Programación Avanzada con Python

Dr. Héctor Selley

1 de agosto de 2025

Índice general

Capítulo 1

Fundamentos de programación

1.1. Estructura de un programa

Un programa es un conjunto de instrucciones codificadas que una computadora puede ejecutar para realizar una tarea específica o resolver un problema.
Definición 1.1 (Algoritmo)**.** Formalmente definimos un algoritmo como un conjunto de pasos, procedimientos o acciones que nos permiten alcanzar un resultado o resolver un problema.

Los programas están escritos en lenguajes de programación y son creados por programadores. Los programas pueden variar en complejidad, desde simples scripts que realizan una acción básica hasta sistemas operativos completos que gestionan los recursos de una computadora.

1.2. Lenguajes de programación

Un lenguaje de programación es un sistema formal diseñado para expresar instrucciones que una computadora puede interpretar y ejecutar. Estos lenguajes permiten a los programadores escribir programas que realizan tareas específicas, desde operaciones simples hasta aplicaciones complejas. Los lenguajes de programación proporcionan una manera de comunicarse con la computadora utilizando una sintaxis y semántica específica.

Algunas características clave de los lenguajes de programación incluyen:

- **Sintaxis**: El conjunto de reglas que define cómo se deben escribir las instrucciones y declaraciones en el lenguaje.
- **Semántica**: El significado de las instrucciones escritas en el lenguaje.
- **Abstracción**: La capacidad de definir estructuras y operaciones de alto nivel que oculten detalles complejos.
- **Paradigmas**: Los enfoques o estilos de programación que el lenguaje soporta, como la programación estructurada, la programación orientada a objetos, la programación funcional, entre otros.

Existen muchos lenguajes de programación, cada uno con sus propias características y propósitos. Algunos ejemplos comunes incluyen:

- **Python**: Conocido por su sintaxis clara y legible, es ampliamente utilizado en ciencia de datos, desarrollo web, automatización y más.

- **Java**: Utilizado en desarrollo de aplicaciones empresariales, aplicaciones móviles y sistemas integrados.
- **C**: Un lenguaje de bajo nivel que proporciona control detallado sobre el hardware de la computadora, utilizado en sistemas operativos y software de sistemas.
- **JavaScript**: Utilizado principalmente en el desarrollo web para crear aplicaciones interactivas y dinámicas.
- **Ruby**: Conocido por su simplicidad y elegancia, utilizado en desarrollo web y automatización.

Cada lenguaje de programación está diseñado con ciertos objetivos en mente y puede ser más adecuado para ciertos tipos de tareas o proyectos.

1.2.1. Los lenguajes de programación más utilizados

En la figura 1.1 se muestran los lenguajes de programación con mayor demanda en el 2023.

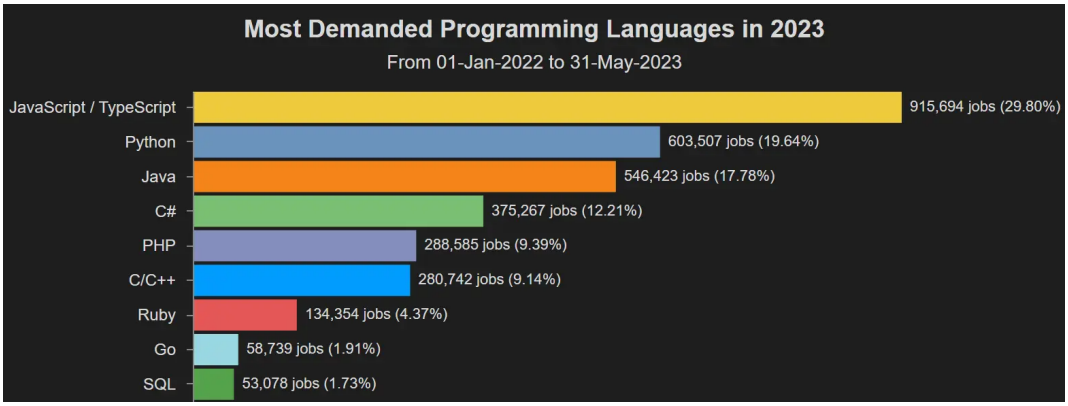


Figura 1.1: Lenguajes más demandados en 2023

1.3. Lenguajes interpretados y compilados

La principal diferencia entre los lenguajes de programación compilados y los interpretados radica en cómo se ejecutan las instrucciones del programa:

1.3.1. Lenguajes de Programación Compilados

1. Proceso de Compilación:
   - Compilador: Un compilador traduce el código fuente completo a código máquina o bytecode antes de que el programa se ejecute. Este proceso se realiza una sola vez y genera un archivo ejecutable independiente.
   - Ejemplo: C, C++, Rust, Go.
2. Ventajas:
   - Rendimiento: Los programas compilados suelen ejecutarse más rápido que los interpretados porque la traducción a código máquina se realiza una sola vez y el ejecutable resultante está optimizado.

- Optimización: Los compiladores pueden realizar optimizaciones avanzadas durante la compilación para mejorar la eficiencia del programa.

3. Desventajas:
   - Tiempo de Compilación: El proceso de compilación puede ser lento, especialmente para programas grandes.
   - Portabilidad: El código compilado suele estar ligado a una plataforma específica, lo que puede dificultar su portabilidad a diferentes sistemas operativos.

1.3.2. Lenguajes de Programación Interpretados

1. Proceso de Interpretación:
   - Intérprete: Un intérprete traduce y ejecuta el código fuente línea por línea en tiempo de ejecución, sin generar un archivo ejecutable intermedio.
   - Ejemplo: Python, JavaScript, Ruby, PHP.
2. Ventajas:
   - Desarrollo Rápido: No es necesario compilar el código antes de ejecutarlo, lo que permite ciclos de desarrollo y pruebas más rápidos.
   - Portabilidad: Los programas interpretados pueden ejecutarse en cualquier plataforma que tenga el intérprete adecuado, lo que mejora su portabilidad.
3. Desventajas:
   - Rendimiento: Los programas interpretados suelen ser más lentos que los compilados porque la traducción a código máquina ocurre en tiempo de ejecución.
   - Optimización: Las oportunidades de optimización en tiempo de ejecución son limitadas en comparación con la compilación previa.

1.3.3. Ejemplos y Consideraciones

Java: Es un caso interesante porque utiliza un enfoque mixto. El código Java se compila en bytecode, que es interpretado por la Máquina Virtual de Java (JVM). Además, la JVM utiliza un compilador JIT (Just-In-Time) para convertir partes del bytecode en código máquina durante la ejecución, combinando ventajas de ambos enfoques.

Python: Aunque es principalmente un lenguaje interpretado, también puede ser compilado a bytecode (.pyc) para ser ejecutado por la máquina virtual de Python, lo que mejora ligeramente su rendimiento, pero no alcanza la velocidad de un lenguaje completamente compilado como C++.

En conclusión, la elección entre un lenguaje compilado y uno interpretado depende de varios factores, incluyendo la necesidad de rendimiento, la rapidez del desarrollo, y la portabilidad del código.

1.4. Python

1.4.1. ¿Qué es Python?

Python es un lenguaje de programación popular. Fue creado por Guido van Rossum y lanzado en 1991.

Se utiliza para:

- Desarrollo web (del lado del servidor),
- Desarrollo de software,
- Matemáticas,
- Inteligencia Artificial,
- Scripting.

1.4.2. ¿Qué puede hacer Python?

- Python se puede utilizar en un servidor para crear aplicaciones web.
- Python se puede utilizar junto con el software para crear flujos de trabajo.
- Python puede conectarse a sistemas de bases de datos. También puede leer y modificar archivos.
- Python se puede utilizar para manejar Big Data y realizar matemáticas complejas.
- Python se puede utilizar para la creación rápida de prototipos o para el desarrollo de software listo para producción.

1.4.3. ¿Por qué Python?

- Python funciona en diferentes plataformas (Windows, Mac, Linux, Raspberry Pi, etc.).
- Python tiene una sintaxis simple similar a la del idioma inglés.
- Python tiene una sintaxis que permite a los desarrolladores escribir programas con menos líneas que otros lenguajes de programación.
- Python se ejecuta en un sistema de interpretación, lo que significa que el código se puede ejecutar tan pronto como se escribe. Esto significa que la creación de prototipos puede ser muy rápida.
- Python se puede tratar de forma procedimental, orientada a objetos o funcional.

1.4.4. Es bueno saber

- La versión principal más reciente de Python es Python 3, que usaremos en este curso. Sin embargo, Python 2, aunque no se actualiza con nada más que actualizaciones de seguridad, sigue siendo bastante popular.
- En este curso, Python se escribirá en un editor de texto. Es posible escribir Python en un entorno de desarrollo integrado, como Thonny, Pycharm, Visual Studio Code, Google Colaboratory, Netbeans o Eclipse, que son particularmente útiles cuando se administran colecciones más grandes de archivos Python.

1.4.5. Sintaxis de Python comparada con otros lenguajes de programación

- Python fue diseñado para facilitar la lectura y tiene algunas similitudes con el idioma inglés con influencia de las matemáticas.
- Python usa nuevas líneas para completar un comando, a diferencia de otros lenguajes de programación que suelen usar punto y coma o paréntesis.

- Python se basa en la sangría, utilizando espacios en blanco, para definir el alcance; como el alcance de los bucles, funciones y clases. Otros lenguajes de programación suelen utilizar llaves para este propósito.

1.4.6. Instalación de Python

Muchas PC y Mac ya tienen Python instalado.

Para comprobar si tiene Python instalado en una PC con Windows, busque Python en la barra de inicio o ejecute lo siguiente en la línea de comandos (cmd.exe):

```
C:\Usuarios\Su nombre>python --version
```

Para verificar si tiene Python instalado en Linux o Mac, en Linux abra la línea de comando o en Mac abra la Terminal y escriba:

```
python --versión
```

Si descubre que no tiene Python instalado en su computadora, puede descargarlo de forma gratuita desde el siguiente sitio web: https://www.python.org/

1.4.7. Mi primer programa en Python

```
print("Hola mundo")
```

1.5. Funciones

Una función es un bloque de código que sólo se ejecuta cuando se llama. Puede pasar datos, conocidos como parámetros, a una función. Una función puede devolver datos como resultado.

1.5.1. Declarar una función

En Python una función se define usando la palabra clave `def`:

```
def mi_función():
    print("Hola desde una función")
```

1.5.2. Llamar a una función

Para llamar a una función, use el nombre de la función seguido de paréntesis:

```
def mi_función():
    print("Hola desde una función")

mi_función()
```

1.5.3. Argumentos

La información se puede pasar a funciones como argumentos. Los argumentos se especifican después del nombre de la función, dentro del paréntesis. Puedes agregar tantos argumentos como quieras, simplemente sepáralos con una coma.

El siguiente ejemplo tiene una función con un argumento (fname). Cuando se llama a la función, pasamos un nombre, que se usa dentro de la función para imprimir el nombre completo:

```
def mi_función(fname):
    print(fnombre + " Refsnes")

mi_funcion("Emil")
mi_función("Tobías")
mi_función("Linus")
```

1.5.4. Argumentos arbitrarios, *args

Si no sabe cuántos argumentos se pasarán a su función, agregue un __*__ antes del nombre del parámetro en la definición de la función. De esta manera, la función recibirá una tupla de argumentos y podrá acceder a los elementos en consecuencia:

```
def mi_funcion(*ninos):
    print("El hijo menor es " + ninos[2])

mi_funcion("Emil", "Tobías", "Linus")
```

1.6. Módulos de Python

1.6.1. ¿Qué es un módulo?

Considere que un módulo es lo mismo que una biblioteca de código. Un archivo que contiene un conjunto de funciones que desea incluir en su aplicación.

1.6.2. Crear un módulo

Para crear un módulo simplemente guarde el código que desea en un archivo con la extensión de archivo `.py`:

Ejemplo Guarde este código en un archivo `llamadomymodule.py`

```
def greeting(name):
    print("Hello, " + name)
```

1.6.3. Utilice un módulo

Ahora podemos utilizar el módulo que acabamos de crear, mediante la importdeclaración: **Ejemplo** Importa el módulo llamado mymodule y llama a la función de saludo:

```
    import mymodule

    mymodule.greeting("Jonathan")
```

1.6.4. Variables en el módulo

El módulo puede contener funciones, como ya se ha descrito, pero también variables de todo tipo (matrices, diccionarios, objetos, etc.):

Ejemplo Guarda este código en el `archivomymodule.py`

```
person1 = {
    "name": "John",
    "age": 36,
```

```
    "country": "Norway"
}
```

Ejemplo Importe el módulo llamado `mymodule` y acceda al diccionario `person1`:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

1.6.5. Nombrar un módulo

Puedes nombrar el archivo del módulo como quieras, pero debe tener la extensión de archivo `.py`

1.6.6. Cambiar el nombre de un módulo

Puedes crear un alias al importar un módulo, utilizando la aspalabra clave:

Ejemplo Crear un alias para mymoduleel llamado `mx`:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

1.6.7. Módulos integrados

Hay varios módulos integrados en Python que puedes importar cuando quieras.

Ejemplo Importar y utilizar el `platformmódulo`:

```
import platform

x = platform.system()
print(x)
```

1.6.8. Usando la función dir()

Hay una función incorporada para enumerar todos los nombres de funciones (o nombres de variables) en un módulo. La función `dir()`:

Ejemplo Enumere todos los nombres definidos que pertenecen al módulo de la plataforma:

```
import platform

x = dir(platform)
print(x)
```

1.6.9. Importar desde módulo

Puede elegir importar solo partes de un módulo, utilizando la frompalabra clave.

Ejemplo El módulo nombrado `mymodule` tiene una función y un diccionario:

```
def greeting(name):
    print("Hello, " + name)
```

```
person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}
```

Ejemplo Importe únicamente el diccionario `person1` del módulo:

```
from mymodule import person1

print (person1["age"])
```

1.7. Referencias

- Python en la W3Schools
- Python.org
- chatGPT en openAI
- Algoritmos y Programación
- Sintaxis básica en Python

Capítulo 2

Sintáxis Básica de Python

2.1. Python

Python es un lenguaje de programación poderoso y rápido, se lleva bien con otros lenguajes, corre en cualquier lugar, es amigable, fácil de aprender y es de software libre.

2.2. Sintáxis

En la gran mayoría de los lenguajes de programación el compilador o intérprete ignora los espacios que el usuario utilice para escribir su código. En Python, los espacios se utilizan para formar grupos de instrucciones y también para la sintaxis de algunas instrucciones.

2.3. If-Elif-Else

La sentencia de control `if-elif-else` tiene la siguiente sintaxis:

Sintaxis:

```
if condición A:
    instrucción A1
    instrucción A2
    ...
    instrucción An
elif condición B:
    instrucción B1
    instrucción B2
    ...
    instrucción Bn
else:
    instrucción C1
    instrucción C2
    ...
    instrucción Cn
```

Código 2.1. if-else

```
if 2<3:
    print("verdadero")
else:
    print("falso")
print("fin")
```

Código 2.2. Solicite dos números x y y. Si x es positivo y se multiplica por dos, si x es cero y se multiplica por 3 y si x es negativo y se multiplica por cuatro.

```
x=int(input("Ingrese el valor de x "))
y=int(input("Ingrese el valor de y "))
if x>0:
    z=y*2
    print("Positivo")
elif x==0:
    z=y*3
    print("Cero")
else:
    z=y*4
    print("Negativo")
print(z)
```

2.4. For

El ciclo `for` tiene la siguiente sintaxis:

```
for iterador in iterando:
    instrucción 1
    instrucción 2
    ...
    instrucción n
```

Código 2.3. Un ciclo que imprime numeros enteros desde 0 hasta 9.

```
for i in range(10):
    print(i)
```

La función `range` genera una secuencia de números enteros comenzando desde 0, con un incremento unitario. Por esa razón es que se generan los números enteros desde 0 hasta 9. Es posible cambiar el valor inicial de la secuencia y el incremento.

Código 2.4. Considere los primeros 10 números naturales, los pimeros 5 urales se multiplican por 2 y los siguientes números naturales se multiplican por 3.

```
for x in range(1,11):
    if x<=5:
        print(f'{x} -> {x*2}')
    else:
        print(f'{x} -> {x*3}')
```

2.5. While

El ciclo `while` repite las instrucciones siempre y cuando la condición sea verdadera. La sintaxis del ciclo `while` es la siguiente:

```
while condición:
    instrucción 1
    instrucción 2
    ...
    instrucción n
```

Código 2.5. Se imprimen los primeros 10 numeros naturales.

```
i = 1
while i <= 10:
    print(i)
    i=i+1
```

2.6. Listas

2.6.1. Creación

Una lista es un conjunto de valores ordenados y modificables que permite repeticiones. Los elementos en la lista son numerados comenzando desde 0, de manera que la localidad donde se encuentra el último elemento para una lista de tamaño `n` será `n-1`.

```
lista = ["manzana", "plátano", "naranja", "mandarina", "maracuyá"]
print(lista)
```

2.6.2. Acceso

El acceso a algún elemento en particular de la lista se hace utilizando los corchetes, indicando dentro de ellos la posición que ocupa dentro de la lista.

```
lista[posición]
```

2.6.3. Índices negativos

Es posible usar numeración negativa para hacer referencia a los elementos dentro de una lista pero en orden reverso. Es decir, `-1` hace referencia al último elemento, `-2` al penúltimo, `-3` al antepenúltimo y así sucesivamente.

2.6.4. Rango de índices

Es posible seleccionar un subconjunto de elementos contiguos contenidos en la lista especificando el rango de posiciones que ocupan dentro de la lista.

```
lista = ["manzana", "plátano", "naranja", "mandarina", "maracuyá",
         "toronja", "mango", "guayaba"]
print(lista[2:6])
```

Por otro lado, si se omite el límite inferior del rango, python considerará 0 como posición inicial.

```
lista = ["manzana", "plátano", "naranja", "mandarina", "maracuyá",
         "toronja", "mango", "guayaba"]
print(lista[:6])
```

Finalmente, si se omite el límite superior del rango, python considerará el último elemento como tal límite.

```
lista = ["manzana", "plátano", "naranja", "mandarina", "maracuyá",
        "toronja", "mango", "guayaba"]
print(lista[2:])
```

2.6.5. Tamaño de una lista

Para determinar el tamaño de una lista se puede utilizar la función `len()`

```
print(len(lista))
```

2.7. Funciones

En python la declaración de funciones requiere muy poco código. Basta con utilizar la palabra clave `def` para comenzar la definición de la función.

```
def nombreFuncion(parámetroEntrada):
    instrucción 1
    instrucción 2
    ...
    instrucción n
    return parámetroSalida
```

Código 2.6. Escriba una función que calcule el factorial de una función. Verifique que el número ingresado por el usuario sea positivo y considere que por definición el factorial de cero es uno.

```
def factorial(x):
    if x>=0:
        f=1
    if x>0:
        f=1
        for i in range(1,x+1):
        f=f*i
        return f
    else:
        print("El factorial no está definido")

x = int(input("Introduzca un número: "))
print(f'{x}! = {factorial(x)}')
```

Código 2.7. Escriba una función que calcule el factorial de una función. Considere la definción recursiva del factorial. Verifique que el número ingresado por el usuario sea positivo y considere que por definición el factorial de cero es uno.

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

x = int(input("Ingrese un número"))
if x>=0:
    fac=factorial(x)
```

```
        print(f'{x}! = {fac}')
else:
    print("El factorial no está definido en los negativos")
```

Código 2.8. Escriba una función recursiva que imprima los elementos de una lista anidada en varios niveles.

```
lista1 = ["manzana", "plátano", "naranja", "mandarina", "maracuyá", "toronja",
        "mango", "guayaba"]
lista2 = [1, 2, 3, 4, 5]
lista3 = ["A", "B", "C", "D"]
lista4 = ["a", 1, "b", 2, "c", 3, "d", 4]
superLista = [lista1 + lista2 + lista3] + [[lista4]] + lista1

def impresionRecursiva(listaAnidada):
for elemento in listaAnidada:
    if isinstance(elemento, list):
        impresionRecursiva(elemento)
    else:
        print(elemento)

impresionRecursiva(superLista)
```

2.8. Arreglos

Python no tiene de forma nativa soporte para arreglos, en su lugar opta por usar listas anidadas. Sin embargo, es posible utilizar el paquete NumPy para utilizar los arreglos de manera semejante a la existente en otros lenguajes. **NumPy** además resulta ser más eficiente en el manejo de datos que su contraparte nativa de python mediante listas anidadas. Adicionalmente, **NumPy** incluye más herramientas que extienden la funcionalidad de Python.

2.8.1. Declaración

Sintaxis:

```
import numpy
arreglo = numpy.array([1, 2, 3, 4, 5])
print(arreglo)
```

Esta sintaxis puede resultar incómoda porque será necesario escribirla todas las veces que necesite declarar un arreglo. Una alternativa para simplificar un poco esta declaración es mediante la creación de un alias, esto de la siguiente manera:

```
import numpy as np
arreglo = np.array([1, 2, 3, 4, 5])
print(arreglo)
```

Para declarar un arreglo bidimensional se utiliza la siguiente sintáxis:

```
import numpy as np
arreglo = np.array([[1, 2, 3], [4, 5, 6]])
print(arreglo)
```

El atributo `ndim` devuelve la cantidad de dimensiones que tiene un arreglo.

```
import numpy as np
arreglo = np.array([[1, 2, 3], [4, 5, 6]])
print(arreglo.ndim)
```

2.8.2. Acceso a elementos de un arreglo

El acceso a elementos dentro de un arreglo en `numpy` es similar a la forma que se utiliza para las listas. Recuerde que elos índice para los elementos dentro del arreglo comienza en 0.

```
import numpy as np
arreglo = np.array([1, 2, 3, 4, 5])
print(arreglo[0])
```

Para el caso de un arreglo bidimensional se utiliza una coma para separar la posición de las dimensiones.

```
import numpy as np
arreglo = np.array([[1, 2, 3], [4, 5, 6]])
print(arreglo[1, 2])
```

Si el arreglo tiene más dimensiones se utiliza la misma idea para cada una de ellas.

```
import numpy as np
arreglo = np.array([[[1, 2, 3], [4, 5, 6]],[[7, 8, 9], [10, 11, 12]])
print(arreglo[0, 1, 2])
```

De la misma forma que con las listas, también es posible utilizar índices negativos.

```
arreglo = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('El último elemento en el arreglo bidimensional', arreglo[1, -1])
```

2.8.3. Cortes de arreglos

Es posible *cortar* un subconjunto de un arreglo para definir uno nuevo. Esto es de especial utilidad para extraer vectores de una matriz existente, ya sea para definir un nuevo vector o bien realizar operaciones con el.

El corte (o rebanada) de la matriz se hace indicando un rango de posiciones, es decir `[inicio : fin]`. Además se puede especificar un incremento `[inicio : fin : incremento]`. Si no se especifica un inicio, de asume como 0, y si no se especifica un final se asume el último elemento de la matriz. Si no se especifica un incremento, se asume como 1.

```
import numpy as np
arreglo = np.array([1, 2, 3, 4, 5])
arreglo2 = arreglo[1:4]
print(arreglo2)
```

En el siguiente ejemplo, se hace una rebanada de la matriz especificando un incremento diferente a uno:

```
import numpy as np
arreglo = np.array([1, 2, 3, 4, 5, 6, 7])
print(arreglo[1:5:2])
```

2.8.4. Cortes de arreglos bidimensionales

Es posible realizar rebanadas de arreglos de 2 ó más dimensiones, resultando un vector o una matriz según sea el caso.

El siguiente ejemplo realiza una rebanada de un arreglo bidimensional y el resultado es un vector. Observe que el vector es una rebanada de la segunda dimensión de la matriz.

```
import numpy as np
arreglo = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arreglo[1, 1:4])
```

En este ejemplo se hace una rebanada de un arreglo bidimensional y el resultado es nuevamente un vector. En esta ocasión el vector es una rebanada vertical por lo que el resultado contiene elementos de ambas dimensiones de la matriz.

```
import numpy as np
arreglo = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arreglo[0:2, 2])
```

En este último caso, se hace una rebanada que resulta una matriz que contiene elementos de ambas dimensiones de la matriz original.

```
import numpy as np
arreglo = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arreglo[0:2, 1:4])
```

2.8.5. Arreglos aleatorios

Es posible generar un arreglo o matriz lleno de números (pseudo) aleatorios, para ello se puede utilizar el método `rand`. Para la generación, basta con especificar las dimensiones del arreglo que se desea generar.

Código 2.9. Generación de un vector unidimensional de 20 elementos aleatorios.

```
import numpy as np
arreglo = np.random.rand(20)
print(arreglo)
```

Código 2.10. Generación de una matriz bidimensional de tamaño `5x3`, es decir 5 renglones y 3 columnas.

```
import numpy as np
arreglo = np.random.rand(5, 3)
print(arreglo)
```

Código 2.11. Generación de una matriz tridimensional de tamaño `5x3x2`, es decir 5 matrices de 3 renglones y 2 columnas cada una de ellas.

```
import numpy as np
arreglo = np.random.rand(5, 3, 2)
print(arreglo)
```

2.8.6. Añadir elementos a un arreglo

Para añadir elementos al final de un arreglo, se puede utilizar el método `append` contenido en la biblioteca `Numpy`. El argumento `axis` permite especificar el lugar donde serán añadidos los elementos al arreglo.

Código 2.12. Añadir una matriz al final de una matriz, justo debajo de la matriz inicial (`axis=0`).

```
import numpy as np
arreglo = np.append([[1, 2], [3, 4]], [[10, 20], [30, 40]], axis=0)
print(arreglo)
```

Código 2.13. Añadir una matriz al final de la matriz, a la derehca de la matriz inicial (`axis=1`).

```
import numpy as np
arreglo = np.append([[1, 2], [3, 4]], [[10, 20], [30, 40]], axis=1)
print(arreglo)
```

2.8.7. Formateo de impresión de un arreglo

Si se desea controlar la forma en la que los números contenidos en un arreglo serán impresos, se puede utilizar el método `printoptions` de la biblioteca `Numpy`.

Código 2.14. Generar una matriz bidimensional de tamaño 10x5 llena con números aleatorios no enteros con valores entre 0 y 1. Imprimir la matriz mostrando únicamente 4 cifras significativas no enteras.

```
import numpy as np
x = np.random.rand(10,5)
    with np.printoptions(precision=4, suppress=True):
        np.set_printoptions(formatter={'float': '{: 0.4f}'.format})
        print(x)
```

La función `with` permite especificar el formato mediante el cual serán impresos unicamente los números en el arreglo, dejando intactos los parámetros de los demás `print` que pudiesen existir. El parámetro `suppress=True` indica que los números deben ser impresos en la forma de punto flotante y evitando la notación científica.

Capítulo 3

Git

Figura 3.1: Git

Git es un sistema de control de versiones de software libre diseñado para manejar desde proyectos pequeños hasta muy grandes con rapidez y eficiencia.

Git es facil de aprender y es liviano con rápido desempeño. Supera a software similares como Subversion, CVS, Perforce y ClearCase gracias a características como manejo de ramas locales, areas de "stagingz múltiples flujos de trabajo.

Git realiza un control de versiones del código, esto quiere decir que almacena el código escrito en todas sus veriones haciendo un registro de los cambios realizados, cuando y quien los hizo. Es posible además volver a un estado anterior del código. Git realiza un registro de los cambios en el código y almacena un `snapshot` del código pudiendo regresar a una versión previa con facilidad.

Aquí puede descargar la versión de Git que corresponda para su sistema:

- Windows
- macOS

3.1. Mini Tutorial Git

3.1.1. Estados de Git

Git tiene tres estados para el código:

1. **Working Directory**: El directorio de trabajo, lugar donde se encuentra el código que estamos escribiendo.
2. **Staging Area**: Archivos de código listos para ser llevados al repositorio.
3. **Respository**: Archivos dentro del repositorio.

3.1.2. Configuración inicial de Git

Git está disponible para instalar en Windows, Linux y macOS. Descarge e instale la versión que corresponda para su sistema operativo.

Una vez instalado, puede acceder a Git a través de la terminal en Linux y macOS o bien a través de GitBash en Windows.

Recién instalado Git, es necesario configurar el nombre y correo del usuario. El siguiente comando asignará el nombre de usuario, este nombre es el que quedará registrado en cada commit que se haga en el repositorio.

```
$ git config --global user.name "Juan Perez"
```

Si ejecuta el comando sin argumentos, mostrará el nombre que se encuentra configurado actualmente:

```
$ git config --global user.name
```

Para registrar el correo del usuario el proceso es similar, el comando es el siguiente:

```
$ git config --global user.email "juan.perez@correo.com"
```

Si ejecuta el comando sin argumentos, mostrará el correo que se encuentra configurado actualmente:

```
$ git config --global user.email
```

La terminal/WindowsBash puede mostrar los resultados de Git en colores, esto facilita su lectura. Para habilitar esta característica basta con ejecutar el comando:

```
$ git config --global color.ui true
```

Para ver la configuración actual de Git, ejecute el siguiente comando:

```
$ git config --global --list
```

```
$ cat .gitconfig
```

3.1.3. Comenzando con Git

El comando `git help` muestra información del manual de git para algún comando específico.

```
$ git help comando
```

El comando `git init` inicializa el proyecto. Indica a Git que este es el "Working Directory", que este directorio tiene el código que habrá de guardarse en el repositorio. Debe usar este comando cada que comienza un proyecto nuevo.

```
$ git init
```

El comando `git stauts` muestra el estado actual del repositorio.

```
$ git status
```

3.1.4. Añadir archivos al Staging Area

Para añadir archivos al Staging Area se usa el comando git add. Puede agregarse un archivo o varios a la vez.

El siguiente comando añade el archivo file.txt al staging:

```
$ git add file.txt
```

El siguiente comando añade todos los archivos en el directorio actual:

```
$ git add -A
```

3.1.5. Añadir archivos al Repository

Para añadir archivos al Repository se utiliza el comando `git commit`. Este comando permite agregar un mensaje, el cual nos permite especificar el cambio que hemos realizado en el código. Este mensaje es para nosotros mismos como desarrolladores, ya que en un futuro que consultemos los cambios, podremos saber con precisión que cambio fue realizado en esa etapa del desarrollo del código. Por esa razón se recomienda que el mensaje sea claro y conciso.

```
$ git commit -m "Mensaje claro y conciso que describe el cambio en el código"
```

Una vez que se haga algún cambio en el código o se agreguen archivos, haga un `git status` y este indicará que el código en el *Working Directory* difiere del que se encientra en el *Repository*. Este será el momento para hacer un `git add` para los archivos modificados y un `git commit` nuevamente.

Es posible utilizar el comando `git commit` sin más argumentos. Esto llevará los cambios del Staging Area al Repository, pero dado que no se especificó un mensaje, se abrirá el editor por defecto en el sistema (`vi` en los sistemas *NIX) para escribir el mensaje correspondiente para el *commit*.

3.1.6. Bitácora de Cambios

Uno de las grandes ventajas de utiliar Git es que guarda un registro de los cambios y un `snapshot` del código en el momento del `commit`. Para ver el registro se utiliza el comando `git log`.

```
$ git log
```

3.1.7. Ver estados anteriores del código

El comando `git checkout` permite ver una versión específica del código para la ocurrencia de un *commit* específico. Se dice que este comando permite "*viajar en el tiempo*"del código. Este comando requiere de un identificador SHA (Secure Hash Algorithm) del *commit*, el cual podemos ver en la bitácora. Por ejemplo:

```
$ git checkout [<commit>]
```

Este comando llevará el código al estado en el que se encontraba al momento de haber hecho el coommit correspondiente al ID. De esta forma podremos examinar el código en ese momento y llevar a cabo cualquier acción que deseemos con el.

Si desearamos viajar nuevamente a otro estado anterior del código podríamos hacerlo con `git chechout ID`, de acuerdo al ID específico a donde quisieramos ir. Por otro lado, si quisieramos regresar al último commit realizado (antes del primer `git checkout`), basta con escribir:

```
$ git checkout master
```

3.1.8. Regresar a estados anteriores del código.

Una de las funcionalidades de Git es que permite ver y regresar a estados anteriores del código. Esto es útil por si algún commit tuviera un error peligroso o cambios no deseados. Utilice esta instrucción con precaución.

Existen cinco tipos de `reset`: `soft`, `mixed`, `hard`, `merge` y `keep`. A continuación se describne los más comunes:

soft Mantiene los cambios de nuestros archivos intacto, simplemente es para que Git tenga presente que está en otro *commit*.

mixed Mantiene nuestros archivos, pero limpia el index de git de las cambios realizadas.

hard Elimina todo los cambios que tenemos en nuestros archivos para dejarlo exactamente igual que en el repositorio.

3.2. GitHub

Figura 3.2: GitHub

GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Utiliza el framework Ruby on Rails por GitHub, Inc. (anteriormente conocida como Logical Awesome).

GitHub es una plataforma de desarrollo inspirada en su forma de trabajo. Usted puede almacenar y revisar código, administrar proyectos y construir software desde código abierto hasta empresarial junto a millones de desarrolladores.

Desde enero de 2010, opera bajo el nombre de GitHub, Inc. El código se almacena de forma pública, aunque también se puede hacer de forma privada, creando una cuenta de pago.

GitHub es un sitio que crea una comunidad de desarrolladores, se puede decir que es la red social de los desarrolladores.

3.3. MiniTutorial Git + GitHub

3.3.1. Clonar un repositorio

Si deseamos descargar todos los archivos de código de un proyecto que se encuentra en GitHub, podemos hacer una copia a través del comando:

```
$ git clone URL
```

Este comando es útil si nos interesa obtener el código y no necesariamente hacer contribuciones.

3.3.2. Manipular repositorios remotos

Para vincular un proyecto de código local a un repositorio remoto en GitHub se utiliza el comando *git remote*.

```
$ git remote add origin URL_Repo
```

Este comando establecerá un vinculo entre el código del repositorio local *origin* con el repositorio remoto que se encuentra en GitHub a través de *URL_Repo*.

El siguiente comando mostrará si existe un vinculo entre el repositorio local y uno remoto:

```
$ git remote -v
```

El siguiente comando permite eliminar el vinculo que exista entre el repositorio local y remoto:

```
$ git remote remove origin
```

Se puede comprobar el efecto de estas operaciones con el comando `git remote -v`.

El comando `git push` permite subir el código que se encuentra en el repositorio local al repositorio remoto en GitHub.

```
$ git push origin master
```

El comando solicitará el *username* y *password* de la cuenta de GitHib donde se encuentra el repositorio remoto.

El comando `git push origin master` puede ejecutarse cada vez que se hagan cambios en el código, de manera que los repositorios local y remoto se encuentren en sincronía.

Tenga en cuenta que este comando sincroniza la rama *master* del repositorio local con el repositorio remoto en GitHub.

3.4. Referencias

- Git
- GitHub
- GitLab

Capítulo 4

Clases y Objetos

Python es un lenguaje de programación orientado a objetos. Casi todo en Python es un objeto, con sus propiedades y métodos. Una clase es como un constructor de objetos, o un "*blueprint*" para crear objetos.

4.1. Crear una Clase

Para crear una clase, use la palabra clave `class`:
Código 4.1. Cree una clase llamada `MyClass`, con una propiedad llamada `x`:

```
class MyClass:
    x = 5
```

4.2. Crear Objeto

Ahora podemos usar la clase llamada `MyClass` para crear objetos:
Código 4.2. Cree un objeto llamado `p1` e imprima el valor de `x`:

```
p1 = MyClass()
print(p1.x)
```

4.3. La función `__init__()`

Los ejemplos anteriores son clases y objetos en su forma más simple, y son no es realmente útil en aplicaciones de la vida real.

Para entender el significado de las clases tenemos que entender el `__init__()` incorporado función.

Todas las clases tienen una función llamada `__init__()`, que siempre se ejecuta cuando la clase está siendo iniciada.

Utilice la función `__init__()` para asignar valores a propiedades de objetos u otros operaciones que son necesarias para hacer cuando el objeto está siendo creado:
Código 4.3. Cree una clase llamada Persona, use la función `__init__()` para asignar valores para nombre y edad:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Nota: La función `__init__()` se llama automáticamente cada vez que la clase se utiliza para crear un nuevo objeto.

En el contexto de la programación orientada a objetos en Python, el término `self` se refiere a la instancia actual de la clase. Es un parámetro que se utiliza en los métodos de una clase para acceder a las variables y métodos del objeto.

El método `__init__` es un constructor especial en Python. Se llama automáticamente cuando se crea una nueva instancia de la clase. Recibe los parámetros `name` y `age`.

Dentro del constructor, `self` entra en juego. El término `self` se refiere a la instancia actual de la clase. Al asignar `self.name = name` y `self.age = age`, estás almacenando los valores `name` y `age` en los atributos `name` y `age` de la instancia actual de la clase `Person`.

4.4. La función `__str__()`

La función `__str__()` controla lo que debe devolverse cuando el objeto de clase se representa como una cadena.

Si la función `__str__()` no está establecida, la representación de cadena del objeto se devuelve:
Código 4.4. La representación de cadena de un objeto SIN la función `__str__()`:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1)
```

Código 4.5. La representación de cadena de un objeto CON la función `__str__()`:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)
```

```
print(p1)
```

4.5. Métodos de Objeto

Los objetos también pueden contener métodos. Los métodos en los objetos son funciones que pertenecen al objeto.

Creemos un método en la clase `Person`:

Código 4.6. Inserte una función que imprima un saludo y ejecútelo en el objeto `p1`:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Nota: El parámetro `self` es una referencia a la instancia actual de la clase, y se utiliza para acceder a las variables que pertenecen a la clase.

Código 4.7. Registro de perros con promedio de peso.

