

Python



Capítulo 1: Historia del lenguaje

Fue creado por Guido Van Rossum, un programador holandés, a finales de la década de 1980 y principios de la de 1990.

En ese periodo, Guido Van Rossum estaba involucrado con un sistema operativo llamado Amoeba, el cual experimentaba dificultades de integración con la interfaz de usuario denominada Bourne Shell. Dada la incompatibilidad entre ambos, decidió abordar el problema mediante la creación de un lenguaje de programación que facilitara una comunicación más fluida entre ellos.

El origen del nombre del lenguaje se debe al aprecio de Guido Van Rossum por el grupo cómico Monty Python.

Algunas de sus características más destacadas incluyen:

- 1-. Es un **lenguaje de alto nivel** con una **gramática simple, clara y altamente legible**. Python prescinde de muchos de los símbolos y convenciones presentes en otros lenguajes de programación, lo que contribuye a su accesibilidad y facilidad de comprensión.
- 2-. También presenta un **tipado dinámico**, lo que significa que el tipo de variable se establece de manera dinámica o durante la ejecución del programa, y un **tipado fuerte**, donde se realiza una distinción clara entre los diversos tipos que una variable puede tener.
- 3-. Es un **lenguaje orientado a objetos** que incorpora conceptos como la sobrecarga de constructores, la herencia múltiple, la encapsulación, las interfaces y el polimorfismo.
- 4-. Además, destaca por ser de **código abierto** (Open Source), lo que significa que su código fuente está disponible públicamente para su estudio, modificación y distribución.
- 5-. Cuenta con una **librería estándar sumamente extensa**, lo que implica que se incluyen de manera predeterminada numerosas clases, bibliotecas, y demás recursos que facilitan la ejecución de diversas tareas en este lenguaje de programación.
- 6-. Se caracteriza por ser un **lenguaje interpretado**, lo que significa que las instrucciones del programa son ejecutadas línea por línea en tiempo real.
- 7-. Destaca por su **versatilidad**, ya que es capaz de ser utilizado en la creación de una amplia gama de aplicaciones, abarcando desde aplicaciones de escritorio y servidores hasta aplicaciones web.
- 8-. Es **multiplataforma**, lo que significa que los programas desarrollados en Python pueden ejecutarse indistintamente en sistemas operativos como Windows, Linux, Mac, y diversas plataformas, proporcionando una portabilidad considerable y flexibilidad en la implementación de software.

Nota: En el caso de tener que declarar la variable antes de ejecutar el programa, se trata de un tipado estático.

Capítulo 2: Sintaxis básica

El símbolo `>>>`, conocido como "**prompt**", señala el punto preciso en la consola (dentro del entorno IDLE) donde ingresaremos nuestras instrucciones, brindando una indicación visual clara y distintiva para la interacción del usuario con el entorno de desarrollo.

La **instrucción print** en Python se utiliza para imprimir o mostrar información en la consola o en la salida estándar. Permite visualizar el valor de variables, mensajes, o cualquier expresión que desees mostrar.

```
>>> print("Hola mundo") —▶ Hola mundo
```

En Python, generalmente, **cada instrucción debe ocupar una línea única**. Aunque es posible, se desaconseja la práctica de introducir varias instrucciones en la misma línea mediante el uso de un punto y coma, como comúnmente se hace en otros lenguajes de programación.

Esta práctica se desaconseja por dos razones principales:

- 1-. Primero, afecta la legibilidad del código al tener múltiples instrucciones en una línea, lo cual puede dificultar la lectura y la identificación de errores durante la depuración.
- 2-. Segundo, reintroducir elementos, como el punto y coma, que se habían eliminado en Python puede generar confusiones, especialmente para aquellos que están comenzando a programar.

```
>>> print("Hola alumnos") ; print("Mundo cruel") —▶ Hola alumnos
                                         Mundo cruel
```

Los **comentarios** son porciones de texto que se incluyen en el código fuente con el propósito de proporcionar información adicional, explicaciones o anotaciones que no afectan la ejecución del programa. Estos comentarios son ignorados por el compilador o intérprete y sirven exclusivamente para la comprensión del código por parte de los programadores.

En Python, los comentarios se crean precediendo el texto con el símbolo de numeral (`#`).

```
>>> # Este es un comentario
```

Nota: Para poner el símbolo del numeral teclea `Alt + 35`

En Python, tenemos la opción de dividir una instrucción en varias líneas utilizando el símbolo de **barra invertida** (`\`). Al hacerlo, logramos que una misma instrucción se extienda a lo largo de varias líneas, facilitando la lectura y organización del código.

```
>>> mi_nombre = "Hola, mi nombre es \
Juan"
```

Nota: Para poner el símbolo de la diagonal teclea Alt + 92

Una práctica esencial en la programación es la **indentación**, una especie de tabulación utilizada para diferenciar secciones de código y estructuras. Cada bloque de código ya sea una declaración o una función, se desplaza ligeramente hacia la derecha, creando un espacio o sangría.

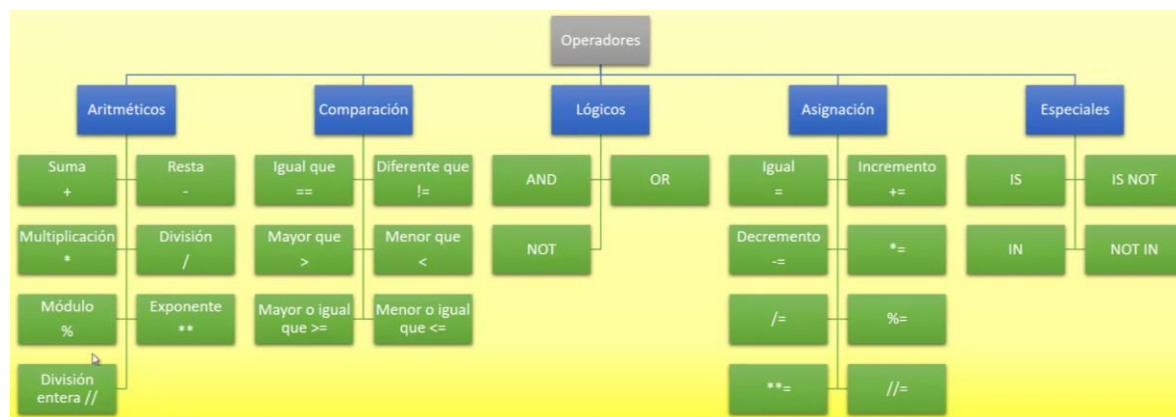
Este enfoque resulta particularmente beneficioso al leer y comprender el código, ya que la indentación proporciona una clara indicación de la relación jerárquica entre las líneas de código. Esto facilita la rápida identificación de qué líneas pertenecen a un bloque específico, mejorando así la legibilidad y la comprensión del código.

Capítulo 3: Tipos de datos, operadores y variables

Tipos de datos:

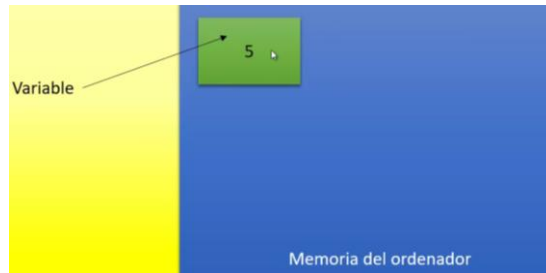


Tipos de operadores:



Una **variable** es un espacio en la memoria del ordenador (con un nombre simbólico, llamado identificador) destinada a almacenar un valor que puede alterarse durante la ejecución del programa.

Un **identificador** se refiere al nombre dado a una variable, función, clase u otro elemento del código fuente. Es una secuencia de caracteres (letras, números y/o guiones bajos) que sirve para identificar y diferenciar ese elemento dentro del programa.



```
>>> nombre = 5
```

En este ejemplo, el tipo de la variable se determina por su contenido (en este caso, el valor 5), a diferencia de otros lenguajes donde el tipo puede ser explícitamente declarado al crear la variable.

