Curso de Python Avanzado

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Bienvenida

Bienvenidos al curso de **Python Avanzado** para Scotiabank, brindado por Belatrix.

En este curso aprenderás a desarrollar una programación avanzada con Python.

El curso está dirigido a profesionales relacionados al área de desarrollo, soporte y análisis de datos.

Se requieren conocimientos previos de Python Intermedio para poder cubrir satisfactoriamente este curso.

Temario

Módulo 1: Programación Funcional en Python

1. Introducción a la programación funcional
2. Funciones de orden superior
3. Lambdas y expresiones generadoras

Módulo 2: Diseño de Patrones Avanzados

1. Patrones de diseño comunes en Python
2. Aplicación práctica de patrones en el desarrollo de software

Módulo 3: Manipulación Eficiente de Datos

1. Uso avanzado de NumPy y Pandas
2. Operaciones vectorizadas y broadcasting

Módulo 4: Concurrencia y Paralelismo en Python

1. Hilos y procesos en Python
2. Multiprocessing y Asyncio

Módulo 5: Optimización de Código

1. Estrategias para mejorar el rendimiento
2. Perfilado de código y herramientas de optimización

Módulo 6: Seguridad en Desarrollo Python

1. Principios básicos de seguridad
2. Mejores prácticas de codificación segura

Módulo 7: Despliegue y Escalabilidad

1. Configuración de entornos de producción
2. Estrategias para escalabilidad horizontal y vertical

Módulo 3: Manipulación Eficiente de Datos

1. Uso avanzado de NumPy y Pandas
2. Operaciones vectorizadas y broadcasting

301. Introudcción a Numpy y su uso avazado

NumPy es una biblioteca para el lenguaje de programación Python, que le agrega soporte para grandes arrays y matrices, junto con una colección de funciones matemáticas de alto nivel para operar con estas estructuras de datos. Es una pieza fundamental en el ecosistema de ciencia de datos en Python, debido a su eficiencia y facilidad de uso. Aquí te dejo una introducción a sus características y capacidades principales:

Características Principales de NumPy

1. Arrays de N-dimensiones: El núcleo de NumPy es el objeto `ndarray`, que representa una colección de elementos de tipos similares. Los arrays pueden tener varias dimensiones, permitiendo almacenar datos en 1D (vectores), 2D (matrices), o cualquier número de dimensiones

(tensores).

2. Tipos de Datos: NumPy soporta una variedad de tipos de datos numéricos, más allá de los tipos básicos de Python. Esto permite un manejo más eficiente de la memoria y un control detallado sobre cómo se deben tratar los datos numéricos.

3. Vectorización: Una de las mayores ventajas de NumPy es su capacidad para vectorizar operaciones, lo que significa que las operaciones matemáticas y lógicas se aplican elemento por elemento, permitiendo un cálculo eficiente sin la necesidad de bucles explícitos en Python.

4. Broadcasting: El broadcasting es un mecanismo que permite a NumPy trabajar con arrays de diferentes formas durante operaciones aritméticas, ajustando automáticamente los tamaños para que coincidan.

5. Indexación y Slicing Avanzados: NumPy ofrece opciones avanzadas para seleccionar y manipular partes de los arrays, incluida la indexación booleana, la indexación basada en slices, y la indexación por arrays de enteros.

6. Herramientas Matemáticas y Estadísticas: Incluye funciones para realizar operaciones estadísticas básicas (como media, mediana, desviación estándar), matemáticas (suma, producto, transformadas), y lineales (inversión de matrices, descomposición, eigenvalores) directamente sobre los arrays.

7. Integración con otros Paquetes: Dado su uso generalizado, muchas otras bibliotecas científicas y de análisis de datos en Python están construidas sobre NumPy, lo que facilita la integración y manipulación de datos entre diferentes herramientas.

Cómo Empezar con NumPy

Para comenzar a utilizar NumPy, primero debes instalarlo, usualmente mediante `pip`:

```
pip install numpy
```

Luego, puedes importar NumPy en tu script o entorno de notebook:

```
import numpy as np
```

Crear un array en NumPy es sencillo. Por ejemplo, para crear un array de una dimensión, puedes hacer lo siguiente:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5])
print(mi_array)
```

Este es solo un punto de partida. NumPy es una herramienta poderosa con muchas más funcionalidades y profundidad de lo que se puede cubrir en una breve introducción. Te animo a explorar la documentación oficial y tutoriales en línea para aprender más sobre cómo utilizar NumPy en tus proyectos de análisis de datos y ciencia de datos.

```
import numpy as np

x = np.array([1, 2, 3, 4, 5, 100])

x
```

```
array([  1,   2,   3,   4,   5, 100])
```

```
x.ndim
```

```
1
```

```
x.shape
```

```
(6,)
```

```
s1 = np.random.normal(30, 5, 10)

s1
```

```
array([26.24930582, 24.34025184, 24.58162844, 33.04994911, 22.81332149,
       39.33222856, 39.05007467, 36.10575628, 25.48219891, 32.14285241])
```

```
x1 = np.array([1, 4, 3, 5, 6])
x2 = np.array([300, 350, 600, 78, 200])

y1 = x2 / x1 # element-wise (operación elemento por elemento)

print(x1)
print(x2)
print(y1)
```

```
[1 4 3 5 6]
[300 350 600  78 200]
[300.          87.5        200.          15.6         33.33333333]
```

Arreglos N-Dimensionales

A continuación, se muestran distintos ejemplos que van desde un arreglo unidimensional hasta un arreglo tridimensional, cada uno con una breve descripción de su posible uso:

1. Arreglo Unidimensional (1D)

Los arreglos unidimensionales son similares a las listas en Python, pero con una funcionalidad mucho más amplia en términos de operaciones numéricas.

```
import numpy as np

# Crear un arreglo 1D de números enteros
arreglo_1d = np.array([1, 2, 3, 4, 5])
print("Arreglo 1D:", arreglo_1d)
```

Uso: Este tipo de arreglo se puede utilizar para almacenar y operar sobre cualquier secuencia de números, como puntuaciones en un juego, temperaturas a lo largo de una semana, etc.

2. Arreglo Bidimensional (2D)

Los arreglos bidimensionales se pueden pensar como matrices, que son listas de listas con la misma longitud.

```
# Crear un arreglo 2D (por ejemplo, una matriz de 3x3)
arreglo_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Arreglo 2D:\n", arreglo_2d)
```

Uso: Este formato es ideal para datos tabulares, como información en una hoja de cálculo, donde cada fila podría representar un ítem diferente (por ejemplo, un estudiante) y cada columna un atributo de ese ítem (por ejemplo, su calificación en diferentes asignaturas).

3. Arreglo Tridimensional (3D)

Los arreglos tridimensionales se pueden visualizar como una lista de matrices, es decir, un "cubo" de números.

```
# Crear un arreglo 3D (por ejemplo, dos matrices de 3x3, pensándolo como un cubo de datos)
arreglo_3d = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                       [[10, 11, 12], [13, 14, 15], [16, 17, 18]]])
print("Arreglo 3D:\n", arreglo_3d)
```

Uso: Los arreglos 3D son útiles en aplicaciones más complejas, como el procesamiento de imágenes (donde una dimensión puede ser el ancho, otra el alto, y otra los canales de color), o en datos de series temporales donde se registra múltiple información en diferentes momentos (por ejemplo, datos meteorológicos horarios a lo largo de varios días).

Cada uno de estos ejemplos muestra cómo se pueden estructurar los datos de diferentes maneras para adaptarse a las necesidades de distintos problemas y análisis. NumPy ofrece las herramientas para manipular eficientemente estos arreglos, permitiéndote realizar desde operaciones matemáticas básicas hasta complejas transformaciones y cálculos estadísticos.

```
arreglo_1d = np.array([1, 2, 3, 4, 5])
print("Arreglo 1D:", arreglo_1d)
```

```
Arreglo 1D: [1 2 3 4 5]
```

```
arreglo_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Arreglo 2D:\n", arreglo_2d)
```

```
Arreglo 2D:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
arreglo_2d[0]
```

```
array([1, 2, 3])
```

```
arreglo_2d[2]
```

```
array([7, 8, 9])
```

```
arreglo_2d[:, 0] # Pedimos todas las filas, en la columna 0 (primera)
```

```
array([1, 4, 7])
```

```
ventas = np.array([
    [800, 900, 700, 600, 500],
    [850, 800, 780, 900, 300],
    [700, 850, 320, 2200, 800],
    [800, 950, 220, 3200, 800],
    [600, 750, 420, 4200, 900],
    [900, 650, 520, 200, 1000],
    [200, 1050, 1120, 200, 600],
])

ventas
```

```
array([[ 800,  900,  700,  600,  500],
       [ 850,  800,  780,  900,  300],
       [ 700,  850,  320, 2200,  800],
       [ 800,  950,  220, 3200,  800],
       [ 600,  750,  420, 4200,  900],
       [ 900,  650,  520,  200, 1000],
       [ 200, 1050, 1120,  200,  600]])
```

```
ventas.T
```

```
array([[ 800,  850,  700,  800,  600,  900,  200],
       [ 900,  800,  850,  950,  750,  650, 1050],
       [ 700,  780,  320,  220,  420,  520, 1120],
       [ 600,  900, 2200, 3200, 4200,  200,  200],
       [ 500,  300,  800,  800,  900, 1000,  600]])
```

```
ventas[:, 2]
```

```
array([ 700,  780,  320,  220,  420,  520, 1120])
```

```
np.sum(ventas[:, 2])
```

```
4080
```

```
np.mean(ventas[:, 2])
```

```
582.8571428571429
```

```
np.min(ventas[:, 2])
```

```
220
```

```
np.max(ventas[:, 2])
```

```
1120
```

```
for index in [0, 1, 2, 3, 4]:
  print(f"Vendedor ({index})")
  print("-" * 35)
  suma_ventas = np.sum(ventas[:, index])
  print("Suma Ventas:             ${:8.2f}".format(suma_ventas))
  prom_ventas = np.mean(ventas[:, index])
  print("Promedio Ventas:         ${:8.2f}".format(prom_ventas))
  min_ventas = np.min(ventas[:, index])
  print("Mínima Venta:            ${:8.2f}".format(min_ventas))
  max_ventas = np.max(ventas[:, index])
  print("Máxima Venta:            ${:8.2f}".format(max_ventas))
  ds_ventas = np.std(ventas[:, index])
  print("Desviación de Ventas:    ${:8.2f}".format(ds_ventas))
  print("-" * 35)
  print("Intervalo bajo 66%:      ${:8.2f}".format(prom_ventas - ds_ventas))
  print("Intervalo alto 66%:      ${:8.2f}".format(prom_ventas + ds_ventas))
  print("-" * 35)
  print("Intervalo bajo 95%:      ${:8.2f}".format(prom_ventas - 2 * ds_ventas))
  print("Intervalo alto 95%:      ${:8.2f}".format(prom_ventas + 2 * ds_ventas))
  print("-" * 35)
  rango_ventas = max_ventas - min_ventas
  print("Rango de Ventas:         ${:8.2f}".format(rango_ventas))
  print()
```

```
Vendedor (0)
-----------------------------------
Suma Ventas:             $ 4850.00
Promedio Ventas:         $  692.86
Mínima Venta:            $  200.00
Máxima Venta:            $  900.00
Desviación de Ventas:    $  221.08
-----------------------------------
Intervalo bajo 66%:      $  471.77
Intervalo alto 66%:      $  913.94
-----------------------------------
Intervalo bajo 95%:      $  250.69
Intervalo alto 95%:      $ 1135.02
-----------------------------------
Rango de Ventas:         $  700.00

Vendedor (1)
-----------------------------------
```

```
Suma Ventas:             $ 5950.00
Promedio Ventas:         $  850.00
Mínima Venta:            $  650.00
Máxima Venta:            $ 1050.00
Desviación de Ventas:    $  122.47
-----------------------------------
Intervalo bajo 66%:      $  727.53
Intervalo alto 66%:      $  972.47
-----------------------------------
Intervalo bajo 95%:      $  605.05
Intervalo alto 95%:      $ 1094.95
-----------------------------------
Rango de Ventas:         $  400.00

Vendedor (2)
-----------------------------------
Suma Ventas:             $ 4080.00
Promedio Ventas:         $  582.86
Mínima Venta:            $  220.00
Máxima Venta:            $ 1120.00
Desviación de Ventas:    $  285.94
-----------------------------------
Intervalo bajo 66%:      $  296.91
Intervalo alto 66%:      $  868.80
-----------------------------------
Intervalo bajo 95%:      $   10.97
Intervalo alto 95%:      $ 1154.74
-----------------------------------
Rango de Ventas:         $  900.00

Vendedor (3)
-----------------------------------
Suma Ventas:             $11500.00
Promedio Ventas:         $ 1642.86
Mínima Venta:            $  200.00
Máxima Venta:            $ 4200.00
Desviación de Ventas:    $ 1467.61
-----------------------------------
Intervalo bajo 66%:      $  175.25
Intervalo alto 66%:      $ 3110.47
```

Transposición de Arreglos N-Dimensionales

Transponer un arreglo n-dimensional en NumPy es un proceso sencillo gracias a la propiedad `.T` de los objetos `ndarray`, o utilizando la función `np.transpose()`. La transposición es un cambio de las dimensiones del arreglo, donde la forma del arreglo se invierte. En el caso de una matriz 2D (un arreglo de dos dimensiones), transponer implica cambiar filas por columnas y viceversa. Para arreglos de más de dos dimensiones, puedes especificar un orden para las dimensiones al usar `np.transpose()`.

Aquí te dejo algunos ejemplos de cómo transponer arreglos de diferentes dimensiones:

Transponiendo un arreglo 2D

Para un arreglo bidimensional, la transposición es directamente el intercambio de sus filas y columnas.