```
class Perro:
    peso = 30

    def __init__(self, peso):
        self.peso = peso

    @classmethod
    def get_peso_promedio(cls):
        return cls.peso

labrador = Perro(25)
print(f'El peso de un perro labrador es {labrador.peso} kilos')
# El peso de un perro labrador es 25 kilos
print(f'El peso promedio de un perro es {Perro.get_peso_promedio()} kilos')
# El peso promedio de un perro es 30 kilos
```

4.6. El parámetro `self`

El parámetro `self` es una referencia a la instancia actual de la clase, y se utiliza para acceder a las variables que pertenecen a la clase.

No tiene que ser nombrado `self` , puede ser llamado como sea, pero tiene que ser el primer parámetro de cualquier función en la clase:

Código 4.8. Usar las palabras `mysillyobject` y `abc` en lugar de `self`:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

4.7. Modificar Propiedades de Objeto

Se pueden modificar propiedades en objetos como este:

Código 4.9. Establezca la edad de `p1` a 40:

```
p1.age = 40
```

4.8. Eliminar Propiedades de Objeto

Puede eliminar propiedades en objetos utilizando el del palabra clave `del`:

Código 4.10. Eliminar la propiedad `age` del objeto `p1`:

```
del p1.age
```

4.9. Eliminar Objetos

Puede eliminar objetos utilizando el del palabra clave `del`:
Código 4.11. Eliminar el objeto `p1`:

```
del p1
```

4.10. La sentencia `pass`

La declaración de una clase no puede quedar vacía, pero si por alguna razón es necesario dejar vacía dicha declaración, se puede emplear la palabra clave `pass`.

Ejemplo

```
class Person:
  pass
```

4.11. Herencia

La herencia nos permite definir una clase que hereda todos los métodos y propiedades de otra clase. La clase padre es la clase de la que se hereda, también llamada clase base. La clase hija es la clase que

hereda de otra clase, también llamada clase derivada.

4.11.1. Crear una clase padre

Cualquier clase puede ser una clase padre, por lo que la sintaxis es la misma que la de crear cualquier otra clase:

Código 4.12. Crea una clase llamada `Person`, con propiedades `firstnamey` `lastname`, y un método `printname`:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

4.11.2. Crear una clase hija

Para crear una clase que herede la funcionalidad de otra clase, envíe la clase principal como parámetro al crear la clase secundaria:

Código 4.13. Cree una clase llamada `Student`, que heredará las propiedades y métodos de la clase `Person`:

```
class Student(Person):
  pass
```

Nota: Utilice la `pass` palabra clave cuando no desee agregar ninguna otra propiedad o método a la clase.

Ahora la clase `Student` tiene las mismas propiedades y métodos que la clase `Person`.

Código 4.14. Utilizar la clase `Student` para crear un objeto y luego ejecute el método `printname`:

```
x = Student("Mike", "Olsen")
x.printname()
```

4.11.3. Agregue la función init()

Hasta ahora hemos creado una clase hija que hereda las propiedades y métodos de su clase principal. Queremos agregar la función `__init__()` a la clase hija (en lugar de la palabra clave `pass`).

Nota: La función `__init__()` se llama automáticamente cada vez que se utiliza la clase para crear un nuevo objeto.

Código 4.15. Añade la función `__init__()` a la clase `Student`:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

Cuando se agrega la función `__init__()`, la clase hija ya no heredará la función `__init__()` de la clase padre.

Nota: La función `__init__()` del hijo reemplaza la herencia de la función del padre `__init__()`.

Si se desea mantener la herencia de la función padre `__init__()`, agregue una llamada a la función padre `__init__()`:

Código 4.16. Mantener la herencia del objeto padre.

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Ahora hemos agregado exitosamente la función `__init__()` y conservamos la herencia de la clase padre, y estamos listos para agregar funcionalidad en la función `__init__()`.

4.11.4. La función super()

Python también tiene una función super() que hará que la clase hija herede todos los métodos y propiedades de su clase padre:

Código 4.17. Herencia de los métodos del objeto padre.

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

Al utilizar la función super(), no es necesario utilizar el nombre del elemento padre, ya que heredará automáticamente los métodos y propiedades de su padre.

4.11.5. Agregar propiedades

Código 4.18. Añade una propiedad llamada `graduationyear` a la clase `Student`:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

En el ejemplo siguiente, el año 2019 debe ser una variable y pasarse a la clase `Student` al crear objetos de estudiantes. Para ello, agregue otro parámetro en la función `__init__()`:

Código 4.19. Agregue un parámetro `year` y pase el año correcto al crear objetos:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

4.11.6. Agregar métodos

Código 4.20. Agregue un método llamado `welcome` a la clase `Student`:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

Si agrega un método en la clase hija con el mismo nombre que una función en la clase padre, se anulará la herencia del método principal.

4.12. Ejercicios de POO

Ejercicio 4.1. Cree una clase llamada Persona, use la función *__init__()* para asignar valores para nombre y edad:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1)
```

```
<__main__.Person object at 0x7d8759ac9240>
```

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"{self.name} is {self.age} years old."

p1 = Person("John", 36)

print(p1)
```

```
John is 36 years old.
```

Ejercicio 4.2. Crear una clase que reciba la parte real y la parte imaginaria de un número complejo. Debe devolver la representación del número.

$$2 \pm 3i$$

$$-4 \pm 5i$$

```
class complex_number:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __str__(self):
        return f'{self.real} ± {self.imag}i'

num1 = complex_number(2,3)
print(num1)
num2 = complex_number(-4,5)
print(num2)
```

```
2 ± 3i
-4 ± 5i
```

Ejercicio 4.3. Inserte una función que imprima un saludo y ejecútelo en el objeto p1:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name + " and I'm " + str(self.age) + " years old.")

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John and I'm 36 years old.
```

Ejercicio 4.4. Registro de perros

```
class Perro:
    def __init__(self, name, raza, talla, peso):
        self.name = name
        self.raza = raza
        self.talla = talla
        self.peso = peso

    def obtener_peso(self):
        return self.peso

miPerro = Perro("Firulais", "mestizo", "Mediano", 15)
print(f'Mi perro pesa {miPerro.obtener_peso()} kilos')
```

```
Mi perro pesa 15 kilos
```

Ejercicio 4.5. Registro de perros con promedio de peso

```
class Perro:
    # Peso promedio
    peso = 20

    def __init__(self, name, raza, talla, peso):
        self.name = name
        self.raza = raza
        self.talla = talla
        self.peso = peso

    @classmethod # Método de clase: Acceso a los valores de clase en lugar de instancia
    def obtener_peso(self):
        return self.peso

miPerro = Perro("Firulais", "mestizo", "Mediano", 12)
print(f'Mi perro pesa {miPerro.peso} kilos')
print(f'El peso promedio de los perros es {miPerro.obtener_peso()} kilos')
```

```
Mi perro pesa 12 kilos
El peso promedio de los perros es 20 kilos
```

`@classmethod` convierte un método para que trabaje con la clase en sí (y sus atributos de clase) y no con instancias específicas.

Ejercicio 4.6. Calculadora

```
class calcu:

    @staticmethod
    def sumar(num1, num2):
        return num1 + num2

    @staticmethod
    def resta(num1, num2):
        return num1 - num2

    @staticmethod
```

```
    def multiplicacion(num1, num2):
        return num1 * num2

    @staticmethod
    def division(num1, num2):
        return num1 / num2

print(calcu.sumar(4,5))
print(calcu.resta(8,4))
print(calcu.multiplicacion(2,9))
print(calcu.division(6,3))
```

```
9
4
18
2.0
```

```
x = calcu.sumar(4,5)
print(x)
```

```
9
```

Ejercicio 4.7. Realizar una clase que reciba un par de argumentos numéricos y realice las operaciones aritméticas básicas

```
class calculadora:
    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 = num2

    def suma(self):
        return self.num1 + self.num2

    def resta(self):
        return self.num1 - self.num2

    def multiplicacion(self):
        return self.num1 * self.num2

    def division(self):
        return self.num1 / self.num2

operacion = calculadora(5,9)
print(operacion.num1)
print(operacion.num2)
print(operacion.suma())
print(operacion.resta())
print(operacion.multiplicacion())
print(operacion.division())
```

```
5
9
14
-4
```

```
45
0.5555555555555556
```

Ejercicio 4.8. Realice un clase *Circulo* que reciba las coordenadas (x, y) donde se ubica el centro y el radio r. La clase debe tener métodos que determinen el perímetro, área, cuadrante donde se ubica el cirulo en el plano cartesiano.

```
import math

class Circulo:
    def __init__(self, x, y, r):
        self.x = x
        self.y = y
        self.r = r

    def perimetro(self):
        return 2 * math.pi * self.r

    def area(self):
        return math.pi * self.r**2

    def cuadrante(self):
        if self.x > 0 and self.y > 0:
            return "I"
        elif self.x < 0 and self.y > 0:
            return "II"
        elif self.x > 0 and self.y < 0:
            return "IV"
        else:
            return "III"

c1 = Circulo(-6,-7,2)
print(c1.perimetro())
print(c1.area())
print(c1.cuadrante())
```

```
12.566370614359172
12.566370614359172
III
```

Ejercicio 4.9. Realice una clase que reciba la parte real y la parte imaginaria de un número complejo. Debe tener métodos que determine el módulo del número complejo, la suma y resta de dos números complejos.

$$|2 + -3i| = \sqrt{2^2 + 3^2} = \sqrt{13}$$
$$(2 + -3i) + (4 + -5i) = 6 + -8i$$
$$(2 + -3i) - (4 + -5i) = -2 + -2i$$

```
import math

class ComplexNumber:
    def __init__(self, r, i):
```

```
        self.r = r
        self.i = i

    def modulo(self):
        return math.sqrt(self.r**2 + self.i**2)

    def __str__(self):
        return f'{self.r} ± {self.i}i'

    # Métodos Mágicos (Dunder -> double under (score))
    def __add__(self, other):
        return ComplexNumber(self.r + other.r, self.i + other.i)

    def __sub__(self, other):
        return ComplexNumber(self.r - other.r, self.i - other.i)

# Ejemplo
c1 = ComplexNumber(2,3)
print(c1)
print(c1.modulo())

c2 = ComplexNumber(4,5)
print(c2)
print(c2.modulo())

c3r = c1.r + c2.r
c3i = c1.i + c2.i
c3 = ComplexNumber(c3r, c3i)
print(c3)

c4 = c1 + c2
print(c4)

c5 = c1 - c2
print(c5)
```

```
2 ± 3i
3.605551275463989
4 ± 5i
6.4031242374328485
6 ± 8i
6 ± 8i
-2 ± -2i
```

En Python, los métodos *__add__* y *__sub__* son métodos especiales que permiten definir el comportamiento de los operadores de suma (+) y resta (-) para los objetos de una clase personalizada. Estos métodos se denominan "**métodos mágicos**.º "**métodos dunder**"(double underscore, por sus siglas en inglés).

Ejercicio 4.10. Clase Persona.

Crea una clase *Persona* con atributos nombre, edad y métodos para mostrar esta información y calcular si la persona es mayor de edad.

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

    def mostrar_info(self):
        return(f'Nombre: {self.nombre}, Edad: {self.edad}')

    def mayor_de_edad(self):
        if self.edad >= 18:
            return True
        else:
            return False

# Ejemplo
persona1 = Persona("Juan", 23)
print(persona1.mostrar_info())
print(persona1.mayor_de_edad())
```

```
Nombre: Juan, Edad: 23
True
```

Ejercicio 4.11. Clase Rectángulo. Crea una clase *Rectángulo* con atributos largo y ancho. Incluye métodos para calcular el área y el perímetro del rectángulo.

```
class Rectangulo:
    def __init__(self, largo, ancho):
        self.largo = largo
        self.ancho = ancho

    def perimetro(self):
        return 2 * (self.largo + self.ancho)

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


# Ejemplo
r1 = Rectangulo(9, 5)
print(r1.perimetro())
print(r1.area())
```

```
28
45
```

Ejercicio 4.12. Clase Círculo. Crea una clase *Círculo* con un atributo radio. Añade métodos para calcular el área y la circunferencia del círculo.

```
import math

class Circulo:
    def __init__(self, r):
        self.r = r

    def area(self):
```

```
        return math.pi * self.r**2

    def circunferencia(self):
        return 2*math.pi*self.r

# Ejemplo
c1 = Circulo(4)
print(f'Area = {c1.area()} u²')
print(f'Perimetro = {c1.circunferencia()} u')
```

```
Area = 50.26548245743669 u²
Perimetro = 25.132741228718345 u
```

Ejercicio 4.13. Clase CuentaBancaria. Crea una clase *CuentaBancaria* con atributos titular y saldo. Incluye métodos para depositar, retirar y mostrar el saldo.

```
class CuentaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, monto):
        self.saldo += monto

    def retirar(self, monto):
        if monto < self.saldo:
            self.saldo = self.saldo - monto # self.saldo -= monto
        else:
            print("Saldo insuficiente")

    def mostrarSaldo(self):
        print(f'Saldo de {self.titular}: $ {self.saldo}')

# Ejemplo
cliente1 = CuentaBancaria("Juan Perez", 4500)
cliente1.depositar(430)
cliente1.mostrarSaldo()
cliente1.retirar(10000)
```

```
Saldo de Juan Perez: $ 4930
Saldo insuficiente
```

Ejercicio 4.14. Clase Estudiante. Crea una clase Estudiante que herede de Persona. Añade un atributo promedio y un método para determinar si el estudiante está aprobado (promedio >= 6).

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

    def mostrarInfo(self):
        print(f'Nombre: {self.nombre}, Edad: {self.edad}')

class Estudiante(Persona):
    def __init__(self, nombre, edad, promedio):
```

```
        super().__init__(nombre, edad)
        self.promedio = promedio

    def aprobado(self):
        if self.promedio >= 6:
            return True
        else:
            return False

# Ejemplo
estudiante1 = Estudiante("Maria", 20, 7.5)
print(estudiante1.aprobado())
estudiante1.mostrarInfo()
```

```
True
Nombre: Maria, Edad: 20
```

Ejercicio 4.15. Clase Libro. Crea una clase Libro con atributos título, autor y año. Incluye un método para mostrar la información del libro y otro para determinar si es un libro antiguo (año < 2000).

```
class Libro:
    def __init__(self, titulo, autor, anio):
        self.titulo = titulo
        self.autor = autor
        self.anio = anio

    def info(self):
        return f'{self.autor}; {self.titulo}; {self.anio}'

    def esAntiguo(self):
        return True if self.anio < 2000 else False

libro1 = Libro("Algebra", "Baldor", 1980)
print(libro1.info())
print(libro1.esAntiguo())
```

```
Baldor; Algebra; 1980
True
```

Ejercicio 4.16. Clase Vehículo. Crea una clase Vehículo con atributos marca, modelo y año. Añade un método para mostrar la información del vehículo y otro para determinar si es un vehículo clásico (año < 1980).

Ejercicio 4.17. Clase Empleado. Crea una clase Empleado con atributos nombre, salario y años_de_experiencia. Añade un método para calcular un aumento salarial basado en los años de experiencia (5 %).

```
class Empleado:
    def __init__(self, nombre, salario, exp):
        self.nombre = nombre
        self.salario = salario
        self.exp = exp
```

```
    def __str__(self):
        return f'Nombre: {self.nombre}, Salario: {self.salario}, Años de Experiencia: {self.exp}'

    def mostrarInfo(self):
        print(f'Nombre: {self.nombre}, Salario: {self.salario}, Años de Experiencia: {self.exp}')

    def aumento(self):
        aumento = 0.05*self.exp
        self.salario += self.salario*aumento

# Ejemplo
empleado1 = Empleado("Carlos", 15000, 10)
print(empleado1)
empleado1.mostrarInfo()
empleado1.aumento()
print(f'Nuevo salario: {empleado1.salario}')
```

```
Nombre: Carlos, Salario: 15000, Años de Experiencia: 10
Nombre: Carlos, Salario: 15000, Años de Experiencia: 10
Nuevo salario: 22500.0
```

Ejercicio 4.18. Clase CuentaDeAhorro. Crea una clase CuentaDeAhorro que herede de CuentaBancaria. Añade un atributo interés y un método para aplicar el interés al saldo.

Ejercicio 4.19. Clase Empresa. Crea una clase Empresa con atributos nombre y empleados (una lista de objetos de la clase Empleado). Incluye métodos para agregar empleados, eliminar empleados y calcular el salario total pagado por la empresa.

4.13. Referencias

- w3 Schools
- Cosas de Devs

Capítulo 5

Métodos Mágicos

5.1. Atributos de clase protegidos

Los atributos dentro de una clase pueden ser modificados por el usuario directamente. Volvamos al ejemplo de la clase `Persona`.

[5.1]:
```
class Persona:
    def __init__(self, nombre, edad, num_cuenta):
        self.nombre, self.edad, self.num_cuenta = nombre, edad, num_cuenta

    def mostrarInformación(self):
        print(f'{self.nombre} -> {self.edad}')
```

Con esta clase `Persona` se crea un objeto llamado `cliente1`.

[5.2]:
```
cliente1 = Persona("Juan", 34, 123456789)
cliente1.mostrarInformación()
```

```
Juan -> 34
```

El objeto `cliente1` tiene los valores se sus propiedades pasadas por el constructor al momento de ser creado, y no existe limitante alguna para ser modificados directamente. Por ejemplo:

[5.3]:
```
cliente1.nombre = "Juan Perez"
cliente1.mostrarInformación()
```

```
Juan Perez -> 34
```

Aunque esto es permitido por el intérprete en `Python`, es considerado como una mala práctica de programación. Lo que se debe hacer es escribir un módulo dentro de la clase que reciba el nuevo nombre y haga el cambio del valor de la propiedad. De esta manera se tiene un control y certeza sobre los cambios en las propiedades. Considere además el caso de que en la propiedad se almacene información sensible que no se desea revelar, sería necesario limitar el acceso a la propiedad que contenga esa información.

[5.4]:
```
cliente1.num_cuenta
```

[5.5]:
```
123456789
```

Dado que Python no posee mecanismo para evitar el acceso o que se modifiquen los valores de las propiedades, existe una convención para marcar los atributos como protegidos. Para ello se utiliza el prefijo guión bajo _ en el nombre de la propiedad. Esto indica a cualquier otro programador que dicha propiedad no debería ser accedida o modificada fuera de la clase.

[5.6]:
```python
class Persona:
    def __init__(self, nombre, edad, num_cuenta):
        self.nombre, self.edad, self._num_cuenta = nombre, edad, num_cuenta
        # Nótese que num_cuenta se marcó como propiedad protegida

    def mostrarInformación(self):
        print(f'{self.nombre} -> {self.edad}')
```

[5.7]:
```python
cliente1 = Persona("Juan", 34, 123456789)
cliente1.mostrarInformación()
```

```
Juan -> 34
```

[5.8]:
```python
cliente1._num_cuenta
```

[5.9]:
```
123456789
```

Aunque la propiedad *_num_cuenta* aún puede ser impresa e incluso modificada, marcarla como protegida indica al programador que hacerlo va en contra de las buenas prácticas de codificación.

5.2. Métodos especiales

Los métodos especiales en Python, también conocidos como **métodos mágicos** o **dunder methods**, son funciones integradas dentro de las clases que permiten definir comportamientos específicos para operaciones estándar. Tienen dos guiones bajos al principio y al final de sus nombres, como `__init__` o `__str__`.

5.2.1. Algunos métodos especiales comunes:

- `__init__`: Inicializa una nueva instancia de una clase.
- `__str__`: Define cómo representar un objeto como una cadena de texto, usado en `str(obj)` y `print(obj)`.
- `__repr__`: Proporciona una representación oficial de un objeto, usada en `repr(obj)`.
- `__len__`: Devuelve el tamaño o longitud de un objeto, usado en `len(obj)`.
- `__getitem__`, `__setitem__`, `__delitem__`: Permiten acceder, modificar y eliminar elementos por índice.
- `__iter__`, `__next__`: Permiten que un objeto sea iterable, como en un bucle `for`.
- `__eq__`, `__lt__`, `__gt__`: Implementan comparaciones (igualdad, menor que, mayor que, etc.).
- `__add__`, `__sub__`, `__mul__`, **etc.**: Definen el comportamiento de operadores aritméticos.

Ya hemos utilizado el método `__init__` que es el constructor, y se ejecuta automáticamente cada vez que se crea un objeto/instancia de la clase.

El nombre **método especial** o más aún **método mágico** puede ser engañoso, ya que técnicamente no hay algo especial o mágico en ellos. Lo único especial acerca de ellos es el nombre, el cual asegura que serán llamados en situaciones especiales. Por ejemplo, el método `__init__` que se ejecuta al crear un objeto.

Por ejemplo, considere la *fórmula general barométrica* (5.1) para calcular la presión atmosférica p dada la altura h.

$$p = p_0 e^{-Mgh/RT} \tag{5.1}$$

donde M es la masa molar del aire, g es la constante gravitacional, R es la constante del gas, T la temperatura y p_0 la presión del aire a nivel del mar. Se define además $h_0 = \frac{RT}{Mg}$.

$$p = p_0 e^{-h/h_0} \tag{5.2}$$

Ahora se define una clase para el cálculo barométrico.

[5.10]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81         # m/s²
        R = 8.314        # J/(K*mol)
        M = 0.02896      # kg/mol
        self.h0 = R*T/M/g
        self.p0 = 100         # kPa

    def value(self, h):
        return self.p0 * math.exp(-h/self.h0)
```

[5.11]:
```
bar1 = Barometric(292.15)
bar1.value(2200)
```

[5.12]:
```
77.31204126500637
```

Esta forma de la clase permite obtener el valor de la presión para cierto valor de T y h. El valor de la presión de obtiene al llamar al método `value` y pasarle el argumento h.

Sería más simple de utilizar si se pudiese llamar directamente al objeto sin necesidad de emplear el método intermedio. Para ello existe el método especial `__call__`.

Veamos una nueva versión de la clase empleando este método especial.

[5.13]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81         # m/s²
        R = 8.314        # J/(K*mol)
        M = 0.02896      # kg/mol
        self.h0 = R*T/M/g
        self.p0 = 100         # kPa
```

```
    def __call__(self, h):
        return self.p0 * math.exp(-h/self.h0)
```

[5.14]:
```
bar2 = Barometric(292.15)
print(bar2(2200))

# Es equivalente a esta forma de la llamada
print(bar2.__call__(2200))
```

```
77.31204126500637
77.31204126500637
```

5.2.2. Método especial para imprimir

Es posible imprmir un objeto a empleando un `print(a)`, lo cual funciona bien para los objetos propios de Python como cadenas y listas. Sin embrago, si nosotros creamos una clase, ese print no necesariamente mostrará información útil. Por ello tendremos que resolver ese problema definiendo el método `__str__` dentro de la clase. El método `__str__` debe devolver de preferencia una cadena y no debe recibir argumentos excepto por `self`.

Redefiniendo la clase `Barometric`, queda así:

[5.15]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81         # m/s²
        R = 8.314        # J/(K*mol)
        M = 0.02896      # kg/mol
        self.h0 = R*T/M/g
        self.p0 = 100         # kPa
        self.T = T

    def __call__(self, h):
        return f'p(h = {h}, T = {self.T}) = {self.p0 * math.exp(-h/self.h0)} kPa'

    def __str__(self):
        return f'p0 * exp(-Mgh/(RT)) [kPa]; T = {self.T}ºK'
```

[5.16]:
```
bar3 = Barometric(292.15)
print(bar3(2200))
print(bar3)
```

```
p(h = 2200, T = 292.15) = 77.31204126500637 kPa
p0 * exp(-Mgh/(RT)) [kPa]; T = 292.15ºK
```

5.2.3. Métodos especiales para operaciones matemáticas

Hasta ahora hemos cubierto los métodos `__init__`, `__call__` y `__str__`, pero hay más de ellos. Por ejemplo, los métodos `__add__`, `__sub__` y `__mul__`. Definir estos métodos dentro de la clase nos permite emplear expresiones como $c = a + b$, donde a y b son instancias de una clase.

```
c = a + b       # c = a.__add__(b)

c = a - b       # c = a.__sub__(b)

c = a * b       # c = a.__mul__(b)

c = a / b       # c = a.__div__(b)

c = a ** b      # c = a.__pow__(b)
```

Para la mayoría de los casos, cualquiera de estas operaciones devuelve un objeto de la misma clase que los operandos.

De manera similar, también existen métodos especiales para comparar objetos:

```
a == b          # a.__eq__(b)

a != b          # a.__ne__(b)

a < b           # a.__lt__(b)

a <= b          # a.__le__(b)

a > b           # a.__gt__(b)

a >= b          # a.__ge__(b)
```

Estos métodos deben ser implementados para devolver un booleano, para que sea consistente con el comportamiento de los operadores de comparación.

El contenido de los métodos al momento de definirlos dependen del desarrollador, lo único especial acerca de los métodos es su nombre, ya que mediante este pueden ser llamados automáticamente por varios operadores.

Por ejemplo, si se desea multiplicar dos objetos $c = a * b$, Python buscará el método llamado `__mul__` en la instancia a. Si el método existe, será llamado pasandole como argumento la instancia b y cualquiera que sea la devolución del método `__mul__` se asigna a c.

5.2.4. Método especial `__repr__`

Este método especial es semejante al método `__str__`, ya que devuelve una cadena con información acerca del objeto. Por un lado la cadena devuelta por `__str__` muestra información que es fácilmente leíble y por otro lado la cadena devuelta por `__repr__` contiene la información necesaria para recrear el objeto.

Para un objeto `a` el método `__repr__` se puede llamar mediante la función nativa de Python llamada `repr(a)`.

[5.17]:
```
import math

class Barometric:
    def __init__(self, T):
        g = 9.81        # m/s²
        R = 8.314       # J/(K*mol)
        M = 0.02896     # kg/mol
```

```
        self.h0 = R*T/M/g
        self.p0 = 100         # kPa
        self.T = T

    def __call__(self, h):
        return f'p(h = {h}, T = {self.T}) = {self.p0 * math.exp(-h/self.h0)} kPa'

    def __str__(self):
        return f'p0 * exp(-Mgh/(RT)) [kPa]; T = {self.T}ºK'

    def __repr__(self):
        """ Return code for regenerating this instance """
        return f'Barometric({self.T})'
```

[5.18]:
```
b3 = Barometric(292.15)
print(b3)
repr(b3)
```

```
p0 * exp(-Mgh/(RT)) [kPa]; T = 292.15ºK
```

[5.19]:
```
'Barometric(292.15)'
```

[5.20]:
```
b4 = eval(repr(b3))
print(b4)
```

```
p0 * exp(-Mgh/(RT)) [kPa]; T = 292.15ºK
```

Estos resultados confirman que el método `__repr__` funciona de acuerdo a lo esperado, dado que `eval(repr(b3))` devuelve un objeto idéntico a `b3`.

Ambos métodos `__str__` y `__repr__` muestran información acerca de un objeto, la diferencia es que una muestra información legible para humanos y la segunda información legible para Python.

5.2.5. Mostrar contenido de una clase

Algunas veces resulta útil mostrar el contenido de una clase, por ejemplo para realizar debugging.

Considere la siguiente clase de ejemplo que solo contiene un comentario, el constructor y una propiedad:

[5.21]:
```
class A:
    """ Una clase de muestra """
    def __init__(self, value):
        self.v = value
```

Si se realiza un dir(A) se mostrarán varios métodos y propiedades que se han definido automáticamente en la clase.

[5.22]:
```
dir(A)
```

[5.23]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
```

```
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

Además, si se crea un objeto de la clase y se ejecuta el método `dir(a)` observaremos la misma salida mostrado con la clase pero también los valores creados por el constructor al momento de la creación del objeto.

[5.24]:
```
a = A(2)
dir(a)
```

[5.25]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
```

```
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'v']
```

5.2.6. Método especial `__doc__`

El método especial `__doc__` muestra los comentarios que existen dentro de la definición de la clase que han sido escritos dentro de triples comillas dobles. Estos textos y este método son los que sirven para construir la documentación del código posteriormente.

[5.26]:
```
a.__doc__
```

[5.27]:
```
' Una clase de muestra '
```

[5.28]:
```
A.__doc__
```

[5.29]:
```
' Una clase de muestra '
```

[5.30]:
```
import numpy
numpy.__doc__
```

[5.31]:
```
'\nNumPy\n=====\n\nProvides\n  1. An array object of arbitrary homogeneous
items\n  2. Fast mathematical operations over arrays\n  3. Linear Algebra,
Fourier Transforms, Random Number Generation\n\nHow to use the
documentation\n----------------------------\nDocumentation is available in two
forms: docstrings provided\nwith the code, and a loose standing reference guide,
available from\n`the NumPy homepage <https://numpy.org>`_.\n\nWe recommend
exploring the docstrings using\n`IPython <https://ipython.org>`_, an advanced
Python shell with\nTAB-completion and introspection capabilities.  See below for
further\ninstructions.\n\nThe docstring examples assume that `numpy` has been
imported as ``np``::\n\n  >>> import numpy as np\n\nCode snippets are indicated
by three greater-than signs::\n\n  >>> x = 42\n  >>> x = x + 1\n\nUse the built-
in ``help`` function to view a function\'s docstring::\n\n  >>> help(np.sort)\n
... # doctest: +SKIP\n\nFor some objects, ``np.info(obj)`` may provide
additional help.  This is\nparticularly true if you see the line "Help on ufunc
object:" at the top\nof the help() page.  Ufuncs are implemented in C, not
Python, for speed.\nThe native Python help() does not know how to view their
help, but our\nnp.info() function does.\n\nTo search for documents containing a
keyword, do::\n\n  >>> np.lookfor(\'keyword\')\n  ... # doctest:
+SKIP\n\nGeneral-purpose documents like a glossary and help on the basic
concepts\nof numpy are available under the ``doc`` sub-module::\n\n  >>> from
numpy import doc\n  >>> help(doc)\n  ... # doctest: +SKIP\n\nAvailable
subpackages\n---------------------\nlib\n    Basic functions used by several
sub-packages.\nrandom\n    Core Random Tools\nlinalg\n    Core Linear Algebra
Tools\nfft\n    Core FFT routines\npolynomial\n    Polynomial tools\ntesting\n
NumPy testing tools\ndistutils\n    Enhancements to distutils with support for\n
Fortran compilers support and more  (for Python <=
```

```
3.11).\n\nUtilities\n---------\ntest\n    Run numpy unittests\nshow_config\n
Show numpy build configuration\nmatlib\n    Make everything
matrices.\n__version__\n    NumPy version string\n\nViewing documentation using
IPython\n-----------------------------------\n\nStart IPython and import `numpy`
usually under the alias ``np``: `import\nnumpy as np`.  Then, directly past or
use the ``%cpaste`` magic to paste\nexamples into the shell.  To see which
functions are available in `numpy`,\ntype ``np.<TAB>`` (where ``<TAB>`` refers
to the TAB key), or use\n``np.*cos*?<ENTER>`` (where ``<ENTER>`` refers to the
ENTER key) to narrow\ndown the list.  To view the docstring for a function,
use\n``np.cos?<ENTER>`` (to view the docstring) and ``np.cos??<ENTER>`` (to
view\nthe source code).\n\nCopies vs. in-place
operation\n-----------------------------\nMost of the functions in `numpy`
return a copy of the array argument\n(e.g., `np.sort`).  In-place versions of
these functions are often\navailable as array methods, i.e. ``x =
np.array([1,2,3]); x.sort()``.\nExceptions to this rule are documented.\n\n'
```

5.2.7. Otors métodos especiales

El método `__module__` devuelve el nombre del módulo al cual pertenece la clase, en el siguiente ejemplo se devuelve `__main__` dado que el objeto fue creado dentro de ese módulo.

[5.32]:
```
a.__module__
```

[5.33]:
```
'__main__'
```

El método especial `__dict__` devuelve un diccionario con las propiedades y los valores de un objeto.

[5.34]:
```
a.__dict__
```

[5.35]:
```
{'v': 2}
```

Una instancia contiene todos los atributos de la clase creados automáticamente por Python. Si se agregan nuevos valores al objeto, éstos son añadidos al diccionario.

[5.36]:
```
a.myVar = 10
a.__dict__
```

[5.37]:
```
{'v': 2, 'myVar': 10}
```

[5.38]:
```
a.__getattribute__("v")
a.__getattribute__("myVar")
```

[5.39]:
```
10
```

Ejemplo 5.1. Crear una clase que calcule la evaluación de la derivada de una función. Utilizar una definición genérica para el cálculo de una aproximación de la derivada.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

[5.40]:
```
class Derivada:
    def __init__(self, f, h=1e-5):
        self.f = f  # Función f(x) a derivar
```

```
        self.h = h

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

[5.41]:
```
def f(x):
    return x**3

dfdx = Derivada(f)

dfdx(1)
```

[5.42]:
```
3.000030000110953
```

Ejemplo 5.2. Crear una clase que calcule la aproximación de una integral definida. Para ello consideremos la regla compuesta del trapecio.

$$A = \int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right]$$

considere:

$$h = \frac{b-a}{n}$$

y

$$x_j = a + j \cdot h,$$

donde n es la cantidad de trapecios y h la altura de cada uno de ellos.

[5.43]:
```
class TrapecioCompuesta:
    def __init__(self, f, n=1):
        self.f, self.n = f, n

    def __call__(self, a, b):
        f, n = self.f, self.n
        h = (b-a)/n
        suma = 0
        for j in range(1,n):
            suma += f(a + j*h)

        return f'A = {h/2*(f(a) + 2*suma + f(b))} u^2'
```

[5.44]:
```
import math
def f(x):
    return math.sin(x)

intf = TrapecioCompuesta(f, 10000)

intf(math.pi, 2*math.pi)
```

[5.45]:
```
'A = -1.9999999835506608 u^2'
```

Ejemplo 5.3. Crear una clase que permita construir un polinomio

$$P(x) = a_0 + a_1 x + a_2 x^2$$

La clase debe incluir la funcionalidad de evaluar un polinomio en un valor dado, y sumar dos polinomios. Se deben utilizar los métodos especiales para usarlos de la forma indicada:

- `__init__`: para constuir un polinomio de la forma `p = Polynomial([1,-1])`
- `__str__`: para imprimir el polinomio
- `__call__`: para evaluar el polinomio de la forma `p(2.0)`
- `__add__`: para realizar la suma de polinomios
- `__mul__`: para realizar la multiplicación de polinomios

Además se debe incluir un método para realizar la derivada del polinomio.

Solución

Creación de la clase Polynomial, el constructor y el método ´**call**´

[5.46]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s
```

[5.47]:
```
p1 = Polynomial([1,-1])
p1(4)
```

[5.48]:
```
-3
```

Implementación de la suma de polinomios

[5.49]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other

        # we start with longest list and add it to the other
```

```
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]

        else:
            coeffsum = other.coeff[:] # copy list
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]

        return Polynomial(coeffsum)
```

[5.50]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
p2 = Polynomial([1, 1, 1])          # x² + x + 1
p3 = p1 + p2    # x³ + 3x² + x - 2
print(p3.coeff)
```

```
[-2, 1, 3, 1]
```

Para la multiplicación se requiere realizar un proceso un poco más complicado. Nos referiremos a la expresión matemática para la multiplicación de polinomios.

$$p_1 \cdot p_2 = \left(\sum_{i=0}^{M} c_i x^i\right)\left(\sum_{j=0}^{N} d_j x^j\right) = \sum_{i=0}^{M}\sum_{j=0}^{N} c_i d_j x^{i+j},$$

donde

$$p_1 = c_0 + c_1 x + c_2 x^2 + \cdots + c_M x^M$$

y

$$p_2 = d_0 + d_1 x + d_2 x^2 + \cdots + d_N x^N$$

[5.51]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other

        # we start with longest list and add it to the other
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]
```

```
        else:
            coeffsum = other.coeff[:] # copy list
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]

        return Polynomial(coeffsum)

    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # [0 for i in range(10)]    # List of M+N+1 zeros
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i] * other.coeff[j]

        return Polynomial(coeff)
```

[5.52]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
p2 = Polynomial([1, 1, 1])         # x² + x + 1
p4 = p1 * p2    # x³ + 3x² + x - 2
print(p4.coeff)
```

```
[-3, -3, -1, 3, 3, 1]
```

Para el cálculo de la derivada del polinomio, se puede utilizar la regla:

$$\frac{d}{dx}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

Por lo tanto, si c es la lista de coeficientes del polinomio, la derivada tiene una lista de coeficientes en dc, donde $dc[i-1] = i \cdot c[i]$ para todos los valores de i desde 1 hasta el índice mayor en c. Recuerde que la derivada de un polinomio se reduce en grado en 1, por lo tanto la lista dc tendrá un elemento menos que la lista c.

[5.53]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other

        # we start with longest list and add it to the other
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]
```

```
        else:
            coeffsum = other.coeff[:] # copy list
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]

        return Polynomial(coeffsum)

    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # [0 for i in range(10)]    # List of M+N+1 zeros
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i] * other.coeff[j]

        return Polynomial(coeff)

    def differentiate(self):
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i * self.coeff[i]
        del self.coeff[-1]

    def derivative(self):
        dpdx = Polynomial(self.coeff[:])
        dpdx.differentiate()
        return dpdx
```

[5.54]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
p1.derivative()
```

[5.55]:
```
<__main__.Polynomial at 0x7fa9cc5e0fd0>
```

Por último, hace falta agregar la función `__str__` para mostrar el polinomio de una forma legible. El método debe devolver una representación del polinomio lo más cercana posible a como se escribe en Matemáticas.

[5.56]:
```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other

        # we start with longest list and add it to the other
```

```
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # copy list
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]

        else:
            coeffsum = other.coeff[:] # copy list
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]

        return Polynomial(coeffsum)

    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # [0 for i in range(10)]    # List of M+N+1 zeros
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i] * other.coeff[j]

        return Polynomial(coeff)

    def differentiate(self):
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i * self.coeff[i]
        del self.coeff[-1]

    def derivative(self):
        dpdx = Polynomial(self.coeff[:])
        dpdx.differentiate()
        return dpdx

    def __str__(self):
        s = ''
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += f' + {self.coeff[i]:g}*x^{i:g}'
        # fix layout
        s = s.replace('+ - ', '- ')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^0','1')
        s = s.replace('*1', '')
        s = s.replace('x^1', 'x')
        if s[0:3] == ' + ':
            s = s[3:]
        if s[0:3] == ' - ':
            s = '-' + s[3:]
        return s
```

$$p_1(x) = x^3 + 2x^2 - 3$$

[5.57]:
```
p1 = Polynomial([-3, 0, 2, 1])      # x³ + 2x² -3
print(p1)
```

```
-3 + 2*x^2 + x^3
```

$$\frac{dp_1}{dx} = 3x^2 + 4x$$

[5.58]:
```
p2 = p1.derivative()
print(p2)
```

```
4*x + 3*x^2
```

5.3. Ejercicios

Ejercicio 5.1. .