La **función type()** se utiliza para obtener el tipo de un objeto o variable. Al pasar un objeto como argumento a la función `type()`, esta devuelve el tipo de datos al que pertenece ese objeto.

```
>>> nombre = 5
```

```
>>> type(nombre) —> <class 'int'>
```

En Python, un texto puede llevar una **triple comilla**. Eso nos va a ser útil cuando queramos incluir saltos de línea dentro de un texto.

```
>>> mensaje = """ Esto es un mensaje
```

```
con tres saltos
```

```
de linea """
```

```
>>> print(mensaje) —>
```

```
Esto es un mensaje
```

```
con tres saltos
```

```
de linea
```



Capítulo 4: Funciones

Una **función** es un bloque de código que realiza una tarea específica y puede ser ejecutado o invocado desde otras partes del programa. Las funciones permiten dividir un programa en secciones más pequeñas y modulares, lo que facilita la organización, la reutilización y el mantenimiento del código.

En términos más simples, una función toma ciertos datos como entrada (llamados argumentos o parámetros), realiza una operación o serie de operaciones, y luego devuelve un resultado o realiza alguna acción.

Nota: A las funciones también se las denomina “métodos” cuando se encuentran definidas dentro de una clase.

La sintaxis básica para definir una función es la siguiente:

```
def nombre_de_la_función(parámetro1, parámetro2, ...):  ← Declaración de la función
instrucciones  ← Cuerpo de la función
return  ← Opcional
nombre_de_la_función(parámetros)  ← Llamada a la función (Ejecución)
```

Las **funciones predefinidas** (incorporadas):

- Son funciones que ya están integradas en el lenguaje de programación
- Están disponibles para ser utilizadas sin necesidad de definir las previamente.
- Ejemplos en Python incluyen funciones como `print()`, `len()`, `type()`, entre otras.

Las **funciones propias** (definidas por el usuario):

- Son funciones creadas por el programador durante el desarrollo del programa.
- Se definen para realizar tareas específicas y modularizar el código, facilitando su reutilización y mantenimiento.
- Los nombres y la lógica de estas funciones son determinados por el programador.

Ejemplo 1 – Función sin parámetros:

```
def mensaje():
```

```
    print("Estamos aprendiendo Python")
    print("Estamos aprendiendo instrucciones básicas")
    print("Poco a poco iremos avanzando")
```

```
mensaje() →
```

Estamos aprendiendo Python

Estamos aprendiendo instrucciones básicas

Poco a poco iremos avanzando

Ejemplo 2 – Función sin parámetros:

```
def suma():
```

```
    num1 = 5
    num2 = 7
```

```
print(num1+num2)
suma() → 12
```

Ejemplo 1 – Función con parámetros:

```
def suma(num1, num2):
    print(num1+num2)
suma(5,7)
suma(2,3) →
```

12
5

Ejemplo 2 – Función con parámetros:

```
def suma(num1, num2):
    resultado = num1 + num2
    return resultado
print(suma(5,7))
print(suma(2,3)) →
```

12
5

La **instrucción return** se utiliza para finalizar la ejecución de una función y devolver un valor al lugar desde el cual fue llamada. Cuando una función alcanza una declaración return, la ejecución de la función se detiene y el valor especificado después de return se devuelve como resultado de la función.

Nota: Los argumentos o parámetros de una función pueden ser de diferente tipo.

Ejemplo 3 – Función con parámetros:

```
def suma(num1, num2):
    resultado = num1 + num2
    return resultado
almacena_resultado = suma(5,8)
print(almacena_resultado) → 13
```

Capítulo 5: Listas

Una **lista** es una estructura de datos que nos permite almacenar una colección ordenada de elementos. Estos elementos pueden ser de diferentes tipos, como números, cadenas de texto, booleanos, u otros objetos.

La sintaxis básica de las listas es:

```
nombre_de_la_lista = [ elem1, elem2, elem3, ... ]
```

↓ ↓ ↓
0 1 2

Un **índice** se refiere a un valor numérico que se utiliza para identificar la posición de un elemento dentro de una estructura de datos, como una lista, una cadena de texto, o un arreglo.

Operaciones básicas de las listas:

miLista=["María", "Pepe", "Marta", "Antonio"] ← **Lista de referencia**

1-. Imprimir el contenido de una lista

print(miLista[:]) → ['María', 'Pepe', 'Marta', 'Antonio']

2-. Acceder a un elemento en concreto

print(miLista[2]) → Marta

3-. Cuando introducimos índices negativos, Python invertirá los índices

print(miLista[-2]) → Marta

4-. Acceder a una porción de lista

print(miLista[0:3]) o print(miLista[:3]) → ['María', 'Pepe', 'Marta']

print(miLista[1:3]) → ['Pepe', 'Marta']

print(miLista[2:]) → ['Marta', 'Antonio']

5-. Agregar elementos al final de la lista

miLista.append("Sandra") → ['María', 'Pepe', 'Marta', 'Antonio', 'Sandra']

6-. Agregar elementos en un punto intermedio

miLista.insert(2,"Sandra") → ['María', 'Pepe', 'Sandra', 'Marta', 'Antonio']

7-. Agregar varios elementos al final de la lista

miLista.extend(["Sandra", "Ana"]) → ['María', 'Pepe', 'Marta', 'Antonio', 'Sandra', 'Ana']

8-. Devolver el índice de un elemento

print(miLista.index("Antonio")) → 3

9-. Comprobar si un elemento se encuentra o no se encuentra en una lista

```
print("Pepe" in miLista) —> TRUE
```

10-. Eliminar elementos

```
miLista.remove("Marta") —> ['María', 'Pepe', 'Antonio']
```

11-. Eliminar el último elemento

```
miLista.pop() —> ['María', 'Pepe', 'Marta']
```

12-. El operador suma concatena una o varias listas

```
miLista2=[5, True, 79.35]
```

```
miLista3=miLista+miLista2 —> ['María', 'Pepe', 'Marta', 'Antonio', 5, True, 79.35]
```

13-. El operador producto repite “n” veces el contenido de una lista

```
miLista=["María", "Pepe", "Marta", "Antonio"] * 3 —> ['María', 'Pepe', 'Marta', 'Antonio',  
'María', 'Pepe', 'Marta', 'Antonio', 'María', 'Pepe', 'Marta', 'Antonio']
```

Nota: En una lista puede haber varios elementos repetidos

Nota: Para poner los corchetes teclea Alt 91 y 93

Capítulo 6: Tuplas

Una **tupla** es una estructura de datos similar a una lista, pero con la diferencia clave de que las tuplas son inmutables. Esto significa que, una vez creada una tupla, no se pueden modificar sus elementos ni agregar nuevos elementos ni eliminar existentes.