```
import numpy as np

# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("Original:\n", arreglo_2d)

# Transponer el arreglo
transpuesto_2d = arreglo_2d.T
print("Transpuesto:\n", transpuesto_2d)
```

Transponiendo un arreglo 3D

Para arreglos de tres o más dimensiones, transponer puede implicar un cambio más complejo, dependiendo de cómo quieras reordenar las dimensiones.

```
# Crear un arreglo 3D
arreglo_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print("Original:\n", arreglo_3d)

# Transponer el arreglo 3D, cambiando las dimensiones
# Por ejemplo, intercambiar la primera y la última dimensión
transpuesto_3d = np.transpose(arreglo_3d, (2, 1, 0))
print("Transpuesto:\n", transpuesto_3d)
```

En el ejemplo 3D, `np.transpose(arreglo_3d, (2, 1, 0))` reordena las dimensiones del arreglo; la dimensión 0 (profundidad) se mueve a la posición de la dimensión 2, la dimensión 1 (filas) se queda igual, y la dimensión 2 (columnas) se mueve a la posición de la dimensión 0.

Estas operaciones demuestran cómo puedes reorganizar los datos dentro de tus arreglos NumPy, lo cual es especialmente útil en matemáticas de matrices, procesamiento de señales, y muchas otras áreas donde la orientación de los datos es importante.

Redimensión de Arreglos N-Dimensionales

Las funciones `reshape` y `ravel` de NumPy son herramientas poderosas para cambiar la forma y dimensiones de los arreglos n-dimensionales, permitiendo una manipulación eficiente de los datos para diversas necesidades de análisis.

Usando `reshape`

La función `reshape` permite cambiar la forma de un arreglo sin cambiar sus datos. Esto es útil cuando necesitas reorganizar tus datos para que se ajusten a una determinada operación o visualización. Sin embargo, para que `reshape` funcione, el nuevo shape debe ser compatible con el tamaño original del arreglo, es decir, el número total de elementos debe permanecer constante.

```
import numpy as np

# Crear un arreglo 1D con 12 elementos
arreglo_1d = np.arange(12)
print("Original 1D:\n", arreglo_1d)

# Redimensionar a un arreglo 2D 3x4
arreglo_2d = arreglo_1d.reshape((3, 4))
print("Redimensionado a 2D (3x4):\n", arreglo_2d)

# Redimensionar a un arreglo 3D 2x2x3
arreglo_3d = arreglo_1d.reshape((2, 2, 3))
print("Redimensionado a 3D (2x2x3):\n", arreglo_3d)
```

Usando `ravel`

La función `ravel` se utiliza para aplanar un arreglo n-dimensional en un arreglo unidimensional. Esto es, convierte cualquier arreglo de N dimensiones en un arreglo lineal 1D. La operación es generalmente un "vista" si es posible; esto significa que el aplanado no copia los datos, sino que devuelve una vista del arreglo original en forma de 1D. Esto hace que `ravel` sea muy eficiente en términos de memoria y tiempo.

```
# Aplanar el arreglo 2D anterior
aplanado = arreglo_2d.ravel()
print("Aplanado desde 2D:\n", aplanado)

# También se puede usar `flatten` para obtener el mismo resultado con una copia en lugar de una vista
aplanado_con_copia = arreglo_2d.flatten()
print("Aplanado (copia) desde 2D:\n", aplanado_con_copia)
```

Diferencias clave

- **`reshape`** permite cambiar las dimensiones de un arreglo a cualquier otra forma, siempre y cuando el número total de elementos sea el mismo. Es una operación esencial para reorganizar los datos para diferentes análisis o algoritmos.
- **`ravel`** convierte cualquier arreglo de N-dimensiones en un arreglo unidimensional, proporcionando una vista del arreglo original, lo que es útil para operaciones que requieren una estructura de datos lineal. Si necesitas una copia en lugar de una vista, puedes usar `flatten`.

Ambas funciones son cruciales para el preprocesamiento de datos y la manipulación de estructuras de datos en análisis numéricos y científicos, permitiendo a los usuarios adaptar sus arreglos para una variedad de aplicaciones y algoritmos específicos.

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
# x = np.arange(1, 13)

x
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
x.reshape((3, 4))
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
x.reshape((4, 3))
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
x.reshape((4, 3)).reshape((2, 6))
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

```
x.reshape((6, 2))
```

```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
```

```
           [ 9, 10],
           [11, 12]])
```

```
x.reshape((4, -1))
```

```
     array([[ 1,  2,  3],
            [ 4,  5,  6],
            [ 7,  8,  9],
            [10, 11, 12]])
```

```
x.reshape((-1, 6))
```

```
     array([[ 1,  2,  3,  4,  5,  6],
            [ 7,  8,  9, 10, 11, 12]])
```

```
A = np.array([
    [6, 4, 78, 1],
    [22, 45, 63, 12],
    [18, 26, 99, 3],
])

A
```

```
     array([[ 6,  4, 78,  1],
            [22, 45, 63, 12],
            [18, 26, 99,  3]])
```

```
A.ravel()
```

```
     array([ 6,  4, 78,  1, 22, 45, 63, 12, 18, 26, 99,  3])
```

```
P = np.array([
    [[6, 1], [4, 3], [78, 8], [1, 4]],
    [[22, 19], [45, 33], [63, 56], [12, 12]],
    [[18, 1], [26, 2], [99, 7], [3, 3]],
])

P
```

```
     array([[[ 6,  1],
             [ 4,  3],
             [78,  8],
             [ 1,  4]],

            [[22, 19],
             [45, 33],
             [63, 56],
             [12, 12]],

            [[18,  1],
             [26,  2],
             [99,  7],
             [ 3,  3]]])
```

```
p = P.ravel() # Aplanado lógico

p
```

```
     array([ 6,  1,  4,  3, 78,  8,  1,  4, 22, 19, 45, 33, 63, 56, 12, 12, 18,
             1, 26,  2, 99,  7,  3,  3])
```

```
p[::2] # i:j (2 en 2)
```

```
     array([ 6,  4, 78,  1, 22, 45, 63, 12, 18, 26, 99,  3])
```

```
P[:, :, 0] # ???
```

```
     array([[ 6,  4, 78,  1],
            [22, 45, 63, 12],
            [18, 26, 99,  3]])
```

```
P.flatten() # Aplinado físico (copia)
```

```
     array([ 6,  1,  4,  3, 78,  8,  1,  4, 22, 19, 45, 33, 63, 56, 12, 12, 18,
             1, 26,  2, 99,  7,  3,  3])
```

Ejemplo de redimensión de Vector Fila a Vector Columna

Para convertir un vector plano unidimensional en un vector columna en NumPy, puedes usar la función `reshape` mencionada anteriormente. Al hacerlo, especificarás que deseas un arreglo de `n` filas y 1 columna, donde `n` es el número de elementos en tu vector original. Si no conoces de antemano el número de elementos, puedes usar `-1` para la dimensión que quieres que NumPy infiera automáticamente. En el caso de convertir un vector 1D en un vector columna, querrías que NumPy infiriera el número de filas basado en la longitud del arreglo, y explicitar que quieres 1 columna.

Aquí tienes cómo se haría con el vector `[1, 2, 3, 4, 5]`:

```
import numpy as np

# Vector original 1D
vector_1d = np.array([1, 2, 3, 4, 5])

# Convertir a vector columna
vector_columna = vector_1d.reshape(-1, 1)

print("Vector columna:\n", vector_columna)
```

En este código, `reshape(-1, 1)` le dice a NumPy que quieres un arreglo que tenga 1 columna, y el número de filas necesario para acomodar todos los elementos del arreglo original. El `-1` es efectivamente un comodín que le permite a NumPy determinar automáticamente el tamaño necesario para esa dimensión, basado en la longitud del arreglo y la otra dimensión especificada (en este caso, 1 columna).

Esta técnica es muy útil cuando trabajas con bibliotecas de aprendizaje automático como scikit-learn, donde a menudo necesitas que tus datos de entrada sean explícitamente bidimensionales incluso si estás trabajando con una sola característica.

Tipos de datos en Numpy

NumPy ofrece una amplia gama de tipos de datos numéricos que puedes utilizar para crear arrays. Estos tipos de datos están optimizados para operaciones matemáticas y permiten un control preciso sobre cómo se almacenan y manipulan los datos en memoria, lo cual es crucial para el rendimiento de las aplicaciones científicas y numéricas. Aquí te muestro algunos de los tipos de datos más comunes en NumPy y ejemplos de cómo usarlos:

Enteros

- `np.int8`, `np.int16`, `np.int32`, `np.int64`: Representan números enteros con signo de 8, 16, 32 y 64 bits, respectivamente.
- `np.uint8`, `np.uint16`, `np.uint32`, `np.uint64`: Representan números enteros sin signo de 8, 16, 32 y 64 bits, respectivamente.

```
import numpy as np

# Crear un array de enteros de 32 bits
enteros_32_bits = np.array([1, 2, 3], dtype=np.int32)
print(enteros_32_bits)

# Crear un array de enteros sin signo de 16 bits
enteros_sin_signo_16_bits = np.array([1, 2, 3], dtype=np.uint16)
print(enteros_sin_signo_16_bits)
```

Flotantes

- `np.float16`, `np.float32`, `np.float64`: Representan números de punto flotante de 16, 32 y 64 bits, respectivamente. `np.float64` es el tipo de punto flotante estándar en NumPy.

```
# Crear un array de flotantes de 64 bits
flotantes_64_bits = np.array([1.5, 2.5, 3.5], dtype=np.float64)
print(flotantes_64_bits)
```

Complejos

- `np.complex64`, `np.complex128`: Representan números complejos con partes real e imaginaria de 32 y 64 bits de punto flotante, respectivamente.

```
# Crear un array de números complejos
complejos_128_bits = np.array([1+2j, 3+4j], dtype=np.complex128)
print(complejos_128_bits)
```

Otros tipos de datos

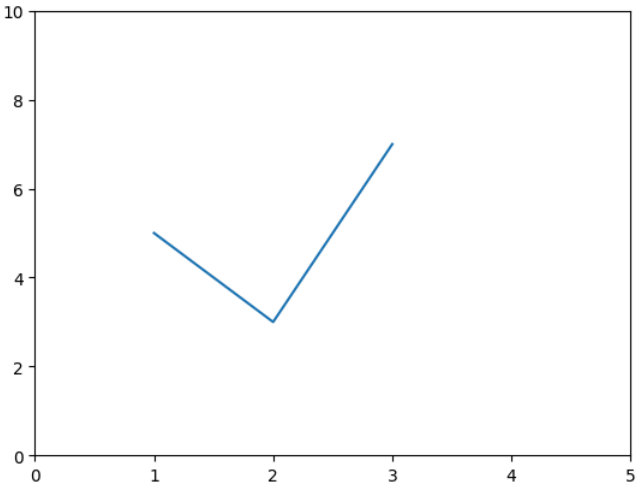
- `np.bool_`: Representa valores booleanos (`True` o `False`).
- `np.object_`: Puede contener cualquier tipo de objeto Python.
- `np.string_`, `np.unicode_`: Para representar cadenas de texto.