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

    def __str__(self):
        return f'{self.nombre}, {self.edad} años'

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

persona = Persona('Juan', 30)
print(persona)
print(repr(persona))
```

```
Juan, 30 años
Persona(Juan, 30)
```

Ejercicio 5.2. .

```
class MiLista:
    def __init__(self, elementos):
        self.elementos = elementos

    def __len__(self):
        return len(self.elementos)

    def __getitem__(self, posicion):
        return self.elementos[posicion]

lista = MiLista([1,2,3,4])
listaNormal = [1,2,3,4]

print(len(lista))
print(len(listaNormal))
```

```
print(lista[2])
print(listaNormal[2])
```

```
4
4
3
3
```

Ejercicio 5.3. .

```
class GestorRecursos:
    def __enter__(self):
        print("Entrando al contexto")
        return self

    def __exit__(self, tipo, valor, traza):
        print("Saliendo del contexto")

# Declaración por contexto
with GestorRecursos() as recurso:
    print("Dentro del bloque with")
```

```
Entrando al contexto
Dentro del bloque with
Saliendo del contexto
```

Ejercicio 5.4. .

```
class Saludador:
    def __init__(self, saludo):
        self.saludo = saludo

    def __call__(self, nombre):
        return f'{self.saludo}, {nombre}'


saludador = Saludador('Hola')
print(saludador('mundo'))
```

```
Hola, mundo
```

Ejercicio 5.5. .

```
class Singleton:
    _instancia = None

    def __new__(cls, *args, **kwargs):
        if cls._instancia is None:
            cls._instancia = super().__new__(cls)
        return cls._instancia

    def __init__(self, valor):
        self.valor = valor

obj1 = Singleton(10)
```

```
obj2 = Singleton(20)

print(obj1)
print(obj2)

print(obj1.valor)
print(obj2.valor)

print(obj1 is obj2)
```

```
<__main__.Singleton object at 0x7b04d2ea1450>
<__main__.Singleton object at 0x7b04d2ea1450>
20
20
True
```

Ejercicio 5.6. .

```
class AtributosDinamicos:
    def __init__(self):
        self.datos = {} # Diccionario vacio

    def __getattr__(self, nombre):
        return self.datos.get(nombre, f'{nombre} no encontrado')

    def __setattr__(self, nombre, valor):
        if nombre == 'datos':
            super().__setattr__(nombre, valor)
        else:
            self.datos[nombre] = valor


obj = AtributosDinamicos()
obj.name = 'Python'
print(obj.name)

print(obj.age)
```

```
Python
age no encontrado
```

Ejercicio 5.7. .

```
class DiccionarioPersonalizado:
    def __init__(self):
        self.datos = {}

    def __setitem__(self, llave, valor):
        self.datos[llave] = valor

    def __getitem__(self, llave):
        return self.datos.get(llave, 'Llave no encontrada')

    def __delitem__(self, llave):
```

```
        del self.datos[llave]

dic = DiccionarioPersonalizado()
dic['llave'] = 'valor'  # __setitem__
print(dic['llave'])     # __getitem__
del dic['llave']        # __delitem__
print(dic['llave'])
```

```
valor
Llave no encontrada
```

Ejercicio 5.8. .

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

    def __eq__(self, otro):
        return self.nombre == otro.nombre and self.edad == otro.edad

    def __lt__(self, otro):
        return self.edad < otro.edad

    def __gt__(self, otro):
        return self.edad > otro.edad

    def __hash__(self):
        return hash((self.nombre, self.edad))

p1 = Persona('Juan', 25)
p2 = Persona('Juan', 25)
p3 = Persona('Ana', 30)

print(p1 == p2)
print(p1 == p3)
print(p1 < p3)
print(p3 > p1)
```

```
True
False
True
True
```

Ejercicio 5.9. .

```
class Optimizado:
    __slots__ = ['x', 'y']

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

obj = Optimizado(1, 2)
```

```
print(obj.x, obj.y)
```

```
1 2
```

Ejercicio 5.10. .

```
class AccesoControlado:
    def __init__(self, valor):
        self.valor = valor

    def __getattribute__(self, nombre):
        print(f'Accediendo a {nombre}')
        return super().__getattribute__(nombre)

obj = AccesoControlado(10)
print(obj.valor)
```

```
Accediendo a valor
10
```

Ejercicio 5.11. .

```
class Coleccion:
    def __init__(self, elementos):
        self.elementos = elementos

    def __contains__(self, item):
        return item in self.elementos

col = Coleccion([1,2,3])
print(2 in col)
```

```
True
```

```
3 in [1,2,3]
```

```
True
```

```
5 in [1,2,3]
```

```
False
```

Ejercicio 5.12. .

```
class Contador:
    def __init__(self, inicio, fin):
        self.actual = inicio
        self.fin = fin

    def __iter__(self):
        return self

    def __next__(self):
        if self.actual < self.fin:
            actual = self.actual
            self.actual += 1
            return actual
```

```
        else:
            raise StopIteration

contador = Contador(0, 5)
for numero in contador:
    print(numero)
```

```
0
1
2
3
4
```

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

```
0
1
2
3
4
```

Ejercicio 5.13. .

```
class Recurso:
    def __init__(self):
        print("Recurso creado")

    def __del__(self):
        print("Recurso liberado")

r = Recurso()
del r
#print(r)
```

```
Recurso creado
Recurso liberado
```

Capítulo 6

Jerarquía de clases y herencia

La jerarquía de clases es una familia de clases que heredan métodos y atributos entre sí.

Una jerarquía de clases es una familia de clases estrechamente relacionadas organizadas de manera jerárquica.

Un concepto clave es la *herencia,* lo que significa que las clases hijo pueden heredar atributos y métodos de las clases padre. Una estrategia típica es escribir una clase general como clase base (o clase principal) y luego dejar que los casos especiales se representen como subclases (clases secundarias). Este enfoque a menudo puede ahorrar mucha escritura y duplicación de código.

Código 6.1. Clases para líneas y parábolas.

En este ejemplo crearemos una clase que represente y evalúe líneas rectas $y = c_0 + c_1 x$.

```
import numpy as np
from prettytable import PrettyTable

class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1

    def __call__(self, x):
        return self.c0 + self.c1 * x

    def table(self, a, b, n):
        """ Devuelve una tabla con n puntos dado a <= x <= b """
        tabla = PrettyTable(field_names = ['x', 'f(x)'], float_format='.6')
        for x in np.linspace(a, b, n):
            y = self(x)
            tabla.add_row([x, y])
        return tabla

    def __str__(self):
        return f'y = {self.c0} + {self.c1}x'
```

La clase tiene su constructor estándar y un método especial `__str__` para la impresión de la función:

```
linea = Line(1,2)
```

```
print(linea)
```

```
y = 1 + 2x
```

Tiene un método especial `__call__` que permite la evaluación de la función:

```
linea(3)
```

```
7
```

Y tiene un método table que construye una tabla tipo `PrettyTable` con la evaluación de la función en un rango de valores de acuerdo a los valores recibidos por los argumentos a, b, y n.

```
linea.table(a=1, b=12, n=20)
```

```
+-----------+-----------+
|     x     |    f(x)   |
+-----------+-----------+
|  1.000000 |  3.000000 |
|  1.578947 |  4.157895 |
|  2.157895 |  5.315789 |
|  2.736842 |  6.473684 |
|  3.315789 |  7.631579 |
|  3.894737 |  8.789474 |
|  4.473684 |  9.947368 |
|  5.052632 | 11.105263 |
|  5.631579 | 12.263158 |
|  6.210526 | 13.421053 |
|  6.789474 | 14.578947 |
|  7.368421 | 15.736842 |
|  7.947368 | 16.894737 |
|  8.526316 | 18.052632 |
|  9.105263 | 19.210526 |
|  9.684211 | 20.368421 |
| 10.263158 | 21.526316 |
| 10.842105 | 22.684211 |
| 11.421053 | 23.842105 |
| 12.000000 | 25.000000 |
+-----------+-----------+
```

Digamos que ahora se requiere escribir una clase similar para la evaluación de la parábola $y = c_0 + c_1x + c_2x^2$, el código 6.2 lo muestra.

Código 6.2. Clase parábola con método *__call__*.

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0=c0, self.c1=c1, self.c2=c2

    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2

    def table(self, a, b, n):
        """ Devuelve una tabla con n puntos dado a <= x <= b """
        tabla = PrettyTable(field_names = ['x', 'f(x)'], float_format='.6')
        for x in np.linspace(a, b, n):
```

```
            y = self(x)
            tabla.add_row([x, y])
        return tabla

    def __str__(self):
        return f'y = {self.c0} + {self.c1}x + {self.c2}x²'
```

Observe que la mayoría del código es la misma excepto por las partes que involucran a c_2. Quizá copiar, pegar y modificar el código no sea tardado ni problemático, pero tal repetición de código es una mala práctica.

Imáginese que fuese necesario cambiar la funcionalidad de la generación de la tabla o corregir un error en el código replicado, sería necesario cambiar en todos los lugares donde el código fue replicado. Esto sería lento, impráctico y una posible fuente de más errores.

Por ello, se puede reutilizar el código de la clase `Line` para la construcción de la clase `Parabola` mediante la herencia.

```
class Parabola(Line):
    pass
```

El término `pass` es una palabra clave que indica a Python que la clase se ha dejado intencionalmente vacía. Sin embargo, aunque no haya código en la definición de la clase Parabola, ésta clase no está vacía. De esta forma `Parabola` ha heredado los atributos c_0 y c_1 y los métodos `__init__`, `__call__`, `__str__` y `table` de la clase `Line`.

En este sentido se dice que `Line` es la clase base (padre o superclase) y `Parabola` es una subclase (clase hijo o clase derivada). En ese momento, `Parabola` es una copia exacta de `Line` pero es posible modificarla para los propósitos de su uso. Para ello es necesario agregar su propio constructor `__init__`, sus métodos `__call__` y `__str__` de acuerdo a lo requerido.

Código 6.3. Clase Parábola con herencia.

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        super().__init__(c0, c1)    # La clase Linea almacena c0 y c1
        self.c2 = c2

    def __call__(self, x):
        return super().__call__(x) + self.c2*x**2

    def __str__(self):
        return super().__str__() + f' + {self.c2}x²'

par = Parabola(1,2,3)
print(par.table(1,2,10))
print(par)
```

```
+----------+-----------+
|    x     |    f(x)   |
+----------+-----------+
| 1.000000 |  6.000000 |
| 1.111111 |  6.925926 |
| 1.222222 |  7.925926 |
| 1.333333 |  9.000000 |
| 1.444444 | 10.148148 |
```

```
| 1.555556 | 11.370370 |
| 1.666667 | 12.666667 |
| 1.777778 | 14.037037 |
| 1.888889 | 15.481481 |
| 2.000000 | 17.000000 |
+----------+-----------+
y = 1 + 2x + 3x²
```

Observe que para maximizar la reutilización del código se ha llamado los métodos de la superclase `Line` y añadido las partes faltantes. Siempre será posible tener acceso a los métodos del padre mediante la función `super()`.

Es posible llamar directamente al padre mediante su nombre, pero será necesario agregar en los argumentos el contexto del objeto mediante `self`. Esto debido a que el nombre genérico no tiene una relación con la subclase, mientras que con `super()` si se hace una referencia directa a la superclase.

Esto es:

```
super().__init__(c0, c1)        # super() hace referencia a la superclase del objeto
Line.__init__(self, c0, c1)     # Se hace referencia a la clase `Line`, que en general no tiene relac
```

En términos generales sería lo siguiente:

```
SuperClassName.method(self, arg1, arg2, ...)
super(arg1, arg2, ...)
```

6.1. El verdadero significado de la herencia

Desde un punto de vista práctico, la herencia permite reutilizar código y minimizar la duplicidad del mismo. Pero desde un punto de vista teórico, la herencia representa la relación que hay entre dos clases.

Esto significa que si `Parabola` es una subclase de `Line`, un objeto `Parabola` también es un objeto de `Line`. En otras palabras, la clase `Parabola` es un caso especial de la clase `Line`, por lo tanto cualquier instancia de `Parabola` es una instancia de `Line` pero no viceversa.

Código 6.4. Ejemplos de herencia y jerarquía.

```
l = Line(-1,1)
print(isinstance(l, Line))
print(isinstance(l, Parabola))
```

```
True
False
```

```
p = Parabola(1,2,3)
print(isinstance(p, Parabola))
print(isinstance(p, Line))
```

```
True
True
```

```
print(issubclass(Parabola, Line))
print(issubclass(Line, Parabola))
```

```
True
False
```

```
print(p.__class__ == Parabola)
print(p.__class__.__name__)
```

```
True
Parabola
```

Se ha dicho que una subclase es un caso especial de una superclase. En el ejemplo, la clase `Parabola` es un caso específico de la clase `Line`, sin embargo matemáticamente una parábola no es un caso específico de una linea, en realidad una línea es un caso específico de una parábola cuando $c_2 = 0$.

Dado este hecho quizá valga la pena redefinir las clases para corregir esta discrepancia.

Código 6.5. Mejora del código.

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0 = c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2

    def table(self, a, b, n):
        """ Devuelve una tabla con n puntos dado a <= x <= b """
        tabla = PrettyTable(field_names = ['x', 'f(x)'], float_format='.6')
        for x in np.linspace(a, b, n):
            y = self(x)
            tabla.add_row([x, y])
        return tabla

    def __str__(self):
        return f'y = {self.c0} + {self.c1}x + {self.c2}x²'

class Line(Parabola):
    def __init__(self, c0, c1):
        super().__init__(c0, c1, 0)
```

Código 6.6. Diferenciación Numérica.

Una tarea común en el cómputo científico es la diferenciación e integración que pueden ser resueltas por diversos métodos numéricos. Muchos de esos métodos están muy relacionados entre sí, por lo que pueden ser agrupados en familias de métodos. Una fórmula simple de cálculo es la siguiente:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

que puede ser implementada por la clase:

```
class Derivative:
    def __init__(self, f, h=1e-5):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h)-f(x))/h
```

Para hacer uso del método, sólo creamos una función y una instancia de la clase. Posteriormente hacemos una llamada a la instancia con el valor a evaluar. Por ejemplo:

$$f(x) = e^{-x}\sin(4\pi x)$$
$$f'(1.2) = ?$$

```
from math import exp, sin, pi

def f(x):
    return exp(-x) * sin(4*pi*x)

dfdx = Derivative(f)
print(dfdx(1.2))
```

```
-3.239208844119101
```

Sim embargo, dado que hay diversas fórumlas para encontrar la aproximación a la derivada:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$
$$f'(x) \approx \frac{4}{3}\frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h) - f(x-2h)}{4h}$$

entre otras.

Por ello se puede escribir fácilmente un módulo que ofrezca múltiples fórmulas.

```
class Forward:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h)-f(x))/h

class Central2:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h)-f(x-h))/2/h

class Central4:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h

    def __call__(self, x):
        f, h = self.f, self.h
        return 4/3* (f(x+h) - f(x-h))/(2*h) - 1/3 * (f(x+2*h) - f(x-2*h))*(4*h)
```

El problema en este código es la repetición de código para el constructor. Para resolverlo se puede crear una superclase que contenga el constructor e implementar una subclase por método.

La superclase queda de la siguiente forma:

```
class Diff:
    def __init__(self, f, h=1e-5):
        self.f, self.h = f, h
```

Y las subclases se definen de la siguiente forma:

```
class Forward(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

class Central2(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/2/h

class Central4(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (4/3) * (f(x+h) - f(x-h))/(2*h) - (1/3) * (f(x+2*h) - f(x-2*h))/(4*h)
```

```
from math import exp, sin, pi

def f(x):
    return exp(-x) * sin(4*pi*x)

dfdx = Forward(f)
print(dfdx(1.2))

dfdx = Central2(f)
print(dfdx(1.2))

dfdx = Central4(f)
print(dfdx(1.2))
```

```
-3.239208844119101
-3.2391005667389834
-3.2391005760477407
```

Ahora hagamos una concentración de aproximaciones con todas las fórmulas:

```
# from Diff import Forward, Central2, Central4
from math import pi, sin, cos
from prettytable import PrettyTable

H = [(1/2)**i for i in range(10)]
x0 = pi/4

table = PrettyTable(field_names=['h', 'Forward', 'Central 2', 'Central 4'])
for h in H:
    f1 = Forward(sin, h)
```

```
    c2 = Central2(sin, h)
    c4 = Central4(sin, h)
    table.add_row([h, f1(x0), c2(x0), c4(x0)])

print(table)
```

```
+-------------+--------------------+--------------------+--------------------+
|      h      |      Forward       |     Central 2      |     Central 4      |
+-------------+--------------------+--------------------+--------------------+
|     1.0     | 0.2699544827129282 | 0.5950098395293859 | 0.6861847232685281 |
|     0.5     | 0.5048856975964859 | 0.6780100988420897 | 0.7056768519463243 |
|     0.25    | 0.611835119448811  | 0.6997640691250939 | 0.707015392552762  |
|    0.125    | 0.6611301360648314 | 0.7052667953545546 | 0.7071010374310415 |
|    0.0625   | 0.6845566203276636 | 0.7066465151141275 | 0.7071064217006517 |
|   0.03125   | 0.6959440534591259 | 0.706991697811663  | 0.7071067587108415 |
|   0.015625  | 0.7015538499518499 | 0.7070780092891873 | 0.7071067797816953 |
|  0.0078125  | 0.7043374663312676 | 0.7070995881463489 | 0.7071067810987361 |
|  0.00390625 | 0.705723916746507  | 0.7071049829223881 | 0.7071067811810678 |
| 0.001953125 | 0.706415797873774  | 0.7071063316202526 | 0.7071067811862074 |
+-------------+--------------------+--------------------+--------------------+
```

6.2. Funciones recursivas

La recursividad es un fenómeno presente en la naturaleza. La recursividad es un concepto donde una función, procedimiento o proceso se define en términos de sí mismo. Es una técnica fundamental en matemáticas y ciencias de la computación, pero también se puede observar en la naturaleza y en otros campos.

Ejemplos de Recursividad en la Naturaleza

1. **Fractales:**
   - **Helechos:** Las hojas de un helecho son ejemplos clásicos de recursividad. Cada hoja está compuesta de pequeñas hojas que tienen una forma similar a la hoja completa.
   - **Brócoli Romanesco:** Este vegetal tiene una estructura fractal, donde cada florete se asemeja a una versión en miniatura de la planta completa.
2. **Conchas Marinas:**
   - Muchas conchas marinas, como las de los nautilos, muestran patrones recursivos en su estructura, donde el crecimiento de la concha sigue un patrón logarítmico.
3. **Ramas de Árboles:**
   - La ramificación de los árboles es un ejemplo de recursividad. Cada rama principal se divide en ramas más pequeñas, y estas a su vez se dividen en ramas aún más pequeñas, siguiendo un patrón similar.
4. **Ríos y Deltas:**
   - Los sistemas fluviales también muestran recursividad. Un río principal se divide en afluentes, y estos a su vez en arroyos más pequeños, siguiendo una estructura jerárquica.
5. **Sistemas Circulatorios:**

- El sistema circulatorio de muchos organismos, donde los grandes vasos sanguíneos se dividen en vasos más pequeños y así sucesivamente, es un ejemplo de recursividad biológica.

6. **Nervios y Neuronas:**
   - Las estructuras neuronales, con dendritas que se ramifican en subdendritas y axones que se ramifican en terminales más pequeños, también son recursivas.

6.2.1. Concepto Matemático

En matemáticas, un ejemplo clásico de recursividad es la secuencia de Fibonacci, donde cada término se define como la suma de los dos términos anteriores:

$$F(n) = F(n-1) + F(n-2) \forall n \geq 2$$

con $F(0) = 0$ y $F(1) = 1$.

La recursividad es una herramienta poderosa porque permite resolver problemas complejos dividiéndolos en subproblemas más simples del mismo tipo.

Código 6.7. Serie de Fibonacci

```
# Serie de Fibonacci

def Fibonacci(n):
    if n==0 or n==1:
        return n
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

for i in range(20):
    print(Fibonacci(i))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

6.2.2. Razón áurea (proporción dorada)

La razón áurea, también conocida como número áureo o proporción dorada, es un número irracional denotado por la letra griega phi (φ).

Su valor es aproximadamente 1.6180339887.

Se define algebraicamente como:

$$\varphi = \frac{1+\sqrt{5}}{2}$$

6.2.3. Propiedades:

1. **Proporción Ideal:** El número surge de la división en dos de un segmento guardando las siguientes proporciones: La longitud total $a+b$ es al segmento más largo a, como a es al segmento más corto b, entonces esa proporción es la razón áurea:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi$$

1. **Aparición en la Naturaleza:** La razón áurea aparece en diversas formas en la naturaleza, como en las proporciones de las conchas de nautilo, la disposición de las hojas, y las espirales de las piñas.
2. **Uso en Arte y Arquitectura:** Ha sido utilizada en obras de arte y arquitectura para crear estéticamente agradables proporciones, como en el Partenón de Atenas o en las obras de Leonardo da Vinci.
3. **Relación con la Secuencia de Fibonacci:** La razón áurea se relaciona con la serie de Fibonacci a través de la relación de los términos consecutivos de la serie. A medida que avanzas en la secuencia de Fibonacci, la proporción de un número con el anterior se aproxima a la razón áurea.

$$\lim_{n\to\infty} \frac{F(n+1)}{F(n)} = \varphi$$

   Este límite tiende a $ \varphi \approx 1.6180339887$ cuando n tiende a infinito.

La razón áurea ha fascinado a matemáticos, artistas y arquitectos durante siglos debido a su presencia en patrones naturales y su aplicación en el diseño.

Número áureo

```
n=35
Fibonacci(n)/Fibonacci(n-1)
```

```
1.6180339887499087
```

Código 6.8. Suma recursiva. Para sumar una lista de números se puede utilizar la función nativa `sum` de Python, pero hagamos una propia que utilice el concepto de recursividad.

```
def miSuma(L):
    if not L:
        return 0
    else:
        return L[0] + miSuma(L[1:])
```

```
print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
```

```
15
5050
```

Código 6.9. Una versión alternativa del código empleando la expresión tenaria `if-else` de Python.

```
def miSuma(L):
    return 0 if not L else L[0] + miSuma(L[1:])

print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
```

```
15
5050
```

Código 6.10. Otra versión que soporta no sólo números.

```
def miSuma(L):
    return L[0] if len(L) == 1 else L[0] + miSuma(L[1:])


print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
print(miSuma(['hola', 'mundo', 'cruel']))
```

```
15
5050
holamundocruel
```

Código 6.11. Una versión que además de no sólo soportar números, utiliza el desempaquetado de la lista.

```
def miSuma(L):
    first, *rest = L    # Desempaquetado: Toma el primer elemento de la lista L y lo asigna a first,
    return first if not rest else first + miSuma(rest)

print(miSuma([1,2,3,4,5]))
print(miSuma(list(range(101))))
print(miSuma(['hola', 'mundo', 'cruel']))
```

```
15
5050
holamundocruel
```

6.2.4. Recursividad vs Ciclos

La recursividad y los ciclos son dos enfoques para repetir acciones en un programa. Aquí están sus diferencias:

Recursividad

1. **Definición:**

- Una función se llama a sí misma para resolver subproblemas más pequeños.

2. **Estructura:**
   - Requiere un caso base para detener las llamadas recursivas.
   - Cada llamada crea un nuevo contexto en la pila de llamadas.
3. **Ventajas:**
   - Más intuitiva para problemas que tienen una naturaleza divisoria, como el recorrido de árboles o la generación de fractales.
   - Puede ser más fácil de entender y escribir para ciertos problemas.
4. **Desventajas:**
   - Puede causar desbordamiento de pila si la profundidad de la recursión es muy grande.
   - A menudo es menos eficiente en términos de memoria y tiempo debido a las múltiples llamadas a funciones.

Ciclos

1. **Definición:**
   - Utilizan estructuras de control (como `for`, `while`) para repetir acciones.
2. **Estructura:**
   - No involucran llamadas repetidas de función.
   - Mantienen el control en un único contexto de ejecución.
3. **Ventajas:**
   - Más eficiente en términos de uso de memoria, ya que no hay múltiples contextos.
   - Evita problemas de desbordamiento de pila.
4. **Desventajas:**
   - A veces puede ser menos intuitivo para problemas que se modelan naturalmente de manera recursiva.

Podría pensarse que las estructuras cíclicas son más sencillas y pueden realizar todo aquello que las formas recursivas pueden. Por ejemplo, la versión cíclica de la función suma anterior es:

Código 6.12. Sumas cíclicas

```
# Suma cíclica
L = [1, 2, 3, 4, 5]

sum = 0
while L:
    sum += L[0]
    L = L[1:]

sum
```

```
15
```

```
# Suma cíclica, versión alternativa
L = [1, 2, 3, 4, 5]

sum = 0
for x in L:
    sum += x

sum
```

```
15
```

Sin embargo hay ocasiones donde la forma recursiva es más simple que una cíclica. Por ejemplo, considere el caso de la suma de números en una lista anidada:

```
[1, [2, [3, 4], 5], 6, [7, 8], [[9], [10]]]
```

En casos como este la programación cíclica no sirve dado que no es una iteración lineal. Para ello sería necesario agregar más código de manera que los ciclos puedan funcionar. Por otro lado, una versión recursiva se acomoda mejor debido a la construcción recursiva de la propia lista anidada.

Veamos el código 6.13:

Código 6.13. Suma recursiva

```
def miSuma(L):
    total = 0
    for x in L:
        if not isinstance(x, list):
            total += x
        else:
            total += miSuma(x)
    return total

print(miSuma([1, [2, [3, 4], 5], 6, [7, 8], [[9], [10]]]))
```

```
55
```

6.2.5. Llamada indirecta de funciones

Debido a que en Python las funciones son objetos, se pueden escribir programas que las procesen de forma genérica. Las funciones como objeto pueden ser asignadas a otro nombre, pasadas a otras funciones, incrustarlas en estructuras de datos, devueltas por una función, entre otras cosas más.

El nombre utilizado al declarar una función a través de `def`, es sólo una variable asignada al contexto actual. Luego de que el operador `def` sea ejecutado, el nombre sólo es una referencia al objeto.

```
# Llamada directa
def echo(message):
    print(message)

echo("Hola mundo cruel")
```

```
Hola mundo cruel
```

```
# Llamada indirecta
x = echo
```

```
x("Hola mundo cruel")
```

```
Hola mundo cruel
```

Debido a que los argumentos se pasan como una asignación de objetos, es posible pasar funciones como argumento para otras funciones.

```
def indirect(func, arg):
    func(arg)

indirect(echo, "Llamada por argumento")
```

```
Llamada por argumento
```

Incluso es posible incluir funciones en estructuras de datos, como si fueran números enteros o cadenas. El siguiente ejemplo, incluye la función dos veces en una lista de tuplas, como una especie de tabla de acciones. Debido a que los tipos compuestos de Python como estos pueden contener cualquier tipo de objeto, tampoco hay ningún caso especial aquí:

```
myList = [(echo, "Hola"), (echo, "mundo"), (echo, "cruel")]
for (func, arg) in myList:
    func(arg)
```

```
Hola
mundo
cruel
```

En el siguiente ejemplo, se define una función como y se devuelve resultado de otra función.

```
def make(label):
    def echo(message):
        print(label + ':' + message)
    return echo

F = make("Hola")
F(" Mundo Cruel")
```

```
Hola: Mundo Cruel
```

En este ejemplo, se define una función `make` que recibe un argumento `label` que se pasa como argumento a la función `echo`. Esta última función recibe el argumento, forma un mensaje y lo imprime. Finalmente la función make devuelve la función `echo`.

6.3. Programación funcional

6.3.1. Funciones anónimas `lambda`

Además del parámetro `def` para definir funciones, Python ofrece la expresión `lambda` para definir funciones. Al igual que con `def`, `lambda` permite crear una función para ser utilizada posteriormente pero devuelve la función en lugar de asignarlo a un nombre. Es por ello que se les conoce como *funciones anónimas*.

Para declarar una función anónima, se utiliza la palabra clave `lambda` seguida de uno o más argumentos, y separado con dos puntos la expresión a evaluar por la función.

```
lambda argument1, argument2, ..., argumentN : expression using arguments
```

Las funciones `lambda` operan de forma idéntica a las funciones `def`, pero existen algunas diferencias entre ellas.

- `lambda` **es una expresión, no una sentencia**. Debido a esto, las funciones lambda pueden aparecer en lugares donde la sintáxis de Python no lo permite (dentro de una lista, en los argumentos de una función, entre muchos otros). Debido a que es una expresión, devuelve un valor (una función) que puede ser opcionalmente asignada a un nombre. A diferencia de `def` que siempre asigna la función al nombre indicado.
- **El cuerpo de** `lambda` **es una sola expresión, no un bloque de sentencias.** El contenido de las funciones `lambda` es similar al de las funciones `def`, pero sólo se escribe la expresión a evlauar sin ser necesario devolverlo explícitamente. Dado que las funciones `lambda` están limitadas a una sola expresión, son menos generales que las funciones `def`. Esta limitante es intencional por diseño, de manera que `lambda` está diseñada para usarse para operaciones simples y `def` para operaciones más largas.

A pesar de sus diferencias, las funciones `def` y `lambda` pueden hacer el mismo trabajo:

```
# Función def
def funcion(x,y,z):
    return x+y+z
funcion(3,4,5)
```

```
12
```

```
# Función lambda
funcion = lambda x,y,z : x+y+z
funcion(3,4,5)
```

```
12
```

Otro ejemplo con función `lambda`:

```
def knights():
    title = 'Sir'
    action = lambda x : title + ' ' + x
    return action

act = knights()
act("Robin")
```

```
'Sir Robin'
```

En este ejemplo, la función `knigths` devuelve una función `lambda`. Por ello `act` es nombre de variable que apunta a un objeto función `lambda` asociado en la llamada indirecta. Cuado se invoca `act('Robin')`, la función `lambda` recibe el argumento en `x` y realiza la operación definida.

```
act
```

```
<function __main__.knights.<locals>.<lambda>(x)>
```

6.3.2. ¿Porqué utilizar funciones `lambda`?

Este tipo de funciones resultan útiles en casos donde se quiere incrustar la definición de la función dentro del código que la utiliza. Aunque su uso es opcional, tienden a generar código más simple en escenarios donde se necesita crear bloques de código concisos e independientes.

Las funciones lambda se utilizan comunmente para codificar *jump tables*, que son listas o diccionarios de acciones que se deben realizar bajo demanda. Por ejemplo:

```
L = [lambda x:x**2, lambda x:x**3, lambda x:x**4]   # Lista de 3 funciones

for f in L:
    print(f(2))

print(L[0](3))
```

```
4
8
16
9
```

Este código es muy simple e intuitivo, sin embargo, es posibble obtener el mismo resultado mediante funciones `def`.

```
def f1(x): return x**2
def f2(x): return x**3
def f3(x): return x**4

L = [f1, f2, f3]

for f in L:
    print(f(2))
```

```
4
8
16
```

El resultado es el mismo pero el código es más simple.

Para el caso de los diccionarios veamos el ejemplo:

```
key = 'got'
D = {
    'already': lambda x: x+4,
    'got': lambda x: x*4,
    'one' : lambda x: x**4
}

D[key](3)
```

```
12
```

Ahora veamos el caso de funciones `lambda` anidadas, pero primero hagamos un ejemplo con uina función `def` y dentro una `lambda`.

```
def action(x):
    return lambda y : x+y

act = action(99)
print(act)
act(2)
```

```
<function action.<locals>.<lambda> at 0x7fa3ccd81580>
```

```
101
```

Ahora la misma operación pero con un par de funciones `lambda`, en una sola línea.

```
action = lambda x : (lambda y : x+y)
act = action(99)
act(3)
```

```
102
```

6.3.3. Funciones `map` sobre secuencias

Otra tarea común al utilizar listas es aplicar una operación a todos los elementos y recolectar un resultado. Por ejemplo, actualizar todos los elementos en una lista se puede hacer fácilmente con un ciclo empleando programación convencional.

```
counters = [1,2,3,4]
updated = []

for x in counters:
    updated.append(x+10)
updated
```

```
[11, 12, 13, 14]
```

Pero dado que este tipo de operaciones son muy comúnes, Python ofrece una función nativa que hace este trabajo. La función `map` aplica una función pasada com oargumento a cada uno de los elementos de una lista y devuelve otra lista con los resultados. Por ejemplo:

```
def inc(x):
    return x+10

list(map(inc, counters))
```

```
[11, 12, 13, 14]
```

La función `map` llama a la función pre-existente inc para cada uno de los elementos de la lista y los almacena en una nueva lista.

Dado que `map` espera recibir una función como argumento, es un buen lugar para utilizar una función `lambda`.

```
counters = [1,2,3,4]
list(map(lambda x : x+10, counters))
```

```
[11, 12, 13, 14]
```

Es posible replicar el funcionamiento de `map` con programación ciclos y programación convencional. Sin embargo `map` es una función nativa de Python, por lo que siempre está disponible, funciona siempre igual y es más rápida que su versión equivalente con ciclos `for`.

Además `map` puede utilizarse en situaciones más complicadas. Por ejemplo, si la función recibida como argumento por `map` requiere a su vez más de un argumento, se pueden agregar todos los argumentos necesarios y `map` se los hace llegar a la función.

Por ejemplo, considere la función `pow` que requiere dos argumentos:

```
pow(2,3)
```

```
list(map(pow, [1,2,3,4], [5,6,7,8]))
```

```
[1, 64, 2187, 65536]
```

Para una función que requiere n argumentos, `map` espera n secuencias para esa función.

6.3.4. Herramientas de programación funcional: `filter` y `reduce`

La función `map` es la representación más simple de las funciones nativas de Python para la *programación funcional*, que son herramientas que aplican funciones a secuencias y otros iterables.

Estas herramientas filtran elementos de acuerdo a una función de prueba (filtro) y aplican funciones a pares de elementos y resultados de ejecución (reduce). Debido a que devuelven iterables, range y filter requieren llamadas de lista para mostrar todos sus resultados.

Filter

Por ejemplo, el siguiente filtro selecciona elementos en una secuencia dada por `range(-5, 5)` que son mayores que cero:

```
list(filter(lambda x:x>0, range(-5, 5)))
```

```
[1, 2, 3, 4]
```

Los elementos de la secuencia o iterable que cumplen la condición dada por la función `lambda` son añadidos a la lista resultante. Esta función, al igual que con `map`, puede ser construida con ciclos `for`, pero es más rápida y nativa.

```
res = []
for x in range(-5, 5):
    if x>0:
        res.append(x)
res
```

```
[1, 2, 3, 4]
```

Reduce

`reduce` es una función nativa contenida dentro del modulo `functools` y acepta un iterador para proceder pero `reduce` no es un iterador ya que devuelve un sólo resultado.

Por ejemplo, `reduce` llama la suma y multiplicación para los elementos de la lista:

```
from functools import reduce

print(reduce(lambda x,y : x+y, [1,2,3,4]))
print(reduce(lambda x,y : x*y, [1,2,3,4]))
```

```
10
24
```

En cada paso, `reduce` pasa el valor actual de la suma o multiplicación junto con el siguiente valor de la lista a la función `lambda`. Por defecto, el primer valor de la lista inicializa el acumulador para el resultado final.

El siguiente código es una versión equivalente empleando ciclo `for`.

```
L = [1,2,3,4]
res = L[0]
for x in L[1:]:
    res += x
res
```

```
10
```

Para fines ilustrativos y entender el funcionamiento de `reduce` a fondo, éste es un código que replica el funcionamiento de `reduce`.

```
def myReduce(function, sequence):
    result = sequence[0]
    for next in sequence[1:]:
        result = function(result, next)
    return result

print(myReduce(lambda x,y : x+y, [1,2,3,4]))
print(myReduce(lambda x,y : x*y, [1,2,3,4]))
```

```
10
24
```

En el código se puede observar que `result` fue inicializado en el primer elemento de la lista. Sin embargo `reduce` acepta un tercer argumento que se utiliza como valor inicial, o incluso final si la lista estuviese vacía.

```
print(reduce(lambda x,y : x+y, [1,2,3,4], 10))
print(reduce(lambda x,y : x*y, [1,2,3,4], 5))
```

```
20
120
```

El valor inicial no debe ser necesariamente un número, puede ser cualquier tipo de dato hasta incluso un objeto. Debe considerarse el tipo que sea dicho valor inicial para las operaciones que `reduce` llevará a cabo con dicho valor inicial.

6.4. Referencias

- Sundnes J., Introduction to Scientific Programming with Python, Springer Open, 2020.
- Lutz M., Learning Python, O'Reilly, 2009.