La sintaxis básica de las tuplas es:

```
nombre_de_la_tupla = ( elem1, elem2, elem3, ...)
```

Operaciones básicas de las tuplas:

```
miTupla=("Juan", 13, 1, 1995) ◀ Tupla de referencia
```

1-. Imprimir el contenido de una tupla

```
print(miTupla[:]) —> ('Juan', 13, 1, 1995)
```

2-. Acceder a un elemento en concreto

```
print(miTupla[2]) —> 1
```

3-. Devolver el índice de un elemento

```
print(miTupla.index(1)) —> 2
```


4-. De tuplas a listas

```
miLista=list(miTupla) —> ['Juan', 13, 1, 1995]
```

5-. De listas a tuplas

```
miTupla=tuple(miLista) —> ('Juan', 13, 1, 1995)
```

6-. Comprobar si un elemento se encuentra o no se encuentra en una lista

```
print("Juan" in miTupla) —> TRUE
```

7-. Saber cuántas veces se repite un elemento

```
print(miTupla.count(13)) —> 1
```

8-. Saber la longitud de una tupla

```
print(len(miTupla)) —> 4
```

Una **tupla unitaria** es una tupla que contiene exactamente un elemento. En Python, se utiliza una coma después del único elemento, incluso si no se utilizan un paréntesis:

```
miTupla=("Juan",)
```

El **empaquetado de tuplas** ocurre cuando se agrupan varios valores en una única tupla. Esto se hace simplemente separando los valores por comas, sin necesidad de utilizar paréntesis.

```
miTupla="Juan", 13, 1, 1995
```

El **desempaquetado de tuplas ocurre** cuando se asignan los elementos individuales de una tupla a variables separadas. Esto se hace asignando la tupla a variables, el número de variables debe coincidir con el número de elementos en la tupla.

```
miTupla=("Juan", 13, 1, 1995)
```

```
nombre, día, mes, año = miTupla
```

```
print(nombre)
```

```
print(día)
```

```
print(mes)
```

```
print(año) —>
```

```
Juan
```

```
13
```

```
1
```

```
1995
```

Nota: Para poner los paréntesis teclea Alt 40 y 41

Capítulo 7: Diccionarios

Un **diccionario** es una estructura de datos que permite almacenar y organizar datos de manera flexible, asociando cada valor con una clave única. En lugar de utilizar índices numéricos, como en las listas y tuplas, los diccionarios utilizan claves para acceder y recuperar valores. Cada par clave-valor en un diccionario representa una entrada individual.

La sintaxis básica de los diccionarios es:

```
nombre_del_diccionario = { elem1, elem2, elem3, ... }
```

Operaciones básicas de los diccionarios:

```
miDiccionario = {"Alemania": "Berlín", "Francia": "París", "Reino Unido": "Londres"} ←
```

Diccionario de referencia

1-. Imprimir el contenido de un diccionario

```
print(miDiccionario) → {'Alemania': 'Berlín', 'Francia': 'París', 'Reino Unido': 'Londres'}
```

2-. Acceder a un elemento en concreto

```
print(miDiccionario["Francia"]) → París
```

3-. Agregar elementos al final de la lista

```
miDiccionario["Italia"]="Roma" → {'Alemania': 'Berlín', 'Francia': 'París', 'Reino Unido': 'Londres', 'Italia': 'Roma'}
```

4-. Eliminar elementos

```
del miDiccionario["Reino Unido"] → {'Alemania': 'Berlín', 'Francia': 'París'}
```

5-. Saber la longitud de un diccionario

```
print(len(miDiccionario)) → 5
```

6-. Asignarle a cada clave de la tupla, un valor del diccionario

```
miTupla = ["España", "Francia", "Reino Unido", "Alemania"]
```

```
miDiccionario = {miTupla[0]: "Madrid", miTupla[1]: "París", miTupla[2]: "Londres",  
miTupla[3]: "Berlín"} →
```

```
{'España': 'Madrid', 'Francia': 'París', 'Reino Unido': 'Londres', 'Alemania': 'Berlín'}
```

7-. Almacenar una tupla en un diccionario

```
miDiccionario = {23: "Jordan", "Nombre": "Michael", "Equipo": "Chicago",  
"Anillos": [1991, 1992, 1993, 1996]}
```

8-. Almacenar un diccionario en un diccionario

```
miDiccionario={23:"Jordan", "Nombre":"Michael", "Equipo":"Chicago",  
"Anillos":{"temporadas":[1991,1992,1993,1996]}}
```

9-. Saber cuáles son las claves de nuestro diccionario

```
print(miDiccionario.keys()) —> dict_keys(['Alemania', 'Francia', 'Reino Unido'])
```

10-. Saber cuáles son los valores de nuestro diccionario

```
print(miDiccionario.values()) —> dict_values(['Berlín', 'París', 'Londres'])
```

Nota: Si introducimos un valor erróneo en algún punto del código, podemos corregirlo reescribiendo la clave-valor correspondiente.

Nota: Para poner los paréntesis teclea Alt 123 y 125

Capítulo 8: Estructuras de control (Condicionales)

Las **estructuras de control** son bloques de código que permiten controlar el flujo de ejecución de un programa. Estas estructuras determinan el orden en que las instrucciones son ejecutadas, lo que es fundamental para crear programas con comportamientos específicos y para tomar decisiones basadas en ciertas condiciones.

Hay tres tipos básicos de estructuras de control:

Secuenciales: Las instrucciones se ejecutan en secuencia, una tras otra, en el orden en que aparecen en el código.

Condicionales: Permiten ejecutar cierto bloque de código si se cumple una condición específica.

Bucles (o Loops): Permiten repetir un bloque de código varias veces, mientras se cumpla una determinada condición.

Tipos de condicionales en Python:

1-. El condicional **if** ejecuta un bloque de código si una condición específica es verdadera.

La sintaxis básica del condicional es:

if condición:

```
    # Bloque de código a ejecutar si la condición es verdadera
```

Ejemplo 1:

```
def evaluacion(nota):
```

```
    valoracion = "aprobado"
```

```
    if nota < 5:
```

```

        valoracion = "reprobado"

    return valoracion

print(evaluacion(4)) → reprobado

```

La función **input()** se emplea para solicitar al usuario que ingrese datos desde el teclado mientras el programa se está ejecutando. La función espera a que el usuario ingrese alguna entrada desde el teclado y luego devuelve una cadena que representa lo que el usuario ha ingresado.

La función **int()** transforma en un entero cualquier cosa que tenga sentido como número entero. Sin embargo, si intentas convertir en entero algo que no tenga sentido como número entero, como una cadena que contenga letras o caracteres especiales, o un objeto que no sea compatible con la conversión a entero, se generará un error.

Ejemplo 2:

```

print("Programa de evaluacion de notas de alumnos")

nota_alumno = input()

def evaluacion(nota):
    valoracion = "aprobado"
    if nota < 6:
        valoracion = "reprobado"
    return valoracion

print(evaluacion(int(nota_alumno))) → Programa de evaluacion de notas de alumnos
5
Reprobado

```

El **ámbito de una variable** se refiere a la región del código donde esa variable es válida y puede ser utilizada. En Python, el ámbito de una variable está determinado por dónde se declara la variable.

Por ejemplo, si una variable se declara dentro de una función, su **ámbito** es **local** a esa función, lo que significa que solo puede ser accedida desde dentro de esa función. Si una variable se declara fuera de cualquier función o estructura de control, su **ámbito** es **global**, lo que significa que puede ser accedida desde cualquier parte del código.

2-. El condicional **if** también puede ir seguido de un bloque **else** para manejar el caso en que la condición no se cumple.

La sintaxis básica del condicional es:

if condición:

```
# Bloque de código a ejecutar si la condición es verdadera
else:
```

```
# Bloque de código a ejecutar si la condición es falsa
```

Ejemplo 1:

```
print("Verificación de acceso")
edad_usuario = int(input("Introduce tu edad, por favor: "))
if edad_usuario < 18:
    print("No puede pasar")
else:
    print("Puede pasar") → Verificación de acceso
```

Introduce tu edad, por favor: 34

Puede pasar

3-. También se puede utilizar la estructura **else if o elif** para manejar múltiples condiciones.

Ejemplo 1:

```
print("Verificación de acceso")
edad_usuario = int(input("Introduce tu edad, por favor: "))
if edad_usuario < 18:
    print("No puede pasar")
elif edad_usuario > 100:
    print("Edad incorrecta")
else:
    print("Puede pasar") → Verificación de acceso
```

Introduce tu edad, por favor: 102

Edad incorrecta

Ejemplo 2:

```
print("Verificación de acceso")
nota_alumno = int(input("Introduce tu nota, por favor: "))
if nota_alumno < 5:
    print("Insuficiente")
```

```

elif nota_alumno < 6:
    print("Reprobado")
elif nota_alumno < 7:
    print("Bien")
elif nota_alumno < 9:
    print("Muy bien")
else:
    print("Sobresaliente") —> Verificación de acceso

```

Introduce tu nota, por favor: 8

Muy bien

La **concatenación de operadores de comparación**, también conocida como encadenamiento de operadores de comparación, es una técnica en la programación que te permite realizar múltiples comparaciones en una sola expresión. Esto es útil cuando necesitas evaluar varias condiciones al mismo tiempo.