```
# Crear un array booleano
booleanos = np.array([True, False, True], dtype=np.bool_)
print(booleanos)

# Crear un array de strings
strings = np.array(["apple", "banana", "cherry"], dtype=np.string_)
print(strings)
```

Cada uno de estos tipos de datos se utiliza dependiendo de las necesidades específicas de tu aplicación, como el manejo eficiente de la memoria, la precisión de los cálculos numéricos, o la representación adecuada de los datos. Elegir el tipo de dato correcto puede tener un impacto significativo en el rendimiento y la eficiencia de tus programas.

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [5, 3, 7])

plt.xlim((0, 5))
plt.ylim((0, 10))
```

```
(0.0, 10.0)
```




```
# x = np.linspace(-1, 1, 11)
x = np.linspace(-np.pi, np.pi, 5)
x
```

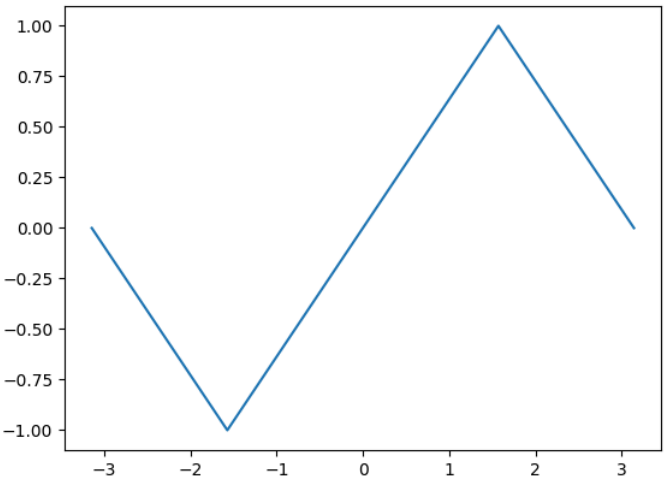
```
array([-3.14159265, -1.57079633,  0.        ,  1.57079633,  3.14159265])
```

```
# y = x ** 2
y = np.sin(x)
y
```

```
array([-1.2246468e-16, -1.0000000e+00,  0.0000000e+00,  1.0000000e+00,
        1.2246468e-16])
```

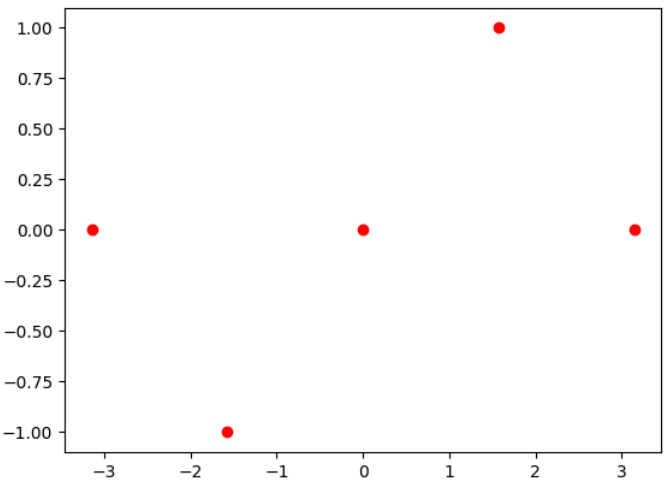
```
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x793bac382fb0>]
```



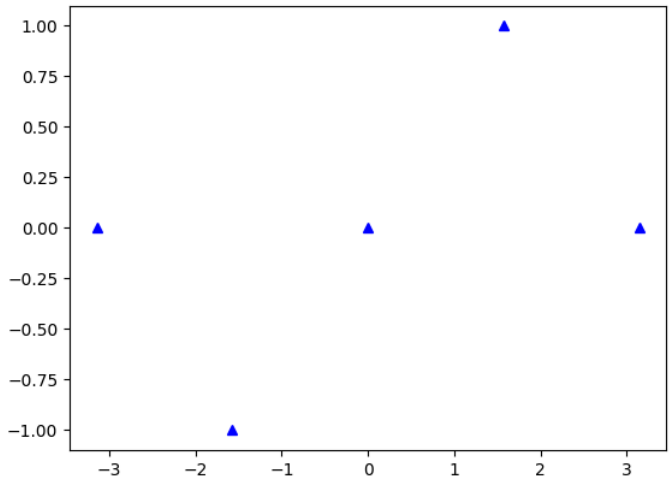

```
plt.plot(x, y, 'ro')
```

```
[<matplotlib.lines.Line2D at 0x793bac1d3010>]
```



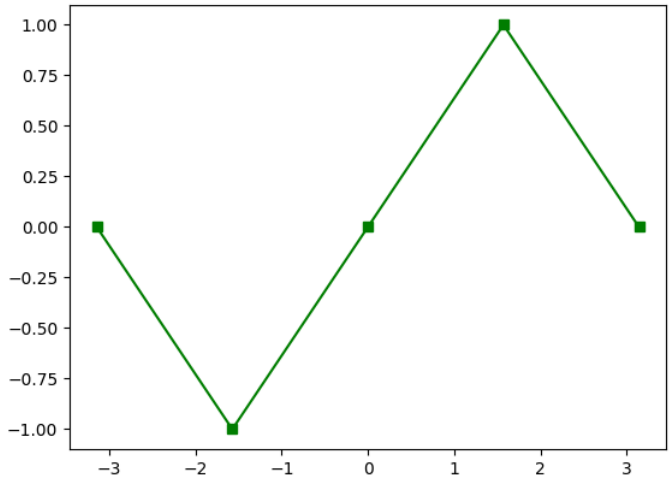
```
plt.plot(x, y, 'b^')
```

```
[<matplotlib.lines.Line2D at 0x793bac182320>]
```



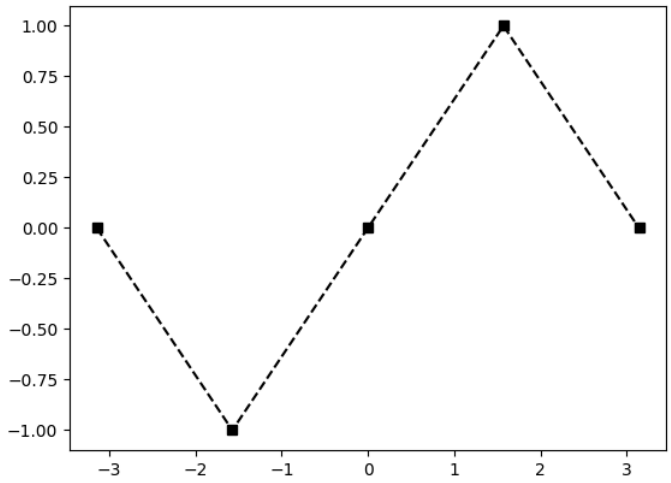

```
plt.plot(x, y, 'gs-')
```

```
[<matplotlib.lines.Line2D at 0x793bac4c9390>]
```

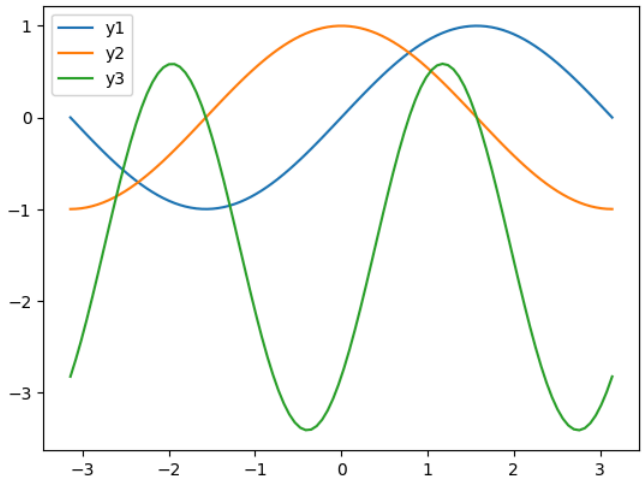



```
plt.plot(x, y, 'ks--')
```

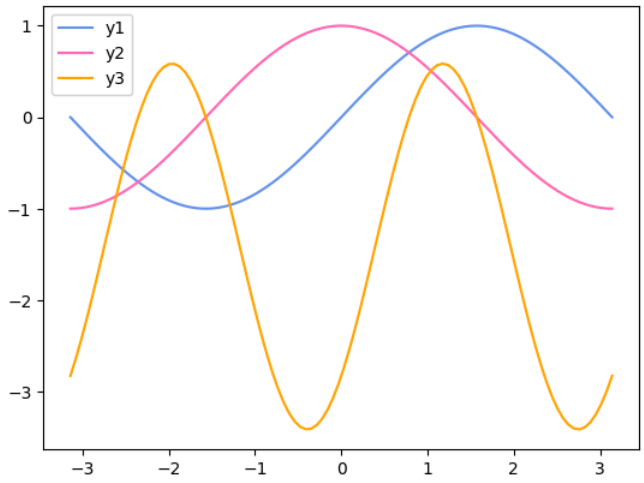
```
[<matplotlib.lines.Line2D at 0x793babfa89a0>]
```



```
x = np.linspace(-np.pi, np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = 4 * np.cos(x) * np.sin(x - np.pi / 4)

plt.plot(x, y1, label="y1")
plt.plot(x, y2, label="y2")
plt.plot(x, y3, label="y3")

# Activamos las etiquetas para las series
plt.legend()

# Guardamos la gráfica en una imagen
plt.savefig("reporte1.png")

plt.show()
```




```
x = np.linspace(-np.pi, np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = 4 * np.cos(x) * np.sin(x - np.pi / 4)

plt.plot(x, y1, color="cornflowerblue", label="y1")
plt.plot(x, y2, color="hotpink", label="y2")
plt.plot(x, y3, color="orange", label="y3")

# Activamos las etiquetas para las series
plt.legend()

# Guardamos la gráfica en una imagen
plt.savefig("reporte1.png")

plt.show()
```



```
A = np.array([
    [1, 2, 4],
    [2, 3, 1],
    [5, 7, 9],
])

b = np.array([24, 36, 94])

x = np.dot(np.linalg.inv(A), b)

x
```

```
array([10.,  5.,  1.])
```

302. Introducción a Pandas y su uso avanzado

Pandas es una biblioteca de Python que proporciona estructuras de datos y herramientas de análisis de datos de alto rendimiento y fáciles de usar. Es ampliamente utilizada en la ciencia de datos y en la analítica debido a su eficiencia, flexibilidad y facilidad de uso. Pandas se basa en NumPy, lo que le permite integrarse bien en el ecosistema de ciencia de datos de Python. Los dos componentes principales de Pandas son las Series y los DataFrames.

Series

Una `Series` es un arreglo unidimensional etiquetado capaz de contener cualquier tipo de datos (enteros, cadenas, flotantes, objetos de Python, etc.). Las etiquetas de los ejes se conocen colectivamente como el índice. Piensa en una `Series` como una columna en una tabla. A diferencia de un arreglo de NumPy, una `Series` puede tener un índice que le da un nombre o etiqueta a cada elemento, lo que añade una capa de información y flexibilidad.

Creación de una Serie

```
import pandas as pd

datos = [1, 3, 5, 7, 9]
serie = pd.Series(datos)
print(serie)
```

También puedes especificar explícitamente un índice al crear la `Series`:

```
serie_con_indice = pd.Series(datos, index=['a', 'b', 'c', 'd', 'e'])
print(serie_con_indice)
```

DataFrame

Un `DataFrame` es una estructura de datos bidimensional, esencialmente una tabla con filas y columnas, donde cada columna puede ser de un tipo de dato diferente (número, cadena, booleano, etc.). Piensa en un `DataFrame` como una hoja de cálculo o una tabla de SQL. Es la estructura de datos más comúnmente utilizada en Pandas y es muy adecuada para representar datos complejos y heterogéneos de una manera fácil de entender y manipular.

Creación de un DataFrame

```
data = {
    'Nombre': ['Juan', 'Ana', 'Pedro', 'Luisa'],
    'Edad': [28, 34, 29, 42],
    'Ciudad': ['Madrid', 'Barcelona', 'Madrid', 'Sevilla']
}
df = pd.DataFrame(data)
print(df)
```

En este ejemplo, `data` es un diccionario de listas. Las claves del diccionario se convierten en los nombres de las columnas y las listas asociadas son los valores en esas columnas. Pandas automáticamente asigna un índice a las filas, comenzando por 0.

Operaciones básicas

Tanto las `Series` como los `DataFrames` admiten una amplia gama de operaciones, como la selección de un subconjunto de datos, la agregación y la combinación de datos, la manipulación de índices, y mucho más. Pandas se integra bien con otras bibliotecas de ciencia de datos en Python, incluyendo NumPy y Matplotlib, permitiendo análisis de datos complejos y la creación de visualizaciones con relativa facilidad.

En resumen, Pandas es una herramienta esencial para la manipulación y el análisis de datos en Python, ofreciendo estructuras de datos poderosas y flexibles que hacen que el trabajo con datos sea más intuitivo y eficiente.