6.5. Ejercicios

6.5.1. map, filter, lambda

Ejercicio 6.1. Escribe una función lambda que tome dos números y devuelva su producto.

```
product = lambda x, y: x * y
print(product(4, 5))
```

```
20
```

Ejercicio 6.2. Usa una función lambda dentro de map para sumar 10 a cada número en una lista.

```
numbers = [1, 2, 3, 4, 5]
result = list(map(lambda x: x + 10, numbers))
print(result)
```

```
[11, 12, 13, 14, 15]
```

Ejercicio 6.3. Usa una función lambda para ordenar una lista de tuplas en función del segundo elemento de cada tupla.

```
tuples = [(1, 2), (3, 1), (5, 4)]
sorted_tuples = sorted(tuples, key=lambda x: x[1])
print(sorted_tuples)  # Salida esperada: [(3, 1), (1, 2), (5, 4)]
```

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

Ejercicio 6.4. Usa una función lambda dentro de sorted para ordenar una lista de cadenas en función de su longitud.

```
strings = ["Hola", "Mundo", "Python", "es", "genial"]
sorted_strings = sorted(strings, key=lambda s: len(s))
print(sorted_strings)
```

```
['es', 'Hola', 'Mundo', 'Python', 'genial']
```

Ejercicio 6.5. Filtrar y transformar una lista de diccionarios. Escribe un programa que tome una lista de diccionarios, donde cada diccionario representa a una persona con llaves nombre y edad. Devuelve una nueva lista de nombres de personas que tengan al menos 18 años.

```
def mayores_de_edad(personas):
    return list(
                map(
                    lambda persona: persona['nombre'],
                    filter(
                        lambda persona: persona['edad'] >= 18,
                        personas
                    )
                )
            )

# Ejemplo de uso
personas = [
    {"nombre": "Ana", "edad": 22},
    {"nombre": "Luis", "edad": 17},
    {"nombre": "Marta", "edad": 19},
    {"nombre": "Carlos", "edad": 15}
]
nombres_mayores = mayores_de_edad(personas)
print(nombres_mayores)  # Output: ['Ana', 'Marta']
```

```
['Ana', 'Marta']
```

Ejercicio 6.6. Aplicar múltiples funciones a una lista de números. Escribe un programa que tome una lista de números y aplique dos funciones diferentes a cada número: una que calcule el cuadrado y otra que calcule el cubo. Devuelve una lista de tuplas donde cada tupla contiene el resultado de ambas funciones.

```
def cuadrado_y_cubo(lista):
    return list(
        map(
            lambda x: (x ** 2, x ** 3),
            lista
        )
    )
```

```
# Ejemplo de uso
numeros = [1, 2, 3, 4]
resultados = cuadrado_y_cubo(numeros)
print(resultados)  # Output: [(1, 1), (4, 8), (9, 27), (16, 64)]
```

```
[(1, 1), (4, 8), (9, 27), (16, 64)]
```

Ejercicio 6.7. Ordenar una lista de tuplas basada en la suma de sus elementos. Escribe un programa que tome una lista de tuplas, donde cada tupla contiene dos números. Devuelve una nueva lista de tuplas ordenadas por la suma de sus elementos.

```
def ordenar_por_suma(lista):
    return list(
        map(
            lambda x: x,
            sorted(
                lista,
                key = lambda x: x[0] + x[1]
            )
        )
    )

# Ejemplo de uso
tuplas = [(1, 2), (3, 4), (1, 1), (2, 2)]
tuplas_ordenadas = ordenar_por_suma(tuplas)
print(tuplas_ordenadas)  # Output: [(1, 1), (1, 2), (2, 2), (3, 4)]
```

```
[(1, 1), (1, 2), (2, 2), (3, 4)]
```

Ejercicio 6.8. Convertir una lista de tuplas a un diccionario. Escribe un programa que tome una lista de tuplas, donde cada tupla contiene una llave y un valor. Devuelve un diccionario construido a partir de estas tuplas.

```
def tuplas_a_diccionario(lista):
    return dict(map(lambda x: (x[0], x[1]), lista))

# Ejemplo de uso
tuplas = [("llave1", "valor1"), ("llave2", "valor2"), ("llave3", "valor3")]
diccionario = tuplas_a_diccionario(tuplas)
print(diccionario)  # Output: {'llave1': 'valor1', 'llave2': 'valor2', 'llave3': 'valor3'}
```

```
{'llave1': 'valor1', 'llave2': 'valor2', 'llave3': 'valor3'}
```

Ejercicio 6.9. Aplicar una función a una lista de listas. Escribe un programa que tome una lista de listas de números y aplique una función que calcule el promedio de cada lista interna. Devuelve una lista de promedios.

```
def promedios_de_listas(lista):
    return list(
        map(
            lambda x: sum(x) / len(x) if len(x) > 0 else 0,
            lista
        )
    )
```

```
# Ejemplo de uso
listas = [[1, 2, 3], [4, 5, 6, 7], [8, 9], []]
promedios = promedios_de_listas(listas)
print(promedios)  # Output: [2.0, 5.5, 8.5, 0]
```

```
[2.0, 5.5, 8.5, 0]
```

6.5.2. Reduce

Ejercicio 6.10. Usa una función lambda dentro de reduce para calcular el producto de todos los números en una lista.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Salida esperada: 120
```

```
120
```

Ejercicio 6.11. Suma de una lista de números. Escribe un programa que tome una lista de números y devuelva la suma de todos los números en la lista.

```
from functools import reduce

def suma(lista):
    return reduce(lambda x, y: x + y, lista)

# Ejemplo de uso
numeros = [1, 2, 3, 4, 5]
resultado = suma(numeros)
print(resultado)  # Output: 15
```

```
15
```

Ejercicio 6.12. Encontrar el máximo en una lista de números. Escribe un programa que tome una lista de números y devuelva el número más grande en la lista.

```
from functools import reduce

def maximo(lista):
    return reduce(lambda x, y: x if x > y else y, lista)

# Ejemplo de uso
numeros = [1, 7, 3, 9, 5]
resultado = maximo(numeros)
print(resultado)  # Output: 9
```

```
9
```

Ejercicio 6.13. Contar la frecuencia de elementos en una lista. Escribe un programa que tome una lista de elementos y devuelva un diccionario con la frecuencia de cada elemento en la lista.

```
from functools import reduce
```

```
def frecuencia(lista):
    return reduce(lambda acc, x: {**acc, x: acc.get(x, 0) + 1}, lista, {})

# Ejemplo de uso
elementos = ['a', 'b', 'a', 'c', 'b', 'a']
resultado = frecuencia(elementos)
print(resultado)  # Output: {'a': 3, 'b': 2, 'c': 1}
```

```
{'a': 3, 'b': 2, 'c': 1}
```

Ejercicio 6.14. Calcular la diferencia entre el número más grande y el más pequeño en una lista. Escribe un programa que tome una lista de números y devuelva la diferencia entre el número más grande y el más pequeño en la lista.

```
from functools import reduce

def diferencia_max_min(lista):
    maximo = reduce(lambda x, y: x if x > y else y, lista)
    minimo = reduce(lambda x, y: x if x < y else y, lista)
    return maximo - minimo

# Ejemplo de uso
numeros = [1, 7, 3, 9, 5]
resultado = diferencia_max_min(numeros)
print(resultado)  # Output: 8
```

```
8
```

Ejercicio 6.15. Encontrar el número de palabras en una lista de cadenas. Escribe un programa que tome una lista de cadenas y devuelva el número total de palabras en todas las cadenas.

```
from functools import reduce

def contar_palabras(lista):
    return reduce(lambda acc, x: acc + len(x.split()), lista, 0)

# Ejemplo de uso
cadenas = ["hola mundo", "python es genial", "reduce es útil"]
resultado = contar_palabras(cadenas)
print(resultado)  # Output: 8
```

```
8
```

Capítulo 7

Iterables

Un iterador es un objeto que contiene una cantidad contable de valores.

Un iterador es un objeto que se puede iterar, lo que significa que se pueden recorrer todos los valores.

Técnicamente, en Python, un iterador es un objeto que implementa el protocolo `iterator`, que consta de los métodos `__iter__()` y `__next__()`.

Python tiene varios métodos nativos que procesan iterables:

- `sort`: ordena elementos en un iterable.
- `zip`: combina elementos de iterables.
- `enumerate`: empareja elementos en un iterable con posiciones relativas.
- `filter`: selecciona elementos para los que una función es verdadera.
- `reduce`: ejecuta pares de elementos en un iterable a través de una función.

7.1. Iterator vs Iterable

Las listas, tuplas, diccionarios y conjuntos son objetos iterables. Son contenedores iterables de los que se puede obtener un iterador.

Todos estos objetos tienen un método `iter()` que se utiliza para obtener un iterador:

Código 7.1. Creación y uso de un Iterador.

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
apple
banana
cherry
```

Incluso las cadenas String son objetos iterables, y pueden devolver un iterador.

Código 7.2. Iterando un *String*.

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

```
b
a
n
a
n
a
```

7.2. Recorriendo un iterador

Se puede utilizar un ciclo for para iterar a través del objeto iterable.

Código 7.3. Iterar los valores de una tupla.

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

```
apple
banana
cherry
```

Código 7.4. Iterar los caracteres de una cadena.

```
 mystr = "banana"

for x in mystr:
  print(x)
```

```
b
a
n
a
n
a
```

El ciclo `for` en realidad crea un objeto iterador y ejecuta el método `next()` para cada bucle.

7.3. Crear un iterador

Para crear un objeto/clase como iterador, debe implementar los métodos `__iter__()` y `__next__()` en su objeto.

Recordando la creación de clases, todas ellas tienen una función llamada `__init__()`, que le permite realizar algunas inicializaciones cuando se crea el objeto.

El método `__iter__()` actúa de manera similar, puede realizar operaciones (inicializar, etc.), pero siempre debe devolver el objeto iterador en sí.

El método `__next__()` también le permite realizar operaciones y debe devolver el siguiente elemento de la secuencia.

Código 7.5. Crear un iterador que devuelve números, comenzando con 1 y cada secuencia se incrementa en uno.

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
1
2
3
4
5
```

7.4. StopIteration

El código anterior iteraría indefinidamente para un número indefinido de llamadas `next()` o si fuese utilizado en un ciclo `for`. Para prevenir esto se puede utilizar la sentencia `StopIteration`.

En la definición del método `__next__()` se puede definir la condición de terminación que lanzará un error si se sobrepasa la cantidad de iteraciones definidas.

Código 7.6. Detener la iteración luego de 20 repeticiones.

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
```

```
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

7.5. Iterador `range`

Ya hemos utilizado con anterioridad el método `range`, pero ahora ya podemos especificar que el método devuelve un iterador que genera números bajo demanda, en lugar de construir la lista resultante en la memoria. Se puede forzar a que el iterador sea una lista de números mediante el método `list`.

Código 7.7. Forzar un iterador en una lista.

```
R = range(10)
R
```

```
range(0, 10)
```

```
I = iter(R)
print(next(I))
```

```
print(next(I))
print(I.__next__())
```

```
0
1
2
```

```
I = iter(R)
for x in R:
    print(next(I))
```

```
0
1
2
3
4
5
6
7
8
9
```

```
list(R)
```

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

7.6. Iteradores map, zip y filter

Al igual que `range`, los métodos `map`, `zip` y `filter` son iteradores en lugar de producir una lista con los resultados. Pero a diferencia de `range`, estos métodos métodos son sus propios iteradores. Es decir, después de recorrer sus resultados una vez, se agotan. En otras palabras, no puede tener múltiples iteradores en sus resultados que mantengan diferentes posiciones en esos resultados.

Código 7.8. En el siguiente código, `map` devuelve un iterador y no una lista.

```
M = map(abs, (-1, 0, 1))
M
```

```
<map at 0x7fa89c5382b0>
```

```
next(M)
```

```
1
```

```
M = map(abs, (-1, 0, 1))

for m in M:
    print(m)
```

```
1
0
1
```

```
M = map(abs, (-1, 0, 1))
list(M)
```

```
[1, 0, 1]
```

Código 7.9. La función `zip` devuelve un iterador que funciona de la misma forma.

```
Z = zip((1,2,3), (10,20,30))
Z
```

```
<zip at 0x7fa89c1c7500>
```

```
list(Z)
```

```
[(1, 10), (2, 20), (3, 30)]
```

```
Z = zip((1,2,3), (10,20,30))

for pair in Z:
    print(pair)
```

```
(1, 10)
(2, 20)
(3, 30)
```

```
Z = zip((1,2,3), (10,20,30))
next(Z)
```

```
(1, 10)
```

```
next(Z)
```

```
(2, 20)
```

El método nativo filter devuelve elementos en un iterable para los cuales la función pasada como argumento devolvió `True`. El método `filter` acepta un iterable para procesarlo y devuelve un iterable para los resultados generados.

Código 7.10. Filter devuelve un iterable.

```
filter(bool, ['spam', '', 'ni'])
```

```
<filter at 0x7fa89c7d9c90>
```

```
list(filter(bool, ['spam', '', 'ni']))
```

```
['spam', 'ni']
```

7.6.1. Iteradores múltiples y sencillos

El objeto `range` difiere de los demás métodos mencionados, ya que soporta `len` e indexación pero no es su propio iterador, (se crea uno con `iter` cuando se itera manualmente) y admite múltiples iteradores sobre su resultado que recuerdan sus posiciones de manera independiente.

Código 7.11. `range` no es su propio iterador.

```
R = range(3)
next(R)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[58], line 2
      1 R = range(3)
----> 2 next(R)
```

```
TypeError: 'range' object is not an iterator
```

```
I1 = iter(R)
next(I1)
```

```
0
```

```
next(I1)
```

```
1
```

```
I2 = iter(R)
next(I2)
```

```
0
```

```
next(I1)
```

```
2
```

Por el contrario, `zip`, `map` y `filter` no soportan múltiples iteradores activos en el mismo resultado.

Código 7.12. `zip`, `map` y `filter`

```
Z = zip((1, 2, 3), (10, 20, 30))
I1 = iter(Z)
I2 = iter(Z)
```

```
next(I1)
```

```
(1, 10)
```

```
next(I2)
```

```
(2, 20)
```

```
next(I1)
```

```
(3, 30)
```

```
next(I2)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[69], line 1
----> 1 next(I2)

StopIteration:
```

Para el caso de `map`:

Código 7.13. `map`.

```
M = map(abs, (-1, 0, 1))
I1 = iter(M)
I2 = iter(M)
print(next(I1), next(I1), next(I1))
print(next(I2))
```

```
1 0 1
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[74], line 5
      3 I2 = iter(M)
```

```
      4 print(next(I1), next(I1), next(I1))
----> 5 print(next(I2))

StopIteration:
```

Y en el caso de `range` para un código similar:
Código 7.14. `range`

```
R = range(3)
I1 = iter(R)
I2 = iter(R)
print(next(I1), next(I1), next(I1))
print(next(I2))
```

```
0 1 2
0
```

7.6.2. Iteradores de vista de Diccionario

En Python las claves, valores y métodos de elementos del diccionario devuelven objetos iterables `view` que generan elementos de resultado de a uno por vez, en lugar de producir listas de resultados de una sola vez en la memoria. Los elementos `view` mantienen el mismo orden físico que el del diccionario y reflejan los cambios realizados en el diccionario subyacente.

Código 7.15. `dict` devuelve un objeto iterable `view`.

```
D = dict(a=1, b=2, c=3)
D
```

```
{'a': 1, 'b': 2, 'c': 3}
```

```
K = D.keys()
K
```

```
dict_keys(['a', 'b', 'c'])
```

```
next(K)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[82], line 1
----> 1 next(K)

TypeError: 'dict_keys' object is not an iterator
```

```
I = iter(K)
next(I)
```

```
'a'
```

```
next(I)
```

```
'b'
```

```
for k in D.keys():
    print(k, end=' ')
```

```
a b c
```

Al igual que con todos los iteradores, siempre puedes forzar una vista de diccionario para que construya una lista real. Sin embargo, esto no suele ser necesario, excepto para mostrar resultados de forma interactiva o para aplicar operaciones de lista como la indexación:

Código 7.16. Forzar que `view` sea una lista.

```
K = D.keys()
list(K)
```

```
['a', 'b', 'c']
```

```
V = D.values()
list(V)
```

```
[1, 2, 3]
```

```
D.items()
```

```
dict_items([('a', 1), ('b', 2), ('c', 3)])
```

```
list(D.items())
```

```
[('a', 1), ('b', 2), ('c', 3)]
```

```
for (k, v) in D.items():
    print(f'{k} -> {v}')
```

```
a -> 1
b -> 2
c -> 3
```

Además, los diccionarios también tienen iteradores, que devuelven claves sucesivas, por lo que no suele ser necesario llamar a las claves directamente en este contexto.

Código 7.17. Los diccionarios también tienen iteradores.

```
D
```

```
{'a': 1, 'b': 2, 'c': 3}
```

```
I = iter(D)
next(I)
```

```
'a'
```

```
next(I)
```

```
'b'
```

No se necesita llamar a la función `keys()` para iterar el diccionario, pero `keys` también es un iterador.

Código 7.18. `keys` es un iterador.

```
for key in D:
    print(key)
```

```
a
b
c
```

Por último, recuerde nuevamente que, dado que `keys` ya no devuelve una lista, el patrón de codificación tradicional para escanear un diccionario por claves ordenadas no funcionará. En su lugar, convierta primero las vistas de `keys` con una llamada de lista, o utilice la llamada `sorted` en una vista de `keys` o en el diccionario mismo, de la siguiente manera.

Código 7.19. `sorted` en una vista de `keys`.

```
D = dict(a=1, c=3, b=2)
D
```

```
{'a': 1, 'c': 3, 'b': 2}
```

```
for k in sorted(D.keys()):
    print(f'{k} -> {D[k]}')
```

```
a -> 1
b -> 2
c -> 3
```

O mejor aún, una buena práctica para el ordenamiento de las llaves:

Código 7.20. Ordenamiento de llaves

```
D
```

```
{'a': 1, 'c': 3, 'b': 2}
```

```
for k in sorted(D):
    print(f'{k} -> {D[k]}')
```

```
a -> 1
b -> 2
c -> 3
```

7.7. Listas por Comprensión

En el capítulo anterior, estudiamos herramientas de programación funcional como `map` y `filter`, que mapean operaciones sobre secuencias y recopilan resultados. Debido a que esta es una tarea tan común en la codificación, Python finalmente generó una nueva funcionalidad: **listas por comprensión** (o la comprensión de listas), que es incluso más flexible que las herramientas anteriormente mencionadas.

En resumen, las listas por comprensión aplican una expresión arbitraria a los elementos de un iterable, en lugar de aplicar una función. Como tal, pueden ser herramientas más generales.

7.7.1. Listas por Comprensión vs. map

Trabajemos con un ejemplo que demuestra los conceptos básicos. La función nativa de Python `ord` devuelve el código ASCII de un caracter (la función nativa `chr` hace lo opuesto, devuelve el caracter de un código ASCII).

Código 7.21. Función `ord`.

```
ord('H')
```

```
72
```

Ahora supongamos que se desea tener una lista con el código ASCII de cada uno de los caracteres de una cadena. Una forma de realizarlo sería a través de un ciclo `for` e ir añadiendo cada valor en una lista.

Código 7.22. Lista de código ASCII.

```
res = []
for x in 'Hola':
    res.append(ord(x))
res
```

```
[72, 111, 108, 97]
```

Ahora empleando el método `map` para realizar la misma tarea.

Código 7.23. Usando `map` para la generación de la lista.

```
res = list(map(ord, "Hola"))
res
```

```
[72, 111, 108, 97]
```

Esta operación requirió menos código utilizando `map`.

Sin embargo, es posible obtener el mismo resultado empleando ahora listas por comprensión.

Mientras que `map` asocia (mapea) una *función* sobre una *secuencia*, las listas por comprensión asocian una *expresión* sobre una *secuencia*.

Código 7.24. Lista por comprensión.

```
res = [ord(x) for x in "Hola"]
res
```

```
[72, 111, 108, 97]
```

Las listas por comprensión recopilan los resultados de aplicar una expresión arbitraria a una secuencia de valores y los devuelven en una nueva lista. Sintácticamente, las listas por comprensión se encierran entre corchetes para enfatizar que construye una lista.

En su forma simple, dentro de los corchetes se codifica una expresión que nombra una variable seguida de un de bucle `for` que nombra la misma variable. Luego, Python recopila los resultados de la expresión para cada iteración del bucle implícito.

El resultado obtenido por el último código es el mismo con respecto a sus equivalentes con `for` y `map`. Sin embargo, las listas por comprensión se vuelven más convenientes cuando deseamos aplicar una expresión arbitraria a una secuencia.

Código 7.25. Lista por comprensión: aplicar una expresión a una secuencia.

```
[x**2 for x in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

La lista del código 7.25 recopila el cuadrado de los números enteros de 0 hasta 9. Si deseamos una versión del código empleando `map` se necesita de una función pequeña que haga el cuadrado de un número. Esta es una oportunidad de utilizar a su vez una función `lambda`, dado que no se utilizará para otro propósito adicional.

Código 7.26. Usando `map` y `lambda` para la generación de la lista.

```
list(map(lambda x:x**2, range(10)))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A pesar de que esta línea del código 7.26 realiza el mismo trabajo, requiere un poco más de código y resulta también un poco más difícil de leer debido a la función `lambda`. Para expresiones más avanzadas, las listas por comprensión típicamente requieren menos código.

7.7.2. Añadiendo pruebas y ciclos anidados

Las listas por comprensión son incluso más generales que lo que se ha mostrado hasta ahora. Por ejemplo, se puede codificar una cláusula `if` después del `for` para agregar lógica de selección.

Las listas por comprensión con cláusulas `if` pueden considerarse análogas al filtro incorporado que se analizó previamente: omiten elementos de secuencia para los que la cláusula `if` no es verdadera.

Para demostrarlo, aquí se muestran dos códigos que recogen números pares del 0 al 4; al igual que con la alternativa `map` a la lista por comprensión, la versión con `filter` requiere una pequeña función `lambda` para la expresión de prueba. A modo de comparación, aquí también se muestra el bucle `for` equivalente.

Código 7.27. Lista por comprensión vs. filter/lambda vs. for

```
# Lista por comprensión
[x for x in range(5) if x %2 ==0]
```

```
[0, 2, 4]
```

```
# filter y lambda
list(filter(lambda x:x%2==0, range(5)))
```

```
[0, 2, 4]
```

```
# ciclo for
res = []
for x in range(5):
    if x%2==0:
        res.append(x)
res
```

```
[0, 2, 4]
```

En todos los casos se utiliza el operador módulo (residuo de la división) para identificar números pares. El ćodigo que emplea `filter` no es considerablemente más largo que la versió de listas por comprensión. Sin embargo, podemos combinar una condicional `if` y una expresión arbitraria en nuestra lista por comprensión para darle el efecto de `filter` y `map`, en una sola expresión.

Código 7.28. Lista por comprensión combinado con un condicional.

```
[x**2 for x in range(10) if x%2==0]
```

```
[0, 4, 16, 36, 64]
```

Esta vez, recopilamos los cuadrados de los números pares del 0 al 9: el ciclo `for` omite los números para los cuales el condicional `if` a la derecha es falso, y la expresión a la izquierda calcula los cuadrados.

La versión del código equivalente con `map` requiere más trabajo: tenemos que combinar selecciones `filter` con iteración `map`, lo que daría como resultado una expresión notablemente más compleja.

Código 7.29. `list`, `map` y `filter`.

```
list(map(lambda x:x**2, filter(lambda x:x%2==0, range(10))))
```

```
[0, 4, 16, 36, 64]
```

Las listas por comprensión son aún más generales. Se puede codificar cualquier cantidad de ciclos `for` anidados en una lista por comprensión, y cada una puede tener una condicional `if` asociada opcional.

La estructura general de las listas por comprensión es la siguiente:

```
    [
        expression for target1 in iterable1 [if condition1]
                   for target2 in iterable2 [if condition2]
                   ...
                   for targetN in iterableN [if conditionN]
    ]
```

Cuando los ciclos `for` están anidados dentro de una lista por comprensión, funcionan como ciclos `for` anidados. Por ejemplo:

Código 7.30. Lista por comprensión con for anidados.

```
res = [x+y for x in [0, 1, 2] for y in [100, 200, 300]]
res
```

```
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

La versión equivalente del código:

Código 7.31. .

```
res = []
for x in [0, 1, 2]:
    for y in [100, 200, 300]:
        res.append(x+y)

res
```

```
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Aunque las listas por comprensión construyen listas, recuerde que pueden iterar sobre cualquier secuencia u otro tipo iterable. El siguiente código 7.32 recorre una cadena en lugar de una lista de números y, por lo tanto recopila resultados de una concatenación.

Código 7.32. Recorrido de dos cadenas en una lista por comprensión.

```
res = [x+y for x in 'hola' for y in 'HOLA']
res
```

```
['hH',
 'hO',
 'hL',
 'hA',
 'oH',
```

```
 'oO',
 'oL',
 'oA',
 'lH',
 'lO',
 'lL',
 'lA',
 'aH',
 'aO',
 'aL',
 'aA']
```

Por último, se muestra a contuación un código que muestra el efecto de añadir condicionales `if` a ciclos `for` anidados. El código debe formar pares ordenados donde su primer elemento sea par y su segundo elemento impar, empleando valores desde 0 hasta 4.

Código 7.33. .

```
[(x,y) for x in range(5) if x%2==0 for y in range(5) if y%2==1]
```

```
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

La versión equivalente empleando ciclos `for`:

```
res = []
for x in range(5):
    if x%2==0:
        for y in range(5):
            if y%2==1:
                res.append((x,y))

res
```

```
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

El código equivalente empleando `map` y `filter` es extremadamente complejo y profundamente anidado, así que ni siquiera intentaré mostrarlo aquí.

7.7.3. Listas por comprensión y matrices

Una forma básica de codificar matrices en Python es con listas anidadas. A continuación, se definen dos matrices de 3×3 como listas de listas anidadas.

Código 7.34. Listas anidadas

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

Dada esta estructura, se puede ubicar por índice los renglones y columnas, utilizando operaciones de índice ordinarias:

Código 7.35. Acceso a elementos en listas anidadas.

```
M[1]
```

```
[4, 5, 6]
```

```
M[1][2]
```

```
6
```

Sin embargo, las listas por comprensión son herramientas poderosas para procesar dichas estructuras, ya que escanean automáticamente las filas y columnas por nosotros. Por ejemplo, aunque esta estructura almacena la matriz por filas, para recolectar la segunda columna podemos simplemente iterar a través de las filas y extraer la columna deseada, o iterar a través de las posiciones en las filas e indexar a medida que avanzamos.

Código 7.36. Recorrido de renglones mediante lista anidada.

```
[row[1] for row in M]
```

```
[2, 5, 8]
```

```
[M[row][1] for row in (0, 1, 2)]
```

```
[2, 5, 8]
```

Dadas las posiciones, también podemos realizar tareas fácilmente, como extraer una diagonal.

La siguiente expresión usa `range` para generar la lista de elementos que se encuentran en el mismo índice para fila y columna, esto es, `M[0][0]`, `M[1][1]`, y así sucesivamente (suponiendo que la matriz tiene la misma cantidad de filas y columnas):

Código 7.37. Recorrido de elementos en la matriz mediante lista anidada.

```
[M[i][i] for i in range(len(M))]
```

```
[1, 5, 9]
```

También podemos usar listas por comprensión para combinar varias matrices. A continuación, primero se crea una lista plana que contiene el resultado de multiplicar las matrices por pares y luego se crea una estructura de lista anidada que tiene los mismos valores anidando listas por comprensión:

Código 7.38. Operaciones entre matrices con listas por comprensión.

```
[M[row][col] * N[row][col] for row in range(len(M)) for col in range(len(N))]
```

```
[2, 4, 6, 12, 15, 18, 28, 32, 36]
```

Y si anidamos la lista, para crear una lista bidimiensional:

```
[[M[row][col] * N[row][col] for col in range(len(M))] for row in range(len(N))]
```

```
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Esta última expresión funciona dado que la iteración del renglón ocurre en el ciclo exterior del anidamiento de ciclos `for`. Es decir, para cada renglón se iteran todas las columnas y se almacena en la lista resultante.

Código 7.39. Un código equivalente empleando ciclos `for`.

```
res = []
for row in range(len(M)):
    temp = []
    for col in range(len(N)):
        temp.append(M[row][col] * N[row][col])
    res.append(temp)

res
```

```
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Comparando ambas versiones, el código que emplea listas por comprensión sólo requiere una línea. Además de acuerdo a la documentación, se ejecuta más rápido para matrices más grandes.

7.8. Regresando a los iteradores: Generadores

Python proporciona herramientas que producen resultados solo cuando son necesarios, en lugar de hacerlo todos a la vez. En particular, dos construcciones del lenguaje retrasan la creación de resultados siempre que sea posible:

- *Funciones generador*. Las funciones de generador se codifican como declaraciones `def` normales, pero utilizan declaraciones `yield` para devolver los resultados de a uno por vez, suspendiendo y reanudando su estado entre cada uno.
- *Expresiones generador*. Las expresiones de generador son similares a las comprensiones de listas, pero devuelven un objeto que produce resultados a pedido en lugar de construir una lista de resultados.

Funciones generador: yield vs. return

Suspensión de estado A diferencia de las funciones normales que devuelven un valor y salen, las funciones generadoras suspenden y reanudan automáticamente su ejecución y estado alrededor del punto de generación del valor. Por eso, suelen ser una alternativa útil tanto para calcular una serie completa de valores por adelantado como para guardar y restaurar manualmente el estado en las clases. Debido a que el estado que las funciones generadoras conservan cuando se suspenden incluye todo su ámbito local, sus variables locales conservan información y la ponen a disposición cuando se reanudan las funciones.

La principal diferencia entre generador y funciones normales, es que el generador *genera* un valor, en lugar de devolverlo: la sentencia `yield` suspende la función y envía un valor de vuelta al invocador, pero conserva su estado para permitir que la función reanude desde allí. Por ende, cuando la función se reanuda continúa la ejecución en ese estado. Esto permite que el código genere una serie de valores a lo largo del tiempo, conforme se requieran, en lugar de calcularlos todos a la vez.

El protocolo de iteración Para verdaderamente comprender las funciones generadoras, es necesario saber que están estrechamente relacionadas con la noción del protocolo de iteración en Python. Como hemos visto, los objetos iterables definen un método `__next__`, que devuelve el siguiente elemento en la iteración o genera la excepción especial `StopIteration` para finalizar la iteración. El iterador de un objeto se obtiene con la función nativa `iter`.

El ciclo `for` de Python y todos los demás contextos de iteración, utilizan el protocolo de iteración para recorrer una secuencia o un generador de valores, si es que el protocolo está soportado. Si no, la iteración recurre a la indexación repetida de secuencias.

Para implementar este protocolo, las funciones que contienen `yield` se compilan como generadores. Cuando se les llama, devuelven un objeto generador que soporta la interfaz de iteración con un método creado automáticamente llamado `__next__` para reanudar la ejecución.

Las funciones generadoras también pueden tener un `return` que, además de definir el final del bloque def, terminan la generación de valores, técnicamente generando una excepción `StopIteration` después de cualquier salida de la función.

Desde la perspectiva de quien llamó a la función, el método `__next__` reanuda la función y se ejecuta hasta que se devuelva el siguiente resultado de `yield` o bien se genere un `StopIteration`.

Código 7.40. El siguiente código define una función generadora que se puede utilizar para generar el cuadrado de una serie de números.

```
def gensquares(N):
    for i in range(N):
        yield i**2
```

```
x = gensquares(5)
x
```

```
<generator object gensquares at 0x7fca5ac08790>
```

```
next(x)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[9], line 1
----> 1 next(x)

StopIteration:
```

```
next(x)
```

```
9
```

Código 7.41. Ahora, incrustando la función generadora en un ciclo.

```
for i in gensquares(10):
    print(i)
```

```
0
1
4
9
16
25
36
49
64
81
```

Código 7.42. Generador de Números Pares.

```
def generador_pares():
    n = 0
    while True:
        yield n
```

```
        n += 2

# Ejemplo de uso
pares = generador_pares()
for _ in range(20):
    print(next(pares))  # Output: 0, 2, 4, 6, 8
```

```
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
```

Código 7.43. Generador de Fibonacci

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Ejemplo de uso
fib = fibonacci()
for _ in range(10):
    print(next(fib))  # Output: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

```
0
1
1
2
3
5
8
13
21
34
```

Código 7.44. Generador de una Secuencia con un Rango Dado

```
def generador_rango(inicio, fin):
    while inicio < fin:
        yield inicio
        inicio += 1

# Ejemplo de uso
for numero in generador_rango(1, 5):
    print(numero)  # Output: 1, 2, 3, 4
```

```
1
2
3
4
```

Código 7.45. Generador que Filtra Números Pares

```
def filtrar_pares(lista):
    for numero in lista:
        if numero % 2 == 0:
            yield numero

# Ejemplo de uso
lista = [1, 2, 3, 4, 5, 6, 7, 8]
pares = filtrar_pares(lista)
for par in pares:
    print(par)  # Output: 2, 4, 6, 8
```

```
2
4
6
8
```

Código 7.46. Generador de aproximaciones al número π

$$\pi = 4 \times \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

```
iter = int(input("Introduzca el número de iteraciones: "))

pi = 0
for n in range(iter+1):
    pi += 4*(-1)**n / (2*n+1)

print(f'pi({iter}) = {pi}')
```

```
pi(100000) = 3.1416026534897203
```

```
def aproximacionPi():
    pi, n = 0, 0
    while True:
        yield n, pi
        pi += 4*(-1)**n / (2*n+1)
        n += 1
```

```
x = aproximacionPi()
```

```
next(x)
```

```
(1, 4.0)
```

```
N = int(input("Introduzca el número de iteraciones: "))
p = aproximacionPi()

for i in range(N):
    print(next(p))
```

```
(0, 0)
(1, 4.0)
(2, 2.666666666666667)
(3, 3.466666666666667)
(4, 2.8952380952380956)
```

7.8.1. Expresiones generadoras: Los iteradores se aproximan a las comprensiones

En todas las versiones recientes de Python, la noción de *iteradores* y *listas por comprensión* se combinan en una nueva característica del lenguaje: *expresiones generadoras*. En térmios de la sintáxis, las expresiones generadoras son como las listas por comprensión normales, pero se incluyen entre paréntesis en lugar de corchetes:

Código 7.47. Listas por comprensión vs. Generadores

```
[x**2 for x in range(5)]
```

```
[0, 1, 4, 9, 16]
```

```
(x**2 for x in range(5))
```

```
<generator object <genexpr> at 0x7fa0ca13e4d0>
```

De hecho, al menos en términos de su funcionalidad, codificar una lista por comprensión es esencialmente lo mismo que envolver una expresión de generador en una llamada incorporada de lista para obligarla a producir todos sus resultados en una lista a la vez, esto es:

Código 7.48. Un generador produce una lista.

```
list(x**2 for x in range(5))
```

```
[0, 1, 4, 9, 16]
```

A pesar de este hecho, sin embargo, las expresiones generadoras son muy diferentes: devuelven un *objeto generador* que soporta el protocolo de iteración, que permite emplear *yield* para obtener el siguiente resultado conforme se pida.

Código 7.49. Objeto generador.

```
G = (x**2 for x in range(5))
```

```
next(G)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[7], line 1
----> 1 next(G)

StopIteration:
```

Normalmente no vemos la sentencia *next* en una expresión generadora como esta, debido a que los ciclos *for* la lanzan automáticamente y por detrás:

Código 7.50. Ciclo `for` iterando un generador.

```
for num in (x**2 for x in range(5)):
    print(num)
```

```
0
1
4
9
16
```

Tal como lo hemos visto, este contexto de iteración lo hace de esta forma, ésto incluye a las funciones nativas `sum`, `map` y `sorted`, así como las listas por comprensión, entre muchos otros.

Código 7.51. Iteración de generadores de `sum`, `map` y `sorted`.

```
sum(x**2 for x in range(5))
```

```
30
```

```
sorted(x**2 for x in range(5))
```

```
[0, 1, 4, 9, 16]
```

```
sum(map(lambda x:x**2, range(5)))
```

```
30
```

```
from functools import reduce

reduce(lambda acum, x : acum + x**2, range(5))
```

```
30
```

7.8.2. Generación de valores con tipos y clases nativos

Python permite varias formas de generar sus tipos avanzados de datos, por ejemplo, para generar un diccionario:

Código 7.52. Generación de un diccionario.

```
D = {'a':1, 'b':2, 'c':3}
x = iter(D)
```

```
next(x)
```

```
'c'
```

El diccionario puede ser iterado manualmente o con las herramientas de iteración nativas: for, map y listas por comprensión.

Código 7.53. Iteración del diccionario.

```
for key in D:
    print(key, D[key])
```

```
a 1
b 2
c 3
```

7.9. Recapitulando

Código 7.54. Lista por comprensión.

```
# Listas por comprensión
[x*x for x in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Código 7.55. Expresión generadora.

```
# Expresión generadora
(x*x for x in range(10))
```

```
<generator object <genexpr> at 0x7fce1b5df6b0>
```

Código 7.56. Conjunto por comprensión.

```
# Conjunto por comprensión
{x*x for x in range(10)}
```

```
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Código 7.57. Diccionario por comprensión.

```
# Diccionario por comprensión
{x:x*x for x in range(10)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

7.9.1. Diccionarios y conjuntos por comprensión

Código 7.58. Conjunto por comprensión.

```
{x*x for x in range(10)}
```

```
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Código 7.59. Conjunto por generación y tipo.

```
set(x*x for x in range(10))
```

```
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Código 7.60. Diccionario por comprensión.

```
{x:x*x for x in range(10)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Código 7.61. Diccionario por generación y tipo.

```
dict((x, x*x) for x in range(10))
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

7.10. Referencias

- Lutz M., Learning Python, O'Reilly. 2009

7.11. Ejercicios

7.11.1. Listas por compresión

Ejercicio 7.1. Crea una lista que contenga los cuadrados de los números del 1 al 10.

```
cuadrados = [x**2 for x in range(1, 11)]
print(cuadrados)  # Salida: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Ejercicio 7.2. Genera una lista que contenga los números pares del 1 al 20.

```
pares = [x for x in range(1, 21) if x % 2 == 0]
print(pares)  # Salida: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Ejercicio 7.3. Dada una lista de palabras, crea una nueva lista que contenga la longitud de cada palabra.

```
palabras = ["manzana", "banana", "cereza", "durazno"]
longitudes = [len(palabra) for palabra in palabras]
print(longitudes)  # Salida: [7, 6, 6, 7]
```

```
[7, 6, 6, 7]
```

Ejercicio 7.4. Crea una lista que contenga los números del 1 al 30 que sean divisibles por 3.

```
divisibles_por_3 = [x for x in range(1, 31) if x % 3 == 0]
print(divisibles_por_3)  # Salida: [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Ejercicio 7.5. Dada una palabra, crea una lista que contenga solo las vocales de la palabra.