Ejemplo 1:

```

edad = 7
if 0 < edad < 100:
    print("La edad es correcta")
else:
    print("La edad es incorrecta") —> La edad es correcta

```

La **función str()** se usa para convertir un objeto en su representación de cadena. Esta función toma un objeto como argumento y devuelve una cadena que representa ese objeto.

Ejemplo 2:

```

salario_presidente = int(input("Introduce salario presidente: "))
print("Salario presidente: " + str(salario_presidente))

salario_director = int(input("Introduce salario director: "))
print("Salario director: " + str(salario_director))

salario_jefe_area = int(input("Introduce salario jefe área: "))
print("Salario jefe_area: " + str(salario_jefe_area))

salario_administrativo = int(input("Introduce salario administrativo: "))

```

```

print("Salario administrativo: " + str(salario_administrativo))
if salario_administrativo < salario_jefe_area < salario_director < salario_presidente:
    print("Todo funciona correctamente")
else:
    print("Algo falla en esta empresa") →

```

Introduce salario presidente: 3500

Salario presidente: 3500

Introduce salario director: 2500

Salario director: 2500

Introduce salario jefe área: 2000

Salario jefe_area: 2000

Introduce salario administrativo: 1500

Salario administrativo: 1500

Todo funciona correctamente

Los **operadores lógicos** son herramientas fundamentales que te permiten combinar expresiones de comparación para evaluar condiciones más complejas. Sirven para controlar el flujo del programa y tomar decisiones basadas en múltiples condiciones.

Ejemplo 3:

```

print("Programa de becas año 2017")
distancia_escuela = int(input("Introduce la distancia a la escuela en Km: "))
numero_hermanos = int(input("Introduce el número de hermanos: "))
salario_familiar = int(input("Introduce el salario anual bruto: "))
if distancia_escuela > 40 and numero_hermanos > 2 or salario_familiar <= 20000:
    print("Tienes derecho a una beca")
else:
    print("No tienes derecho a una beca") →

```

Programa de becas año 2017

Introduce la distancia a la escuela en Km: 39

Introduce el número de hermanos: 1

Introduce el salario anual bruto: 15000

Tienes derecho a una beca

El **operador and** produce un resultado verdadero cuando todas las condiciones son verdaderas, mientras que el **operador or** da un resultado verdadero si al menos una de las condiciones es verdadera.

Ejemplo 4:

```
print("Optativas año 2017")  
  
print("Materias: Informática grafica - Pruebas de software - Usabilidad y accesibilidad")  
  
opción = input("Escribe la asignatura escogida: ")  
  
asignatura = opción.lower()  
  
if asignatura in ("informática grafica", "pruebas de software", "usabilidad y accesibilidad"):  
    print("Asignatura elegida: " + asignatura)  
else:  
    print("La asignatura escogida no está contemplada") —> Optativas año 2017
```

Materias: Informática grafica - Pruebas de software - Usabilidad y accesibilidad

Escribe la asignatura escogida: PrUebas de SOftware

Asignatura elegida: pruebas de software

El **operador in** sirve para comprobar si un valor está presente en una secuencia, como una lista, una tupla, un diccionario o una cadena de caracteres.

Python es **case sensitive**, lo que significa que distingue entre mayúsculas y minúsculas en los nombres de variables, funciones, clases y otros identificadores.

Los métodos **lower()** y **upper()** transforman todos los caracteres de una cadena a minúsculas y mayúsculas, respectivamente, y retornan la cadena resultante.




Capítulo 9: Estructuras de control (Bucles)

Los bucles se pueden clasificar en dos categorías principales: bucles determinados e indeterminados.

En los **bucles determinados**, el número de iteraciones o repeticiones está predeterminado y conocido antes de que comience el bucle (se utilizan principalmente cuando sabemos exactamente cuántas veces queremos que se ejecute el bloque de código).


La sintaxis básica del bucle for es:

for elemento in secuencia:  **Declaración del bucle**

Código a ejecutar para cada elemento  **Cuerpo del bucle**

Ejemplo 1:

```
for i in [1,2,3]:
```

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

Hola

Hola

Nota: El bloque de código indentado debajo del bucle for se ejecutará en cada iteración del bucle. En este caso, simplemente imprime "Hola" en cada iteración, por lo que veremos tres veces la palabra "Hola" impresa en la consola, una vez por cada elemento de la lista.

Ejemplo 2:

```
for i in ["primavera", "verano", "otoño", "invierno"]:
```

```
    print(i)  primavera
```

verano

otoño

invierno

Nota: En cada iteración del bucle, se imprime el valor de i, que corresponde a una estación del año.

Ejemplo 3:

```
for i in "sebas@gmail.com":
```

```
    print("Hola", end = " ")  Hola Hola Hola Hola Hola Hola Hola Hola Hola  
Hola Hola Hola Hola Hola
```

Nota: En este caso, el bloque simplemente imprime la palabra "Hola", pero en lugar de imprimir en una nueva línea después de cada impresión, utiliza el parámetro `end = " "` para que la salida se mantenga en la misma línea y se separen con un espacio. Por lo tanto, verás la palabra "Hola" impresa tantas veces como caracteres tenga la cadena "sebas@gmail.com", separados por espacios.

Ejemplo 4:

```
mi_email = input("Introduce tu dirección de email: ")
```

```
for i in mi_email:
```

```
    if(i == "@"):
```

```
        email = True
```

```

if email == True:
    print("Email correcto")
else:
    print("Email incorrecto") —> Introduce tu dirección de email: sebasolvil@gmail.com

```

Email correcto

Nota: Cuando estamos evaluando una variable booleana en un condicional, podemos simplificar el código eliminando la comparación `== True`.

En programación, un **contador** es una variable que se utiliza para realizar un seguimiento del número de veces que ocurre cierto evento o acción en un programa. Los contadores son comunes en situaciones donde necesitas realizar un recuento o un seguimiento de la frecuencia de ocurrencia de algo.

Ejemplo 5:

```

contador = 0

mi_email = input("Introduce tu dirección de email: ")

for i in mi_email:
    if(i == "@" or i == "."):
        contador = contador+1

if contador == 2:
    print("Email correcto")
else:
    print("Email incorrecto") —> Introduce tu dirección de email: sebasolvil@gmail.com

```

Email correcto

Nota: En este código, contador se incrementa cada vez que el carácter actual en la dirección de correo electrónico (`mi_email`) es “@” o “.”. Luego, se verifica si el contador es igual a 2, ya que se espera que una dirección de correo electrónico tenga exactamente un carácter “@” y un carácter “.”. Si el contador es igual a 2, se considera que la dirección de correo electrónico es correcta; de lo contrario, se considera incorrecta.

El **tipo range** en Python es un tipo de dato que representa una secuencia inmutable de números enteros. Se utiliza comúnmente con bucles for para iterar sobre una secuencia de números en un rango específico.

La sintaxis básica es:

`range(inicio, fin, paso)` —> Donde “inicio” es el número inicial, “fin” es el número final y “paso” es el tamaño del paso entre los números en el rango

Ejemplo 6:

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

```
    print("Hola", end=" ") → Hola Hola Hola Hola Hola
```

Ejemplo 7:

```
for i in range(2):
```

```
    print(f"valor de la variable {i}") → valor de la variable 0
```

```
valor de la variable 1
```

Nota: Este bucle imprimirá la frase "valor de la variable" seguida del valor de la variable *i* en cada iteración del bucle. Esto es debido a las **f-strings o cadenas formateadas**, las cuales facilitan una interpolación de cadenas más simple y legible.