Haz doble clic (o ingresa) para editar

```
import pandas as pd

edades = pd.Series([
    34,
    37,
    35,
    45,
    56,
    18,
], name="Edad")

edades
```

```
0    34
1    37
2    35
3    45
4    56
5    18
Name: Edad, dtype: int64
```

```
edades.min(), edades.max()
```

```
(18, 56)
```

```
print(f"Min  {edades.min()}")
print(f"Prom {edades.mean()}")
print(f"Max  {edades.max()}")
print(f"D.S. {edades.std()}")

print("Intervalo de confianza de las edades al 66% de los datos")
print("({:.0f}, [{:.0f}], {:.0f})".format(edades.mean() - edades.std(), edades.mean(), edades.mean() + edades.std()))

print("Intervalo de confianza de las edades al 95% de los datos")
print("({:.0f}, [{:.0f}], {:.0f})".format(edades.mean() - 2 * edades.std(), edades.mean(), edades.mean() + 2 * edades.std()))
```

```
Min  18
Prom 37.5
Max  56
D.S. 12.62933094031509
Intervalo de confianza de las edades al 66% de los datos
(25, [38], 50)
Intervalo de confianza de las edades al 95% de los datos
(12, [38], 63)
```

Ejemplo de operaciones entre Series

Las operaciones entre `Series` en Pandas se pueden realizar de manera muy similar a las operaciones aritméticas en NumPy, con la ventaja añadida de alinear datos basándose en sus índices. Esto significa que cuando realizas operaciones entre `Series`, Pandas automáticamente alinea los datos según los índices de cada `Series` antes de realizar la operación. Aquí te dejo algunos ejemplos:

Ejemplo de Suma

Supongamos que tienes dos `Series` que representan cantidades de ventas en dos trimestres diferentes para un conjunto de productos:

```
import pandas as pd

ventas_q1 = pd.Series([100, 200, 300], index=['manzanas', 'naranjas', 'plátanos'])
ventas_q2 = pd.Series([90, 210, 250, 300], index=['manzanas', 'naranjas', 'plátanos', 'kiwis'])

total_ventas = ventas_q1 + ventas_q2
print(total_ventas)
```

Este código sumará las ventas de `ventas_q1` y `ventas_q2` basándose en el índice de cada `Series`. Para los índices que no se encuentran en ambas `Series` (como "kiwis" en este caso), Pandas asignará un valor de `NaN` (Not a Number), indicando datos faltantes.

Ejemplo de Resta

De forma similar, puedes restar una `Series` de otra para, por ejemplo, encontrar la diferencia en ventas entre dos trimestres:

```
diferencia_ventas = ventas_q1 - ventas_q2
print(diferencia_ventas)
```

Esta operación restará las ventas de `ventas_q2` de `ventas_q1`, nuevamente alineando las `Series` por sus índices.

Ejemplo de Multiplicación y División

Las operaciones de multiplicación y división también se pueden realizar de la misma manera, lo que puede ser útil para aplicar factores de escala o calcular ratios, respectivamente:

```
# Multiplicación
ventas_q1_duplicadas = ventas_q1 * 2
print(ventas_q1_duplicadas)
```

```
# División
ratio_ventas = ventas_q1 / ventas_q2
print(ratio_ventas)
```

Tratamiento de NaN

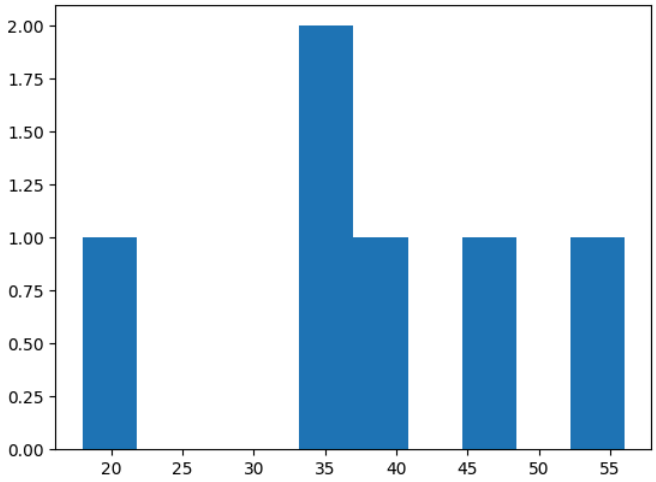
En casos donde quieres evitar resultados con `NaN` cuando los índices no coinciden completamente entre las `Series`, puedes utilizar métodos como `add`, `sub`, `mul`, y `div`, que permiten especificar un valor de relleno para los índices faltantes usando el argumento `fill_value`:

```
# Suma con fill_value
total_ventas_fill = ventas_q1.add(ventas_q2, fill_value=0)
print(total_ventas_fill)
```

Este enfoque te permite mantener operaciones entre `Series` sin introducir valores `NaN`, lo cual puede ser especialmente útil en análisis de datos donde quieres evitar la propagación de valores faltantes.

Estos ejemplos muestran la flexibilidad y potencia de Pandas para realizar operaciones entre `Series`, aprovechando el alineamiento automático por índices para facilitar el manejo de datos.

```
import matplotlib.pyplot as plt

plt.hist(edades)

plt.show()
```



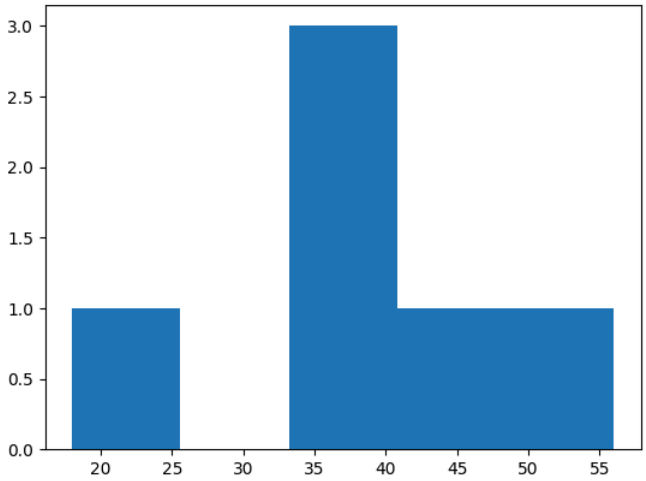

```
np.histogram(edades)
```

```
(array([1, 0, 0, 0, 2, 1, 0, 1, 0, 1]),
 array([18. , 21.8, 25.6, 29.4, 33.2, 37. , 40.8, 44.6, 48.4, 52.2, 56. ]))
```

```
np.histogram(edades, bins=5)
```

```
(array([1, 0, 3, 1, 1]), array([18. , 25.6, 33.2, 40.8, 48.4, 56. ]))
```

```
plt.hist(edades, bins=5)

plt.show()
```



Ejemplo de mapeo de Series

El mapeo en Pandas es una operación poderosa que te permite transformar los datos de una `Series` basándose en un mapeo definido por otro conjunto de valores, una función, o un diccionario. Es particularmente útil para la transformación de datos categóricos, la aplicación de transformaciones a un conjunto de valores, o incluso para reemplazar datos basándose en ciertos criterios. Aquí te muestro cómo hacerlo con un ejemplo práctico:

Ejemplo de Mapeo con un Diccionario

Imagina que tienes una `Series` que representa códigos de categorías para ciertos productos y quieres convertir esos códigos en nombres legibles para humanos.

```
import pandas as pd

# Serie con códigos de categorías
codigos_categorias = pd.Series([1, 2, 3, 1, 2, 3, 1, 1])

# Diccionario para mapear códigos a nombres de categorías
mapeo_categorias = {
    1: 'Alimentos',
    2: 'Bebidas',
    3: 'Ropa'
}

# Usar map para transformar los códigos en nombres de categorías
categorias = codigos_categorias.map(mapeo_categorias)
print(categorias)
```

Este código transformará la `Series` original de códigos numéricos en una nueva `Series` con los nombres de las categorías correspondientes a esos códigos, utilizando el diccionario `mapeo_categorias` como la definición del mapeo.

Ejemplo de Mapeo con una Función

También puedes usar una función como argumento para `.map()`. Esto te permite aplicar cualquier transformación definida por la función a los elementos de la `Series`.

```
# Definir una función para categorizar edades
def categorizar_edad(edad):
    if edad < 18:
        return 'Menor'
    elif edad >= 18 and edad <= 65:
        return 'Adulto'
    else:
        return 'Senior'

edades = pd.Series([5, 20, 67, 34, 15, 52])
categorias_edad = edades.map(categorizar_edad)
print(categorias_edad)
```

En este ejemplo, la función `categorizar_edad` se aplica a cada elemento de la `Series edades`, devolviendo una nueva `Series` con las categorías de edad correspondientes.

Estos ejemplos demuestran cómo el método `.map()` de Pandas te permite realizar transformaciones de datos de forma eficiente y flexible, lo que lo hace una herramienta valiosa para el preprocesamiento y la manipulación de datos en análisis de datos y ciencia de datos.

```
deudas = pd.Series([
    "S",
    "N",
    "S",
    "S",
    "N",
    "S",
], name="deudas")

deudas
```

```
0    S
1    N
2    S
3    S
4    N
5    S
Name: deudas, dtype: object
```

```
# Mapeo por diccionario (sustitución directa)
deudas = pd.Series([
    "S",
    "N",
    "S",
    "S",
    "N",
    "S",
], name="deudas").map({
    "S": 1,
    "N": 0
})

deudas
```

```
0     True
1    False
2     True
3     True
4    False
5     True
Name: deudas, dtype: bool
```

```
s1 = pd.Series(["A", "B", "A", "C", "B"])

s2 = s1.map({"A": 1, "B": 3, "C": 2})

print(s1)
print(s2)
```

```
0    A
1    B
2    A
3    C
4    B
dtype: object
0    1
1    3
2    1
3    2
4    3
dtype: int64
```

```
def T(x):
  return x ** 2

s1 = pd.Series([1, 4, 3, 5, 6])
s2 = s1 ** 2
s3 = s1.map(T)

print(s1)
print(s2)
print(s3)
```

```
0    1
1    4
2    3
3    5
4    6
dtype: int64
0     1
1    16
2     9
3    25
4    36
dtype: int64
0     1
1    16
2     9
3    25
4    36
dtype: int64
```

```
# Clúster de edades:
# 1 - 0-20
# 2 - 21-40
# 3 - 41-60
# 4 - 60+

edades
```

```
0    34
1    37
2    35
3    45
4    56
5    18
Name: Edad, dtype: int64
```

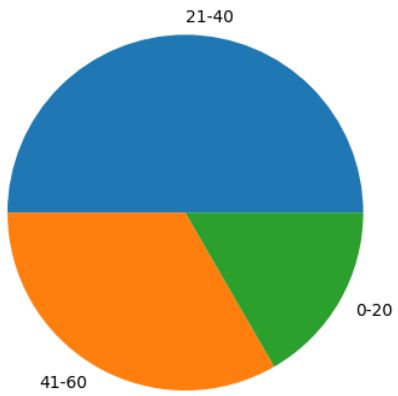
```
def clu_edad(edad):
  if edad <= 20:
    return 1
  if edad <= 40:
    return 2
  if edad <= 60:
    return 3
  return 4

print(edades)

edades.map(clu_edad)
```

```
0    34
1    37
2    35
3    45
4    56
5    18
Name: Edad, dtype: int64
0    2
1    2
2    2
3    3
4    3
5    1
Name: Edad, dtype: int64
```

```
cluster_edades = edades.map(clu_edad)

clusters = pd.Series(cluster_edades.unique())

print(clusters)

sizes = [cluster_edades[cluster_edades == i].count() for i in clusters]

sizes
```

```
0    2
1    3
2    1
dtype: int64
[3, 2, 1]
```

```
plt.pie(sizes, labels=clusters.map({ 1: "0-20", 2: "21-40", 3: "41-60", 4: "6'+"}))
plt.show()
```



Ejemplo de filtros en Series

Puedes filtrar elementos de una `Series` en Pandas utilizando condiciones booleanas. Esta operación es muy común y útil cuando necesitas seleccionar elementos que cumplan con ciertos criterios. Aquí te dejo un ejemplo para ilustrar cómo se hace:

Ejemplo de Filtrado de Series

Supongamos que tienes una `Series` que representa las edades de un grupo de personas y quieres filtrar para obtener solo las edades mayores o iguales a 18 años.

```
import pandas as pd

# Serie con edades
edades = pd.Series([16, 21, 22, 15, 17, 24, 35, 55])

# Filtrar para obtener solo edades mayores o iguales a 18
edades_adultas = edades[edades >= 18]
print(edades_adultas)
```

En este ejemplo, `edades >= 18` produce una `Series` de valores booleanos donde cada elemento es `True` si la condición se cumple (es decir, si la edad es mayor o igual a 18) y `False` en caso contrario. Al usar esta `Series` de valores booleanos para indexar la `Series` original, Pandas selecciona solo aquellos elementos donde el valor correspondiente en la `Series` de condiciones es `True`.