```
palabra = "computadora"
vocales = [letra for letra in palabra if letra in 'aeiou']
print(vocales)  # Salida: ['o', 'u', 'a', 'o', 'a']
```

```
['o', 'u', 'a', 'o', 'a']
```

Ejercicio 7.6. Crea una lista que contenga todos los números primos entre 1 y 50 usando una lista por comprensión.

```
def es_primo(n):
    return n > 1 and all(n % i != 0 for i in range(2, int(n**0.5) + 1))

primos = [x for x in range(1, 51) if es_primo(x)]
print(primos)  # Salida: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Ejercicio 7.7. Genera una lista de listas donde cada sublista contenga un número y su cuadrado, para los números del 1 al 5.

```
lista_de_listas = [[x, x**2] for x in range(1, 6)]
print(lista_de_listas)  # Salida: [[1, 1], [2, 4], [3, 9], [4, 16], [5, 25]]
```

```
[[1, 1], [2, 4], [3, 9], [4, 16], [5, 25]]
```

Ejercicio 7.8. Dada una lista de palabras, crea una nueva lista que contenga solo aquellas palabras que tienen más de 5 letras.

```
palabras = ["sol", "luna", "estrellas", "cometa", "galaxia"]
palabras_largas = [palabra for palabra in palabras if len(palabra) > 5]
print(palabras_largas)  # Salida: ['estrellas', 'cometa', 'galaxia']
```

```
['estrellas', 'cometa', 'galaxia']
```

Ejercicio 7.9. Genera una lista de tuplas, donde cada tupla contenga un número y su cubo, para los números del 1 al 5.

```
lista_de_tuplas = [(x, x**3) for x in range(1, 6)]
print(lista_de_tuplas)  # Salida: [(1, 1), (2, 8), (3, 27), (4, 64), (5, 125)]
```

```
[(1, 1), (2, 8), (3, 27), (4, 64), (5, 125)]
```

Ejercicio 7.10. Dada una lista de listas, usa una lista por comprensión para aplanarla en una sola lista.

```
lista_de_listas = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
lista_aplanada = [elemento for sublista in lista_de_listas for elemento in sublista]
print(lista_aplanada)  # Salida: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

7.11.2. Expresiones generadoras

Ejercicio 7.11. Crea una expresión generadora que calcule la suma de los cuadrados de los números del 1 al 10.

```
suma_cuadrados = sum(x**2 for x in range(1, 11))
print(suma_cuadrados)  # Salida: 385
```

```
385
```

Ejercicio 7.12. Genera una expresión generadora que produzca los números pares entre 1 y 20.

```
pares = (x for x in range(1, 21) if x % 2 == 0)
print(*pares, sep=", ")  # Salida: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

```
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

Ejercicio 7.13. Dada una lista de palabras, usa una expresión generadora para calcular la longitud total de todas las palabras.

```
palabras = ["manzana", "banana", "cereza", "durazno"]
longitud_total = sum(len(palabra) for palabra in palabras)
print(longitud_total)  # Salida: 26
```

```
26
```

Ejercicio 7.14. Crea una expresión generadora que filtre los números divisibles por 3 en el rango de 1 a 30.

```
divisibles_por_3 = (x for x in range(1, 31) if x % 3 == 0)
print(*divisibles_por_3, sep=", ")  # Salida: 3, 6, 9, 12, 15, 18, 21, 24, 27, 30
```

```
3, 6, 9, 12, 15, 18, 21, 24, 27, 30
```

Ejercicio 7.15. Genera una expresión generadora que calcule las raíces cuadradas de los números pares entre 1 y 20.

```
import math
raices_cuadradas = (math.sqrt(x) for x in range(2, 21, 2))
print(*raices_cuadradas, sep=", ")
# Salida: 1.4142135623730951, 2.0, 2.8284271247461903, 3.464101615137754, 4.0, 4.47213595499958, 5.0
```

```
1.4142135623730951, 2.0, 2.449489742783178, 2.8284271247461903, 3.1622776601683795, 3.464101615137754
```

Ejercicio 7.16. Dada una lista de números, utiliza una expresión generadora para calcular el producto de todos los elementos.

```
import math
numeros = [1, 2, 3, 4, 5]
producto = math.prod(x for x in numeros)
print(producto)  # Salida: 120
```

```
120
```

Ejercicio 7.17. Dada una lista de cadenas, utiliza una expresión generadora para concatenarlas en una sola cadena.

```
cadenas = ["Hola", " ", "Mundo", "!"]
resultado = ''.join(cadena for cadena in cadenas)
print(resultado)  # Salida: "Hola Mundo!"
```

```
Hola Mundo!
```

Ejercicio 7.18. Usa una expresión generadora para sumar los primeros **N** números primos.

```
def es_primo(n):
    return n > 1 and all(n % i != 0 for i in range(2, int(n**0.5) + 1))

N = 10
```

```
primos_sum = sum(x for x in range(2, 100) if es_primo(x))  # Generamos una lista amplia para cubrir l
print(primos_sum)  # Salida: 129 (suma de los primeros 10 números primos)
```

```
1060
```

Ejercicio 7.19. Dada una lista de palabras, utiliza una expresión generadora para filtrar solo aquellas palabras que tengan más de 5 letras.

```
palabras = ["sol", "luna", "estrellas", "cometa", "galaxia"]
palabras_largas = (palabra for palabra in palabras if len(palabra) > 5)
print(*palabras_largas, sep=", ")  # Salida: "estrellas", "cometa", "galaxia"
```

```
estrellas, cometa, galaxia
```

Ejercicio 7.20. Usa una expresión generadora para generar números aleatorios entre 1 y 100, pero detente cuando el número generado supere 90.

```
import random
numeros_al_azar = (num for num in iter(lambda: random.randint(1, 100), None))
for num in numeros_al_azar:
    print(num, end=", ")
    if num > 90:
        break  # Detenemos la iteración cuando el número es mayor a 90
```

```
59, 45, 92,
```

Capítulo 8

Manejo de archivos

El manejo de archivos es una parte importante de cualquier aplicación web.

Python tiene varias funciones para crear, leer, actualizar y eliminar archivos.

8.1. Función `open`

La función clave para trabajar con archivos en Python es `open()`. La función `open()` toma dos parámetros: *nombre_archivo* y *modo* .

Hay cuatro métodos (modos) diferentes para abrir un archivo:

- "r" - Leer - Valor predeterminado. Abre un archivo para leerlo. Se produce un error si el archivo no existe.
- "a" - Anexar - Abre un archivo para anexarlo, crea el archivo si no existe.
- "w" - Escribir - Abre un archivo para escribir, crea el archivo si no existe.
- "x" - Crear - Crea el archivo especificado, devuelve un error si el archivo existe.

Además, puede especificar si el archivo debe manejarse en modo binario o de texto.

- "t" - Texto - Valor predeterminado. Modo texto.
- "b" - Binario - Modo binario (por ejemplo, imágenes).

8.2. Sintaxis

La función `open()` abre un archivo y lo devuelve como un objeto de archivo.

```
open(file, mode)
```

8.3. Valores predeterminados

Parámetro	Descripción
file	La ubicación y nombre del archivo

Parámetro	Descripción
mode	Una cadena que define el modo como se abre el archivo ''r'' - **Lectura** - Valor por defecto. Abre un archivo para lectura, envía error si el archivo no existe. ''a'' - **Añadir** - Abre un archivo para añadir, crea el archivo si no existe. ''w'' - **Escritura** - Abre un archivo para escritura, crea el archivo si no existe. ''x'' - **Creación** - Crea un archivo, devuelve un error si el archivo ya existe. Adicionalmente se puede especificar si el archivo será manejado como texto o binario ''t'' - **Texto** - Modo texto, valor por defecto. ''b'' - **Binario** - Modo binario (i.e. imágenes)

Para abrir un archivo para su lectura es suficiente especificar el nombre del archivo:

```
f = open("demofile.txt")
```

Este código es equivalente al siguiente:

```
f = open("demofile.txt", "rt")
```

Dado que `'r'` para lectura y `'t'` para texto son los valores por defecto, no es necesario incluirlos.

8.4. Abrir un archivo localmente

Supongamos que tenemos el archivo `demofile.txt`, ubicado en la misma carpeta que el script de Python:

Archivo demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

Para abrir el archivo, se utiliza la función incorporada `open()`. Esta función devuelve un objeto tipo archivo, que tiene un método `read()` para leer el contenido del archivo:

Código 8.1. Apertura y lectura de un archivo.

```
f = open("demofile.txt", 'r')
print(f.read())
```

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!Now the file has more content!
```

8.5. Abrir un archivo en Google Drive

Supongamos que tenemos el archivo `demofile.txt`, ubicado en Google Drive:

Archivo demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

Además un notebook de Python se encuentra en la misma carpeta. Para abrir un archivo es necesario utilizar la función `google.colab` dentro del paquete `drive`. Una vez que se haya incluído el paquete, ahora es necesario *montar* la unidad de almacenamiento de Google Drive en una ubicación dentro del sistema de archivos virtual: `/drive`. Cabe mencionar que esta dirección apuntará a la raíz de la unidad de almacenamiento en Google Drive.

Código 8.2. Código en Colaboratory para abrir archivos.

```
from google.colab import drive
drive.mount("/drive")
```

Al ejecutar la función `drive.mount`, Google Drive preguntará por autorización para tener acceso a su unidad.

Posterior a esto, sólo es necesario utilizar la función open tal y como lo hicimos anteriormente.

```
f = open("/drive/My Drive/demofile.txt", 'r')
print(f.read())
```

Observe que se precede al nombre del archivo la ubicación `/drive/My Drive/`, que es donde se montó virtualmente la unidad de almacenamiento de Google Drive.

8.6. Leer partes de un archivo

Por defecto el método `read()` devuelve todo el texto, pero se puede especificar cuantos caracteres se desea leer.

Código 8.3. Lectura parcial del archivo.

```
f = open("demofile.txt", "r")
print(f.read(5))
```

```
Hello
```

8.7. Leer líneas del archivo

Es posible leer una línea de un archivo mediante el método `readline()`.

Código 8.4. Lectura de archivo mediante `readline()`.

```
f = open("demofile.txt", "r")
print(f.readline())
```

```
Hello! Welcome to demofile.txt
```

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

```
Hello! Welcome to demofile.txt

This file is for testing purposes.
```

Si se utiliza un ciclo `for`, se puede leer todo el archivo línea por línea.

Código 8.5. Lectura mediante ciclo `for`.

```
f = open("demofile.txt", "r")
for x in f:
    print(x)
```

```
Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!Now the file has more content!
```

8.8. Cerrar un archivo

Es una biena práctica cerrar un archivo después de utilizarse. Para ello se debe utilizar el método `close()`.

Código 8.6. Cerrar archivo después de su apertura.

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

```
Hello! Welcome to demofile.txt
```

Se puede verificar si un archivo está cerrado mediante la propiedad booleana `closed`.

```
print(f'El archivo está cerrado?: {f.closed}')
```

```
El archivo está cerrado?: True
```

8.9. Escribir en un archivo existente

Para escribir en un archivo existente se debe agregar un parámetro al método `open()`.

- `'a'`. Añadir contenido al final del archivo
- `'w'`. Escribir en el archivo. Sobreescribe cualquier contenido existente.

Código 8.7. Abrir el archivo `demofile2.txt` y añadir contenido.

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!Now the file has more content!Now the file has more content!
```

Código 8.8. Abrir el archivo `demofile2.txt` y sobrescribir contenido.

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the overwriting:
f = open("demofile3.txt", "r")
print(f.read())
```

```
Woops! I have deleted the content!
```

8.10. Crear un nuevo archivo

Para crear un nuevo archivo en Python se debe utilizar el método `open()` con uno de los siguientes parámetros.

- 'x'. Crear - crea el archivo y devuelve un error si el archivo existe.
- 'a'. Añadir - crea el archivo si es que no existe previamente.
- 'w'. Escribir - crea el archivo si es que no existe previamente.

Código 8.9. Crea un archivo llamado `myfile.txt`.

```
f = open("myfile.txt", "x")
```

Código 8.10. Crea un nuevo archivo `myfile.txt` si es que no existe previamente.

```
f = open("myfile.txt", "w")
```

8.11. Borrar un archivo

Para borrar archivos en Python se debe importar el módulo `os` y utlizar el módulo `os.remove()`.

Código 8.11. Borrar archivos desde Python.

```
import os
os.remove("myfile.txt")
```

```
import os
if os.path.exists("myfile.txt"):
  os.remove("myfile.txt")
else:
  print("The file does not exist")
```

```
The file does not exist
```

8.12. Referencias

- Manejo de archivos en Python.
- Lutz M., Learning Python, O'Reilly. 2009

Capítulo 9

Pandas

9.1. ¿Qué es Pandas?

Pandas es una biblioteca de Python y sirve para analizar datos. Tiene funciones para analizar, limpiar, explorar y manipular datos.

Pandas es una herramienta de análisis y manipulación de datos de código abierto rápida, potente, flexible y fácil de usar, construida sobre el lenguaje de programación Python.

El nombre "Pandas"hace referencia tanto a "Panel Dataçomo a "Python Data Analysisz fue creado por Wes McKinney en 2008.

El código fuente de Pandas se encuentra en un repositorio de GitHub.

9.2. Historia de su desarrollo

En 2008, AQR Capital Management comenzó a desarrollar pandas. A fines de 2009, ya era de código abierto y, en la actualidad, cuenta con el apoyo activo de una comunidad de personas con ideas afines en todo el mundo que contribuyen con su valioso tiempo y energía para ayudar a que el código abierto de pandas sea posible. Gracias a todos nuestros colaboradores.

Desde 2015, pandas es un proyecto patrocinado por NumFOCUS. Esto ayudará a garantizar el éxito del desarrollo de pandas como un proyecto de código abierto de clase mundial.

9.3. Documentación

Pandas posee una muy buena documentación en su sitio web oficial. En ella se puede encontrar una guía rápida de uso, además de la documentación completa del paquete.

9.4. ¿Porqué usar Pandas?

Pandas nos permite analizar grandes cantidades de datos y sacar conclusiones basadas en teorías estadísticas.

Pandas puede limpiar conjuntos de datos desordenados y hacerlos legibles y relevantes.

Los datos relevantes son muy importantes en la ciencia de datos.

Pandas puede dar respuestas sobre los datos. Por ejemplo:

- ¿Existe una correlación entre dos o más columnas?
- ¿Qué es el valor promedio?
- ¿Valor máximo?
- ¿Valor mínimo?

Pandas también puede eliminar filas que no son relevantes o que contienen valores incorrectos, como valores vacíos o NULL. Esto se llama limpiar los datos.

9.5. Instalación

Si ya tiene Python y PIP instalados en un sistema, entonces la instalación de Pandas es muy sencilla. Instálelo usando este comando:

```
$ pip install pandas
```

Si este comando falla, entonces use una distribución de Python que ya tenga Pandas instalado, como Anaconda, Spyder, etc.

9.6. Importar Pandas

Una vez que Pandas esté instalado, impórtelo en sus aplicaciones agregando la palabra clave `import`:

```
import pandas
```

Con esto Pandas está importado y listo para usarse.

Código 9.1. Creación de un DataFrame.

```
import pandas

mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

```
    cars  passings
0    BMW         3
1  Volvo         7
2   Ford         2
```

Al importar pandas se puede emplear el alias `pd`. Ahora, el paquete Pandas se puede invocar con `pd` en lugar del nombre completo `pandas`.

Código 9.2. Creación de un alias para un DataFrame.

```
import pandas as pd

mydataset = {
```

```
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pd.DataFrame(mydataset)

print(myvar)
```

```
    cars  passings
0    BMW         3
1  Volvo         7
2   Ford         2
```

9.7. Verificando la versión

La versión de pandas se almacena en una cadena almacenada en el atributo `__version__`.
Código 9.3. Verificación de la versión de `pandas`.

```
import pandas as pd

print(pd.__version__)
```

```
2.2.2
```

9.8. Series Pandas

Una serie de Pandas es como una columna de una tabla. Es una matriz unidimensional que contiene datos de cualquier tipo.

Código 9.4. Crear una serie de pandas a partir de una lista.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

```
0    1
1    7
2    2
dtype: int64
```

Si no se especifica nada más, los valores se etiquetan con su número de índice. El primer valor tiene el índice 0, el segundo valor tiene el índice 1, etc.

Esta etiqueta se puede utilizar para acceder a un valor especifico.

Código 9.5. Devolver el primer valor de la serie.

```
print(myvar[0])
```

```
1
```

Con el argumento `index` se puede nombrar las etiquetas.

Código 9.6. Crear etiquetas propias.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

```
x    1
y    7
z    2
dtype: int64
```

Una vez que haya creado etiquetas, podrá acceder a un elemento haciendo referencia a la etiqueta.

Código 9.7. Devolver el valor de "y".

```
print(myvar["y"])
```

```
7
```

9.9. Objetos llave/valor como series

También puedes utilizar un objeto clave/valor, como un diccionario, al crear una Serie.

Código 9.8. Crea una serie Pandas sencilla a partir de un diccionario.

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories)

print(myvar)
```

```
day1    420
day2    380
day3    390
dtype: int64
```

Observe que las claves del diccionario se convierten en las etiquetas.

Para seleccionar solo algunos de los elementos del diccionario, utilice el argumento de índice y especifique solo los elementos que desea incluir en la serie.

Código 9.9. Cree una serie utilizando solo datos de "día1" y "día2".

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories, index = ["day1", "day2"])
```

```
print(myvar)
```

```
day1    420
day2    380
dtype: int64
```

9.10. DataFrames

Los conjuntos de datos en Pandas suelen ser tablas multidimensionales, llamadas DataFrames.

Una serie es como una columna, un DataFrame es la tabla completa.

Código 9.10. Crear un DataFrame a partir de dos series.

```
import pandas as pd

# Diccionario data
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

df = pd.DataFrame(data)

print(df)
```

```
   calories  duration
0       420        50
1       380        40
2       390        45
```

9.10.1. ¿Qué es un DataFrame?

Un Pandas DataFrame es una estructura de datos de 2 dimensiones, como una de 2 dimensiones array, o una tabla con filas y columnas.

Como puede ver en el resultado anterior, el DataFrame es como una tabla con filas y columnas.

9.10.2. Localizar una fila

Los pandas usan el atributo `loc` a devolver una o más filas especificadas.

Código 9.11. Devolver la fila 0.

```
print(df.loc[0])
```

```
calories    420
duration     50
Name: 0, dtype: int64
```

Este código devuelve una serie Pandas.

Código 9.12. Devolver filas 0 y 1.

```
print(df.loc[[0, 1]])
```

```
   calories  duration
0       420        50
1       380        40
```

En este caso, el código devuelve un DataFrame Pandas.

9.10.3. Índices nombrados

Con el argumento `index`, se puede nombrar sus propios índices. Nótese que los índices deben ser una lista.

Código 9.13. DataFrame con índices nombrados.

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)
```

```
      calories  duration
day1       420        50
day2       380        40
day3       390        45
```

9.10.4. Localizar Índices Nombrados

Utilice el índice nombrado en el atributo `loc` para devolver la fila especificada.

Código 9.14. Retorno “day2”.

```
# Referencia al índice nombrado:
print(df.loc["day2"])
```

```
calories    380
duration     40
Name: day2, dtype: int64
```

9.10.5. Carga de archivos en un DataFrame

Una forma sencilla de almacenar grandes conjuntos de datos es usar archivos CSV (separados por comas archivos).

Los archivos CSV contienen texto sin formato y es un formato bien conocido que puede ser leído por todos, incluyendo Pandas.

Si sus conjuntos de datos se almacenan en un archivo, Pandas puede cargarlos en un DataFrame.

Código 9.15. Cargue un archivo separado por comas (Archivo CSV) en un DataFrame.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

print(df)
```

```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
..        ...    ...       ...       ...
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4

[169 rows x 4 columns]
```

Si tiene un DataFrame grande con muchas filas, Pandas solo devolverá las primeras 5 filas y las últimas 5 filas.

Se puede emplear el método `to_string()` para convertir el DataFrame en una sola representación String. La salida es una tabla amigable para la consola.

Código 9.16. Método `to_string()` para convertir el DataFrame en una sola representación String.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

print(df.to_string())
```

```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
5          60    102       127     300.0
6          60    110       136     374.0
7          45    104       134     253.3
8          30    109       133     195.1
9          60     98       124     269.0
10         60    103       147     329.3
11         60    100       120     250.7
12         60    106       128     345.3
13         60    104       132     379.3
14         60     98       123     275.0
15         60     98       120     215.2
16         60    100       120     300.0
17         45     90       112       NaN
```

18	60	103	123	323.0
19	45	97	125	243.0
20	60	108	131	364.2
21	45	100	119	282.0
22	60	130	101	300.0
23	45	105	132	246.0
24	60	102	126	334.5
25	60	100	120	250.0
26	60	92	118	241.0
27	60	103	132	NaN
28	60	100	132	280.0
29	60	102	129	380.3
30	60	92	115	243.0
31	45	90	112	180.1
32	60	101	124	299.0
33	60	93	113	223.0
34	60	107	136	361.0
35	60	114	140	415.0
36	60	102	127	300.0
37	60	100	120	300.0
38	60	100	120	300.0
39	45	104	129	266.0
40	45	90	112	180.1
41	60	98	126	286.0
42	60	100	122	329.4
43	60	111	138	400.0
44	60	111	131	397.0
45	60	99	119	273.0
46	60	109	153	387.6
47	45	111	136	300.0
48	45	108	129	298.0
49	60	111	139	397.6
50	60	107	136	380.2
51	80	123	146	643.1
52	60	106	130	263.0
53	60	118	151	486.0
54	30	136	175	238.0
55	60	121	146	450.7
56	60	118	121	413.0
57	45	115	144	305.0
58	20	153	172	226.4
59	45	123	152	321.0
60	210	108	160	1376.0
61	160	110	137	1034.4
62	160	109	135	853.0
63	45	118	141	341.0
64	20	110	130	131.4
65	180	90	130	800.4
66	150	105	135	873.4
67	150	107	130	816.0
68	20	106	136	110.4
69	300	108	143	1500.2

70	150	97	129	1115.0
71	60	109	153	387.6
72	90	100	127	700.0
73	150	97	127	953.2
74	45	114	146	304.0
75	90	98	125	563.2
76	45	105	134	251.0
77	45	110	141	300.0
78	120	100	130	500.4
79	270	100	131	1729.0
80	30	159	182	319.2
81	45	149	169	344.0
82	30	103	139	151.1
83	120	100	130	500.0
84	45	100	120	225.3
85	30	151	170	300.0
86	45	102	136	234.0
87	120	100	157	1000.1
88	45	129	103	242.0
89	20	83	107	50.3
90	180	101	127	600.1
91	45	107	137	NaN
92	30	90	107	105.3
93	15	80	100	50.5
94	20	150	171	127.4
95	20	151	168	229.4
96	30	95	128	128.2
97	25	152	168	244.2
98	30	109	131	188.2
99	90	93	124	604.1
100	20	95	112	77.7
101	90	90	110	500.0
102	90	90	100	500.0
103	90	90	100	500.4
104	30	92	108	92.7
105	30	93	128	124.0
106	180	90	120	800.3
107	30	90	120	86.2
108	90	90	120	500.3
109	210	137	184	1860.4
110	60	102	124	325.2
111	45	107	124	275.0
112	15	124	139	124.2
113	45	100	120	225.3
114	60	108	131	367.6
115	60	108	151	351.7
116	60	116	141	443.0
117	60	97	122	277.4
118	60	105	125	NaN
119	60	103	124	332.7
120	30	112	137	193.9
121	45	100	120	100.7

```
122        60    119       169     336.7
123        60    107       127     344.9
124        60    111       151     368.5
125        60     98       122     271.0
126        60     97       124     275.3
127        60    109       127     382.0
128        90     99       125     466.4
129        60    114       151     384.0
130        60    104       134     342.5
131        60    107       138     357.5
132        60    103       133     335.0
133        60    106       132     327.5
134        60    103       136     339.0
135        20    136       156     189.0
136        45    117       143     317.7
137        45    115       137     318.0
138        45    113       138     308.0
139        20    141       162     222.4
140        60    108       135     390.0
141        60     97       127       NaN
142        45    100       120     250.4
143        45    122       149     335.4
144        60    136       170     470.2
145        45    106       126     270.8
146        60    107       136     400.0
147        60    112       146     361.9
148        30    103       127     185.0
149        60    110       150     409.4
150        60    106       134     343.0
151        60    109       129     353.2
152        60    109       138     374.0
153        30    150       167     275.8
154        60    105       128     328.0
155        60    111       151     368.5
156        60     97       131     270.4
157        60    100       120     270.4
158        60    114       150     382.8
159        30     80       120     240.9
160        30     85       120     250.4
161        45     90       130     260.4
162        45     95       130     270.0
163        45    100       140     280.9
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4
```

Código 9.17. Método `to_string()` para convertir el DataFrame en una sola representación String.

```
import pandas as pd
```

```
df = pd.read_csv('data/social.csv')

print(df.to_string())
```

```
  Social media platform Number of monthly (million)
0              Facebook                      3 000m
1               YouTube                    2000.5 m
2             Instagram                      2 000m
3                TikTok                    1000.5 m
4              Snapchat                       800 m
5           X (Twitter)                       611 m
6                Reddit                       500 m
7             Pinterest                       498 m
8              LinkedIn                       350 m
9               Threads                       175 m
```

9.10.6. max_filas

El número de filas devueltas se define en la configuración de la opción Pandas. Puede verificar las filas máximas de su sistema con la declaración `pd.options.display.max_rows`.

Código 9.18. Verificación del número máximo de filas.

```
import pandas as pd

print(pd.options.display.max_rows)
```

```
60
```

En mi sistema el número es 60, lo que significa que si el DataFrame contiene más de 60 filas, el `print(df)` la declaración devolverá solo los encabezados y las primeras y últimas 5 filas.

Puede cambiar el número máximo de filas con la misma instrucción.

Código 9.19. Modificación del número máximo de filas.

```
import pandas as pd

pd.options.display.max_rows = 9999
df = pd.read_csv('data/data.csv')

print(df)
```

```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
5          60    102       127     300.0
6          60    110       136     374.0
7          45    104       134     253.3
8          30    109       133     195.1
9          60     98       124     269.0
10         60    103       147     329.3
```

11	60	100	120	250.7
12	60	106	128	345.3
13	60	104	132	379.3
14	60	98	123	275.0
15	60	98	120	215.2
16	60	100	120	300.0
17	45	90	112	NaN
18	60	103	123	323.0
19	45	97	125	243.0
20	60	108	131	364.2
21	45	100	119	282.0
22	60	130	101	300.0
23	45	105	132	246.0
24	60	102	126	334.5
25	60	100	120	250.0
26	60	92	118	241.0
27	60	103	132	NaN
28	60	100	132	280.0
29	60	102	129	380.3
30	60	92	115	243.0
31	45	90	112	180.1
32	60	101	124	299.0
33	60	93	113	223.0
34	60	107	136	361.0
35	60	114	140	415.0
36	60	102	127	300.0
37	60	100	120	300.0
38	60	100	120	300.0
39	45	104	129	266.0
40	45	90	112	180.1
41	60	98	126	286.0
42	60	100	122	329.4
43	60	111	138	400.0
44	60	111	131	397.0
45	60	99	119	273.0
46	60	109	153	387.6
47	45	111	136	300.0
48	45	108	129	298.0
49	60	111	139	397.6
50	60	107	136	380.2
51	80	123	146	643.1
52	60	106	130	263.0
53	60	118	151	486.0
54	30	136	175	238.0
55	60	121	146	450.7
56	60	118	121	413.0
57	45	115	144	305.0
58	20	153	172	226.4
59	45	123	152	321.0
60	210	108	160	1376.0
61	160	110	137	1034.4
62	160	109	135	853.0

63	45	118	141	341.0
64	20	110	130	131.4
65	180	90	130	800.4
66	150	105	135	873.4
67	150	107	130	816.0
68	20	106	136	110.4
69	300	108	143	1500.2
70	150	97	129	1115.0
71	60	109	153	387.6
72	90	100	127	700.0
73	150	97	127	953.2
74	45	114	146	304.0
75	90	98	125	563.2
76	45	105	134	251.0
77	45	110	141	300.0
78	120	100	130	500.4
79	270	100	131	1729.0
80	30	159	182	319.2
81	45	149	169	344.0
82	30	103	139	151.1
83	120	100	130	500.0
84	45	100	120	225.3
85	30	151	170	300.0
86	45	102	136	234.0
87	120	100	157	1000.1
88	45	129	103	242.0
89	20	83	107	50.3
90	180	101	127	600.1
91	45	107	137	NaN
92	30	90	107	105.3
93	15	80	100	50.5
94	20	150	171	127.4
95	20	151	168	229.4
96	30	95	128	128.2
97	25	152	168	244.2
98	30	109	131	188.2
99	90	93	124	604.1
100	20	95	112	77.7
101	90	90	110	500.0
102	90	90	100	500.0
103	90	90	100	500.4
104	30	92	108	92.7
105	30	93	128	124.0
106	180	90	120	800.3
107	30	90	120	86.2
108	90	90	120	500.3
109	210	137	184	1860.4
110	60	102	124	325.2
111	45	107	124	275.0
112	15	124	139	124.2
113	45	100	120	225.3
114	60	108	131	367.6

```
115      60    108      151    351.7
116      60    116      141    443.0
117      60     97      122    277.4
118      60    105      125      NaN
119      60    103      124    332.7
120      30    112      137    193.9
121      45    100      120    100.7
122      60    119      169    336.7
123      60    107      127    344.9
124      60    111      151    368.5
125      60     98      122    271.0
126      60     97      124    275.3
127      60    109      127    382.0
128      90     99      125    466.4
129      60    114      151    384.0
130      60    104      134    342.5
131      60    107      138    357.5
132      60    103      133    335.0
133      60    106      132    327.5
134      60    103      136    339.0
135      20    136      156    189.0
136      45    117      143    317.7
137      45    115      137    318.0
138      45    113      138    308.0
139      20    141      162    222.4
140      60    108      135    390.0
141      60     97      127      NaN
142      45    100      120    250.4
143      45    122      149    335.4
144      60    136      170    470.2
145      45    106      126    270.8
146      60    107      136    400.0
147      60    112      146    361.9
148      30    103      127    185.0
149      60    110      150    409.4
150      60    106      134    343.0
151      60    109      129    353.2
152      60    109      138    374.0
153      30    150      167    275.8
154      60    105      128    328.0
155      60    111      151    368.5
156      60     97      131    270.4
157      60    100      120    270.4
158      60    114      150    382.8
159      30     80      120    240.9
160      30     85      120    250.4
161      45     90      130    260.4
162      45     95      130    270.0
163      45    100      140    280.9
164      60    105      140    290.8
165      60    110      145    300.0
166      60    115      145    310.2
```

```
167         75    120         150      320.4
168         75    125         150      330.4
```

9.10.7. Leer archivos JSON

Los grandes conjuntos de datos a menudo se almacenan o extraen como JSON.

JSON es texto sin formato, pero tiene el formato de un objeto, y es bien conocido en el mundo de la programación, incluyendo Pandas.

En nuestros ejemplos usaremos un archivo JSON llamado `'data.json'`.

Código 9.20. Cargar el archivo `json` en un DataFrame.

```
import pandas as pd

df = pd.read_json('data/data.json')

print(df.to_string())
```

```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
5          60    102       127     300.5
6          60    110       136     374.0
7          45    104       134     253.3
8          30    109       133     195.1
9          60     98       124     269.0
10         60    103       147     329.3
11         60    100       120     250.7
12         60    106       128     345.3
13         60    104       132     379.3
14         60     98       123     275.0
15         60     98       120     215.2
16         60    100       120     300.0
17         45     90       112       NaN
18         60    103       123     323.0
19         45     97       125     243.0
20         60    108       131     364.2
21         45    100       119     282.0
22         60    130       101     300.0
23         45    105       132     246.0
24         60    102       126     334.5
25         60    100       120     250.0
26         60     92       118     241.0
27         60    103       132       NaN
28         60    100       132     280.0
29         60    102       129     380.3
30         60     92       115     243.0
31         45     90       112     180.1
32         60    101       124     299.0
```

33	60	93	113	223.0
34	60	107	136	361.0
35	60	114	140	415.0
36	60	102	127	300.5
37	60	100	120	300.1
38	60	100	120	300.0
39	45	104	129	266.0
40	45	90	112	180.1
41	60	98	126	286.0
42	60	100	122	329.4
43	60	111	138	400.0
44	60	111	131	397.0
45	60	99	119	273.0
46	60	109	153	387.6
47	45	111	136	300.0
48	45	108	129	298.0
49	60	111	139	397.6
50	60	107	136	380.2
51	80	123	146	643.1
52	60	106	130	263.0
53	60	118	151	486.0
54	30	136	175	238.0
55	60	121	146	450.7
56	60	118	121	413.0
57	45	115	144	305.0
58	20	153	172	226.4
59	45	123	152	321.0
60	210	108	160	1376.0
61	160	110	137	1034.4
62	160	109	135	853.0
63	45	118	141	341.0
64	20	110	130	131.4
65	180	90	130	800.4
66	150	105	135	873.4
67	150	107	130	816.0
68	20	106	136	110.4
69	300	108	143	1500.2
70	150	97	129	1115.0
71	60	109	153	387.6
72	90	100	127	700.0
73	150	97	127	953.2
74	45	114	146	304.0
75	90	98	125	563.2
76	45	105	134	251.0
77	45	110	141	300.0
78	120	100	130	500.4
79	270	100	131	1729.0
80	30	159	182	319.2
81	45	149	169	344.0
82	30	103	139	151.1
83	120	100	130	500.0
84	45	100	120	225.3

85	30	151	170	300.1
86	45	102	136	234.0
87	120	100	157	1000.1
88	45	129	103	242.0
89	20	83	107	50.3
90	180	101	127	600.1
91	45	107	137	NaN
92	30	90	107	105.3
93	15	80	100	50.5
94	20	150	171	127.4
95	20	151	168	229.4
96	30	95	128	128.2
97	25	152	168	244.2
98	30	109	131	188.2
99	90	93	124	604.1
100	20	95	112	77.7
101	90	90	110	500.0
102	90	90	100	500.0
103	90	90	100	500.4
104	30	92	108	92.7
105	30	93	128	124.0
106	180	90	120	800.3
107	30	90	120	86.2
108	90	90	120	500.3
109	210	137	184	1860.4
110	60	102	124	325.2
111	45	107	124	275.0
112	15	124	139	124.2
113	45	100	120	225.3
114	60	108	131	367.6
115	60	108	151	351.7
116	60	116	141	443.0
117	60	97	122	277.4
118	60	105	125	NaN
119	60	103	124	332.7
120	30	112	137	193.9
121	45	100	120	100.7
122	60	119	169	336.7
123	60	107	127	344.9
124	60	111	151	368.5
125	60	98	122	271.0
126	60	97	124	275.3
127	60	109	127	382.0
128	90	99	125	466.4
129	60	114	151	384.0
130	60	104	134	342.5
131	60	107	138	357.5
132	60	103	133	335.0
133	60	106	132	327.5
134	60	103	136	339.0
135	20	136	156	189.0
136	45	117	143	317.7

```
137        45    115        137     318.0
138        45    113        138     308.0
139        20    141        162     222.4
140        60    108        135     390.0
141        60     97        127       NaN
142        45    100        120     250.4
143        45    122        149     335.4
144        60    136        170     470.2
145        45    106        126     270.8
146        60    107        136     400.0
147        60    112        146     361.9
148        30    103        127     185.0
149        60    110        150     409.4
150        60    106        134     343.0
151        60    109        129     353.2
152        60    109        138     374.0
153        30    150        167     275.8
154        60    105        128     328.0
155        60    111        151     368.5
156        60     97        131     270.4
157        60    100        120     270.4
158        60    114        150     382.8
159        30     80        120     240.9
160        30     85        120     250.4
161        45     90        130     260.4
162        45     95        130     270.0
163        45    100        140     280.9
164        60    105        140     290.8
165        60    110        145     300.4
166        60    115        145     310.2
167        75    120        150     320.4
168        75    125        150     330.4
```

Los objetos JSON tienen el mismo formato que los diccionarios Python.

Si su código JSON no está en un archivo, sino en un diccionario Python, puede cargarlo en un DataFrame directamente.

Código 9.21. Cargar un diccionario Python en un DataFrame.

```
import pandas as pd

data = {
  "Duration":{
    "0":60,
    "1":60,
    "2":60,
    "3":45,
    "4":45,
    "5":60
  },
  "Pulse":{
    "0":110,
```

```
    "1":117,
    "2":103,
    "3":109,
    "4":117,
    "5":102
  },
  "Maxpulse":{
    "0":130,
    "1":145,
    "2":135,
    "3":175,
    "4":148,
    "5":127
  },
  "Calories":{
    "0":409,
    "1":479,
    "2":340,
    "3":282,
    "4":406,
    "5":300
  }
}

df = pd.DataFrame(data)

print(df)
```

```
   Duration  Pulse  Maxpulse  Calories
0        60    110       130       409
1        60    117       145       479
2        60    103       135       340
3        45    109       175       282
4        45    117       148       406
5        60    102       127       300
```

9.10.8. Vistazo rápido a los Datos

Uno de los métodos más utilizados para obtener una visión general rápida del DataFrame, es el `head()` método.

El `head()` el método devuelve los encabezados y un número específico de filas, comenzando desde la parte superior. Por defecto se muestran las primeras 5 líneas a menos que se indique otra cosa.