Ejemplo 7:

```
for i in range(5,20,5):
```

```
    print(f"valor de la variable {i}") → valor de la variable 5
```

```
valor de la variable 10
```

```
valor de la variable 15
```

La **función len()** sirve para obtener la longitud de un objeto.

Ejemplo 8:

```
valido = False
```

```
email = input("Introduce tu email: ")
```

```
for i in range(len(email)):
```

```
    if email[i] == "@":
```

```
        valido = True
```

```
if valido:
```

```
    print("Email correcto")
```

```
else:
```

```
    print("Email incorrecto") → Introduce tu email: sebas@gmail.com
```

```
Email correcto
```


Si email de 0 es igual a @, etc

En los **bucles indeterminados**, el número de iteraciones no está predeterminado y puede variar durante la ejecución del programa (se utilizan cuando no conocemos de antemano

cuántas veces se ejecutará el bucle, y el bucle continúa hasta que se cumple una condición de salida).

La sintaxis básica del bucle while es:

while condicion:  **Declaración del bucle**

 # Código a ejecutar mientras la condición sea verdadera  **Cuerpo del bucle**

 # Se ejecuta repetidamente hasta que la condición se evalúe como False


Ejemplo 1:

```
i = 1
```

```
while i <= 3:
```

```
    print("Ejecución " + str(i))
```

```
    i = i+1
```

```
print("Termino la ejecución")  Ejecución 1
```

Ejecución 2

Ejecución 3

Termino la ejecución

Nota: Este ejemplo ilustra un bucle while de forma sencilla. Sin embargo, no estamos explorando su naturaleza indeterminada, ya que antes de la ejecución del programa sabemos con certeza que el código dentro del bucle se ejecutará exactamente 10 veces.

Ejemplo 2:


```
edad = int(input("Introduce tu edad por favor: "))
```

```
while edad < 5 or edad > 100:
```

```
    print("Has introducido una edad incorrecta. Vuelve a intentarlo")
```

```
    edad = int(input("Introduce tu edad por favor: "))
```

```
print("Gracias por su atención")
```

```
print("Edad del aspirante " + str(edad)) 
```

Introduce tu edad por favor: -45

Has introducido una edad negativa. Vuelve a intentarlo

Introduce tu edad por favor: 23

Gracias por su atención

Edad del aspirante 23

La **instrucción break** se usa dentro de los bucles para salir inmediatamente del bucle en el que se encuentra, incluso si la condición de control del bucle aún no es falsa. Cuando se ejecuta la instrucción break, la ejecución del programa sale del bucle y continúa con la siguiente línea de código después del bucle.

Ejemplo 3:

```
import math

print("Programa de cálculo de raíz cuadrada")

numero = int(input("Introduce un numero: "))

intentos = 0

while numero < 0:

    print("No se puede hallar la raíz")

    if intentos == 2:

        print("Has consumido demasiados intentos")

        break;

    numero = int(input("Introduce un numero: "))

    if numero < 0:

        intentos = intentos+1

if intentos < 2:

    solucion = math.sqrt(numero)

    print("La raíz cuadrada de " + str(numero) + " es " + str(solucion)) ➡
```

Programa de cálculo de raíz cuadrada

Introduce un numero: -7

No se puede hallar la raíz

Introduce un numero: -9

No se puede hallar la raíz

Introduce un numero: -11

No se puede hallar la raíz

Has consumido demasiados intentos

En Python, un **módulo** es simplemente un archivo que contiene definiciones y declaraciones. Estas definiciones pueden incluir funciones, variables y clases, así como declaraciones de importación de otros módulos.

También permiten organizar el código en archivos separados y reutilizarlo en diferentes partes de un programa, o en diferentes programas. Esto promueve la modularidad y la reutilización del código, lo que hace que el desarrollo de software sea más eficiente y mantenible.

Para utilizar las definiciones y declaraciones de un módulo en un programa, podemos importarlo utilizando la palabra clave **import**.

Por ejemplo: `import math` (esto importa el módulo `math`, que contiene muchas funciones y constantes matemáticas útiles).

Además, puedes importar partes específicas de un módulo utilizando la instrucción **from import**.

Por ejemplo: `from math import sqrt` (esto importa solo la función `sqrt` del módulo `math`, lo que nos permite usarla directamente en el programa sin tener que especificar el nombre del módulo cada vez que la utilicemos).

Palabras clave utilizadas en los bucles (`continue`, `pass` y `else`):

Continue: Se aplica dentro de bucles para saltar la iteración actual y continuar con la siguiente iteración del bucle. Es útil cuando se quiere evitar ejecutar cierto código bajo ciertas condiciones, pero aún así continuar con el bucle.

Ejemplo 1:

```
for letra in "Python":
    if letra == "h":
        continue
    print("Viendo la letra: " + letra) → Viendo la letra: P
```

Viendo la letra: y

Viendo la letra: t

Viendo la letra: o

Viendo la letra: n

Ejemplo 2:

```
nombre = "Píldoras informáticas"
```

```
contador = 0
```

```
for i in nombre:
```

```
    if i == " ":
```

```

        continue
    contador+=1
print(contador) ➡ 20

```

Pass: No realiza ninguna operación. Se emplea como marcador de posición cuando la sintaxis requiere un bloque de código, pero no se desea ejecutar ninguna instrucción dentro de ese bloque. Puede ser útil durante el desarrollo de un programa para definir funciones, clases u otras estructuras sin implementarlas aún.

Ejemplo 1:

```

if x < 0:
    pass
else:
    print("x es positivo")

```

Else: Se ejecuta cuando el bucle ha completado todas sus iteraciones normales. En otras palabras, si el bucle no ha sido interrumpido por una instrucción "break", el bloque de código dentro del "else" se ejecutará al final del bucle.

Ejemplo 1:

```

email = input("Introduce tu email: ")
for i in email:
    if i == "@":
        arroba = True
        break;
else:
    arroba == False

print(arroba) ➡ Introduce tu email: sebas@gmail.com
True

```

Nota: Una vez que se encuentra el símbolo "@" y se ejecuta la instrucción break, el flujo del programa salta directamente fuera del bucle for y no se ejecuta el bloque de código dentro del else. El bloque de código dentro del else solo se ejecutaría si el bucle for se completara completamente sin encontrar el símbolo "@".

Capítulo 10: Generadores

Los **generadores** son funciones especiales que permiten generar una secuencia de valores sobre la marcha. Estos valores se producen dinámicamente y se entregan uno a la vez usando la **instrucción yield**.

Es decir, que cuando la función es llamada y se ejecuta, produce un objeto generador. Luego, se puede iterar sobre este objeto generador para obtener los valores generados uno por uno. Cada vez que se encuentra una declaración `yield` dentro de la función generadora, la ejecución de la función se detiene y el valor indicado después de `yield` se devuelve.

Luego, cuando el generador es llamado nuevamente para producir el siguiente valor, la ejecución de la función se reanuda justo después de la última declaración `yield`.



Ventajas al utilizar un generador:

1-. Eficiencia de memoria: Los generadores no almacenan todos los valores en la memoria al mismo tiempo, lo que los hace ideales para trabajar con conjuntos de datos grandes o potencialmente infinitos. Esto ayuda a reducir el uso de memoria y mejora el rendimiento del programa.

2-. Producción bajo demanda: Los generadores producen valores sobre la marcha, a medida que se solicitan, en lugar de generar todos los valores de una vez. Esto significa que los valores se generan cuando se necesitan, lo que puede ahorrar tiempo y recursos, especialmente cuando se trata de grandes conjuntos de datos.