Filtrado con Múltiples Condiciones

También puedes combinar varias condiciones utilizando operadores lógicos (`&` para AND, `|` para OR, `~` para NOT) para realizar filtrados más complejos. Por ejemplo, si quisieras filtrar las edades que son mayores o iguales a 18 pero menores que 30:

```
# Filtrar para obtener edades mayores o iguales a 18 y menores que 30
edades_joven_adulto = edades[(edades >= 18) & (edades < 30)]
print(edades_joven_adulto)
```

Recuerda utilizar paréntesis alrededor de cada condición para asegurar que las operaciones se evalúen correctamente.

Estos ejemplos muestran cómo puedes filtrar datos en una `Series` de Pandas de manera eficiente, permitiéndote seleccionar subconjuntos de datos que cumplan con criterios específicos, una operación fundamental en la limpieza y preparación de datos para análisis posteriores.

Explicación sobre el uso de Máscaras en Numpy y Pandas

Las máscaras en el contexto de filtrado con Pandas (y, por extensión, con NumPy) son esencialmente arreglos de valores booleanos (True/False) que se utilizan para seleccionar (filtrar) datos de estructuras más grandes como `Series` o `DataFrames`. La "máscara" actúa como un filtro que "oculta" los datos no deseados y solo "muestra" los datos que cumplen con un criterio específico. Veamos en detalle cómo funciona esto.

Creación de una Máscara

Para filtrar datos, primero se evalúa una condición sobre el arreglo o `Series`, lo cual genera una máscara de valores booleanos. Por ejemplo, dada una `Series` de edades:

```
import pandas as pd

edades = pd.Series([18, 22, 19, 17, 30, 21, 23, 16])
```

Si queremos filtrar para obtener solo aquellas edades mayores o iguales a 18, primero creamos una máscara aplicando la condición a la `Series`:

```
mascara = edades >= 18
```

Esto generará:

```
0     True
1     True
2     True
3    False
4     True
5     True
6     True
7    False
dtype: bool
```

Cada `True` indica que el elemento en esa posición cumple con la condición (es mayor o igual a 18), mientras que `False` indica lo contrario.

Aplicación de la Máscara para Filtrar Datos

Para aplicar la máscara y filtrar los datos, usamos la máscara como índice de la `Series` original:

```
edades_filtradas = edades[mascara]
```

Esto devuelve una nueva `Series` que solo contiene los elementos que cumplen con la condición:

```
0    18
1    22
2    19
4    30
5    21
6    23
dtype: int64
```

Combinación de Máscaras

Las máscaras se pueden combinar para aplicar múltiples criterios de filtrado simultáneamente, usando operadores lógicos como `&` (AND), `|` (OR), y `~` (NOT). Por ejemplo, para seleccionar edades mayores o iguales a 18 pero menores que 25:

```
mascara_combinada = (edades >= 18) & (edades < 25)
edades_filtradas = edades[mascara_combinada]
```

Es importante usar paréntesis alrededor de las condiciones individuales para asegurar que las operaciones se evalúen correctamente.

Ventajas de las Máscaras

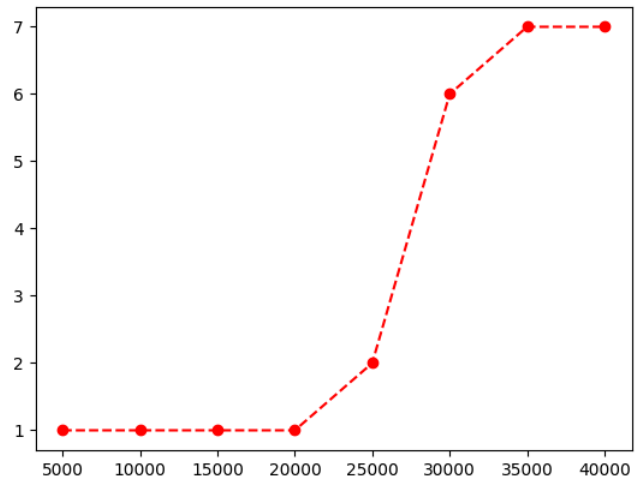
- **Flexibilidad:** Permite combinar múltiples condiciones de filtrado de manera fácil y legible.
- **Eficiencia:** Operar con máscaras es generalmente más rápido que iterar sobre los datos manualmente, especialmente con grandes volúmenes de datos.
- **Integración:** Esta técnica de filtrado con máscaras es consistente entre Pandas y NumPy, facilitando la manipulación de datos entre estas bibliotecas.

En resumen, el filtrado mediante máscaras es una técnica potente y versátil para seleccionar subconjuntos de datos basados en uno o más criterios, permitiendo un análisis de datos eficiente y efectivo.

```
salarios = pd.Series([
  24000,
  28000,
  28000,
  28000,
  28000,
  32000,
  # 19000,
  # 14000,
  4000,
], name="Salarios")

salarios_max = [5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000]

salarios_accum = []

print("MAX SALARIO | TOTAL")
for max_salario in salarios_max:
  salarios_debajo_max = salarios[salarios < max_salario].count()
  print("{:>10}  | {:>4}".format(max_salario, salarios_debajo_max))
  salarios_accum.append(salarios_debajo_max)
```

```
     MAX SALARIO | TOTAL
            5000  |    1
           10000  |    1
           15000  |    1
           20000  |    1
           25000  |    2
           30000  |    6
           35000  |    7
           40000  |    7
```

```
plt.plot(salarios_max, salarios_accum, 'ro--')
plt.show()
```



Trabajar con DataFrames

Los DataFrames de Pandas son estructuras de datos bidimensionales, similares a las hojas de cálculo, que pueden almacenar diferentes tipos de datos en cada columna y son capaces de realizar una amplia gama de operaciones de manipulación de datos. Son increíblemente útiles para el análisis de datos porque permiten realizar operaciones complejas sobre los datos con comandos sencillos y eficientes. A continuación, te presento algunos ejemplos de operaciones comunes que puedes realizar con DataFrames en Pandas.

Creación de un DataFrame

Primero, veamos cómo crear un DataFrame desde un diccionario de listas de Python:

```
import pandas as pd

data = {
    "Nombres": ["Ana", "Luis", "Marta", "Pedro", "Juan"],
    "Edades": [23, 34, 45, 22, 33],
    "Ciudad": ["Madrid", "Barcelona", "Valencia", "Madrid", "Barcelona"]
}

df = pd.DataFrame(data)
```

Este código crea un DataFrame con los datos proporcionados, donde las claves del diccionario se convierten en los nombres de las columnas.

Selección de Datos

Puedes seleccionar columnas específicas de un DataFrame simplemente usando el nombre de la columna:

```
# Seleccionar una columna
edades = df["Edades"]

# Seleccionar múltiples columnas
subset = df[["Nombres", "Ciudad"]]
```

Filtrado de Datos

Para filtrar filas basándose en alguna condición:

```
# Filtrar filas donde la edad es mayor a 30
df_mayores_30 = df[df["Edades"] > 30]
```

Modificación de Datos

Puedes añadir nuevas columnas o modificar las existentes fácilmente:

```
# Añadir una nueva columna
df["Años hasta jubilación"] = 65 - df["Edades"]

# Modificar una columna existente (por ejemplo, incrementar todas las edades en 1)
df["Edades"] += 1
```

Agrupación y Agregación

Los DataFrames permiten agrupar datos y aplicar funciones de agregación, similar a SQL:

```
# Agrupar por ciudad y contar el número de personas en cada una
conteo_ciudades = df.groupby("Ciudad").size()

# Agrupar por ciudad y calcular la edad promedio
promedio_edad_por_ciudad = df.groupby("Ciudad")["Edades"].mean()
```

Fusionar DataFrames

Puedes combinar DataFrames de manera similar a las operaciones JOIN en SQL:

```
# Supongamos otro DataFrame de ciudades y población
data_ciudades = pd.DataFrame({
    "Ciudad": ["Madrid", "Barcelona", "Valencia"],
    "Población": [3200000, 1600000, 790000]
})

# Fusionar los DataFrames basándose en la columna "Ciudad"
df_con_poblacion = pd.merge(df, data_ciudades, on="Ciudad")
```

Lectura y Escritura de Datos

Pandas facilita la lectura y escritura de DataFrames desde y hacia diferentes formatos de archivos:

```
# Leer un CSV
df_from_csv = pd.read_csv("datos.csv")

# Escribir a un Excel
df.to_excel("datos_salida.xlsx", index=False)
```

Estos ejemplos apenas rascan la superficie de lo que puedes hacer con los DataFrames en Pandas. Pandas ofrece una amplia funcionalidad para manipular, filtrar, agrupar, y transformar datos, lo que lo hace una herramienta indispensable en el análisis de datos con Python.

```
sC = pd.Series([19, 2, 5, 4])

df = pd.DataFrame({
    "A": [1, 2, 3, 4],
    "B": [5, 6, 8, 2],
    "C": sC,
    "D": sC % 2 == 0,
})

df["E"] = df["B"] * df["C"]
df["F"] = df["B"] - df["C"].mean()

df
```

	A	B	C	D	E	F
0	1	5	19	False	95	-2.5
1	2	6	2	True	12	-1.5
2	3	8	5	False	40	0.5
3	4	2	4	True	8	-5.5

Leer un DataFrame a partir de un CSV

Leer archivos CSV en un DataFrame de Pandas es una de las operaciones más comunes en el análisis de datos, ya que los CSV son un formato estándar muy utilizado para el almacenamiento de datos tabulares. Pandas hace que esta tarea sea sencilla con la función `pd.read_csv()`. Aquí te explico cómo leer un CSV tanto desde una ruta local como desde una URL, y también te aclaro el concepto de rutas relativas.

Leer un CSV desde una Ruta Local

Cuando trabajas con archivos en tu computadora, puedes especificar la ubicación del archivo usando una **ruta absoluta** o una **ruta relativa**:

- **Ruta Absoluta:** Es la dirección completa en tu sistema de archivos que describe cómo llegar a un archivo. Por ejemplo, en Windows podría ser algo como `C:\usuarios\mi_usuario\documentos\datos.csv`, y en macOS/Linux sería algo como `/usuarios/mi_usuario/documentos/datos.csv`.
- **Ruta Relativa:** Es una dirección que se describe en relación a la ubicación actual de tu script o notebook de Jupyter. Si tu script y tu archivo CSV están en el mismo directorio, simplemente puedes usar el nombre del archivo como una ruta relativa. Si el archivo está en un subdirectorio del directorio actual, podrías usar una ruta como `subdirectorio/datos.csv`.

Ejemplo de cómo leer un CSV usando una ruta relativa:

```
import pandas as pd

# Asumiendo que `datos.csv` está en el mismo directorio que tu script o notebook
df = pd.read_csv('datos.csv')

# Si `datos.csv` está en un subdirectorio llamado `datos`, relativo al script
df = pd.read_csv('datos/datos.csv')
```

Leer un CSV desde una URL

Pandas también puede leer archivos directamente desde una URL, lo cual es muy útil para acceder a conjuntos de datos alojados en internet sin necesidad de descargarlos primero.

Ejemplo de cómo leer un CSV desde una URL:

```
url = 'https://ejemplo.com/datos.csv'
df = pd.read_csv(url)
```

Ejemplo Práctico

Supongamos que tienes un archivo CSV llamado `ejemplo_datos.csv` que contiene datos de ejemplo. Para leer este archivo en un DataFrame de Pandas, simplemente usarías `pd.read_csv()` así:

```
# Para un archivo local
df_local = pd.read_csv('ejemplo_datos.csv')

# O, para un archivo en línea
url = 'https://mi_dominio.com/ejemplo_datos.csv'
df_url = pd.read_csv(url)
```

En ambos casos, `df_local` y `df_url` serán DataFrames de Pandas conteniendo los datos leídos desde el archivo CSV local o en línea, respectivamente.

Nota Adicional sobre Rutas Relativas

El uso de rutas relativas hace tu código más portable, ya que no depende de la estructura de directorios específica de tu sistema. Esto es especialmente útil cuando compartes tu código con otros o lo ejecutas en diferentes entornos (como diferentes computadoras o cuando se publica en plataformas de colaboración como GitHub).