Código 9.22. Obtenga una visión general rápida imprimiendo las primeras 10 filas de DataFrame.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

print(df.head(10))
```

```
   Duration  Pulse  Maxpulse  Calories
0        60    110       130     409.1
```

```
1        60    117       145     479.0
2        60    103       135     340.0
3        45    109       175     282.4
4        45    117       148     406.0
5        60    102       127     300.0
6        60    110       136     374.0
7        45    104       134     253.3
8        30    109       133     195.1
9        60     98       124     269.0
```

También hay un `tail()` método para ver el último filas del DataFrame.

El `tail()` el método devuelve los encabezados y un número específico de filas, comenzando desde la parte inferior.

Código 9.23. Imprima las últimas 5 filas de DataFrame.

```
print(df.tail())
```

```
     Duration  Pulse  Maxpulse  Calories
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4
```

9.10.9. Información Sobre los Datos

El objeto DataFrames tiene un método llamado `info()`, eso le da más información sobre el conjunto de datos.

Código 9.24. Imprimir información sobre los datos.

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
dtypes: float64(1), int64(3)
memory usage: 5.4 KB
None
```

El resultado nos dice que hay 169 filas y 4 columnas, el nombre de cada columna, con el tipo de dato.

El `info()` el método también nos dice cuántos valores no nulos hay presentes en cada columna y en nuestro conjunto de datos parece que hay 164 de 169 valores No Nulos en la columna Çalorías".

Lo que significa que hay 5 filas sin ningún valor en absoluto, en la columna Çalorías", por alguna razón desconocida.

Los valores vacíos, o valores nulos, pueden ser malos al analizar los datos y debe considerar eliminar filas con valores vacíos. Este es un paso hacia lo que se llama datos de limpieza.

9.11. Limpieza de Datos

La limpieza de datos significa arreglar datos incorrectos en su conjunto de datos.

Los datos malos incorrectos pueden ser:

- Celdas vacías
- Datos en formato incorrecto
- Datos incorrectos
- Duplicados

Las celdas vacías pueden potencialmente darle un resultado incorrecto cuando analiza datos.

9.11.1. Eliminar Filas

Una forma de tratar con las celdas vacías es eliminar las filas que las contienen.

Esto suele estar bien, siempre y cuando los conjuntos de datos sean muy grandes, ya que eliminar algunas filas no tendrá un gran impacto en el resultado.

Código 9.25. Devuelve un nuevo Marco de datos sin celdas vacías.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

new_df = df.dropna()

print(new_df.to_string())
```

```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
5          60    102       127     300.0
6          60    110       136     374.0
7          45    104       134     253.3
8          30    109       133     195.1
9          60     98       124     269.0
10         60    103       147     329.3
11         60    100       120     250.7
12         60    106       128     345.3
13         60    104       132     379.3
14         60     98       123     275.0
15         60     98       120     215.2
16         60    100       120     300.0
18         60    103       123     323.0
```

19	45	97	125	243.0
20	60	108	131	364.2
21	45	100	119	282.0
22	60	130	101	300.0
23	45	105	132	246.0
24	60	102	126	334.5
25	60	100	120	250.0
26	60	92	118	241.0
28	60	100	132	280.0
29	60	102	129	380.3
30	60	92	115	243.0
31	45	90	112	180.1
32	60	101	124	299.0
33	60	93	113	223.0
34	60	107	136	361.0
35	60	114	140	415.0
36	60	102	127	300.0
37	60	100	120	300.0
38	60	100	120	300.0
39	45	104	129	266.0
40	45	90	112	180.1
41	60	98	126	286.0
42	60	100	122	329.4
43	60	111	138	400.0
44	60	111	131	397.0
45	60	99	119	273.0
46	60	109	153	387.6
47	45	111	136	300.0
48	45	108	129	298.0
49	60	111	139	397.6
50	60	107	136	380.2
51	80	123	146	643.1
52	60	106	130	263.0
53	60	118	151	486.0
54	30	136	175	238.0
55	60	121	146	450.7
56	60	118	121	413.0
57	45	115	144	305.0
58	20	153	172	226.4
59	45	123	152	321.0
60	210	108	160	1376.0
61	160	110	137	1034.4
62	160	109	135	853.0
63	45	118	141	341.0
64	20	110	130	131.4
65	180	90	130	800.4
66	150	105	135	873.4
67	150	107	130	816.0
68	20	106	136	110.4
69	300	108	143	1500.2
70	150	97	129	1115.0
71	60	109	153	387.6

72	90	100	127	700.0
73	150	97	127	953.2
74	45	114	146	304.0
75	90	98	125	563.2
76	45	105	134	251.0
77	45	110	141	300.0
78	120	100	130	500.4
79	270	100	131	1729.0
80	30	159	182	319.2
81	45	149	169	344.0
82	30	103	139	151.1
83	120	100	130	500.0
84	45	100	120	225.3
85	30	151	170	300.0
86	45	102	136	234.0
87	120	100	157	1000.1
88	45	129	103	242.0
89	20	83	107	50.3
90	180	101	127	600.1
92	30	90	107	105.3
93	15	80	100	50.5
94	20	150	171	127.4
95	20	151	168	229.4
96	30	95	128	128.2
97	25	152	168	244.2
98	30	109	131	188.2
99	90	93	124	604.1
100	20	95	112	77.7
101	90	90	110	500.0
102	90	90	100	500.0
103	90	90	100	500.4
104	30	92	108	92.7
105	30	93	128	124.0
106	180	90	120	800.3
107	30	90	120	86.2
108	90	90	120	500.3
109	210	137	184	1860.4
110	60	102	124	325.2
111	45	107	124	275.0
112	15	124	139	124.2
113	45	100	120	225.3
114	60	108	131	367.6
115	60	108	151	351.7
116	60	116	141	443.0
117	60	97	122	277.4
119	60	103	124	332.7
120	30	112	137	193.9
121	45	100	120	100.7
122	60	119	169	336.7
123	60	107	127	344.9
124	60	111	151	368.5
125	60	98	122	271.0

```
126        60     97       124     275.3
127        60    109       127     382.0
128        90     99       125     466.4
129        60    114       151     384.0
130        60    104       134     342.5
131        60    107       138     357.5
132        60    103       133     335.0
133        60    106       132     327.5
134        60    103       136     339.0
135        20    136       156     189.0
136        45    117       143     317.7
137        45    115       137     318.0
138        45    113       138     308.0
139        20    141       162     222.4
140        60    108       135     390.0
142        45    100       120     250.4
143        45    122       149     335.4
144        60    136       170     470.2
145        45    106       126     270.8
146        60    107       136     400.0
147        60    112       146     361.9
148        30    103       127     185.0
149        60    110       150     409.4
150        60    106       134     343.0
151        60    109       129     353.2
152        60    109       138     374.0
153        30    150       167     275.8
154        60    105       128     328.0
155        60    111       151     368.5
156        60     97       131     270.4
157        60    100       120     270.4
158        60    114       150     382.8
159        30     80       120     240.9
160        30     85       120     250.4
161        45     90       130     260.4
162        45     95       130     270.0
163        45    100       140     280.9
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4
```

```
print(new_df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 164 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  164 non-null    int64
 1   Pulse     164 non-null    int64
 2   Maxpulse  164 non-null    int64
```

```
 3   Calories  164 non-null    float64
dtypes: float64(1), int64(3)
memory usage: 6.4 KB
None
```

Por defecto, el método `dropna()` devuelve un nuevo DataFrame, y no cambia el original.

Si desea cambiar el DataFrame original, utilice el argumento `inplace = True`.

Código 9.26. Eliminar todas las filas con valores NULL.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

df.dropna(inplace = True)

print(df.to_string())
```

```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
5          60    102       127     300.0
6          60    110       136     374.0
7          45    104       134     253.3
8          30    109       133     195.1
9          60     98       124     269.0
10         60    103       147     329.3
11         60    100       120     250.7
12         60    106       128     345.3
13         60    104       132     379.3
14         60     98       123     275.0
15         60     98       120     215.2
16         60    100       120     300.0
18         60    103       123     323.0
19         45     97       125     243.0
20         60    108       131     364.2
21         45    100       119     282.0
22         60    130       101     300.0
23         45    105       132     246.0
24         60    102       126     334.5
25         60    100       120     250.0
26         60     92       118     241.0
28         60    100       132     280.0
29         60    102       129     380.3
30         60     92       115     243.0
31         45     90       112     180.1
32         60    101       124     299.0
33         60     93       113     223.0
34         60    107       136     361.0
35         60    114       140     415.0
```

36	60	102	127	300.0
37	60	100	120	300.0
38	60	100	120	300.0
39	45	104	129	266.0
40	45	90	112	180.1
41	60	98	126	286.0
42	60	100	122	329.4
43	60	111	138	400.0
44	60	111	131	397.0
45	60	99	119	273.0
46	60	109	153	387.6
47	45	111	136	300.0
48	45	108	129	298.0
49	60	111	139	397.6
50	60	107	136	380.2
51	80	123	146	643.1
52	60	106	130	263.0
53	60	118	151	486.0
54	30	136	175	238.0
55	60	121	146	450.7
56	60	118	121	413.0
57	45	115	144	305.0
58	20	153	172	226.4
59	45	123	152	321.0
60	210	108	160	1376.0
61	160	110	137	1034.4
62	160	109	135	853.0
63	45	118	141	341.0
64	20	110	130	131.4
65	180	90	130	800.4
66	150	105	135	873.4
67	150	107	130	816.0
68	20	106	136	110.4
69	300	108	143	1500.2
70	150	97	129	1115.0
71	60	109	153	387.6
72	90	100	127	700.0
73	150	97	127	953.2
74	45	114	146	304.0
75	90	98	125	563.2
76	45	105	134	251.0
77	45	110	141	300.0
78	120	100	130	500.4
79	270	100	131	1729.0
80	30	159	182	319.2
81	45	149	169	344.0
82	30	103	139	151.1
83	120	100	130	500.0
84	45	100	120	225.3
85	30	151	170	300.0
86	45	102	136	234.0
87	120	100	157	1000.1

88	45	129	103	242.0
89	20	83	107	50.3
90	180	101	127	600.1
92	30	90	107	105.3
93	15	80	100	50.5
94	20	150	171	127.4
95	20	151	168	229.4
96	30	95	128	128.2
97	25	152	168	244.2
98	30	109	131	188.2
99	90	93	124	604.1
100	20	95	112	77.7
101	90	90	110	500.0
102	90	90	100	500.0
103	90	90	100	500.4
104	30	92	108	92.7
105	30	93	128	124.0
106	180	90	120	800.3
107	30	90	120	86.2
108	90	90	120	500.3
109	210	137	184	1860.4
110	60	102	124	325.2
111	45	107	124	275.0
112	15	124	139	124.2
113	45	100	120	225.3
114	60	108	131	367.6
115	60	108	151	351.7
116	60	116	141	443.0
117	60	97	122	277.4
119	60	103	124	332.7
120	30	112	137	193.9
121	45	100	120	100.7
122	60	119	169	336.7
123	60	107	127	344.9
124	60	111	151	368.5
125	60	98	122	271.0
126	60	97	124	275.3
127	60	109	127	382.0
128	90	99	125	466.4
129	60	114	151	384.0
130	60	104	134	342.5
131	60	107	138	357.5
132	60	103	133	335.0
133	60	106	132	327.5
134	60	103	136	339.0
135	20	136	156	189.0
136	45	117	143	317.7
137	45	115	137	318.0
138	45	113	138	308.0
139	20	141	162	222.4
140	60	108	135	390.0
142	45	100	120	250.4

```
143        45    122       149     335.4
144        60    136       170     470.2
145        45    106       126     270.8
146        60    107       136     400.0
147        60    112       146     361.9
148        30    103       127     185.0
149        60    110       150     409.4
150        60    106       134     343.0
151        60    109       129     353.2
152        60    109       138     374.0
153        30    150       167     275.8
154        60    105       128     328.0
155        60    111       151     368.5
156        60     97       131     270.4
157        60    100       120     270.4
158        60    114       150     382.8
159        30     80       120     240.9
160        30     85       120     250.4
161        45     90       130     260.4
162        45     95       130     270.0
163        45    100       140     280.9
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4
```

Ahora, el `dropna(inplace = True)` NO devolverá un DataFrame nuevo, pero eliminará todas las filas que contengan valores `NULL` del DataFrame original.

9.11.2. Reemplazar Valores Vacíos

Otra forma de tratar con celdas vacías es insertar un nuevo valor en su lugar.

De esta manera, no tiene que eliminar filas enteras solo por algunas celdas vacías.

El `fillna()` el método nos permite reemplazar vacío celdas con un valor.

Código 9.27. Reemplace los valores NULL con el número 130.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

df.fillna(130, inplace = True)

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
```

```
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  169 non-null    float64
dtypes: float64(1), int64(3)
memory usage: 5.4 KB
None
```

9.11.3. Reemplazar Solo Para Columnas Especificadas

El ejemplo anterior reemplaza todas las celdas vacías en todo el conjunto de datos.

Para reemplazar solo valores vacíos para una columna, se puede especificar el nombre de la columna para el DataFrame.

Código 9.28. Reemplace los valores NULL en las columnas Çaloríasçon el número 130.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

df["Calories"].fillna(130, inplace = True)
```

```
/tmp/ipykernel_502828/10114066.py:5: FutureWarning: A value is trying to be set on a copy of a DataFr
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inp


  df["Calories"].fillna(130, inplace = True)
```

9.11.4. Reemplazar usando Media, Mediana o Moda

Una forma común de reemplazar las celdas vacías es calcular la media, la mediana o la moda de la columna.

Pandas usa los métodos `mean()`, `median()` y `mode()` métodos para calcular los valores respectivos para una columna especificada.

Código 9.29. Calcule la media y reemplazar cualquier valor vacío con ella.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)
```

```
/tmp/ipykernel_502828/3993554943.py:7: FutureWarning: A value is trying to be set on a copy of a Data
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inp


  df["Calories"].fillna(x, inplace = True)
```

Código 9.30. Calcular la mediana y reemplazar cualquier valor vacío con ella.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

x = df["Calories"].median()

df["Calories"].fillna(x, inplace = True)
```

```
/tmp/ipykernel_502828/1269507780.py:7: FutureWarning: A value is trying to be set on a copy of a Data
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inp


  df["Calories"].fillna(x, inplace = True)
```

Código 9.31. Calcule la moda y reemplazar cualquier valor vacío con ella.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

x = df["Calories"].mode()[0]

df["Calories"].fillna(x, inplace = True)
```

```
/tmp/ipykernel_502828/2697854522.py:7: FutureWarning: A value is trying to be set on a copy of a Data
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inp


  df["Calories"].fillna(x, inplace = True)
```

9.11.5. Convertir en un Formato Correcto

Las celdas de datos con formato incorrecto pueden dificultar, o incluso imposibilitar, el análisis de datos.

Para solucionarlo, tiene dos opciones: eliminar las filas o convertir todas las celdas en el columnas en el mismo formato.

En nuestro conjunto de datos, tenemos dos celdas con el formato incorrecto. Echa un vistazo a la fila 22 y 26, la columna 'Fecha' en el código 9.32 debe ser una cadena que represente una fecha.

Código 9.32. Celdas con formato incorrecto.

```
Duration Date Pulse Maxpulse Calories
  0 60 '2020/12/01' 110 130 409,1
  1 60 '2020/12/02' 117 145 479,0
  2 60 '2020/12/03' 103 135 340,0
  3 45 '2020/12/04' 109 175 282,4
```

```
  4 45 '2020/12/05' 117 148 406,0
  5 60 '2020/12/06' 102 127 300,0
  6 60 '2020/12/07' 110 136 374,0
  7 450 '2020/12/08' 104 134 253,3
  8 30 '2020/12/09' 109 133 195,1
  9 60 '2020/12/10' 98 124 269,0
  10 60 '2020/12/11' 103 147 329,3
  11 60 '2020/12/12' 100 120 250,7
  12 60 '2020/12/12' 100 120 250,7
  13 60 '2020/12/13' 106 128 345,3
  14 60 '2020/12/14' 104 132 379,3
  15 60 '2020/12/15' 98 123 275,0
  16 60 '2020/12/16' 98 120 215,2
  17 60 '2020/12/17' 100 120 300,0
  18 45 '2020/12/18' 90 112 NaN
  19 60 '2020/12/19' 103 123 323,0
  20 45 '2020/12/20' 97 125 243,0
  21 60 '2020/12/21' 108 131 364,2
  22 45 NaN 100 119 282,0
  23 60 '2020/12/23' 130 101 300,0
  24 45 '2020/12/24' 105 132 246,0
  25 60 '2020/12/25' 102 126 334,5
  26 60 20201226 100 120 250,0
  27 60 '2020/12/27' 92 118 241,0
  28 60 '2020/12/28' 103 132 NaN
  29 60 '2020/12/29' 100 132 280,0
  30 60 '2020/12/30' 102 129 380,3
  31 60 '2020/12/31' 92 115 243,0
```

Intentemos convertir todas las celdas de la columna 'Fecha' en fechas. Pandas tiene un método `to_datetime()` para esto.

Código 9.33. Convertir fecha.

```
import pandas as pd

df = pd.read_csv('data.csv')

df['Date'] = pd.to_datetime(df['Date'])

print(df.to_string())
```

Resultado:

```
Duración Fecha Pulso Maxpulse Calorías
  0 60 '2020/12/01' 110 130 409,1
  1 60 '2020/12/02' 117 145 479,0
  2 60 '2020/12/03' 103 135 340,0
  3 45 '2020/12/04' 109 175 282,4
  4 45 '2020/12/05' 117 148 406,0
  5 60 '2020/12/06' 102 127 300,0
  6 60 '2020/12/07' 110 136 374,0
```

```
  7 450 '2020/12/08' 104 134 253,3
  8 30 '2020/12/09' 109 133 195,1
  9 60 '2020/12/10' 98 124 269,0
  10 60 '2020/12/11' 103 147 329,3
  11 60 '2020/12/12' 100 120 250,7
  12 60 '2020/12/12' 100 120 250,7
  13 60 '2020/12/13' 106 128 345,3
  14 60 '2020/12/14' 104 132 379,3
  15 60 '2020/12/15' 98 123 275,0
  16 60 '2020/12/16' 98 120 215,2
  17 60 '2020/12/17' 100 120 300,0
  18 45 '2020/12/18' 90 112 NaN
  19 60 '2020/12/19' 103 123 323,0
  20 45 '2020/12/20' 97 125 243,0
  21 60 '2020/12/21' 108 131 364,2
  22 45 NaT 100 119 282,0
  23 60 '2020/12/23' 130 101 300,0
  24 45 '2020/12/24' 105 132 246,0
  25 60 '2020/12/25' 102 126 334,5
  26 60 '2020/12/26' 100 120 250,0
  27 60 '2020/12/27' 92 118 241,0
  28 60 '2020/12/28' 103 132 NaN
  29 60 '2020/12/29' 100 132 280,0
  30 60 '2020/12/30' 102 129 380,3
  31 60 '2020/12/31' 92 115 243,0
```

```
import pandas as pd

df = pd.read_csv('data/calorias.csv')

print(df)
```

```
    Unnamed: 0  Duration          Date  Pulse  Maxpulse Calories
0            0        60  '2020/12/01'    110       130    409,1
1            1        60  '2020/12/02'    117       145    479,0
2            2        60  '2020/12/03'    103       135    340,0
3            3        45  '2020/12/04'    109       175    282,4
4            4        45  '2020/12/05'    117       148    406,0
5            5        60  '2020/12/06'    102       127    300,0
6            6        60  '2020/12/07'    110       136    374,0
7            7       450  '2020/12/08'    104       134    253,3
8            8        30  '2020/12/09'    109       133    195,1
9            9        60  '2020/12/10'     98       124    269,0
10          10        60  '2020/12/11'    103       147    329,3
11          11        60  '2020/12/12'    100       120    250,7
12          12        60  '2020/12/12'    100       120    250,7
13          13        60  '2020/12/13'    106       128    345,3
14          14        60  '2020/12/14'    104       132    379,3
15          15        60  '2020/12/15'     98       123    275,0
16          16        60  '2020/12/16'     98       120    215,2
17          17        60  '2020/12/17'    100       120    300,0
18          18        45  '2020/12/18'     90       112      NaN
19          19        60  '2020/12/19'    103       123    323,0
```

```
20          20        45  '2020/12/20'     97       125    243,0
21          21        60  '2020/12/21'    108       131    364,2
22          22        45           NaN    100       119    282,0
23          23        60  '2020/12/23'    130       101    300,0
24          24        45  '2020/12/24'    105       132    246,0
25          25        60  '2020/12/25'    102       126    334,5
26          26        60      20201226    100       120    250,0
27          27        60  '2020/12/27'     92       118    241,0
28          28        60  '2020/12/28'    103       132      NaN
29          29        60  '2020/12/29'    100       132    280,0
30          30        60  '2020/12/30'    102       129    380,3
31          31        60  '2020/12/31'     92       115    243,0
```

Como puede ver en el resultado, la fecha en la fila 26 fue fija en el código 9.33, pero la fecha vacía en la fila 22 obtuvo un valor de `NaT` (No un tiempo), en otras palabras, un valor vacío. Una forma de lidiar con los valores vacíos es simplemente eliminar toda la fila.

9.11.6. Borrar Filas

El resultado de la conversión en el ejemplo anterior nos dio un valor `NaT`, que se puede manejar como un valor `NULL`, y podemos eliminar la fila utilizando el método `dropna()`.

Código 9.34. Eliminar filas con un valor NULL en la columna "Fecha".

```
df.dropna(subset=['Date'], inplace = True)
```

9.11.7. Datos Incorrectos

Los '`Datos incorrectos`' no tienen que ser '`celdas vacías`' o '`formato incorrecto`', puede ser solo equívoco, como si alguien registrara '`199`' en lugar de '`1.99`'.

A veces puede detectar datos incorrectos mirando el conjunto de datos, porque tiene una expectativa de qué debería ser.

Si echa un vistazo a nuestro conjunto de datos, puede ver que en la fila 7 del código 9.35, la duración es 450, pero para todas las demás filas la duración es entre 30 y 60.

No tiene que estar mal, pero teniendo en cuenta que este es el conjunto de datos del entrenamiento de alguien, concluimos con el hecho de que la persona no entrenó 450 minutos.

Código 9.35. DataFrame con dato incorrecto.

```
    Duration          Date  Pulse  Maxpulse  Calories
0         60  '2020/12/01'    110       130     409.1
1         60  '2020/12/02'    117       145     479.0
2         60  '2020/12/03'    103       135     340.0
3         45  '2020/12/04'    109       175     282.4
4         45  '2020/12/05'    117       148     406.0
5         60  '2020/12/06'    102       127     300.0
6         60  '2020/12/07'    110       136     374.0
7        450  '2020/12/08'    104       134     253.3
8         30  '2020/12/09'    109       133     195.1
9         60  '2020/12/10'     98       124     269.0
10        60  '2020/12/11'    103       147     329.3
11        60  '2020/12/12'    100       120     250.7
```

```
12        60  '2020/12/12'    100       120     250.7
13        60  '2020/12/13'    106       128     345.3
14        60  '2020/12/14'    104       132     379.3
15        60  '2020/12/15'     98       123     275.0
16        60  '2020/12/16'     98       120     215.2
17        60  '2020/12/17'    100       120     300.0
18        45  '2020/12/18'     90       112       NaN
19        60  '2020/12/19'    103       123     323.0
20        45  '2020/12/20'     97       125     243.0
21        60  '2020/12/21'    108       131     364.2
22        45           NaN    100       119     282.0
23        60  '2020/12/23'    130       101     300.0
24        45  '2020/12/24'    105       132     246.0
25        60  '2020/12/25'    102       126     334.5
26        60      20201226    100       120     250.0
27        60  '2020/12/27'     92       118     241.0
28        60  '2020/12/28'    103       132       NaN
29        60  '2020/12/29'    100       132     280.0
30        60  '2020/12/30'    102       129     380.3
31        60  '2020/12/31'     92       115     243.0
```

¿Cómo podemos corregir valores incorrectos, como el de "Duración.en la fila 7?

9.11.8. Sustitución de Valores

Una forma de corregir valores incorrectos es reemplazarlos con otra cosa.

En nuestro ejemplo, lo más probable es un error tipográfico, y el valor debe ser “45” en lugar de “450”, y nosotros podría insertar “45” en la fila 7 del código 9.36:

Código 9.36. Establecer “Duración” = 45 en la fila 7.

```
df.loc[7, 'Duration'] = 45
```

Para conjuntos de datos pequeños, es posible que pueda reemplazar los datos incorrectos uno por uno pero no para grandes conjuntos de datos.

Para reemplazar datos incorrectos para conjuntos de datos más grandes, puede crear algunas reglas, establezca algunos límites para los valores legales y reemplace cualquier valor que esté fuera del límites.

Código 9.37. Recorre todos los valores en la columna “Duración”, si el valor es superior a 120, establezca en 120.

```
for x in df.index:
    if df.loc[x, "Duration"] > 120:
        df.loc[x, "Duration"] = 120
```

9.11.9. Eliminación de Filas

Otra forma de manejar datos incorrectos es eliminar las filas que contienen datos incorrectos.

De esta manera no tiene que averiguar con qué reemplazarlos, y los hay una buena oportunidad de que no los necesite para hacer sus análisis.

Código 9.38. Eliminar filas donde "Duración" es superior a 120.

```
for x in df.index:
    if df.loc[x, "Duration"] > 120:
        df.drop(x, inplace = True)
```

9.11.10. Datos Duplicados

Las filas duplicadas son filas que se han registrado más de una vez.

Código 9.39. DataFrame con datos duplicados.

```
Duración Fecha Pulso Maxpulse Calorías
0 60 '2020/12/01' 110 130 409,1
1 60 '2020/12/02' 117 145 479,0
2 60 '2020/12/03' 103 135 340,0
3 45 '2020/12/04' 109 175 282,4
4 45 '2020/12/05' 117 148 406,0
5 60 '2020/12/06' 102 127 300,0
6 60 '2020/12/07' 110 136 374,0
7 450 '2020/12/08' 104 134 253,3
8 30 '2020/12/09' 109 133 195,1
9 60 '2020/12/10' 98 124 269,0
10 60 '2020/12/11' 103 147 329,3
11 60 '2020/12/12' 100 120 250,7
12 60 '2020/12/12' 100 120 250,7
13 60 '2020/12/13' 106 128 345,3
14 60 '2020/12/14' 104 132 379,3
15 60 '2020/12/15' 98 123 275,0
16 60 '2020/12/16' 98 120 215,2
17 60 '2020/12/17' 100 120 300,0
18 45 '2020/12/18' 90 112 NaN
19 60 '2020/12/19' 103 123 323,0
20 45 '2020/12/20' 97 125 243,0
21 60 '2020/12/21' 108 131 364,2
22 45 NaN 100 119 282,0
23 60 '2020/12/23' 130 101 300,0
24 45 '2020/12/24' 105 132 246,0
25 60 '2020/12/25' 102 126 334,5
26 60 20201226 100 120 250,0
27 60 '2020/12/27' 92 118 241,0
28 60 '2020/12/28' 103 132 NaN
29 60 '2020/12/29' 100 132 280,0
30 60 '2020/12/30' 102 129 380,3
31 60 '2020/12/31' 92 115 243,0
```

Al echar un vistazo a nuestro conjunto de datos de prueba, podemos suponer que las **filas 11 y 12 son duplicados**.

Para descubrir duplicados, podemos usar el método `duplicated()`.

El método `duplicated()` devuelve valores booleanos para cada fila.

Código 9.40. Devuelve `True` por cada fila que es un duplicado, de otra manera `False`.

```
print(df.duplicated())
```

9.12. Correlación de los datos

Un gran aspecto del módulo Pandas es el método `corr()`. Este método calcula la relación entre cada columna en su conjunto de datos.

Código 9.41. Mostrar la relación entre las columnas.

```
import pandas as pd

df = pd.read_csv('data/data.csv')

print(df.corr())
```

```
          Duration     Pulse  Maxpulse  Calories
Duration  1.000000 -0.155408  0.009403  0.922717
Pulse    -0.155408  1.000000  0.786535  0.025121
Maxpulse  0.009403  0.786535  1.000000  0.203813
Calories  0.922717  0.025121  0.203813  1.000000
```

El resultado del método `corr()` es una tabla con muchos números que representa qué tan bien está la relación entre dos columnas.

El número varía de - 1 a 1. Si la correlación vale 1 significa que hay una relación de 1 a 1 (una correlación perfecta) y para este conjunto de datos, cada vez que un valor sube en la primer columna, el otro valor también subía.

Un valor 0.9 también es una buena relación, y si aumenta un valor, el otro probablemente también aumentará.

Para un valor -0.9 habría tan buena relación como con 0.9, pero en este casi si aumenta un valor, el otro probablemente bajará.

Por último, 0.2 no es una buena relación, lo que significa que si un valor sube no significa que el otro lo haga.

¿Qué es una buena correlación? Depende del uso, pero creo que es seguro decir que tienes que tener al menos 0.6 (o -0.6) para llamarlo una buena correlación.

Correlación Perfecta Podemos ver que "Duración" y "Duración" obtuvieron el número 1.000000, lo que tiene sentido cada columna siempre tiene una relación perfecta consigo misma.

Buena Correlación "Duración" y "Calorías" tienen un 0.922721 correlación, lo cual es una muy buena correlación, y podemos predecir que cuanto más tiempo trabaje fuera, cuantas más calorías queme, y al revés: si quemó mucho de calorías, probablemente tuviste un largo ejercicio.

Mala Correlación "Duración" y "Maxpulse" tiene un 0.009403 correlación, lo cual es una correlación muy mala, lo que significa que no podemos predecir el pulso máximo con solo mirar la duración del ejercicio, y viceversa.

9.13. Visualización de datos

Pandas utiliza el métpdo `plot()` para crear gráficas.

Podemos utilizar `Pyplot`, un submódulo de la biblioteca `Matplotlib` para visualizar el diagrama en la pantalla.

Código 9.42. Visualización de un DataFrame.

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

df = pd.read_csv('data/data.csv')

df.plot()

plt.show()
```

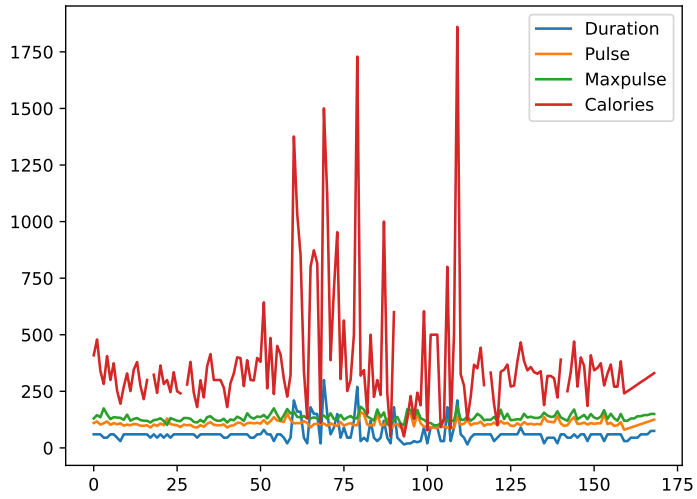


Figura 9.1: Visualización de un DataFrame

9.13.1. Diagrama de dispersión

Especifique que desea un gráfico de dispersión con el argumento `kind`.

```
kind = 'scatter'
```

Un diagrama de dispersión necesita un eje x y un eje y.

En el código 9.43, utilizaremos “Duración” para el eje x y “Calorías” para el eje y.

Incluya los argumentos x e y de la siguiente manera:

```
    x = 'Duration', y = 'Calories'
```

Código 9.43. Diagrama de dispersión.

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

```
df = pd.read_csv('data/data.csv')

df.plot(kind = 'scatter', x = 'Duration', y = 'Calories')

plt.show()
```

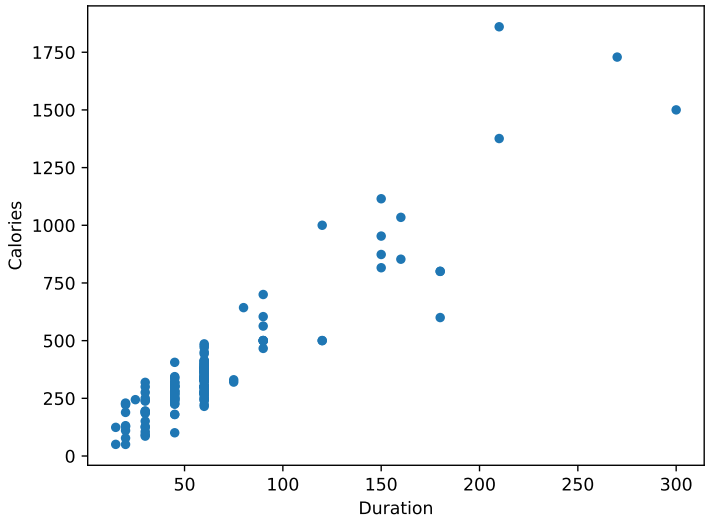


Figura 9.2: Diagrama de dispersión

En la sección anterior, aprendimos que la correlación entre "Duración" y "Calorías" era 0.922721, y concluimos con el hecho de que una mayor duración significa más calorías quemadas.

Al observar el diagrama de dispersión, podemos observar esa relación.

Creemos otro diagrama de dispersión, donde hay una mala relación entre las columnas, como "Duración" y "Pulso máximo", con la correlación 0.009403.

Código 9.44. Un diagrama de dispersión donde no hay relación entre las columnas.

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

df = pd.read_csv('data/data.csv')

df.plot(kind = 'scatter', x = 'Duration', y = 'Maxpulse')

plt.show()
```

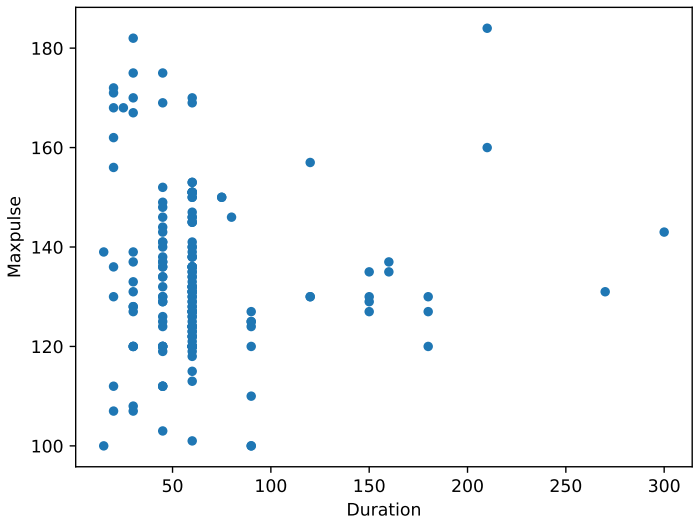


Figura 9.3: Diagrama de dispersión sin relación entre columnas

9.13.2. Histograma

Utilice el kindargumento para especificar que desea un histograma:

```
kind = 'hist'
```

Un histograma solo necesita una columna. Un histograma nos muestra la frecuencia de cada intervalo, por ejemplo ¿cuántos entrenamientos duraron entre 50 y 60 minutos?

En el siguiente ejemplo, utilizaremos la columna "Duración"para crear el histograma.

Código 9.45. Histograma de un DataFrame.

```
df["Duration"].plot(kind = 'hist')
```

```
<Axes: ylabel='Frequency'>
```

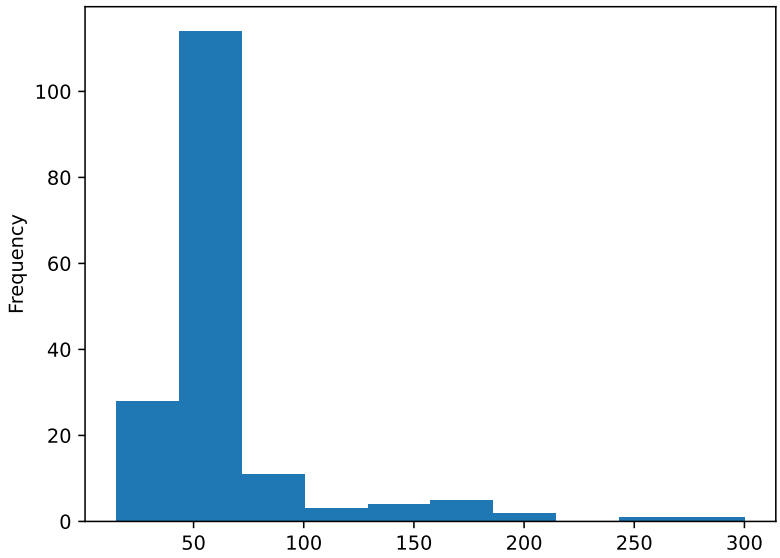


Figura 9.4: Histograma de un DataFrame

El histograma nos dice que hay más de 100 ejercicios que duraron entre 50 y 60 minutos.

Código 9.46. Ahora un histograma para las calorías.

```
df["Calories"].plot(kind = 'hist')
```

```
<Axes: ylabel='Frequency'>
```

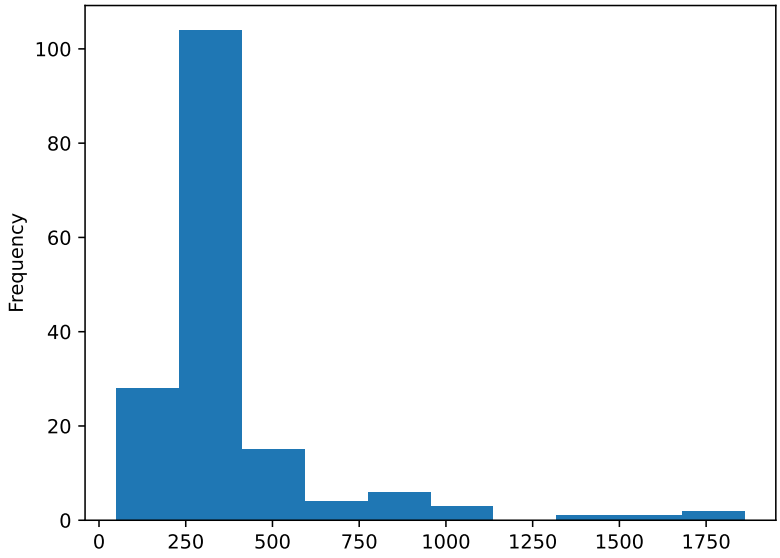


Figura 9.5: Histograma para las calorías

Este histograma nos dice que hubo poco más de 100 casos donde las calorías quemadas fueron entre 250 y 300.

Dado que el método `plot` de Pandas utiliza `matplotlib` como backend, si se desea modificar aspectos finos de la gráfica se debe hacer configurando el gráfico a través de `matplotlib` directamente.

9.14. Referencias

- Manejo de archivos en Python.
- Lutz M., Learning Python, O'Reilly. 2009
- Pandas, sitio oficial.
- Pandas en la W3Schools.
- Pandas Plot

Capítulo 10

Matplotlib

Matplotlib es una biblioteca de trazado de gráficos de bajo nivel en Python que sirve como una utilidad de visualización.

Matplotlib es de código abierto y fue creado por John D. Hunter.

Está escrito principalmente en Python, algunos segmentos están escritos en C, Objective-C y Javascript.

10.1. Comprobar versión de matplotlib

Una vez que `matplotlib` esté instalado es necesario importarlo. Se puede verificar la versión del paquete con el comando.