La sintaxis básica de un generador es:

`def nombre_del_generador():` ← Declaración del generador

`instrucciones` ← Cuerpo del generador

`yield` ← Instrucción `yield`

Ejemplo de una función:

```
def GeneraPares(limite):  
    num = 1  
    miLista = []  
    while num < limite:
```

Ejemplo de un generador:

```
def GeneraPares(limite):  
    num = 1  
    while num < limite:  
        yield num*2
```


<pre>miLista.append(num*2) num = num+1 return miLista</pre>	VS	<pre>num = num+1 DevuelvePares = GeneraPares(10) for i in DevuelvePares:</pre>
<pre>print(GeneraPares(10))</pre>		<pre>print(i, end = " ")</pre>
<pre>[2, 4, 6, 8, 10, 12, 14, 16, 18]</pre>		<pre>2 4 6 8 10 12 14 16 18</pre>

La función **next()** se ejecuta para obtener el siguiente valor de un objeto generador.

Cuando se llama a **next()** en un objeto generador, la ejecución de la función generadora se reanuda desde donde se detuvo la última vez (es decir, justo después de la última declaración **yield**), y continúa hasta encontrar la próxima declaración **yield**.

El valor devuelto por esta declaración **yield** es el valor devuelto por **next()**.

Ejemplo 1:

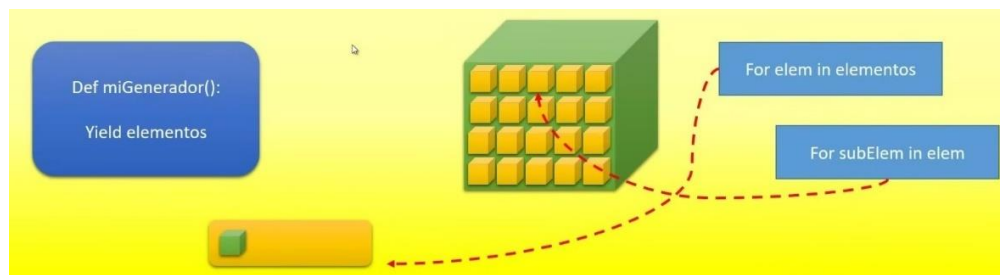
```
def GeneraPares(limite):
    num = 1
    while num < limite:
        yield num*2
        num = num+1

DevuelvePares = GeneraPares(10)
print(next(DevuelvePares))
print("Aqui podría haber más código...")
print(next(DevuelvePares))
```

2

Aqui podría haber más código...

4



La **instrucción yield from** se emplea principalmente en generadores para delegar la generación de valores a otro generador o un objeto iterable. Es una forma más concisa y legible de iterar sobre los valores de otro generador u objeto iterable y "delegar" estos valores al generador actual.

Cuando se utiliza `yield from`, el generador actual "cede" el control al generador u objeto iterable especificado después de `from`. Este último genera valores, que se pasan directamente al generador actual. Cuando el generador u objeto iterable se agota, el control vuelve al generador actual.

Ejemplo sin aplicar la instrucción:

```
def DevuelveCiudades(*ciudades):  
    for elemento in ciudades:  
        for SubElemento in elemento:  
            yield SubElemento
```

```
CiudadesDevueltas = DevuelveCiudades("Madrid", "Barcelona", "Bilbao", "Valencia")
```

```
print(next(CiudadesDevueltas))
```

```
print(next(CiudadesDevueltas)) → M
```

a

Nota: En este caso, la función `DevuelveCiudades` ahora itera sobre cada ciudad pasada como argumento y luego itera sobre cada carácter de la ciudad, utilizando `yield` para devolver cada carácter individualmente. Posteriormente, al imprimir el primer carácter de `CiudadesDevueltas`, se obtiene el primer carácter de la primera ciudad, seguido por el segundo.

Ejemplo simplificado:

```
def DevuelveCiudades(*ciudades):  
    for elemento in ciudades:  
        yield from elemento
```

```
CiudadesDevueltas = DevuelveCiudades("Madrid", "Barcelona", "Bilbao", "Valencia")
```

```
print(next(CiudadesDevueltas))
```

```
print(next(CiudadesDevueltas)) → M
```

a

Nota: Este código define una función llamada `DevuelveCiudades` que utiliza `yield from` para generar caracteres de cada cadena de texto pasada como argumento. Luego, crea un objeto generador llamado `CiudadesDevueltas` utilizando esta función, y finalmente imprime los primeros caracteres de las ciudades generadas utilizando la función `next()`.

Nota: Cuando un parámetro de una función está precedido por un asterisco (*), se le conoce como parámetro `*args` y representa una lista de argumentos de longitud variable. Esto significa que puede aceptar un número variable de argumentos posicionales cuando se llama a la función.

Capítulo 11: Excepciones

Las **excepciones** son errores imprevistos que ocurren durante la ejecución de un programa, a pesar de que la sintaxis del código sea correcta. Estos errores interrumpen el flujo normal de ejecución del programa y pueden provocar su detención si no se gestionan adecuadamente.

Cuando se produce una excepción, el programa puede detenerse y mostrar un mensaje de error en la consola. Sin embargo, es posible controlar estas situaciones utilizando la **captura o manejo de excepciones en el código**. Este proceso implica gestionar y manejar las excepciones que pueden ocurrir durante la ejecución del programa, permitiendo que se manejen los errores de forma controlada y que el programa continúe ejecutándose sin interrupciones completas.

Ejemplo 1:

```
Introduce el primer número: 8
Introduce el segundo número: 0
Introduce la operación a realizar (suma,resta,multiplica,divide): divide
Traceback (most recent call last):
  File "C:\Users\52556\Desktop\Python\Práctica_excepciones.py", line 30, in <module>
    print(divide(op1,op2))
    ~~~~~^~~~~~
  File "C:\Users\52556\Desktop\Python\Práctica_excepciones.py", line 11, in divide
    return num1/num2
           ~~~~~^~~~~~
ZeroDivisionError: division by zero
```

En Python, podemos usar **bloques try y except** para controlar las excepciones:

```
def divide(num1,num2):
    try:
        return num1/num2
    except ZeroDivisionError:
        print("No se puede dividir entre 0")
        return ("Operacion erronea")
```

- El código que puede causar una excepción se coloca dentro de un bloque try.
- Luego, se proporciona un bloque except que especifica qué hacer si ocurre una excepción dentro del bloque try (este bloque maneja la excepción de manera controlada, evitando que el programa se detenga bruscamente).

Nota: Si no ocurre ninguna excepción dentro del bloque try, el bloque except se omite y el programa continúa ejecutándose normalmente. Sin embargo, si ocurre una excepción dentro del bloque try, el flujo de control del programa se desplaza al bloque except, donde se puede manejar la excepción adecuadamente.

Ejemplo 2:

```
def divide():
    try:
        op1 = (float(input("Introduce el primer número: ")))
```

```

        op2 = (float(input("Introduce el segundo número: ")))

        print("La division es: " + str(op1/op2))

    except ValueError:

        print("El valor introducido es erróneo")

    except ZeroDivisionError:

        print("No se puede dividir entre cero")

    print("Calculo finalizado")

divide()

```

Nota: En este ejemplo, significa que el bloque de código de la función puede involucrar varias expresiones distintas o sufrir diversos fallos. Por esta razón, podemos capturar múltiples excepciones de manera consecutiva. Esto contrasta con el enfoque utilizado en el ejemplo anterior, donde solo se capturaba una excepción a la vez.

Cabe mencionar, que es posible utilizar varios bloques except de forma consecutiva para capturar distintas excepciones. Esto permite manejar múltiples tipos de errores de forma individual y personalizada en el código.

Hay otra opción que podemos considerar, que es más genérica aunque menos recomendada. Se trata de capturar algo más general. Imagina que tienes un bloque de código que podría generar muchos errores o excepciones. Aunque en este programa no sea el caso, puede que no desees utilizar bloques try y except uno detrás de otro para manejar cada posible error o excepción. En su lugar, podrías optar por un enfoque más general.

Ejemplo 3:

```

def divide():

    try:

        op1 = (float(input("Introduce el primer número: ")))

        op2 = (float(input("Introduce el segundo número: ")))

        print("La división es: " + str(op1/op2))

    except:

        print("Ha ocurrido un error")

    print("Cálculo finalizado")

divide()

```

Es poco recomendable porque, aunque deja al usuario un tanto confundido al no saber exactamente qué ha ocurrido, al menos el programa no se detiene y puede continuar su ejecución.