Principales funciones estadísticas del DataFrame

Pandas ofrece una variedad de métodos estadísticos que te permiten inspeccionar y analizar tus DataFrames de manera eficiente. Estos métodos son esenciales para explorar y comprender los datos, especialmente al inicio de cualquier proyecto de análisis de datos. Aquí te muestro algunos de los principales métodos estadísticos y su uso:

Descripción General

- **`describe()`**: Proporciona estadísticas resumidas como el conteo, media, desviación estándar, mínimo, máximo y cuartiles de las columnas numéricas. Para datos categóricos, ofrece el conteo, la cantidad de categorías únicas, y la moda.

```
df.describe()
```

Medidas de Tendencia Central

- **`mean()`**: Calcula la media aritmética de las columnas numéricas.

```
df.mean()
```

- **`median()`**: Encuentra la mediana de las columnas numéricas, el valor central en una lista ordenada de números.

```
df.median()
```

- **`mode()`**: Determina la moda de cada columna, el valor que aparece con mayor frecuencia en un conjunto de datos.

```
df.mode()
```

Medidas de Dispersión

- **`std()`**: Calcula la desviación estándar de las columnas numéricas, una medida de la dispersión de un conjunto de datos.

```
df.std()
```

- **`var()`**: Calcula la varianza de las columnas numéricas, que es el cuadrado de la desviación estándar.

```
df.var()
```

- **`min()`** y **`max()`**: Encuentran el valor mínimo y máximo en cada columna numérica, respectivamente.

```
df.min()
df.max()
```

- **`quantile(q=0.5)`**: Calcula los cuantiles de las columnas numéricas; por defecto, calcula la mediana, pero puedes especificar otros valores para `q`.

```
df.quantile(q=0.25)  # Primer cuartil
df.quantile(q=0.75)  # Tercer cuartil
```

Otros Métodos Estadísticos Útiles

- **`count()`**: Cuenta el número de elementos no nulos en cada columna.

```
df.count()
```

- **`sum()`**: Suma los valores de cada columna.

```
df.sum()
```

- **`corr()`**: Calcula la correlación entre columnas numéricas.

```
df.corr()
```

- **`cov()`**: Calcula la covarianza entre columnas numéricas.

```
df.cov()
```

Estos métodos son fundamentales para realizar un análisis exploratorio de datos (EDA) al comenzar con un nuevo conjunto de datos, ya que te permiten obtener rápidamente una visión general de las propiedades estadísticas de tus datos. Utilizarlos eficazmente puede ayudarte a identificar patrones, anomalías, tendencias y relaciones entre variables, lo cual es crucial para cualquier análisis posterior.

Agregar una fila al DataFrame

Agregar una fila a un DataFrame en Pandas se puede hacer de varias maneras, dependiendo de tus necesidades específicas y de la estructura de tus datos. Aquí te muestro dos métodos comunes para hacerlo:

Usando `loc` para Agregar una Fila

Si conoces el índice de la nueva fila (o si quieres especificar uno), puedes usar el método `loc` para agregar una fila directamente. Este método es útil si quieres agregar una fila en una posición específica o con un índice específico.

```
import pandas as pd

# Supongamos que tenemos el siguiente DataFrame
df = pd.DataFrame({
    'Nombre': ['Ana', 'Luis'],
    'Edad': [25, 30],
    'Ciudad': ['Madrid', 'Barcelona']
})

# Agregar una fila usando loc
df.loc[len(df)] = ['Marta', 22, 'Sevilla']  # len(df) da el nuevo índice como el siguiente en la secuencia

print(df)
```

En este ejemplo, `len(df)` se utiliza para obtener el próximo índice disponible, asumiendo que los índices son secuenciales y empiezan en 0. Sin embargo, puedes especificar cualquier otro índice si es necesario.

Usando el Método `append`

El método `append` es otra forma de agregar filas a un DataFrame. Con `append`, creas una nueva `Series` o un diccionario con los datos de la fila y luego los añades al DataFrame. Ten en cuenta que `append` no modifica el DataFrame original; en su lugar, devuelve un nuevo DataFrame con la fila añadida, por lo que deberás asignar el resultado a una variable si deseas conservarlo.

```
# Crear un diccionario con los datos de la nueva fila
nueva_fila = {'Nombre': 'Carlos', 'Edad': 28, 'Ciudad': 'Valencia'}

# Usar append para agregar la nueva fila
df = df.append(nueva_fila, ignore_index=True)  # ignore_index=True para reasignar índices

print(df)
```

Es importante usar el argumento `ignore_index=True` si quieres que Pandas reasigne los índices del DataFrame resultante. Si no, el índice de la nueva fila será 0 por defecto, lo cual podría resultar en índices duplicados si ya tienes una fila con ese índice.

Ambos métodos son útiles y puedes elegir uno u otro dependiendo de tu situación específica. `loc` es más directo y modifica el DataFrame in situ, mientras que `append` es más flexible pero requiere asignar el resultado a un nuevo DataFrame o al existente.

Visualización de Datos en el DataFrame

Seaborn es una biblioteca de visualización de datos en Python basada en Matplotlib que ofrece una interfaz de alto nivel para la creación de gráficos estadísticos atractivos y explicativos. Es particularmente útil para visualizar DataFrames de Pandas gracias a su capacidad para entender directamente las estructuras de datos de Pandas y su enfoque en la visualización de patrones estadísticos. Aquí te presento algunas de las principales formas de visualización de datos en un DataFrame utilizando Seaborn:

Gráficos de Distribución

- **`distplot` / `histplot`**: Muestra la distribución de un conjunto de datos univariado. `histplot` es el sucesor de `distplot` en versiones recientes de Seaborn.
- **`jointplot`**: Crea un gráfico que muestra la relación bivariada entre dos variables junto con la distribución univariada de cada variable en los márgenes.
- **`pairplot`**: Visualiza las relaciones bivariadas entre varias parejas de variables en un DataFrame, útil para explorar correlaciones entre variables.

Gráficos Categóricos

- **`barplot`**: Muestra estimaciones de valores centrales de una variable numérica con la altura de cada rectángulo y proporciona algunas indicaciones de la incertidumbre alrededor de esas estimaciones usando barras de error.
- **`countplot`**: Muestra el número de ocurrencias de cada categoría categórica utilizando barras.

- **`boxplot`**: Muestra la distribución de datos cuantitativos de una manera que facilita comparaciones entre variables o a través de niveles de variables categóricas. Es útil para ver medianas, cuartiles y outliers.
- **`violinplot`**: Combina las características de los boxplots y los KDE plots, mostrando la distribución de los datos a través de diferentes categorías y sus probabilidades de densidad.

Gráficos de Relaciones

- **`scatterplot`**: Dibuja una dispersión de puntos que representa los valores de dos variables diferentes, permitiendo ver cómo se relaciona una variable con otra.
- **`regplot` / `lmplot`**: Muestra una regresión lineal entre dos variables. `lmplot` combina `regplot` y `FacetGrid`, facilitando la creación de gráficos de regresión en facetas.
- **`heatmap`**: Muestra datos de matriz donde los colores representan los valores. Es ideal para visualizar matrices de correlación o tablas de contingencia.

Ejemplo de Visualización con Seaborn

Aquí tienes un ejemplo básico de cómo visualizar la distribución de una variable utilizando `histplot` en Seaborn:

```
import seaborn as sns
import pandas as pd

# Supongamos que df es tu DataFrame
df = pd.DataFrame({
    'Edades': [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
})

# Visualización de la distribución de edades
sns.histplot(df['Edades'])
```

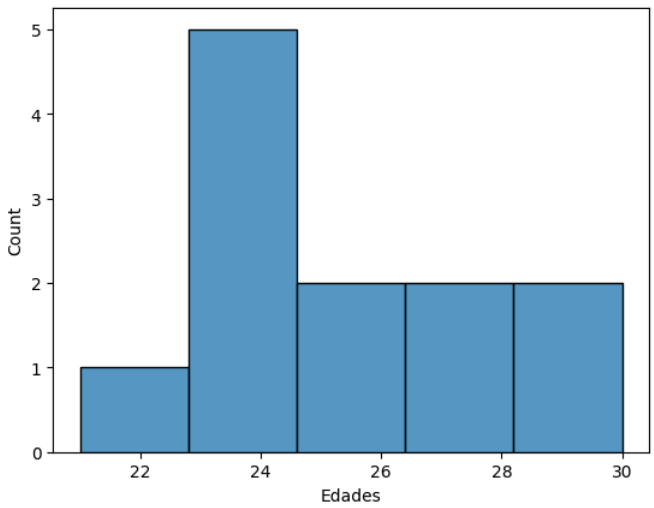
Y un ejemplo usando `scatterplot` para explorar la relación entre dos variables:

```
# Asumiendo que df tiene dos columnas, 'Edades' y 'Alturas'
sns.scatterplot(x='Edades', y='Alturas', data=df)
```

Estos son solo algunos ejemplos de cómo puedes comenzar a visualizar tus datos con Seaborn. La biblioteca ofrece muchas otras opciones y personalizaciones para explorar y presentar tus datos de manera efectiva.

```
import seaborn as sns
import pandas as pd

# Supongamos que df es tu DataFrame
df = pd.DataFrame({
    'Edades': [21, 23, 23, 23, 23, 24, 25, 26, 27, 28, 29, 30]
})

# Visualización de la distribución de edades
sns.histplot(df['Edades'])
```

```
<Axes: xlabel='Edades', ylabel='Count'>
```



Ejemplos de repaso

```
edades = [18, 19, 23, 45, 50] # Son una lista de valores de Python

edad_promedio = sum(edades) / len(edades)

edades, edad_promedio
```

```
([18, 19, 23, 45, 50], 31.0)
```

```
# importación canónica: import mylib
# importación por alias: import mylib as mb
# importación por partes: from mylib import foo, bar, sum

# importación por alias
import numpy as np

edades = np.array([18, 19, 23, 45, 50])

# print(edades.count())
print(edades.shape[0])
print(edades.sum())
print(edades.max())
print(edades.min())
print(edades.mean())
print(edades.std())
```

```
5
155
50
18
31.0
13.66747965061591
```

```
import pandas as pd

edades = pd.Series([18, 19, 23, 45, 50])

print(edades.count())
print(edades.sum())
print(edades.max())
print(edades.min())
print(edades.mean())
print(edades.std())
print(edades.quantile(0.25))
```

```
5
155
50
18
31.0
15.280706789936126
19.0
```

303. Operaciones vectorizadas y broadcasting

Operaciones Vectorizadas

En el contexto de la programación numérica, especialmente con bibliotecas como NumPy en Python, las operaciones vectorizadas se refieren al uso de operaciones que actúan sobre arrays completos de una sola vez, en lugar de utilizar bucles sobre los elementos individuales. Esto es fundamentalmente más eficiente gracias al uso de optimizaciones a nivel de bajo código y operaciones que se pueden paralelizar o ejecutar en hardware especializado como CPUs vectoriales o GPUs.

Ventajas de las Operaciones Vectorizadas:

- **Rendimiento Mejorado**: Al evitar explícitamente los bucles en Python, se reduce la sobrecarga interpretativa y se aprovechan las optimizaciones de las bibliotecas subyacentes.
- **Código más Conciso**: El código es generalmente más corto y fácil de entender, pues se describe la operación a nivel de array en lugar de elemento por elemento.

Ejemplo en NumPy:

```
import numpy as np

# Crear dos arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Operación vectorizada: suma de arrays
c = a + b
print(c)  # Output: [5 7 9]
```

Broadcasting

El broadcasting es una extensión poderosa de las operaciones vectorizadas que permite que estas operaciones se realicen en arrays de diferentes tamaños. NumPy, por ejemplo, sigue un conjunto de reglas para "extender" las dimensiones menores de manera que coincidan con las mayores, permitiendo realizar operaciones como si los arrays involucrados tuvieran las mismas dimensiones.

Reglas del Broadcasting:

1. **Compatibilidad de Dimensiones**: Si los arrays no tienen el mismo número de dimensiones, se añaden dimensiones de tamaño 1 al principio del array con menos dimensiones.
2. **Alineación de Dimensiones**: Dos dimensiones son compatibles cuando son iguales o una de ellas es 1. Si una dimensión es 1, se extiende a lo largo de la dimensión del otro array.
3. **Resultado**: El tamaño de cada dimensión en el resultado es el máximo de las dimensiones de los arrays en esa posición.