```
import matplotlib
print(matplotlib.__version__)
```

```
3.8.3
```

10.2. Pyplot

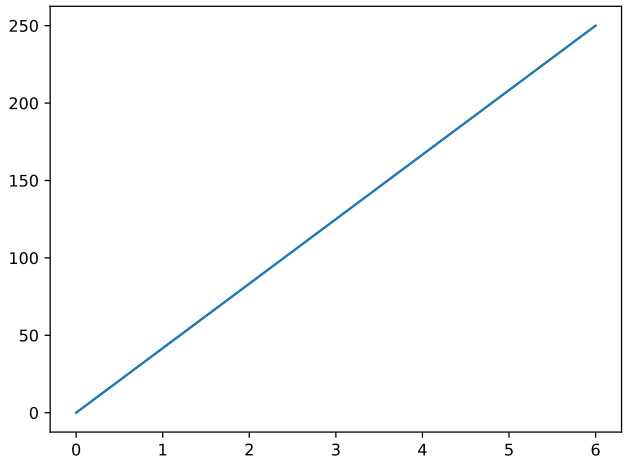
La mayoría de las utilidades de `Matplotlib` se encuentran bajo el submódulo `pyplot`, y generalmente se importan bajo el alias `plt`:

```
import matplotlib.pyplot as plt
```

Ahora el paquete `Pyplot` se puede denominar `plt`.

Código 10.1. Dibuja una línea en un diagrama desde la posición $(0, 0)$ hasta la posición $(6, 250)$.

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```



10.3. Graficas con matplotlib

10.3.1. Gráfica puntos (x, y)

La función `plot()` se utiliza para dibujar puntos en un diagrama. Por defecto, `plot()` dibuja una línea de punto a punto.

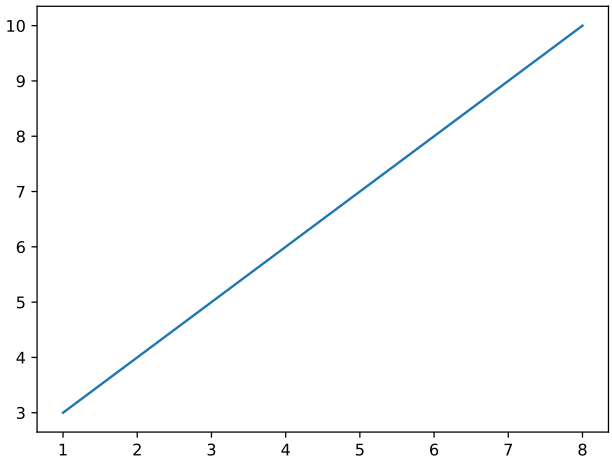
La función toma parámetros para especificar puntos en el diagrama.

- El parámetro 1 es una matriz que contiene los puntos en el eje X.
- El parámetro 2 es una matriz que contiene los puntos en el eje Y.

Si necesitamos trazar una línea de (1, 3) a (8, 10), tenemos que pasar dos matrices [1, 8] y [3, 10] a la función de trazado.

Código 10.2. Dibuje una línea en un diagrama desde la posición (1, 3) a la posición (8, 10)

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

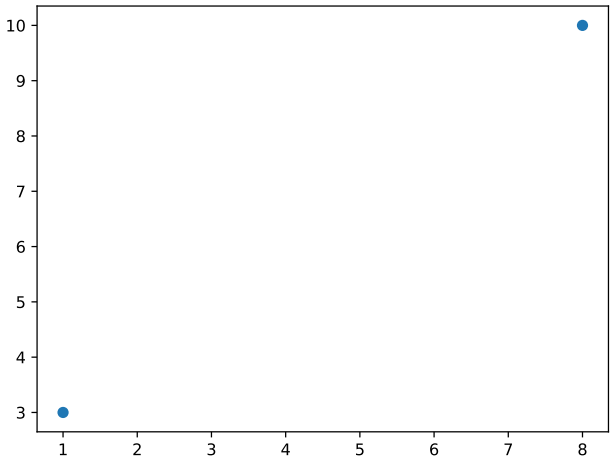


10.4. Gráfica Sin Línea

Para trazar solo los marcadores, puede usar notación de cadena de acceso directo parámetro `'o'`, que significa *'anillos'*.

Código 10.3. Dibuja dos puntos en el diagrama, uno en posición (1, 3) y otro en posición (8, 10).

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints, 'o')
plt.show()
```

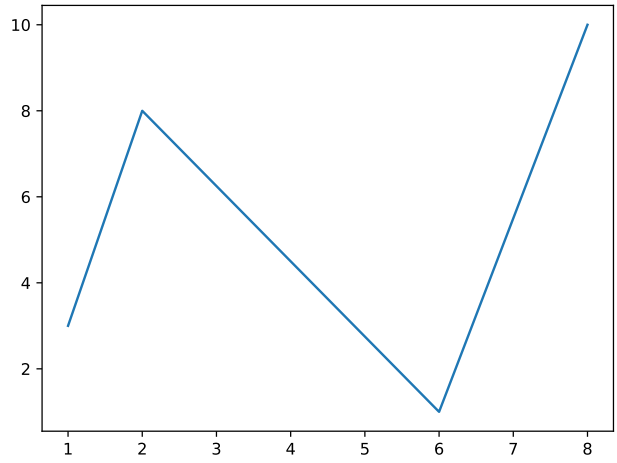


10.5. Múltiples Puntos

Puede trazar tantos puntos como desee, solo asegúrese de tener el mismo número de puntos en ambos ejes.

Código 10.4. Dibuja una línea en un diagrama desde la posición $(1,3)$ a $(2,8)$ luego a $(6,1)$ y finalmente a la posición $(8,10)$

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```



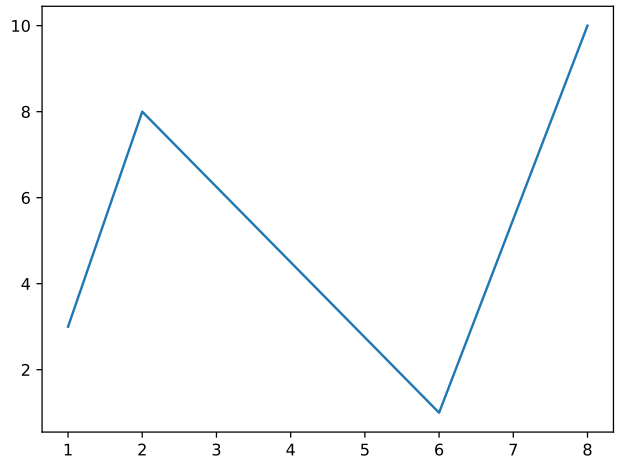
10.6. Valores x predeterminados

Si no especificamos los puntos en el eje x, obtendrán los valores predeterminados $0, 1, 2, 3, \ldots$, dependiendo de la longitud de los puntos y.

Entonces, si tomamos el mismo ejemplo que el anterior y dejamos de lado los puntos x, el diagrama se verá así:

Código 10.5. Gráfica sin valores x.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10, 5, 7])

plt.plot(ypoints)
plt.show()
```

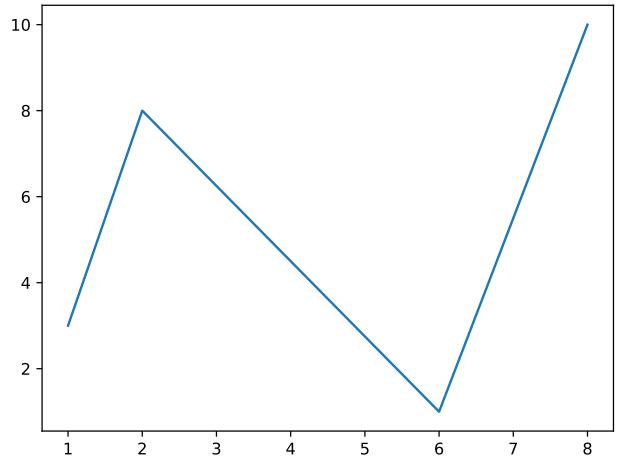


10.7. Marcadores

Puede usar el argumento `marker` para enfatizar cada punto con un marcador especifico.

Código 10.6. Marcar cada punto con un círculo.

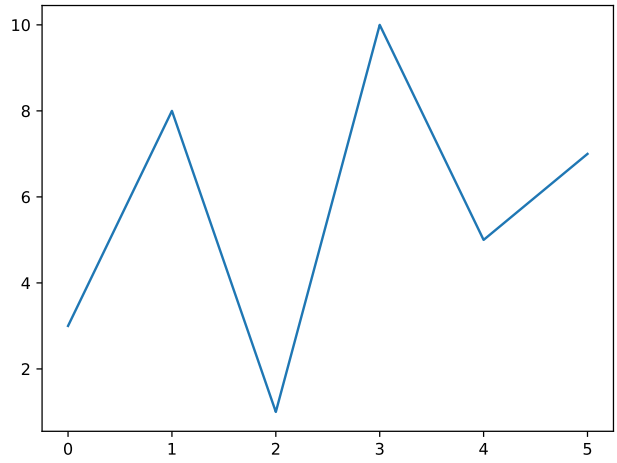
```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o')
plt.show()
```



Código 10.7. Marca cada punto con una estrella.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])
```

```
plt.plot(ypoints, marker = '*')
plt.show()
```



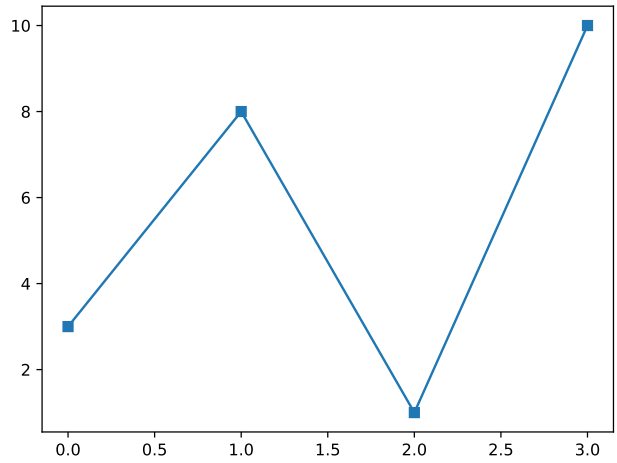
10.7.1. Referencia de los marcadores

Puede elegir cualquiera de estos marcadores:

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	'|'
'_'	Hline

Código 10.8. Marcador cuadrado.

```
plt.plot(ypoints, marker = 's')
plt.show()
```



10.8. Formato cadena `fmt`

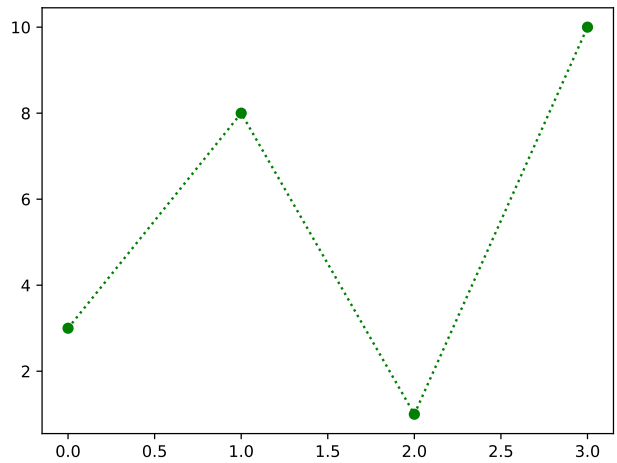
También puedes usar el notación de cadena de acceso directo parámetro para especificar el marcador.

Este parámetro también se llama `fmt` y está escrito con esta sintaxis:

```
marker|line|color
```

Código 10.9. Marque cada punto con un círculo.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, 'o:g')
plt.show()
```



El valor del marcador puede ser cualquier cosa de la Referencia del Marcador anterior.

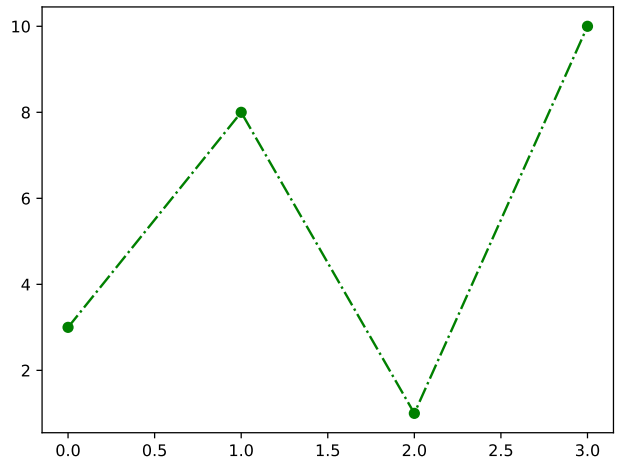
El valor de la línea puede ser uno de los siguientes:

10.8.1. Referencia de Línea

Sintaxis línea	Descripción
'-'	Solid line
':'	Dotted line 1

'--' Dashed line '-.' Dashed/dotted line

```
plt.plot(ypoints, 'o-.g')
plt.show()
```



10.8.2. Referencia de Color

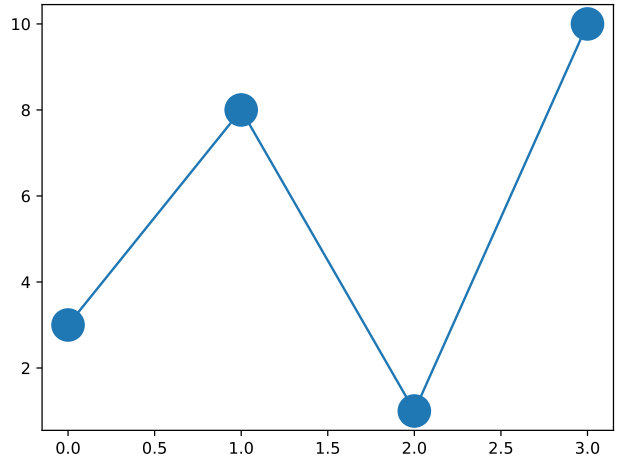
Sintaxis	Descripción
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

10.8.3. Tamaño del Marcador

Puede usar el argumento de palabra clave `markersize` o la versión más corta, `ms` para establecer el tamaño de los marcadores.

Código 10.10. Establezca el tamaño de los marcadores en 20.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20)
plt.show()
```

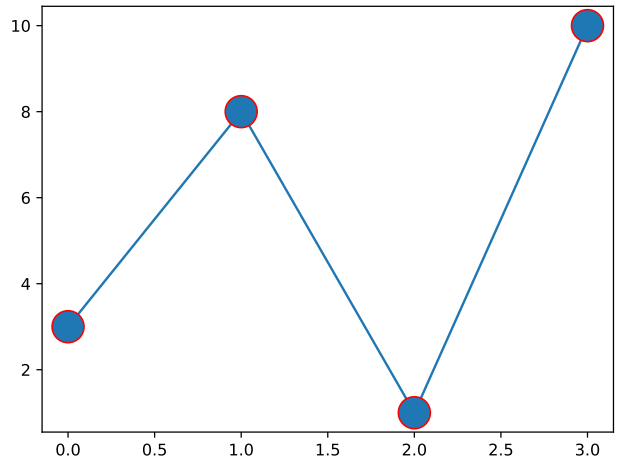


10.8.4. Color del Marcador

Puede usar el argumento de palabra clave `markeredgecolor` o el más corto `mec` para establecer el color de la borde de los marcadores.

Código 10.11. Establezca el color EDGE en rojo.

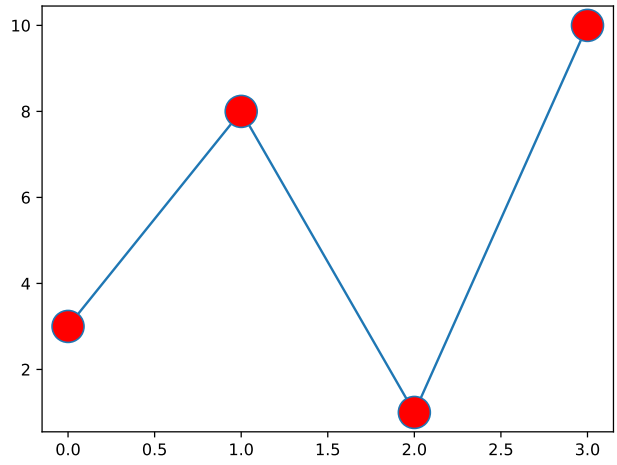
```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r')
plt.show()
```



Puede usar el argumento de palabra clave `markerfacecolor` o el más corto `mfc` para establecer el color dentro del borde de los marcadores:

Código 10.12. Coloque el color FACE en rojo.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mfc = 'r')
plt.show()
```

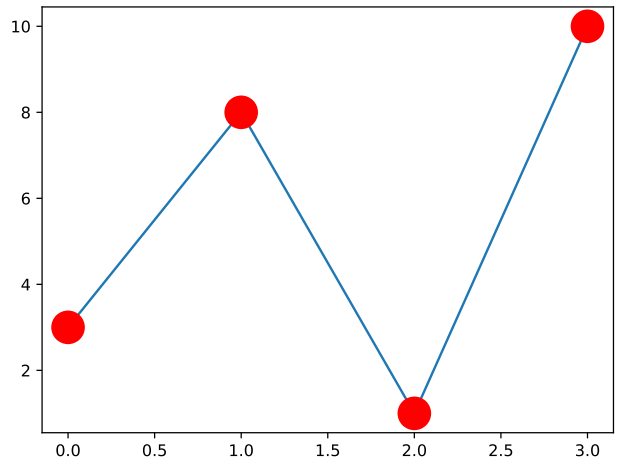


Usar ambos el `mec` y `mfc` argumentos para colorear todo el marcador:

Código 10.13. Establezca el color de ambos borde y el cara a rojo.

```
import matplotlib.pyplot as plt
import numpy as np
```

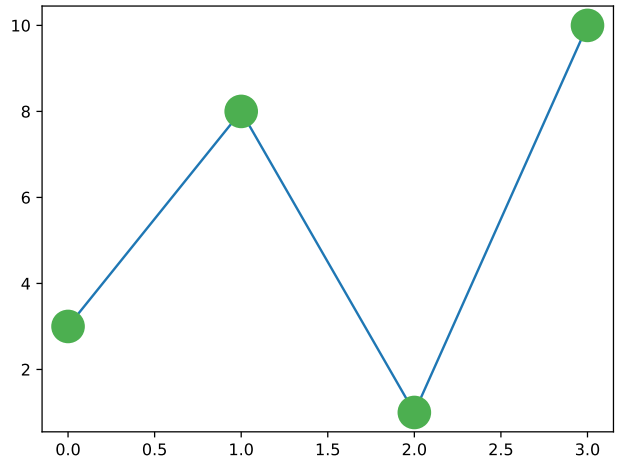
```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r', mfc = 'r')
plt.show()
```



También puedes usar valores de color hexadecimal, o cualquiera de los 140 nombres de color compatibles.

Código 10.14. Marque cada punto con un hermoso color verde.

```
plt.plot(ypoints, marker = 'o', ms = 20, mec = '#4CAF50', mfc = '#4CAF50')
plt.show()
```

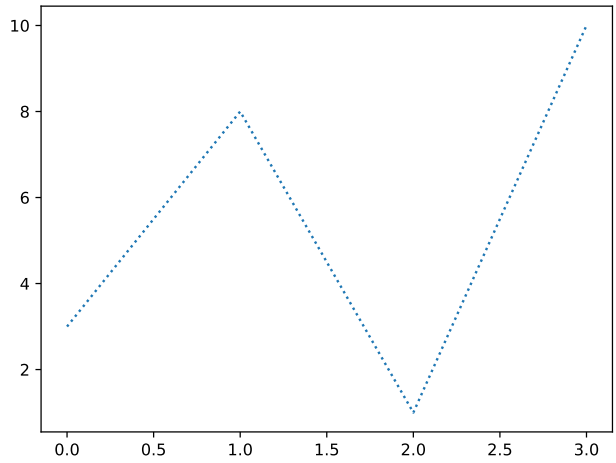


10.9. Estilo de línea

Puede usar el argumento de palabra clave `linestyle`, o más corto `ls`, a cambiar el estilo de la línea trazada.

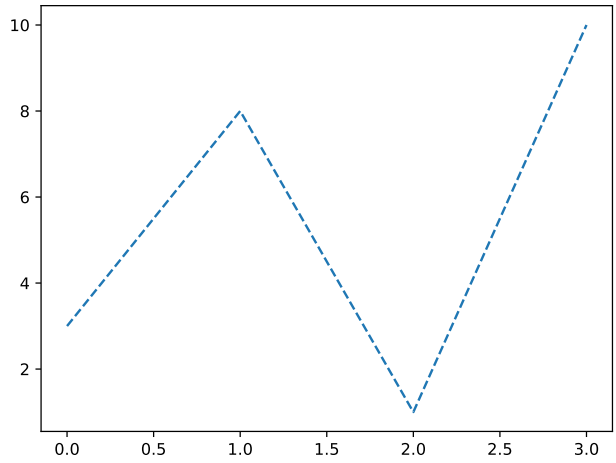
Código 10.15. Usa una línea punteada.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle = 'dotted')
plt.show()
```



Código 10.16. Usa una línea discontinua.

```
plt.plot(ypoints, linestyle = 'dashed')
plt.show()
```



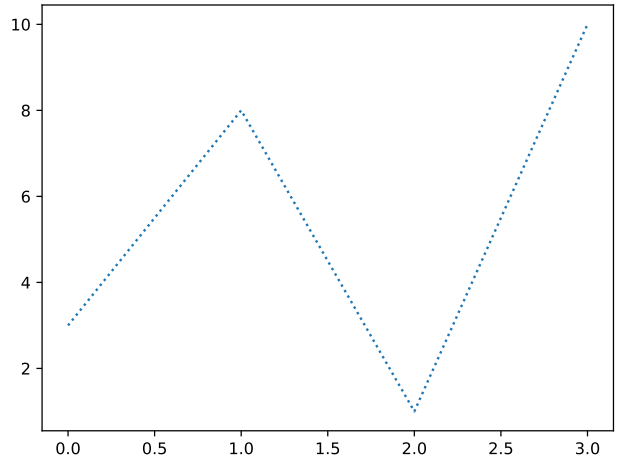
10.9.1. Sintaxis más corta

El estilo de línea se puede escribir en una sintaxis más corta:

- `linestyle` se puede escribir como `ls`.
- `dotted` se puede escribir como `:`.
- `dashed` se puede escribir como `--`.

Código 10.17. Sintaxis más corta.

```
plt.plot(ypoints, ls = ':')
plt.show()
```



10.10. Estilos de Línea

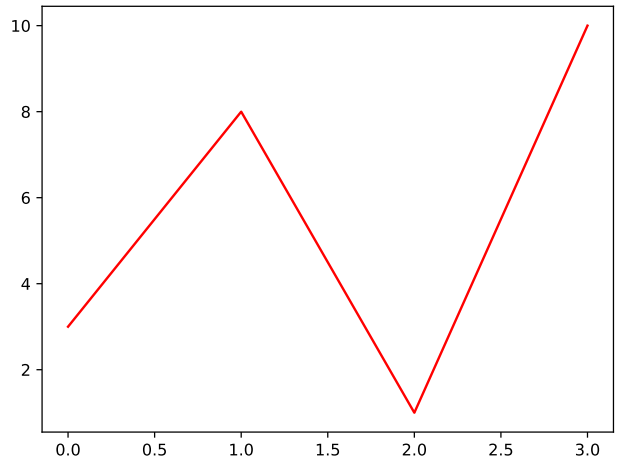
Puedes elegir cualquiera de estos estilos:

| Style | Or | |:-....:|:--:| | 'solid' | (default) '-' | | 'dotted' | ':' | | 'dashed' | '--' | | 'dashdot' | '-.' | | 'None' | '' or ' ' |

10.11. Color de Línea

Puede usar el argumento de palabra clave `color` o el más corto `c` para establecer el color de la línea.
Código 10.18. Establecer el color de la línea en rojo.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, color = 'r')
plt.show()
```



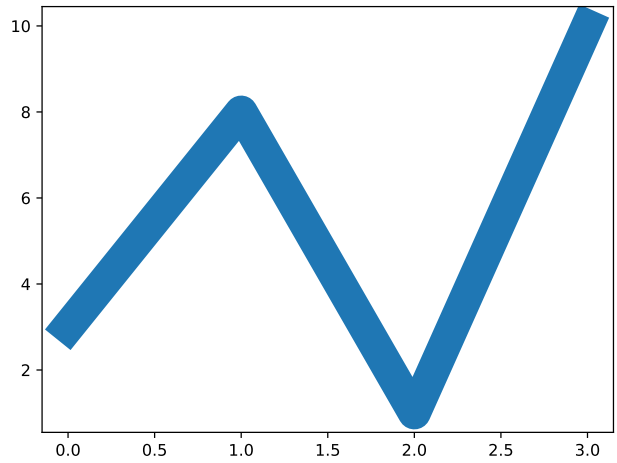
También se puede utilizar el color en su valor hexadecimal o los nombres de color compatibles.

10.12. Ancho de Línea

Puede usar el argumento de palabra clave `linewidth` o el más corto `lw` para cambiar el ancho de la línea.

El valor es un número flotante, en puntos.
Código 10.19. Parcela con una línea ancha de 20.5pt.

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth = '20.5')
plt.show()
```

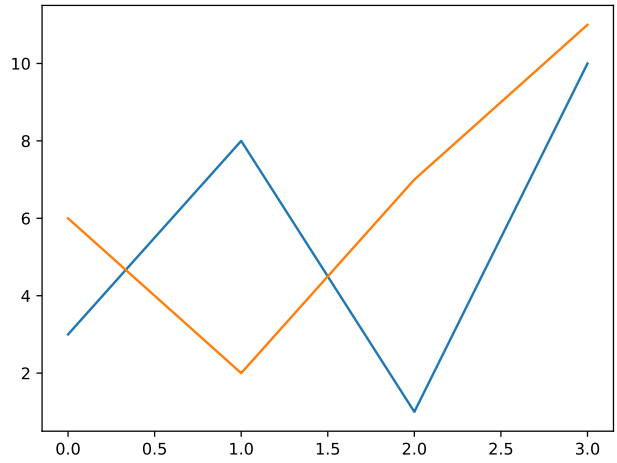


10.13. Múltiples Líneas

Puede trazar tantas líneas como desee simplemente agregando más `plt.plot()` funciones.

Código 10.20. Dibuja dos líneas especificando un `plt.plot()` función para cada línea.

```
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```



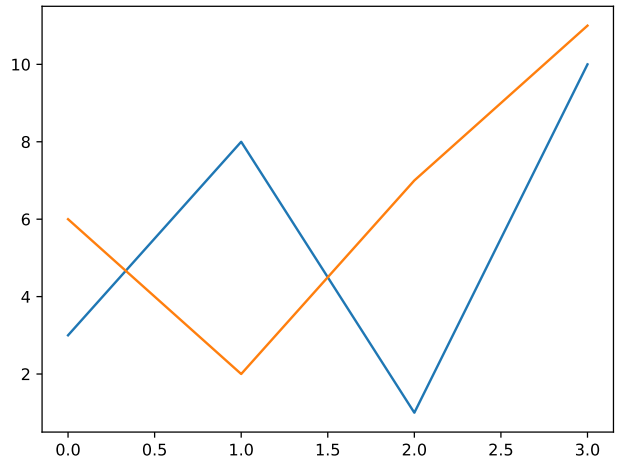
También puede trazar muchas líneas agregando los puntos para el eje x y y para cada línea en la misma función `plt.plot()`.

(En los ejemplos anteriores solo especificamos los puntos en el eje y, lo que significa que los puntos en el eje x obtuvieron los valores predeterminados (0, 1, 2, 3).)

Los valores x y y vienen en pares.

Código 10.21. Dibuje dos líneas especificando los valores x y y para ambas líneas.

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.array([0, 1, 2, 3])
y1 = np.array([3, 8, 1, 10])
x2 = np.array([0, 1, 2, 3])
y2 = np.array([6, 2, 7, 11])

plt.plot(x1, y1, x2, y2)
plt.show()
```

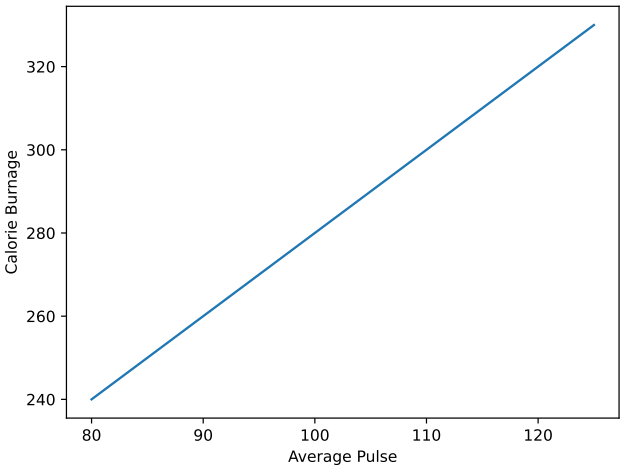


10.14. Crear Etiquetas para una gráfica

Con `Pyplot`, puede usar las funciones `xlabel()` y `ylabel()` para establecer una etiqueta para el eje x y y.

Código 10.22. Agregar etiquetas al eje x y y.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```



10.15. Crear un título para una trama

Con `Pyplot`, puedes utilizar la función `title()` para establecer un título para el gráfico.
Código 10.23. Agregue un título de gráfico y etiquetas para los ejes x e y.

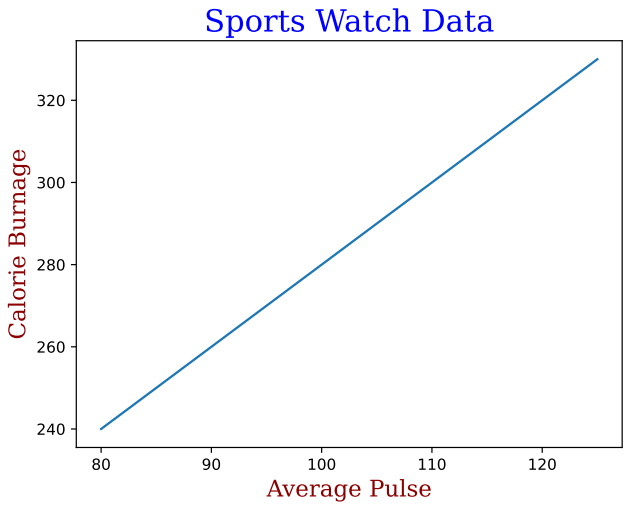
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

10.16. Establecer fuente para títulos y etiquetas

Puede utilizar el parámetro `fontdict` en `xlabel()`, `ylabel()` y `title()` para establecer las propiedades de fuente para el título y las etiquetas.
Código 10.24. Establecer propiedades de fuente para el título y las etiquetas.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

plt.title("Sports Watch Data", fontdict = font1)
```

```
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

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

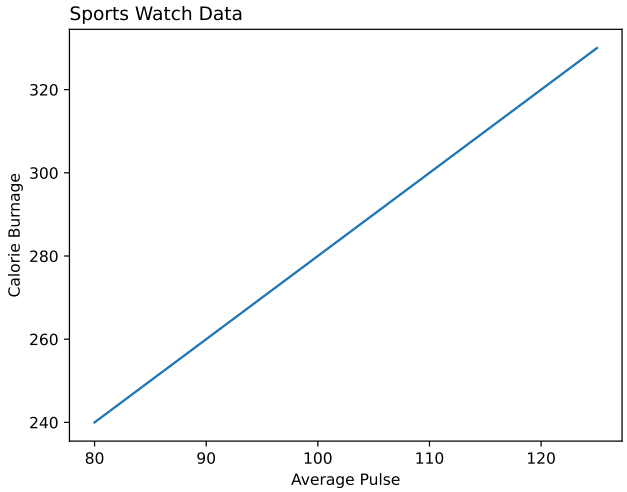


10.17. Posicionar el título

Puedes utilizar el parámetro `loc` en `title()` para posicionar el título.

Los valores admitidos son: *'left'*, *'right'* y *'center'*. El valor predeterminado es *'center'*.
Código 10.25. Coloque el título a la izquierda.

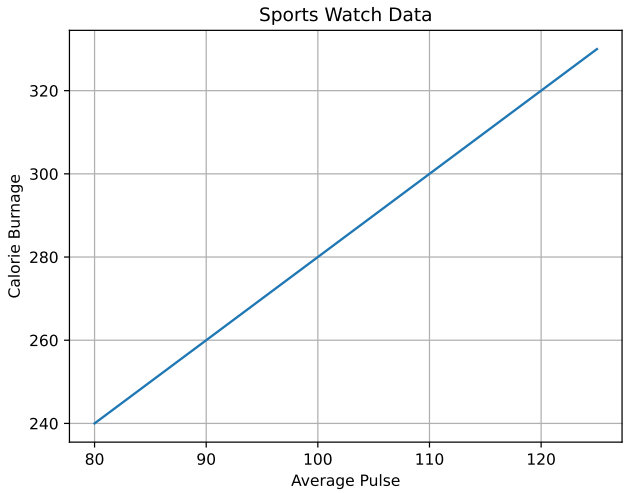
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data", loc = 'left')
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

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



10.18. Agregar cuadrícula a un gráfico

Con `Pyplot`, puedes usar la función `grid()` para agregar cuadrícula al gráfico.
Código 10.26. Añadir líneas de cuadrícula al gráfico.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

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



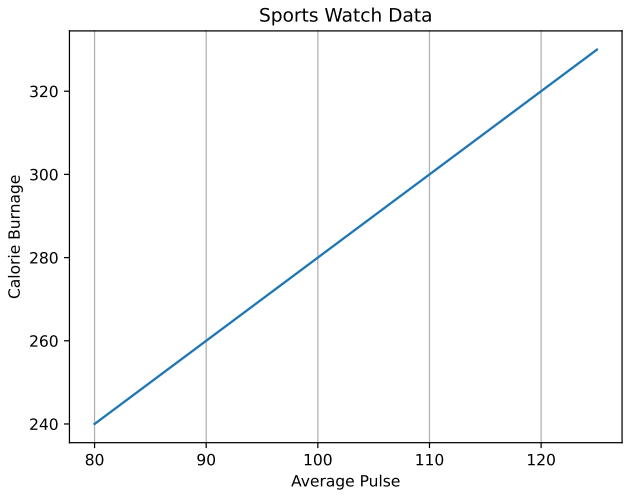
10.19. Especificar qué líneas de cuadrícula se mostrarán

Puede utilizar el parámetro `axis` en la función `grid()` para especificar qué líneas de cuadrícula mostrar.

Los valores admitidos son: *'x'*, *'y'* y *'both'*. El valor predeterminado es *'both'*.

Código 10.27. Mostrar solo líneas de cuadrícula para el eje x.

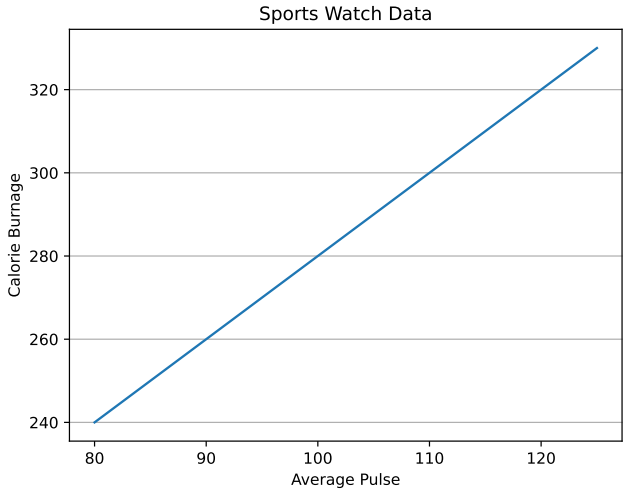
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)
plt.grid(axis = 'x')
plt.show()
```



Código 10.28. Mostrar solo líneas de cuadrícula para el eje y.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)
plt.grid(axis = 'y')
```

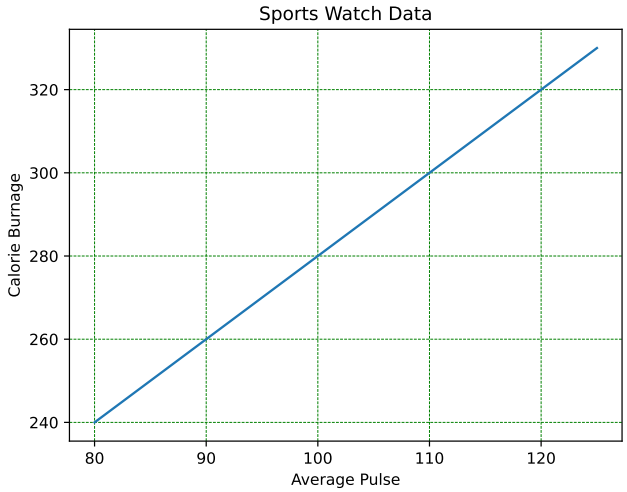
```
plt.show()
```



10.20. Establecer propiedades de línea para la cuadrícula

También puede configurar las propiedades de línea de la cuadrícula, de esta manera: `grid(color = '_color_', linestyle = '_linestyle_', linewidth = _number_)`.
Código 10.29. Establecer las propiedades de línea de la cuadrícula.

