

Voy a dividir el código en varias partes para que sea más fácil de entender:

productos.groupby('Producto')

La primera parte de este código es `productos.groupby('Producto')`.

- `productos` es tu *DataFrame*, es decir, la tabla con los datos.
- `groupby('Producto')` agrupa todas las filas del *DataFrame* por los valores de la columna `'Producto'`. En otras palabras, toma todos los datos del mismo tipo de producto y los agrupa juntos.

.agg(...)

Después de agrupar los datos, usamos `.agg(...)` para aplicar funciones de agregación sobre cada grupo.

Una función de agregación toma un conjunto de valores y los combina para obtener un solo resultado. Por ejemplo, sumar valores, calcular un promedio, o contar cuántos elementos hay.

En este caso, estamos usando `.agg()` para crear tres nuevas columnas: `'Precio_Total'`, `'Cantidad_Total'`, y `'Precio_Promedio'`.

Veamos cada una de ellas:

Precio_Total=('Precio', 'sum')

- Aquí estás diciendo que quieres calcular una nueva columna llamada `'Precio_Total'`.
- La información que quieres sumar proviene de la columna `'Precio'` del *DataFrame*.
- La función que aplicas es `'sum'`, que suma todos los precios de cada grupo.

Por ejemplo, si tienes dos filas con "A" que tienen precios de 100 y 120, entonces el `'Precio_Total'` sería 220.

```
Cantidad_Total=( 'Cantidad', 'sum' )
```

- De manera similar, quieres una nueva columna llamada '`Cantidad_Total`' que sume todos los valores de la columna '`Cantidad`' de cada grupo.

```
Precio_Promedio=( 'Precio', lambda x: x.sum() / dp.iloc[x.index][ 'Cantidad' ].count() )
```

Esta es la parte más complicada. Vamos a desglosarla con detalle:

- Aquí estás creando una columna nueva llamada '`Precio_Promedio`'.
- La agregación se aplica sobre la columna '`Precio`'.
- La función que aplicas aquí es una **función lambda**, que es básicamente una función rápida y sencilla que no tiene nombre. En este caso, la función lambda se ve así:

```
lambda x: x.sum() / dp.iloc[x.index][ 'Cantidad' ].count()
```

¿Qué está haciendo esta función lambda?

- `x` representa una serie de precios dentro de un grupo específico (por ejemplo, los precios de todos los productos).
- `x.sum()` suma todos los precios dentro de ese grupo.
- `dp.iloc[x.index]['Cantidad']` usa `.iloc[]` para seleccionar las filas del *DataFrame* original (`dp`) que corresponden a los índices de los elementos agrupados en `x`.
 - En otras palabras, selecciona las filas de '`Cantidad`' correspondientes al grupo actual de precios.
- `['Cantidad'].count()` toma esas cantidades y las cuenta.

En resumen, lo que está haciendo la función es calcular un **precio promedio**, dividiendo la suma de los precios de un grupo entre la suma de las cantidades de ese grupo.

```
print(grupo_productos)
```

Finalmente, imprimimos el resultado con `print(grupo_productos)`.

- `grupo_productos` es un nuevo *DataFrame* que contiene los resultados de la agrupación y las agregaciones.
- Mostrará una tabla donde cada fila representa un producto y las columnas `'Precio_Total'`, `'Cantidad_Total'`, y `'Precio_Promedio'` mostrarán los valores calculados para cada uno.

Combinar DataFrames

Podemos combinar dos *DataFrames* en uno solo usando el método `merge()`. Por ejemplo, podemos combinar dos *DataFrames* con información de productos y precios.

```
datos_productos = {
    'Producto': ['A', 'B', 'C'],
    'Precio': [100, 150, 200]
}
productos = pd.DataFrame(datos_productos)

datos_precios = {
    'Producto': ['A', 'B', 'D'],
    'Precio': [100, 150, 250]
}

precios = pd.DataFrame(datos_precios)

print(pd.merge(productos, precios, on='Producto', how='inner'))
```

Salida:

	Producto	Precio_x	Precio_y
0	A	100	100
1	B	150	150

En este caso combina los dos *DataFrames* en uno solo usando la columna 'Producto' como clave. El parámetro `how='inner'` indica que solo queremos las filas que tienen un valor común en ambas tablas.