Ejemplo de Broadcasting en NumPy:

```
import numpy as np

# Array de dimensiones (3,)
a = np.array([1, 2, 3])

# Escalar, que se puede pensar como un array de dimensiones ()
b = 2

# Broadcasting: el escalar se extiende a (3,) y se multiplica con a
c = a * b
print(c)  # Output: [2 4 6]

# Ejemplo más complejo
d = np.array([[1, 2, 3], [4, 5, 6]])  # Dimensiones (2, 3)
e = np.array([1, 2, 3])              # Dimensiones (3,)
# e se "extiende" a (2, 3) replicando el array a lo largo de la primera dimensión
f = d + e
print(f)  # Output: [[2 4 6], [5 7 9]]
```

En resumen, las operaciones vectorizadas y el broadcasting son técnicas fundamentales en la ciencia de datos y el análisis numérico, proporcionando no solo un código más eficiente sino también más legible y conciso. Estas técnicas son ampliamente utilizadas en bibliotecas de Python como NumPy y pandas, y son cruciales para el manejo eficiente de grandes volúmenes de datos.

```
calificaciones = [10, 5, 8, 6, 9, 7, 6, 1, 10]

sum_calif_mayores_8 = 0
total_calif_mayores_8 = 0

for calif in calificaciones:
  if calif >= 8:
    sum_calif_mayores_8 += calif
    total_calif_mayores_8 += 1

print(sum_calif_mayores_8)
print(total_calif_mayores_8)

prom_calif_mayores_8 = sum_calif_mayores_8 / total_calif_mayores_8

print(prom_calif_mayores_8)
```

```
37
4
9.25
```

```
import numpy as np

calificaciones = np.array([10, 5, 8, 6, 9, 7, 6, 1, 10]) # <<< vector de datos y no lista de datos

calificaciones_mayores_mascara = calificaciones >= 8 # operaciones element-wise (elemento-a-elemento)

# (PARA CADA calificación EN calificaciones) >= 8 >>> (Vector de datos donde cada elemento determine ser o no mayor 8)

# Toma cada elemento del arreglo (vector de datos) de calificaciones y aplica la operacióne escalar/singular x >= 8 (implica que x sea un escalar)
# para determinar como resultado un vector con la aplicación en la comparación

calificaciones_mayores_mascara
```

```
array([ True, False,  True, False,  True, False, False, False,  True])
```

```
# Filtro directo por máscara

calificaciones[calificaciones_mayores_mascara] # Aplica la búsqueda (filtro) de los elementos "activos" en la máscara
```

```
array([10,  8,  9, 10])
```

```
mascara_uno = [False, True, False, True, False, False, True, False, True]

calificaciones[mascara_uno]
```

```
array([ 5,  6,  6, 10])
```

```
calificaciones[ calificaciones >= 8 ]
```

```
array([10,  8,  9, 10])
```

```
calificaciones
```

```
array([10,  5,  8,  6,  9,  7,  6,  1, 10])
```

```
calificaciones[ calificaciones >= 8 ].shape[0] / calificaciones.shape[0] * 100
```

```
44.44444444444444
```

```
calificaciones[ calificaciones >= 8 ].mean()
```

```
9.25
```

```
import numpy as np

calificaciones = np.array([10, 5, 8, 6, 9, 7, 6, 1, 10])

calificaciones[ calificaciones >= 8 ].mean()
```

```
9.25
```

```
import pandas as pd

data = pd.read_csv("/content/adult.csv", header=None, names=[
    "EDAD",
    "TIPO_TRABAJO",
    "SIMILARES",
    "EDUCACION_CAT",
    "EDUCACION_NUM",
    "ESTADO_CIVIL",
    "OCUPACION",
    "RELACION",
    "ASCENDENCIA",
    "GENERO",
    "GANANCIA_CAPITAL",
    "PERDIDA_CAPITA",
    "HORAS_TRABAJO_SEMANA",
    "PAIS_NATAL",
    "TOTAL_ANUAL"
])

data.loc[len(data)] = [24, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan]

# data.append({
#     "EDAD": 29,
#     "TIPO_TRABAJO": np.nan,
#     "SIMILARES": np.nan,
#     "EDUCACION_CAT": np.nan,
#     "EDUCACION_NUM": np.nan,
#     "ESTADO_CIVIL": np.nan,
#     "OCUPACION": np.nan,
#     "RELACION": np.nan,
#     "ASCENDENCIA": np.nan,
#     "GENERO": np.nan,
#     "GANANCIA_CAPITAL": np.nan,
#     "PERDIDA_CAPITA": np.nan,
#     "HORAS_TRABAJO_SEMANA": np.nan,
#     "PAIS_NATAL": np.nan,
#     "TOTAL_ANUAL": np.nan,
# }, ignore_index=True)

data
```

	EDAD	TIPO_TRABAJO	SIMILARES	EDUCACION_CAT	EDUCACION_NUM	ESTADO_CIVIL	OCUPACION	RELACION	ASCENDENCIA	GENERO	GANANCIA_CAPITAL	PERDII
0	39.0	State-gov	77516.0	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	2174.0	
1	50.0	Self-emp-not-inc	83311.0	Bachelors	13.0	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.0	
2	38.0	Private	215646.0	HS-grad	9.0	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.0	
3	53.0	Private	234721.0	11th	7.0	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0.0	
4	28.0	Private	338409.0	Bachelors	13.0	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	
32557	40.0	Private	154374.0	HS-grad	9.0	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0.0	
32558	58.0	Private	151910.0	HS-grad	9.0	Widowed	Adm-clerical	Unmarried	White	Female	0.0	
32559	22.0	Private	201490.0	HS-grad	9.0	Never-married	Adm-clerical	Own-child	White	Male	0.0	
32560	52.0	Self-emp-inc	287927.0	HS-grad	9.0	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024.0	
32561	24.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

32562 rows × 15 columns

```
data["TIPO_TRABAJO"].unique()
```

```
array([' State-gov', ' Self-emp-not-inc', ' Private', ' Federal-gov',
       ' Local-gov', ' ?', ' Self-emp-inc', ' Without-pay',
       ' Never-worked', nan], dtype=object)
```

```
# data["TIPO_TRABAJO"] = data["TIPO_TRABAJO"].fillna(' ?')

data["TIPO_TRABAJO_NUM"] = data["TIPO_TRABAJO"].map({
    ' State-gov': 1,
    ' Private': 2,
    ' Self-emp-not-inc': 3,
    ' Federal-gov': 9,
    ' Local-gov': 10,
    ' ?': -1,
    ' Self-emp-inc': 4,
    ' Without-pay': 7,
    ' Never-worked': 8,
    np.nan: 100
})

data
```

	EDAD	TIPO_TRABAJO	SIMILARES	EDUCACION_CAT	EDUCACION_NUM	ESTADO_CIVIL	OCUPACION	RELACION	ASCENDENCIA	GENERO	GANANCIA_CAPITAL	PERDII
0	39.0	State-gov	77516.0	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	2174.0	
1	50.0	Self-emp-not-inc	83311.0	Bachelors	13.0	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.0	
2	38.0	Private	215646.0	HS-grad	9.0	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.0	
3	53.0	Private	234721.0	11th	7.0	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0.0	
4	28.0	Private	338409.0	Bachelors	13.0	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	
32557	40.0	Private	154374.0	HS-grad	9.0	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0.0	
32558	58.0	Private	151910.0	HS-grad	9.0	Widowed	Adm-clerical	Unmarried	White	Female	0.0	
32559	22.0	Private	201490.0	HS-grad	9.0	Never-married	Adm-clerical	Own-child	White	Male	0.0	
32560	52.0	Self-emp-inc	287927.0	HS-grad	9.0	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024.0	
32561	24.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

32562 rows × 16 columns

```
data["ESTADO_CIVIL"].unique()
```

```
array([' Never-married', ' Married-civ-spouse', ' Divorced',
       ' Married-spouse-absent', ' Separated', ' Married-AF-spouse',
       ' Widowed', nan], dtype=object)
```

```
data["ESTADO_CIVIL_NUM"] = data["ESTADO_CIVIL"].map({
    ' Never-married': 1,
    ' Married-civ-spouse': 2,
    ' Married-AF-spouse': 3,
    ' Divorced': 4,
    ' Married-spouse-absent': 5,
    ' Separated': 6,
    ' Widowed': 7,
    np.nan: 1,
})

data
```

	EDAD	TIPO_TRABAJO	SIMILARES	EDUCACION_CAT	EDUCACION_NUM	ESTADO_CIVIL	OCUPACION	RELACION	ASCENDENCIA	GENERO	GANANCIA_CAPITAL	PERDI
0	39.0	State-gov	77516.0	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	2174.0	
1	50.0	Self-emp-not-inc	83311.0	Bachelors	13.0	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.0	
2	38.0	Private	215646.0	HS-grad	9.0	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.0	
3	53.0	Private	234721.0	11th	7.0	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0.0	
4	28.0	Private	338409.0	Bachelors	13.0	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	
32557	40.0	Private	154374.0	HS-grad	9.0	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0.0	
32558	58.0	Private	151910.0	HS-grad	9.0	Widowed	Adm-clerical	Unmarried	White	Female	0.0	
32559	22.0	Private	201490.0	HS-grad	9.0	Never-married	Adm-clerical	Own-child	White	Male	0.0	
32560	52.0	Self-emp-inc	287927.0	HS-grad	9.0	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024.0	
32561	24.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

32562 rows × 17 columns

```
data_analysis = data[["TIPO_TRABAJO_NUM", "ESTADO_CIVIL_NUM", "EDAD"]] # Subdataframe (el DataFrame en las columnas especificadas)

data_analysis
```

	TIPO_TRABAJO_NUM	ESTADO_CIVIL_NUM	EDAD
0	1	1	39.0
1	3	2	50.0
2	2	4	38.0
3	2	2	53.0
4	2	2	28.0
...	...	...	...
32557	2	2	40.0
32558	2	7	58.0
32559	2	1	22.0
32560	4	2	52.0
32561	100	1	24.0

32562 rows × 3 columns

```
data_analysis["TIPO_TRABAJO_NUM"] # Serie (columna o eje de datos de los tipos de trabajo)
```

```
0          1
1          3
2          2
3          2
4          2
         ...
32557      2
32558      2
32559      2
32560      4
32561    100
Name: TIPO_TRABAJO_NUM, Length: 32562, dtype: int64
```

```
data_analysis["TIPO_TRABAJO_NUM"] == 1 # Máscara (columna de True | False)
```

```
0         True
1        False
2        False
3        False
4        False
         ...
32557    False
32558    False
32559    False
32560    False
32561    False
Name: TIPO_TRABAJO_NUM, Length: 32562, dtype: bool
```

```
data_analysis[ data_analysis["TIPO_TRABAJO_NUM"] == 1 ] # data[ <máscara> ] ==> subdata (filtro del original)
```

	TIPO_TRABAJO_NUM	ESTADO_CIVIL_NUM	EDAD
0	1	1	39.0
11	1	2	30.0
34	1	2	22.0
48	1	2	41.0
123	1	2	29.0
...	...	...	...
32360	1	2	58.0
32380	1	4	48.0
32479	1	2	64.0
32540	1	6	45.0
32549	1	4	43.0

1298 rows × 3 columns

```
data_analysis[ (data_analysis["TIPO_TRABAJO_NUM"] == 1) & (data_analysis["EDAD"] <= 30) ]
```

	TIPO_TRABAJO_NUM	ESTADO_CIVIL_NUM	EDAD
11	1	2	30.0
34	1	2	22.0
123	1	2	29.0
171	1	2	28.0
174	1	2	28.0
...	...	...	...
31715	1	1	22.0
31754	1	1	26.0
32126	1	1	23.0
32137	1	1	26.0
32197	1	2	22.0

347 rows × 3 columns

```
data_analysis[ (data_analysis["TIPO_TRABAJO_NUM"] == 2) & ((data_analysis["EDAD"] >= 25) & (data_analysis["EDAD"] <= 30)) ]
```

	TIPO_TRABAJO_NUM	ESTADO_CIVIL_NUM	EDAD
4	2	2	28.0
44	2	1	25.0
49	2	1	29.0
50	2	2	25.0
59	2	2	30.0
...	...	...	...
32523	2	4	30.0
32524	2	1	26.0
32529	2	6	29.0
32537	2	1	30.0
32556	2	2	27.0

3933 rows × 3 columns

```
data_analysis.sample(5)
```

	TIPO_TRABAJO_NUM	ESTADO_CIVIL_NUM	EDAD
27832	4	2	38.0
15917	2	2	23.0
13927	4	2	60.0
18416	2	1	24.0
26128	9	2	44.0

```
data_analysis[(data_analysis["ESTADO_CIVIL_NUM"] == 2)]["EDAD"].mean()
```

```
43.24759615384615
```

```
data_analysis[(data_analysis["ESTADO_CIVIL_NUM"] == 1)]["EDAD"].mean()
```

```
28.150599026581805
```

```
data_analysis[(data_analysis["ESTADO_CIVIL_NUM"] == 3)]["EDAD"].mean()
```

```
32.47826086956522
```

```
data_analysis.loc[[1, 2, 3, 32560]] # .loc hace "filtros" por índice (algoritmo de selección)
```

	TIPO_TRABAJO_NUM	ESTADO_CIVIL_NUM	EDAD
1	3	2	50.0
2	2	4	38.0
3	2	2	53.0
32560	4	2	52.0

```
data_analysis.loc[ (data_analysis["ESTADO_CIVIL_NUM"] == 3) ] # (algoritmo de selección) | Más cómputo & Menos memoria (física)
```