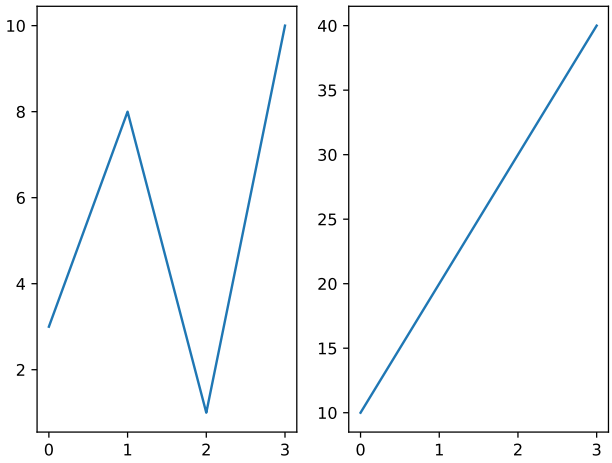
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)
plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)
plt.show()
```



10.21. Mostrar múltiples gráficos

Con esta función `subplot()` puedes dibujar múltiples gráficos en una figura.
Código 10.30. Dibuje 2 gráficos.

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.show()
```



10.22. La función subplot()

La función `subplot()` toma tres argumentos que describen el diseño de la figura.

El diseño está organizado en filas y columnas, que están representadas por el primer y segundo argumento.

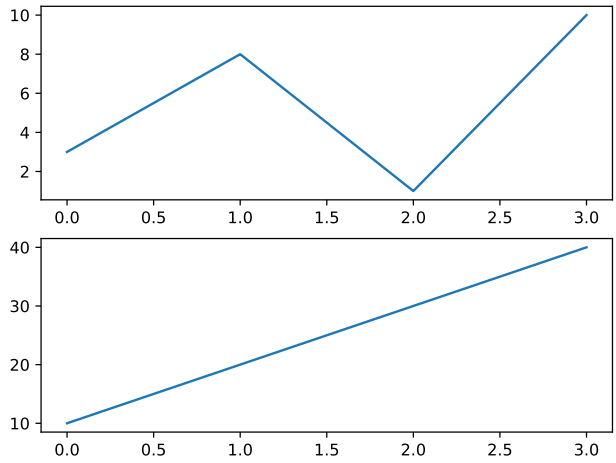
El tercer argumento representa el índice de la malla actual.

```
plt.subplot(1, 2, 1)
#the figure has 1 row, 2 columns, and this plot is the first plot.

plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

Entonces, si queremos una figura con 2 filas y 1 columna (lo que significa que los dos gráficos se mostrarán uno encima del otro en lugar de uno al lado del otro), podemos escribir la sintaxis de esta manera.

Código 10.31. Dibuje 2 gráficos uno encima del otro.

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 1, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 1, 2)
plt.plot(x,y)
plt.show()
```

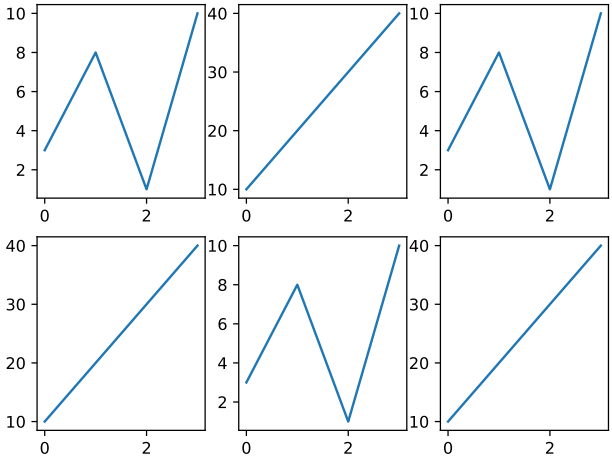


Puede dibujar tantos gráficos como quiera en una figura, simplemente describa el número de filas, columnas y el índice del gráfico.

Código 10.32. Dibuje 6 gráficos.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 1)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 2)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 3)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 4)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 5)
plt.plot(x,y)
```

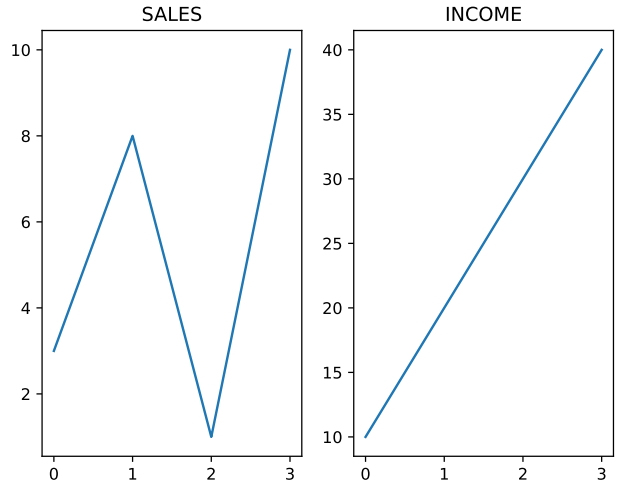
```
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 6)
plt.plot(x,y)
plt.show()
```



10.23. Título

Puede agregar un título a cada gráfico con la función `title()`.
Código 10.33. 2 parcelas, con títulos.

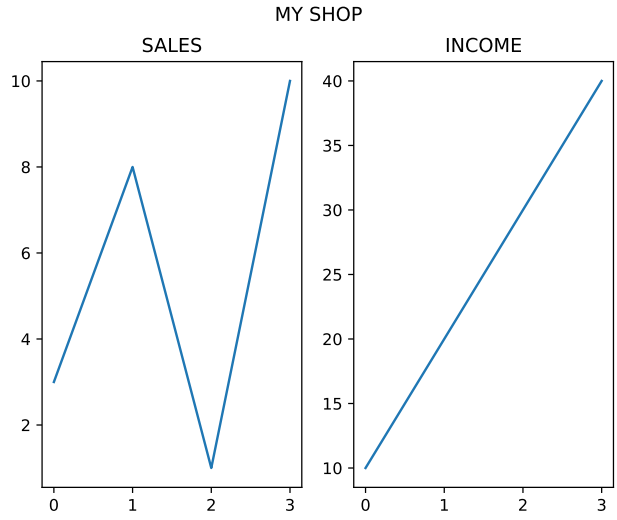
```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")

plt.show()
```



10.24. Súper título

Puede agregar un título a toda la figura con la función `suptitle()`.
Código 10.34. Añade un título para toda la figura.

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")

plt.suptitle("MY SHOP")
plt.show()
```



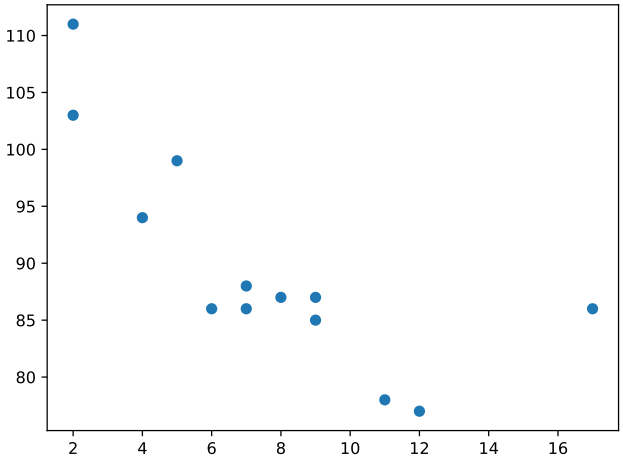
10.25. Gráficos de dispersión

Con `Pyplot`, puede utilizar la función `scatter()` para dibujar un diagrama de dispersión.

La función `scatter()` traza un punto para cada observación. Necesita dos matrices de la misma longitud, una para los valores del eje x y otra para los valores del eje y.

Código 10.35. Un diagrama de dispersión simple.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])

plt.scatter(x, y)
plt.show()
```



La observación en el ejemplo anterior es el resultado de 13 automóviles.

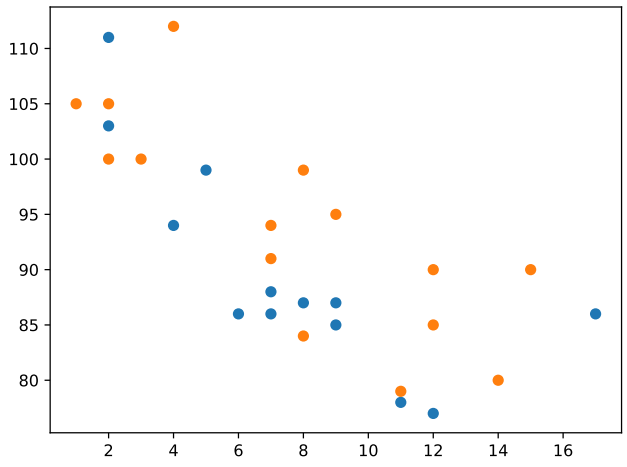
El eje x muestra la antigüedad del coche y el eje y muestra la velocidad del automóvil cuando pasa.

¿Existe alguna relación entre las observaciones?. Parece que cuanto más nuevo es el coche, más rápido va, pero eso podría ser una coincidencia, después de todo sólo registramos 13 coches.

10.26. Comparar gráficos

En el ejemplo anterior, parece haber una relación entre la velocidad y la edad, pero ¿qué pasa si graficamos también las observaciones de otro día? ¿El diagrama de dispersión nos dirá algo más?
Código 10.36. Dibuje dos gráficos en la misma figura.

```
import matplotlib.pyplot as plt
import numpy as np

#day one, the age and speed of 13 cars:
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)

#day two, the age and speed of 15 cars:
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)

plt.show()
```



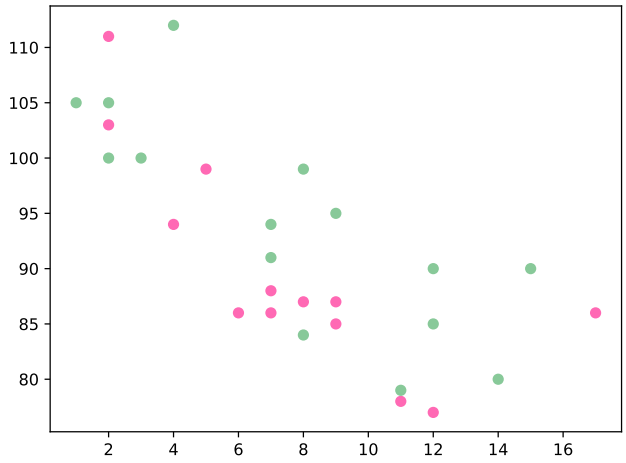
Nota: Los dos gráficos están representados con dos colores diferentes, por defecto azul y naranja.

Comparando ambos gráficos, creo que es seguro decir que ambos nos llevan a la misma conclusión: cuanto más nuevo es el coche, más rápido va.

10.27. Colores

Puede establecer su propio color para cada gráfico de dispersión con `color` o el argumento `c`.
Código 10.37. Establezca su propio color de marcadores.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y, color = 'hotpink')

x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y, color = '#88c999')

plt.show()
```



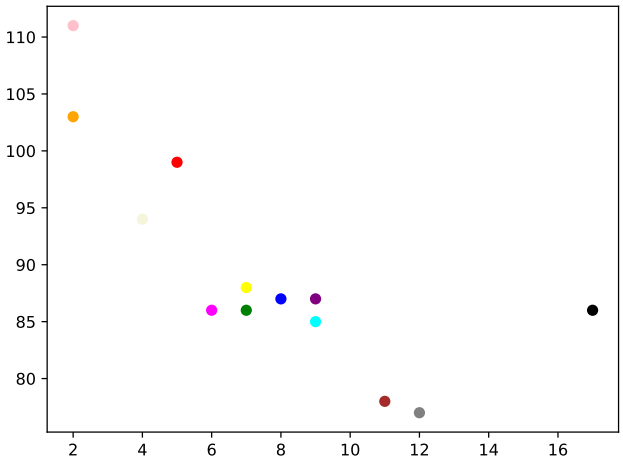
10.28. Colorear cada punto

Es posible establecer un color específico para cada punto utilizando una matriz de colores como valor para el cargumento.

Nota: No puede usar el argumento `color` para esto, solo el argumento `c`.

Código 10.38. Colorear puntos de dispersión.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors = np.array(["red","green","blue","yellow","pink","black","orange","purple",
    "beige","brown","gray","cyan","magenta"])

plt.scatter(x, y, c=colors)

plt.show()
```



10.29. Mapa de colores

El módulo `Matplotlib` tiene varios mapas de colores disponibles.

Un mapa de colores es una lista de colores, donde cada color tiene un valor que varía de 0 a 100.

A continuación se muestra un ejemplo de un mapa de colores:

Este mapa de colores se llama *'viridis'* y como puedes ver, varía desde 0, que es un color púrpura, hasta 100, que es un color amarillo.

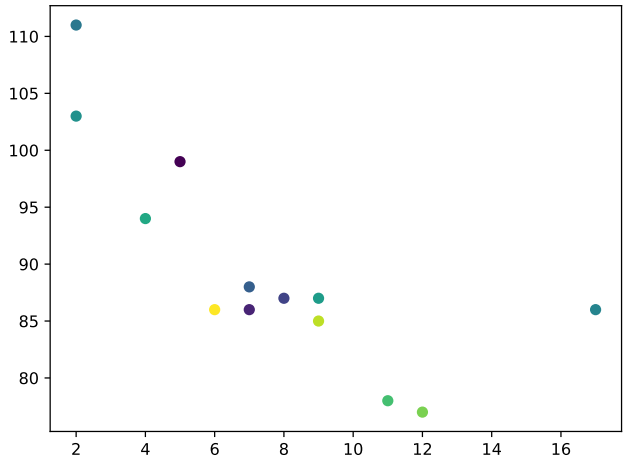
Cómo utilizar el mapa de colores

Puede especificar el mapa de colores con el argumento de palabra clave `cmap`, en este caso *'viridis'* que es uno de los mapas de colores integrados disponibles en `Matplotlib`.

Además, debes crear una matriz con valores (de 0 a 100), un valor para cada punto en el diagrama de dispersión.

Código 10.39. Cree una matriz de colores y especifique un mapa de colores en el gráfico de dispersión.

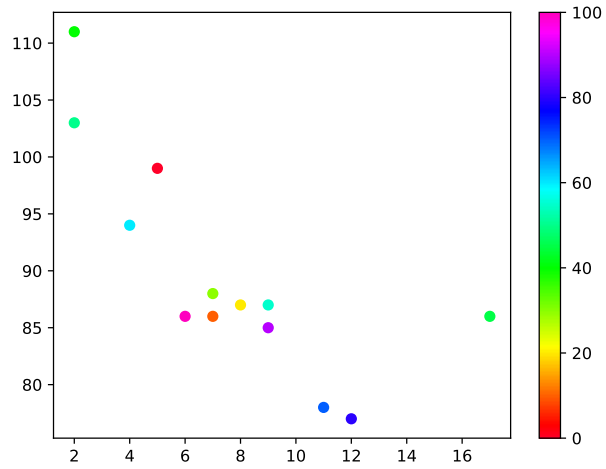
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors = np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])

plt.scatter(x, y, c=colors, cmap='viridis')

plt.show()
```



Puede incluir el mapa de colores en el dibujo incluyendo la declaración `plt.colorbar()`.

Código 10.40. Incluya el mapa de colores real.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors = np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])

plt.scatter(x, y, c=colors, cmap='gist_rainbow')

plt.colorbar()

plt.show()
```



10.30. Mapas de colores disponibles

Puede elegir cualquiera de los mapas de colores incorporados.

```
from matplotlib import colormaps
list(colormaps)
```

```
['magma',
 'inferno',
 'plasma',
 'viridis',
 'cividis',
 'twilight',
 'twilight_shifted',
 'turbo',
 'Blues',
 'BrBG',
 'BuGn',
 'BuPu',
 'CMRmap',
 'GnBu',
 'Greens',
 'Greys',
 'OrRd',
 'Oranges',
 'PRGn',
 'PiYG',
 'PuBu',
 'PuBuGn',
 'PuOr',
 'PuRd',
 'Purples',
 'RdBu',
 'RdGy',
 'RdPu',
 'RdYlBu',
```

```
'RdYlGn',
'Reds',
'Spectral',
'Wistia',
'YlGn',
'YlGnBu',
'YlOrBr',
'YlOrRd',
'afmhot',
'autumn',
'binary',
'bone',
'brg',
'bwr',
'cool',
'coolwarm',
'copper',
'cubehelix',
'flag',
'gist_earth',
'gist_gray',
'gist_heat',
'gist_ncar',
'gist_rainbow',
'gist_stern',
'gist_yarg',
'gnuplot',
'gnuplot2',
'gray',
'hot',
'hsv',
'jet',
'nipy_spectral',
'ocean',
'pink',
'prism',
'rainbow',
'seismic',
'spring',
'summer',
'terrain',
'winter',
'Accent',
'Dark2',
'Paired',
'Pastel1',
'Pastel2',
'Set1',
'Set2',
'Set3',
'tab10',
'tab20',
```

```
'tab20b',
'tab20c',
'grey',
'gist_grey',
'gist_yerg',
'Grays',
'magma_r',
'inferno_r',
'plasma_r',
'viridis_r',
'cividis_r',
'twilight_r',
'twilight_shifted_r',
'turbo_r',
'Blues_r',
'BrBG_r',
'BuGn_r',
'BuPu_r',
'CMRmap_r',
'GnBu_r',
'Greens_r',
'Greys_r',
'OrRd_r',
'Oranges_r',
'PRGn_r',
'PiYG_r',
'PuBu_r',
'PuBuGn_r',
'PuOr_r',
'PuRd_r',
'Purples_r',
'RdBu_r',
'RdGy_r',
'RdPu_r',
'RdYlBu_r',
'RdYlGn_r',
'Reds_r',
'Spectral_r',
'Wistia_r',
'YlGn_r',
'YlGnBu_r',
'YlOrBr_r',
'YlOrRd_r',
'afmhot_r',
'autumn_r',
'binary_r',
'bone_r',
'brg_r',
'bwr_r',
'cool_r',
'coolwarm_r',
'copper_r',
```

```
 'cubehelix_r',
 'flag_r',
 'gist_earth_r',
 'gist_gray_r',
 'gist_heat_r',
 'gist_ncar_r',
 'gist_rainbow_r',
 'gist_stern_r',
 'gist_yarg_r',
 'gnuplot_r',
 'gnuplot2_r',
 'gray_r',
 'hot_r',
 'hsv_r',
 'jet_r',
 'nipy_spectral_r',
 'ocean_r',
 'pink_r',
 'prism_r',
 'rainbow_r',
 'seismic_r',
 'spring_r',
 'summer_r',
 'terrain_r',
 'winter_r',
 'Accent_r',
 'Dark2_r',
 'Paired_r',
 'Pastel1_r',
 'Pastel2_r',
 'Set1_r',
 'Set2_r',
 'Set3_r',
 'tab10_r',
 'tab20_r',
 'tab20b_r',
 'tab20c_r']
```

10.31. Tamaño

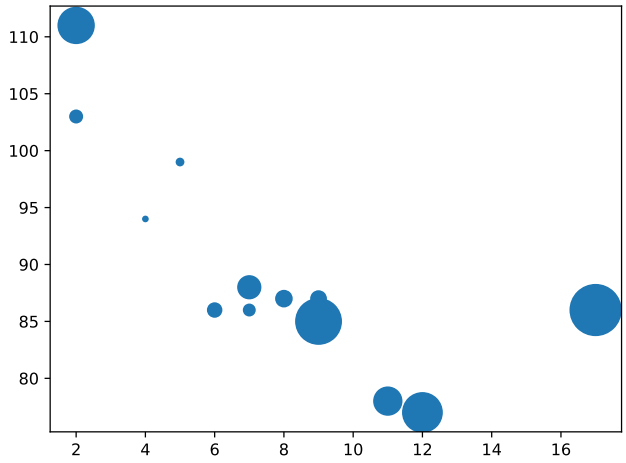
Puede cambiar el tamaño de los puntos con el sargumento.

Al igual que con los colores, asegúrese de que la matriz de tamaños tenga la misma longitud que las matrices de ejes x y y.

Código 10.41. Establezca su propio tamaño para los marcadores.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])
```

```
plt.scatter(x, y, s=sizes)
plt.show()
```



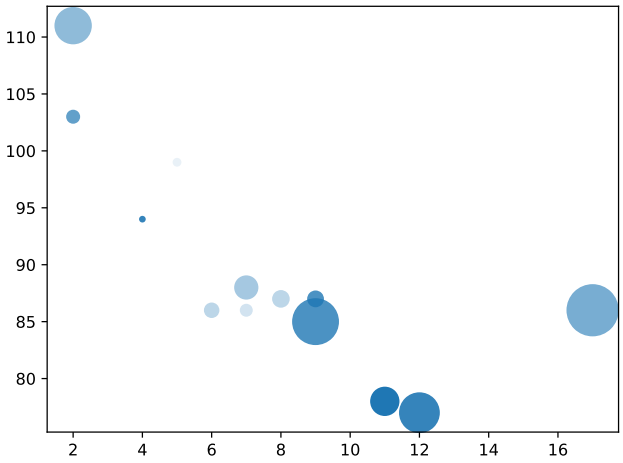
10.32. Transparencia

Puede ajustar la transparencia de los puntos con el argumento `alpha`.

Al igual que con los colores, es posbile ajustar la transparencia a los puntos individualmente, sólo asegúrese de que la matriz de tamaños tenga la misma longitud que las matrices de ejes x y y.

Código 10.42. Establezca su propio tamaño para los marcadores.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])
transparency = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 0.9, 0.8, 0.3])

plt.scatter(x, y, s=sizes, alpha=transparency)
plt.show()
```

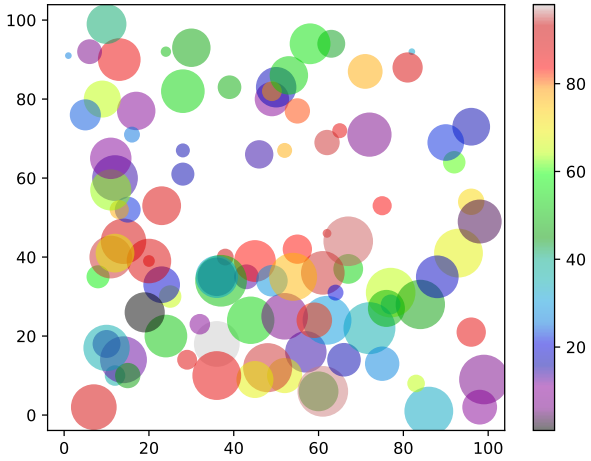


10.33. Combinar color, tamaño y alfa

Se puede combinar un mapa de colores con diferentes tamaños de puntos. Esto se visualiza mejor si los puntos son transparentes.

Código 10.43. Crear matrices aleatorias con 100 valores para puntos *x*, puntos *y*, colores y tamaños.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randint(100, size=(100))
y = np.random.randint(100, size=(100))
colors = np.random.randint(100, size=(100))
sizes = 10 * np.random.randint(100, size=(100))

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='nipy_spectral')
plt.colorbar()
plt.show()
```

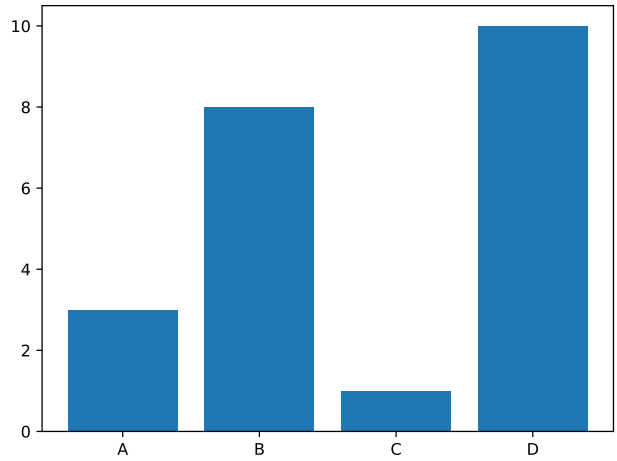


10.34. Diagrama de barras

Con `Pyplot`, se puede utilizar la función `bar()` para dibujar gráficos de barras.

Código 10.44. Dibuja 4 barras.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```



La función `bar()` toma argumentos que describen el diseño de las barras.

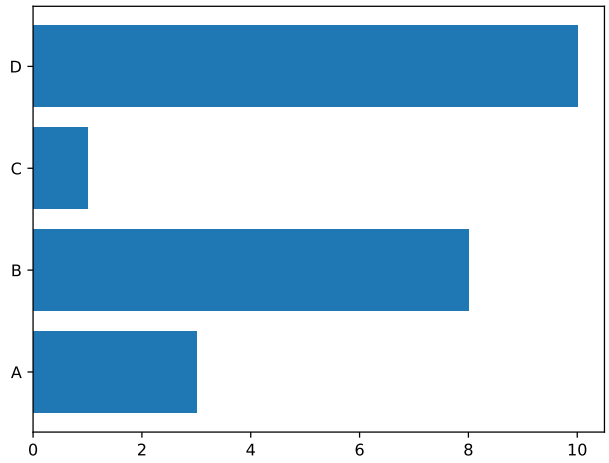
Las categorías y sus valores representados por el primer y segundo argumento como matrices.

10.35. Barras horizontales

Si desea que las barras se muestren horizontalmente en lugar de verticalmente, utilice la función `barh()`.

Código 10.45. Dibuja 4 barras horizontales.

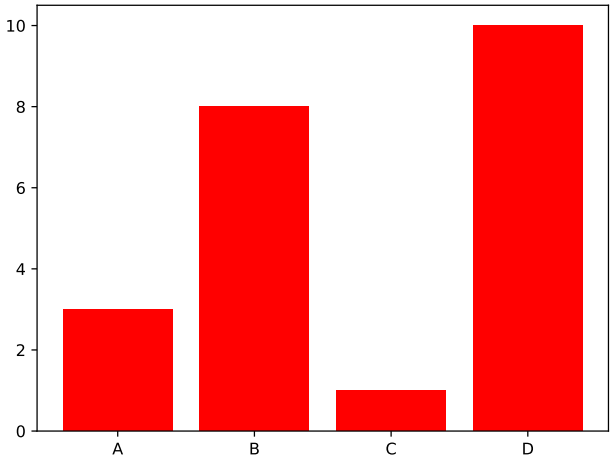
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.barh(x, y)
plt.show()
```



10.36. Color de la barra

El argumento `color` de palabra clave `bar()` y `barh()` se utiliza para establecer el color de las barras.
Código 10.46. Dibuja 4 barras rojas.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, color = "red")
plt.show()
```



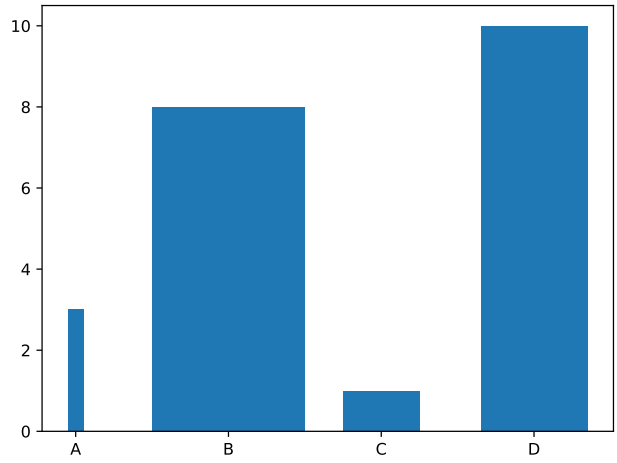
Nota. Puede utilizar cualquiera de los 140 nombres de colores admitidos o valores de color hexadecimales.

10.37. Ancho de barra

El argumento `width` de palabra clave `bar()` permite establecer el ancho de las barras.

Código 10.47. Dibuja 4 barras muy finas.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
width = np.array([0.1, 1, 0.5, 0.7])

plt.bar(x, y, width = width)
plt.show()
```



El valor de ancho predeterminado es 0.8.

Nota: Para barras horizontales, utilice `height` en lugar de `width`.

10.38. Histograma

En Matplotlib, se utiliza la función `hist()` para crear histogramas.

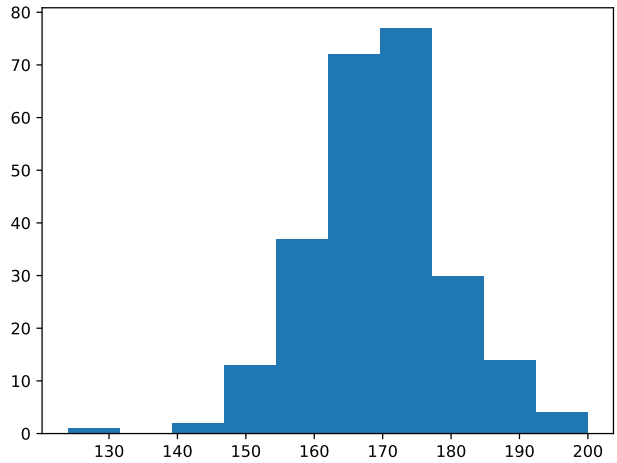
La función `hist()` utilizará una matriz de números para crear un histograma; la matriz se envía a la función como argumento.

A manera de ejemplo, se utiliza `NumPy` para generar una matriz con 250 valores aleatorios, donde los valores se concentrarán alrededor de 170 y la desviación estándar es 10.

Código 10.48. Una distribución de datos normal por `NumPy`.

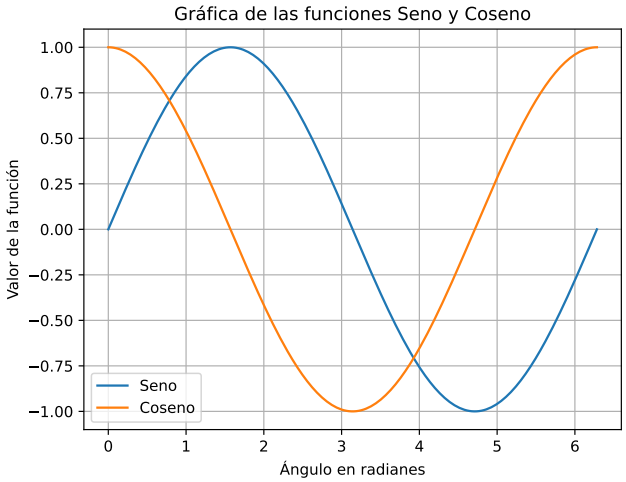
```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.normal(170, 10, 250)
```

```
plt.hist(x)
plt.show()
```



Código 10.49. Graficar las funciones Seno y Coseno.

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib

x = np.linspace(0, 2 * np.pi, 200)
y1 = np.sin(x)
y2 = np.cos(x)

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

plt.legend()
plt.grid()
plt.title('Gráfica de las funciones Seno y Coseno')
plt.xlabel('Ángulo en radianes')
plt.ylabel('Valor de la función')

plt.show()
```



10.39. Referencias

- Matplotlib en GitHub

Capítulo 11

LDA - sklearn

11.1. Carga de los módulos

```
import numpy as np
import pandas as pd
```

11.2. Lectura de los datos

```
datos = pd.read_table("data/datosAB.txt", sep='\t')
```

```
datos
```

```
      a    b clase
0   168  141     r
1   165  143     r
2   170  143     r
3   172  145     r
4   174  145     r
5   167  147     r
6   174  147     r
7   169  149     r
8   170  150     r
9   164  151     r
10  172  151     r
11  175  152     r
12  164  153     r
13  168  154     r
14  170  156     r
15  173  157     r
16  176  159     r
17  175  162     r
18  165  151     n
19  157  153     n
20  167  156     n
21  171  156     n
22  160  155     n
```

```
23  165  150     n
24  177  161     n
25  179  162     n
26  172  163     n
27  168  160     n
28  172  164     n
29  171  165     n
30  178  165     n
31  169  166     n
32  165  168     n
33  174  168     n
34  173  169     n
35  160  143     n
```

11.3. Separación de datos

```
X = datos.iloc[:,:-1]
y = datos.iloc[:,2]
```

```
y
```

```
0     r
1     r
2     r
3     r
4     r
5     r
6     r
7     r
8     r
9     r
10    r
11    r
12    r
13    r
14    r
15    r
16    r
17    r
18    n
19    n
20    n
21    n
22    n
23    n
24    n
25    n
26    n
27    n
28    n
29    n
30    n
```

```
31    n
32    n
33    n
34    n
35    n
Name: clase, dtype: object
```

11.4. Creación de subconjutos CP y CE

```
from sklearn.model_selection import train_test_split
```

```
X_ce, X_cp, y_ce, y_cp = train_test_split(X, y, test_size=0.3, random_state=0)
```

```
X_ce
```

```
      a    b
35  160  143
33  174  168
28  172  164
32  165  168
8   170  150
13  168  154
5   167  147
17  175  162
14  170  156
7   169  149
26  172  163
1   165  143
12  164  153
25  179  162
24  177  161
6   174  147
23  165  150
4   174  145
18  165  151
21  171  156
19  157  153
9   164  151
34  173  169
3   172  145
0   168  141
```

```
y_ce
```

```
35    n
33    n
28    n
32    n
8     r
13    r
5     r
17    r
14    r
```

```
7     r
26    n
1     r
12    r
25    n
24    n
6     r
23    n
4     r
18    n
21    n
19    n
9     r
34    n
3     r
0     r
Name: clase, dtype: object
```

11.5. Creación del Clasificador LDA

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
clasificador = LinearDiscriminantAnalysis(solver="svd", store_covariance=True)
```

11.6. Ajuste

```
clasificador.fit(X_ce, y_ce)
```

```
LinearDiscriminantAnalysis(store_covariance=True)
```

```
X_cp
```

```
      a    b
31  169  166
20  167  156
16  176  159
30  178  165
22  160  155
15  173  157
10  172  151
2   170  143
11  175  152
29  171  165
27  168  160
```

11.7. Predicción

```
y_pred = clasificador.predict(X_cp)
```

```
y_pred
```

```
array(['n', 'n', 'r', 'n', 'n', 'r', 'r', 'r', 'r', 'n', 'n'], dtype='<U1')
```

```
y_cp
```

```
31    n
20    n
16    r
30    n
22    n
15    r
10    r
2     r
11    r
29    n
27    n
Name: clase, dtype: object
```

11.8. Creación de los resultados estadísticos de la clasificación

```
from sklearn.metrics import confusion_matrix
```

```
mconf = confusion_matrix(y_cp, y_pred)
```

```
mconf
```

```
array([[6, 0],
       [0, 5]])
```

```
clasificador.score(X_cp, y_cp)
```

```
1.0
```

```
from sklearn.metrics import accuracy_score
```

```
cc = accuracy_score(y_cp, y_pred)
```

```
cc
```

```
1.0
```

11.9. Preparación del gráfico

```
import matplotlib.pyplot as plt
```

```
from matplotlib.colors import ListedColormap
```

```
print(y_ce)
```

```
35    n
33    n
28    n
32    n
8     r
13    r
5     r
17    r
14    r
7     r
26    n
```

```
1     r
12    r
25    n
24    n
6     r
23    n
4     r
18    n
21    n
19    n
9     r
34    n
3     r
0     r
Name: clase, dtype: object
```

```
y_ce.size
```

```
25
```

11.10. Ajuste del etiquetado de la variable y

```
from sklearn.preprocessing import LabelEncoder
```

```
labelencoder_y = LabelEncoder()
```

```
y_ce = labelencoder_y.fit_transform(y_ce)
```

```
print(y_ce)
```

```
[0 0 0 0 1 1 1 1 1 1 0 1 1 0 0 1 0 1 0 0 0 1 0 1 1]
```

Nota: Es necesario realizar el ajuste de nuevo dado que cambió la variable y debido al proceso de etiquetado

```
clasificador.fit(X_ce, y_ce)
```

```
LinearDiscriminantAnalysis(store_covariance=True)
```

```
X_set, y_set = X_ce, y_ce
```

```
X_set
```

```
      a    b
35  160  143
33  174  168
28  172  164
32  165  168
8   170  150
13  168  154
5   167  147
17  175  162
14  170  156
7   169  149
26  172  163
1   165  143
12  164  153
```

```
25  179  162
24  177  161
6   174  147
23  165  150
4   174  145
18  165  151
21  171  156
19  157  153
9   164  151
34  173  169
3   172  145
0   168  141
```

```
y_set
```

```
rray([0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1])
```

Creación de la malla (plano cartesiano)

```
X1, X2 = np.meshgrid(
    np.arange(start = X_set.iloc[:,0].min()-1, stop = X_set.iloc[:,0].max()+1, step=0.1),
    np.arange(start = X_set.iloc[:,1].min()-1, stop = X_set.iloc[:,1].max()+1, step=0.1)
)
```

11.11. Creación del gráfico

```
plt.contourf(X1, X2,
    clasificador.predict(
        np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
        alpha = 0.75, cmap = ListedColormap(('orange', 'red'))
)

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())

j=0
for i in y_set:
    if i==0:
        color = "orange"
    else:
        color = "red"
    plt.scatter(
        X_set.iloc[j,0],
        X_set.iloc[j,1],
        c = color,
        label = i
    )
    j=j+1

plt.title('LDA (Conjunto de entrenamiento)')
plt.xlabel('a')
plt.ylabel('b')
plt.show()
```

```
/usr/lib/python3/dist-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
  warnings.warn(
```

Es importante notar que los puntos que no se encuentran clasificados correctamente en la gráfica por el LDA, corresponden a aquellos datos del conjunto de entrenamiento que tenían una etiqueta en esa región