Podemos usar otros valores para el parámetro `how`:

- `how='left'`: Devuelve todas las filas del *DataFrame* de la izquierda y las filas coincidentes del *DataFrame* de la derecha.

- `how='right'`: Devuelve todas las filas del DataFrame de la derecha y las filas coincidentes del DataFrame de la izquierda.
- `how='outer'`: Devuelve todas las filas de ambos DataFrames.
- `how='inner'`: Devuelve solo las filas que tienen un valor común en ambos DataFrames.

Guardar y cargar DataFrames

Podemos guardar un DataFrame en un archivo CSV usando el método `to_csv()`. Por ejemplo, podemos guardar el DataFrame de productos en un archivo llamado 'productos.csv'.

```
productos.to_csv('productos.csv', index=False)
```

Podemos cargar un DataFrame desde un archivo CSV usando el método `read_csv()`. Por ejemplo, podemos cargar el archivo 'productos.csv' en un DataFrame.

```
productos = pd.read_csv('productos.csv')

print(productos)
```

Salida:

	Producto	Precio
0	A	100
1	B	150
2	C	200

En este caso carga el archivo 'productos.csv' en un DataFrame.

Podemos guardar un DataFrame en un archivo Excel usando el método `to_excel()`. Por ejemplo, podemos guardar el DataFrame de productos en un archivo llamado 'productos.xlsx'.

```
productos.to_excel('productos.xlsx', index=False)
```

Podemos cargar un DataFrame desde un archivo Excel usando el método `read_excel()`. Por ejemplo, podemos cargar el archivo 'productos.xlsx' en un DataFrame.

```
productos = pd.read_excel('productos.xlsx')
print(productos)
```

O desde un archivo con formato JSON

```
productos.to_json('productos.json', orient='records')
productos = pd.read_json('productos.json', orient='records')
```

Incluso con otros formatos como tamaño fijos

```
productos.to_fwf('productos.fwf', index=False, widths=[10, 5])
productos = pd.read_fwf('productos.fwf', index=False, widths=[10, 5], names=['Produc
```

Manejo de Datos Nulos

Cuando trabajamos con conjuntos de datos reales, es común encontrarnos con valores faltantes o nulos (NaN). Pandas proporciona varias herramientas para manejar estos valores de manera eficiente.

Identificar valores nulos

Podemos identificar valores nulos en un DataFrame utilizando los métodos `isna()` o `isnull()`:

```
data = {'Nombre': ['Juan', 'Ana', 'Pedro', None],
        'Edad': [25, 30, np.nan, 40],
        'Ciudad': ['Madrid', None, 'Sevilla', 'Valencia']}
df = pd.DataFrame(data)
print(df.isna())
```

Salida:

```
Nombre  Edad  Ciudad
0   False  False  False
1   False  False   True
2   False   True  False
3    True  False  False
```

Eliminar valores nulos

Podemos eliminar filas o columnas que contengan valores nulos utilizando el método `dropna()`:

```
# Eliminar filas con valores nulos
df_sin_nulos = df.dropna()
print(df_sin_nulos)
```

Salida:

```
Nombre  Edad      Ciudad
0    Juan  25.0     Madrid
```

También podemos eliminar columnas que contengan valores nulos:

```
# Eliminar columnas con valores nulos
df_sin_nulos_col = df.dropna(axis=1)
print(df_sin_nulos_col)
```

Salida:

```
Edad
0  25.0
1  30.0
2   NaN
3  40.0
```

Rellenar valores nulos

Otra opción es llenar los valores nulos con un valor predeterminado utilizando el método

`fillna()`:

```
# Rellenar valores nulos con un valor
df_rellenado = df.fillna({'Nombre': 'Desconocido', 'Edad': 0, 'Ciudad': 'Desconocida'})
print(df_rellenado)
```

Salida:

	Nombre	Edad	Ciudad
0	Juan	25.0	Madrid
1	Ana	30.0	Desconocida
2	Pedro	0.0	Sevilla
3	Desconocido	40.0	Valencia

También se puede llenar valores nulos con el valor anterior o siguiente:

```
# Rellenar valores nulos con el valor anterior
df_rellenado_ffill = df.fillna(method='ffill')
print(df_rellenado_ffill)
```

Salida:

	Nombre	Edad	Ciudad
0	Juan	25.0	Madrid
1	Ana	30.0	Madrid
2	Pedro	30.0	Sevilla
3	Pedro	40.0	Valencia

Operaciones con Fecha y Hora

Pandas tiene un excelente soporte para trabajar con datos de tipo fecha y hora, lo que es crucial cuando se trabaja con datos temporales. La manipulación de fechas puede realizarse de manera eficiente utilizando la funcionalidad de `pd.to_datetime()` y el uso de índices de tiempo.

Conversión a formato de fecha

Podemos convertir una columna de strings a un objeto `datetime` usando `pd.to_datetime()`:

```
# Crear un DataFrame con fechas en formato de string
fechas = pd.DataFrame({'Fecha': ['2023-01-01', '2023-02-01', '2023-03-01']})
fechas['Fecha'] = pd.to_datetime(fechas['Fecha'])
print(fechas)
```

Salida:

```
    Fecha
0 2023-01-01
1 2023-02-01
2 2023-03-01
```

Filtrado por rango de fechas

Podemos filtrar un DataFrame por un rango de fechas fácilmente:

```
# Crear un rango de fechas
fechas = pd.date_range(start='2023-01-01', end='2023-03-01', freq='D')
df_fechas = pd.DataFrame({'Fecha': fechas})
# Filtrar por un rango de fechas
df_filtrado = df_fechas[(df_fechas['Fecha'] >= '2023-02-01') & (df_fechas['Fecha'] <= '2023-02-15')]
print(df_filtrado)
```

Salida:

```
    Fecha
31 2023-02-01
32 2023-02-02
33 2023-02-03
...
44 2023-02-15
```

Extraer componentes de una fecha

Es posible extraer componentes específicos como el día, mes o año de una columna de tipo `datetime`:

```
# Extraer año, mes y día de la columna Fecha
df_fechas['Año'] = df_fechas['Fecha'].dt.year
df_fechas['Mes'] = df_fechas['Fecha'].dt.month
df_fechas['Día'] = df_fechas['Fecha'].dt.day
print(df_fechas.head())
```

Salida:

	Fecha	Año	Mes	Día
0	2023-01-01	2023	1	1
1	2023-01-02	2023	1	2
2	2023-01-03	2023	1	3
3	2023-01-04	2023	1	4
4	2023-01-05	2023	1	5

Operaciones aritméticas con fechas

Podemos realizar operaciones aritméticas con fechas, como calcular la diferencia entre dos fechas:

```
# Calcular la diferencia de días entre dos fechas
df_fechas['Diferencia_días'] = df_fechas['Fecha'] - pd.to_datetime('2023-01-01')
print(df_fechas.head())
```

Salida:

	Fecha	Año	Mes	Día	Diferencia_días
0	2023-01-01	2023	1	1	0 days
1	2023-01-02	2023	1	2	1 days
2	2023-01-03	2023	1	3	2 days
3	2023-01-04	2023	1	4	3 days
4	2023-01-05	2023	1	5	4 days

Las operaciones con fechas y horas son esenciales para cualquier análisis temporal, y pandas facilita el manejo de este tipo de datos.

Conclusiones

En este artículo hemos visto cómo crear, seleccionar, filtrar, modificar, eliminar, agregar, ordenar, agrupar, combinar, guardar y cargar DataFrames en Pandas. Los DataFrames son una estructura de

datos muy útil para trabajar con datos tabulares en Python. Con Pandas, podemos realizar operaciones aritméticas, aplicar funciones matemáticas y estadísticas, y realizar operaciones de manipulación de datos en los DataFrames. Pandas proporciona una amplia gama de funciones y métodos para trabajar con DataFrames, lo que facilita la manipulación y el análisis de datos en Python.