	TIPO_TRABAJO_NUM	ESTADO_CIVIL_NUM	EDAD
37	2	3	19.0
1987	-1	3	28.0
2887	9	3	19.0
3034	2	3	30.0
3787	2	3	26.0
8045	3	3	47.0
8454	2	3	30.0
11481	1	3	30.0
11918	2	3	27.0
13034	2	3	32.0
14138	2	3	43.0
15365	2	3	24.0
16984	-1	3	75.0
18074	9	3	34.0
18895	2	3	35.0
23501	2	3	29.0
25130	2	3	29.0
25386	2	3	26.0
27035	2	3	35.0
27075	3	3	40.0
27228	2	3	38.0
29626	9	3	29.0
29889	2	3	22.0

```
# <máscara 1> & <máscara 2> (AND)
# <máscara 1> | <máscara 2> (OR)
# ~ <máscara 1> (NOT)

data_analysis[ ~(data_analysis["ESTADO_CIVIL_NUM"] == 3) ].info() # (algoritmo de filtrado) | Menos cómputo & Más memoria física
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 32539 entries, 0 to 32561
Data columns (total 3 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
```

```
 0   TIPO_TRABAJO_NUM  32539 non-null  int64
 1   ESTADO_CIVIL_NUM  32539 non-null  int64
 2   EDAD              32539 non-null  float64
dtypes: float64(1), int64(2)
memory usage: 1016.8 KB
```

SPARK | HADOOP (Diseño de índices y particiones)

```
f = open("/content/adult.csv")

data = []

finish = False

while not finish:
  i = 0

  chunks = []

  while i < 1000:
    line = f.readline()
    if line == None:
      finish = True
      break
    chunks.append(line)
    i += 1

  # print(chunks)
  data.append(chunks)

data
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-108-ecaf4b3ef865> in <cell line: 7>()
     11
     12   while i < 1000:
---> 13     line = f.readline()
     14     chunks.append(line)
     15     if line == None:

/usr/lib/python3.10/codecs.py in decode(self, input, final)
    317         raise NotImplementedError
    318
--> 319     def decode(self, input, final=False):
    320         # decode input (taking the buffer into account)
    321         data = self.buffer + input

KeyboardInterrupt:
```

Operaciones reactivas en Python

Para fusionar dos flujos de datos en Python de manera similar a como se hace en ReactiveX (Rx), se necesitan herramientas diseñadas específicamente para manejar flujos de datos de forma reactiva. NumPy, aunque es excelente para operaciones numéricas y de array, no es adecuado para trabajar con flujos de datos en tiempo real o asincrónicos porque está orientado principalmente al cálculo numérico sincrónico y vectorizado.

Para programación reactiva en Python, puedes utilizar bibliotecas como **RxPY** (Reactive Extensions for Python), que es una implementación de ReactiveX para Python. RxPY proporciona funcionalidades para componer y consumir secuencias asincrónicas de eventos a través de operadores declarativos, lo que te permite manejar flujos de datos de manera reactiva.

Ejemplo de cómo fusionar flujos de datos con RxPY

Aquí podemos ver cómo se pueden combinar dos flujos de datos usando RxPY:

1. **Instalación de RxPY**: Primero, necesitas instalar la biblioteca RxPY. Puedes hacerlo usando pip:

   ```
   pip install rx
   ```

2. **Ejemplo de Código**: En este ejemplo, crearemos dos flujos de datos (observables) y los fusionaremos usando el operador `merge`. Luego, suscribiremos a la salida para procesar los datos combinados.

   ```
   import rx
   from rx import operators as ops

   # Crear el primer flujo de datos
   observable1 = rx.from_iterable([1, 2, 3])

   # Crear el segundo flujo de datos
   observable2 = rx.from_iterable([4, 5, 6])

   # Fusionar los dos flujos
   ```

```
merged_observable = rx.merge(observable1, observable2)

# Suscribirse al flujo combinado para imprimir los resultados
merged_observable.subscribe(
    on_next=lambda x: print(f"Recibido: {x}"),
    on_error=lambda e: print(f"Error: {e}"),
    on_completed=lambda: print("¡Proceso completado!")
)
```

En este código:

- `from_iterable` crea un observable a partir de una lista.
- `merge` combina los dos observables en uno solo que emite los ítems de ambos tan pronto como están disponibles.
- `subscribe` permite procesar los eventos emitidos por el observable combinado.

Uso de operadores adicionales

RxPY también soporta una gran variedad de operadores para transformar, filtrar, combinar y manejar flujos de datos de diversas maneras, similar a las operaciones que puedes realizar en ReactiveX en otros lenguajes.

Estos operadores te permiten realizar tareas complejas de manera concisa y expresiva, ideal para escenarios donde los datos son asincrónicos o provienen de múltiples fuentes que deben ser coordinadas o combinadas, como en el procesamiento de eventos de UI, llamadas de API, o el manejo de flujos de datos en tiempo real.

Vamos a ver ejemplos de algunos operadores principales en RxPY que pueden ayudarte a manipular y gestionar flujos de datos en Python de manera reactiva. Estos ejemplos asumen que ya tienes instalada la biblioteca RxPY.

1. `map` (Transformar)

El operador `map` transforma los elementos emitidos por un Observable aplicando una función a cada elemento.

```
import rx
from rx import operators as ops

# Crear un flujo de datos
observable = rx.from_iterable([1, 2, 3])

# Transformar cada elemento multiplicándolo por 2
mapped_observable = observable.pipe(
    ops.map(lambda x: x * 2)
)

# Suscribirse y imprimir los resultados
mapped_observable.subscribe(lambda x: print(f"Mapeado: {x}"))
```

2. `filter` (Filtrar)

El operador `filter` emite solo aquellos elementos de un Observable que cumplen con una condición específica.

```
import rx
from rx import operators as ops

# Crear un flujo de datos
observable = rx.from_iterable([1, 2, 3, 4, 5])

# Filtrar para emitir solo los elementos mayores que 3
filtered_observable = observable.pipe(
    ops.filter(lambda x: x > 3)
)

# Suscribirse y imprimir los resultados
filtered_observable.subscribe(lambda x: print(f"Filtrado: {x}"))
```

3. `reduce` (Reducir)

El operador `reduce` aplica una función acumulativa a los elementos emitidos por el Observable y emite el resultado final una vez que el Observable completa.

```
import rx
from rx import operators as ops

# Crear un flujo de datos
observable = rx.from_iterable([1, 2, 3, 4])

# Reducir sumando todos los elementos
```

```
reduced_observable = observable.pipe(
    ops.reduce(lambda acc, x: acc + x, 0)
)

# Suscribirse y imprimir el resultado
reduced_observable.subscribe(lambda x: print(f"Reducido: {x}"))
```

4. `merge` (Fusionar)

El operador `merge` combina múltiples Observables en uno solo, emitiendo los ítems de cada uno según se van produciendo.

```
import rx

# Crear dos flujos de datos
observable1 = rx.from_iterable([1, 2, 3])
observable2 = rx.from_iterable([4, 5, 6])

# Fusionar los observables
merged_observable = rx.merge(observable1, observable2)

# Suscribirse y imprimir los resultados
merged_observable.subscribe(lambda x: print(f"Merge: {x}"))
```

5. `concat` (Concatenar)

El operador `concat` concatena los outputs de múltiples Observables en orden, empezando por el primer Observable y siguiendo con el siguiente sólo después de que el anterior ha completado.

```
import rx

# Crear dos flujos de datos
observable1 = rx.from_iterable([1, 2, 3])
observable2 = rx.from_iterable([4, 5, 6])

# Concatenar los observables
concatenated_observable = rx.concat(observable1, observable2)

# Suscribirse y imprimir los resultados
concatenated_observable.subscribe(lambda x: print(f"Concat: {x}"))
```

Estos ejemplos muestran cómo utilizar algunos de los operadores más comunes en RxPY para realizar operaciones comunes sobre flujos de datos, desde transformaciones y filtros hasta combinaciones y reducciones.

Generar flujos de datos mediante un genrador

Puedes utilizar una función generadora para crear flujos de datos en RxPY. Las funciones generadoras en Python son una forma conveniente de implementar iteradores que producen una secuencia de valores utilizando la palabra clave `yield`. RxPY permite integrar estas funciones generadoras en su flujo reactiva a través del operador `from_generator`.

Ejemplo usando una función generadora

Vamos a crear un flujo de datos usando una función generadora que emite valores. Luego, usaremos RxPY para trabajar con estos valores de forma reactiva.

1. **Definir la función generadora**: Esta función simplemente genera una secuencia de números, uno a uno.

```
def number_generator():
    for i in range(1, 6):  # Genera números del 1 al 5
        yield i
```

2. **Crear el Observable**: Usa la función `from_generator` de RxPY para convertir la función generadora en un Observable. Necesitarás importar RxPY y especificar la función generadora.

```
import rx
from rx import operators as ops

# Crear Observable desde la función generadora
observable = rx.from_generator(number_generator)
```

3. **Operar sobre el flujo de datos**: Puedes aplicar varios operadores al flujo, como `map` para transformar los datos, `filter` para filtrarlos, etc. Aquí, vamos a multiplicar cada número por 2.

```
# Aplicar operadores
transformed_observable = observable.pipe(
```

```
    ops.map(lambda x: x * 2)  # Multiplicar cada número por 2
)
```

4. **Suscribirse al Observable**: Finalmente, te suscribes al Observable para procesar los valores emitidos.

```
# Suscribirse y imprimir los resultados
transformed_observable.subscribe(lambda x: print(f"Valor procesado: {x}"))
```

Código completo

Aquí está el código completo que muestra cómo integrar una función generadora con RxPY para crear y manipular un flujo de datos.

```
import rx
from rx import operators as ops

def number_generator():
    for i in range(1, 6):  # Genera números del 1 al 5
        yield i

# Crear Observable desde la función generadora
observable = rx.from_generator(number_generator)

# Aplicar operadores
transformed_observable = observable.pipe(
    ops.map(lambda x: x * 2)  # Multiplicar cada número por 2
)

# Suscribirse y imprimir los resultados
transformed_observable.subscribe(lambda x: print(f"Valor procesado: {x}"))
```

Este enfoque es particularmente útil cuando necesitas crear flujos de datos de fuentes que no son colecciones estáticas sino que se generan dinámicamente, permitiendo una integración suave y eficiente dentro del paradigma reactivo de RxPY.

Flujos de datos temporales

Para obtener datos de un generador durante un período de tiempo específico, como 2 segundos, y considerando que cada dato se genera con un intervalo de tiempo (por ejemplo, algunos milisegundos), puedes combinar el uso de RxPY con operadores específicos que gestionen el tiempo y la concurrencia. Vamos a detallar cómo hacerlo paso a paso.

Paso 1: Crear la función generadora con retraso

Primero, definiremos una función generadora que emite valores con un pequeño retraso, simulando la demora en la generación de cada dato.

```
import time

def delayed_number_generator():
    for i in range(1, 101):  # Genera números del 1 al 100
        time.sleep(0.1)  # Retraso de 100 milisegundos
        yield i
```

Paso 2: Configurar el Observable

Para crear un Observable que maneje esta función generadora, podemos utilizar `rx.create()` para tener un control más detallado, permitiéndonos suscribir y desuscribir basado en el tiempo.

```
import rx
from rx.scheduler import ThreadPoolScheduler
import rx.operators as ops
import threading

# Configurar un scheduler para la concurrencia
scheduler = ThreadPoolScheduler(max_workers=1)

def observable_emitter(observer, scheduler):
    for number in delayed_number_generator():
        observer.on_next(number)
    observer.on_completed()

# Crear Observable
observable = rx.create(observable_emitter)
```

Paso 3: Limitar la Emisión a 2 Segundos

Utiliza el operador `take_until_with_time` para limitar la recepción de los datos a un período específico, en este caso, 2 segundos.

```
# Tomar datos por 2 segundos
limited_time_observable = observable.pipe(
    ops.take_until_with_time(2.0, scheduler=scheduler)  # Limitar a 2 segundos
)
```

Paso 4: Suscribirse al Observable

Finalmente, te suscribes al Observable para procesar los valores emitidos dentro del límite de tiempo establecido.

```
# Suscribirse y imprimir los resultados
limited_time_observable.subscribe(
    on_next=lambda x: print(f"Recibido: {x}"),
    on_error=lambda e: print(f"Error: {e}"),
    on_completed=lambda: print("Proceso completado después de 2 segundos.")
)
```

Código Completo

Aquí tienes todo el código en conjunto, listo para ejecutar:

```
import rx
import rx.operators as ops
from rx.scheduler import ThreadPoolScheduler
import time
import threading

def delayed_number_generator():
    for i in range(1, 101):
        time.sleep(0.1)
        yield i

scheduler = ThreadPoolScheduler(max_workers=1)
```