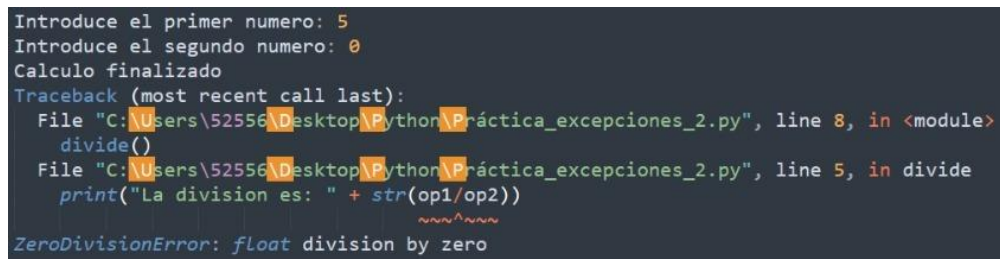
La **cláusula finally** se utiliza para definir un bloque de código que se ejecutará siempre, independientemente de si se produjo una excepción dentro de un bloque try o no.

Esta cláusula se usa comúnmente para realizar tareas de limpieza, como cerrar archivos o conexiones de red. Esto garantiza que los recursos se liberen adecuadamente y que el programa mantenga un comportamiento predecible incluso en caso de errores.

Ejemplo 4:

```
def divide():  
    try:  
        op1 = (float(input("Introduce el primer número: ")))  
        op2 = (float(input("Introduce el segundo número: ")))  
        print("La división es: " + str(op1/op2))  
    finally:  
        print("Cálculo finalizado")  
  
divide()
```

Por lo tanto, al ejecutar este programa, si hay una excepción no manejada dentro del bloque try, el código dentro del bloque finally se ejecutará después de que se haya producido la excepción, pero antes de que el programa genere un mensaje de error indicando que ha ocurrido una excepción no manejada.



```
Introduce el primer numero: 5  
Introduce el segundo numero: 0  
Cálculo finalizado  
Traceback (most recent call last):  
  File "C:\Users\52556\Desktop\Python\Práctica_excepciones_2.py", line 8, in <module>  
    divide()  
  File "C:\Users\52556\Desktop\Python\Práctica_excepciones_2.py", line 5, in divide  
    print("La division es: " + str(op1/op2))  
                                ~~~~~^~~~~  
ZeroDivisionError: float division by zero
```

Nota: La estructura try debe estar acompañada por al menos una de las siguientes cláusulas: except, finally, o ambas.

La **instrucción raise** se aplica para generar manualmente una excepción y lanzarla durante la ejecución del programa. Puede especificarse con el tipo de excepción que desees generar, o incluso con argumentos para personalizar la excepción.

Ejemplo 5:

```
def EvaluaEdad(edad):  
    if edad < 0:  
        raise TypeError("No se permiten edades negativas")
```

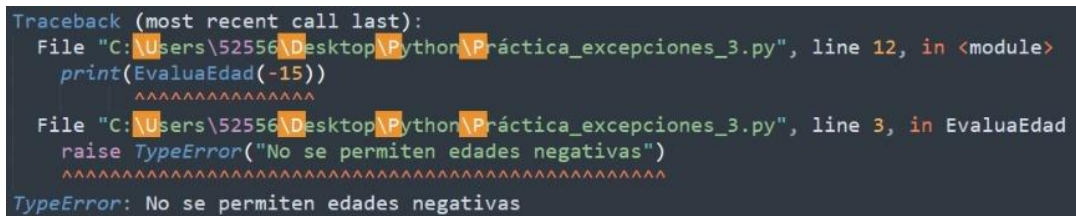
```

if edad < 20:
    return "eres muy joven"
elif edad < 40:
    return "eres joven"
elif edad < 65:
    return "eres maduro"
elif edad < 100:
    return "cuídate"

print(EvaluaEdad(-15))

```

Si la edad es menor que 0, la función genera y lanza una excepción de tipo `TypeError` con el mensaje "No se permiten edades negativas".



```

Traceback (most recent call last):
  File "C:\Users\52556\Desktop\Python\Práctica_excepciones_3.py", line 12, in <module>
    print(EvaluaEdad(-15))
    ^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\52556\Desktop\Python\Práctica_excepciones_3.py", line 3, in EvaluaEdad
    raise TypeError("No se permiten edades negativas")
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: No se permiten edades negativas

```

Ejemplo 6:

```

import math

def CalculaRaiz(num1):
    if num1 < 0:
        raise ValueError("El número no puede ser negativo")
    else:
        return math.sqrt(num1)

op1 = int(input("Introduce un número: "))

try:
    print(CalculaRaiz(op1))
except ValueError as ErrorDeNumeroNegativo:
    print(ErrorDeNumeroNegativo)

print("Programa terminado") ➡ Introduce un número: -144
El número no puede ser negativo

```

Programa terminado

La **palabra clave as** se utiliza para proporcionar un alias o un nombre alternativo a algo, ya sea un módulo durante la importación o un objeto de excepción durante el manejo de excepciones.

Capítulo 12: POO

La **programación orientada a procedimientos** es un paradigma de programación en el que un programa se divide en una serie de procedimientos o funciones, que son secuencias de instrucciones que realizan una tarea específica.

Algunas de las desventajas de la programación orientada a procedimientos incluyen:

- 1-. Unidades de código muy grandes en aplicaciones complejas:** A medida que las aplicaciones se vuelven más complejas, el código tiende a crecer en tamaño, lo que puede llevar a tener unidades de código muy grandes y difíciles de manejar.
- 2-. En aplicación es complejas el código resultaba difícil de descifrar:** Dado que el código puede volverse muy grande y complejo, puede ser difícil de entender, especialmente para aquellos que no están familiarizados con el código. Esto puede hacer que el mantenimiento y la actualización del código sean desafiantes.
- 3-. Limitaciones en la reutilización de código:** Como los procedimientos son específicos para ciertas tareas, puede ser complicado utilizarlos en diferentes partes de la aplicación sin hacer modificaciones. Esto puede llevar a una mayor duplicación de código y a una menor flexibilidad en el diseño del software.
- 4-. Si existe fallo en alguna línea del código es muy probable que el programa caiga:** Un error en una línea de código puede causar que todo el programa falle. Esto se debe a que los procedimientos se ejecutan secuencialmente, por lo que un error en una parte del código puede afectar a todo el programa.
- 5-. Aparición frecuente de código espagueti:** Es un término que se usa para describir el código que es difícil de seguir o entender debido a su estructura compleja y enredada. Este tipo de código puede aparecer con frecuencia debido a que no se siguen buenas prácticas de programación.
- 6-. Difícil de depurar por otros programadores en caso de necesidad o error:** Como los procedimientos pueden ser largos y complejos, y dado que no hay una forma natural de agrupar los datos con las funciones que los manipulan, puede ser difícil para otros programadores entender y depurar el código si surge un error o si es necesario hacer cambios.

Nota: Algunos ejemplos de lenguajes de programación son Fortran, Cobol, Basic y C.

La **programación orientada a objetos (POO)** es un paradigma de programación que se fundamenta en la idea de representar entidades del mundo real como objetos en el código de

programación. Cada objeto puede tener atributos, que representan sus características o estado, y funciones, conocidas como métodos, que definen su comportamiento.

En la POO, los objetos son instancias de "clases", que son estructuras que definen el comportamiento y las propiedades de esos objetos. Este enfoque permite organizar el código de manera más modular y reutilizable, ya que cada objeto puede interactuar con otros objetos a través de métodos y mensajes.

La naturaleza de un objeto de la vida real puede variar según el contexto, pero en términos generales, un objeto suele tener las siguientes características:

- 1-. **Comportamiento:** Es la manera en que el objeto actúa o responde ante diferentes situaciones. Este comportamiento puede estar determinado por sus características y por las interacciones con otros objetos o el entorno.
- 2-. **Atributos:** Son las características o propiedades que definen al objeto. Estos atributos pueden ser tangibles, como el color, tamaño o forma, o intangibles, como el estado emocional o la categoría a la que pertenece.



Por ejemplo:

¿Qué propiedades tiene el coche?

Un auto tiene color, tamaño, peso, velocidad, etc.

¿Qué comportamiento tiene el coche?

Un auto puede arrancar, frenar, acelerar, girar, etc.

Objeto coche

Algunas de las ventajas de la programación orientada a objetos incluyen:

- 1-. **Modularidad:** Permite dividir un programa en módulos independientes (clases y objetos), lo que facilita la organización, mantenimiento y comprensión del código, así como la colaboración en equipos de desarrollo.
- 2-. **Reutilización de código:** Promueve la reutilización de código a través de la creación de clases y objetos, lo que permite utilizar y extender funcionalidades ya existentes en diferentes partes de un programa o en proyectos distintos.
- 3-. **Tratamiento de excepciones:** Si se produce un error o excepción en una parte del código, el programa puede manejar esa situación de manera controlada y continuar su ejecución sin detenerse por completo.
- 4-. **Encapsulamiento:** Consiste en ocultar los detalles de implementación de una clase u objeto y proporcionar una interfaz clara y definida para interactuar con ellos. Esto ayuda a prevenir accesos no autorizados o errores accidentales, y promueve una mayor seguridad y robustez del código.

Nota: Algunos ejemplos de lenguajes de programación son Java, Python, Swift, Ruby y C++.

Conceptos o términos de la programación orientada a objetos:

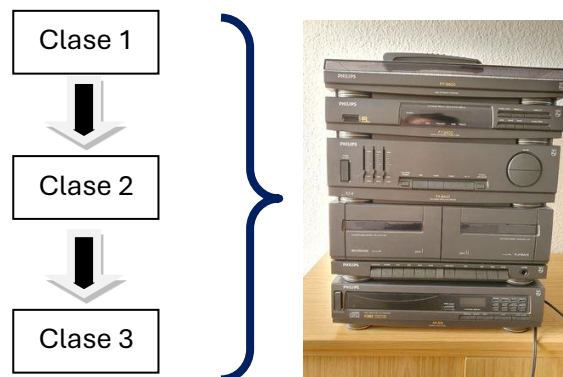
Clase: Proporciona la estructura y el comportamiento que comparten múltiples objetos del mismo tipo. Las clases son como moldes que se utilizan para crear objetos específicos con características y comportamientos similares.



Objeto de clase: También conocido como ejemplar de clase o instancia de clase, es una entidad concreta creada a partir de una clase. Cuando se instancia una clase, se crea un objeto que posee las características y comportamientos definidos por esa clase. Cada objeto de clase es único e independiente de otros objetos creados a partir de la misma clase, pero comparte la misma estructura y comportamiento definidos por la clase.



Modularización: Se refiere a la división de un programa en partes más pequeñas y manejables llamadas módulos. Estos módulos son unidades independientes que realizan una función específica dentro del programa y están diseñados para ser reutilizables en diferentes partes del código.





Encapsulación: Es la capacidad de una clase para ocultar los detalles internos de su implementación y exponer solo una interfaz bien definida para interactuar con otros objetos.


Esto significa que los datos (atributos) y el comportamiento (métodos) de un objeto están agrupados y protegidos dentro de la clase, y solo se pueden acceder y modificar a través de métodos específicos definidos en esa clase.


Métodos de acceso: También conocidos como getters y setters, son métodos especiales que se utilizan para acceder y modificar los atributos de un objeto de forma controlada, permitiendo garantizar el principio de encapsulación al proporcionar una interfaz pública para interactuar con los atributos privados de una clase.





Ejemplo 1:

```
class Coche():  Definición de la clase "Coche"

    LargoChasis = 250
    AnchoChasis = 120
    Ruedas = 4
    EnMarcha = False  Atributos o propiedades de la clase

    def arrancar(self):  Cambia el estado del coche a "en marcha"
        self.EnMarcha = True

    def estado(self):  Devuelve una cadena de texto indicando si el
        if(self.EnMarcha): coche está en marcha
            return "El coche está en marcha"
        else:
            return "El coche está parado"

MiCoche = Coche()  Instancia de la clase "Coche"
print("El largo del coche es:",MiCoche.LargoChasis)
print("El coche tiene",MiCoche.Ruedas, "ruedas")
MiCoche.arrancar()  Se llama al método "arrancar" en el objeto "MiCoche"
print(MiCoche.estado()) para ponerlo en marcha
  Se imprime el estado actual del coche utilizando el método estado()
```

El largo del coche es: 250

El coche tiene 4 ruedas

El coche está en marcha

Un **método** es una función asociada a un objeto. Los métodos definen el comportamiento de un objeto y pueden acceder y modificar sus atributos.

La sintaxis inicial de un método es:

```
def function(self):
```

← “function” es el nombre del método y se puede cambiar

```
    pass
```

← Self es un parámetro que recibe el método; en otras palabras, es una convención que se utiliza para referirse al objeto en sí mismo.

La **notación del punto** es una convención utilizada en muchos lenguajes de programación, para acceder a los miembros (atributos y métodos) de un objeto:

Para acceder a las propiedades de un objeto ponemos:

```
Nombre_del_objeto . propiedad = valor
```

Para acceder a los comportamientos (métodos) de un objeto ponemos:

```
Nombre_del_objeto . comportamiento ()
```

En la programación orientada a objetos, es común que las características compartidas de los objetos formen parte de un **estado inicial**. Esto implica que al crear una instancia u objeto de una clase, este adquiera automáticamente un estado inicial predefinido, que incluye atributos como ancho, alto, largo, número de ruedas, entre otros, según lo especificado en la clase. Este estado inicial se define mediante lo que se conoce como un constructor.

Un **constructor** es un método especial que se utiliza para inicializar el estado de los objetos de una clase. Permite especificar claramente cuál será el estado inicial de los objetos que se creen a partir de esa clase.

En Python, el constructor se define utilizando el método especial `__init__`:

```
class nombre:
```

```
    def __init__(self, parámetro_1, parámetro_2):
        self.atributo_1 = parámetro_1
        self.atributo_2 = parámetro_2
```

Nota: El parámetro self se usa para acceder y modificar los atributos de ese objeto.



Para encapsular variables en un constructor, podemos definir los atributos como privados y luego proporcionar métodos públicos para acceder y modificar estos atributos (esta es una práctica común para garantizar la integridad de los datos y controlar el acceso a ellos desde fuera de la clase).

Recordemos que el encapsulamiento consiste en ocultar los detalles de implementación de una clase y exponer una interfaz pública para interactuar con los objetos.

Los atributos de la clase que comienzan con dos guiones bajos (__) se consideran "privados" y no deben ser accedidos desde fuera de la clase. Intentar acceder a estos miembros desde fuera de la clase generará un error.

Nota: Si has definido un atributo como privado, puedes acceder a este atributo dentro de la misma clase utilizando self.__propiedad

Ejemplo 2:

```
class Coche():  
    def __init__(self):  Constructor de la clase  
        self.__LargoChasis = 250  Estos atributos son privados y se acceden  
        self.__AnchoChasis = 120 mediante métodos de acceso.  
        self.__Ruedas = 4  
        self.__EnMarcha = False  
    def arrancar(self, arrancamos):  
        self.__EnMarcha = arrancamos  
        if (self.__EnMarcha):  
            chequeo = self.__chequeo_interno()  
            if (self.__EnMarcha and chequeo):  
                return "El coche está en marcha"  
            elif (self.__EnMarcha and chequeo == False):  
                return "Algo ha ido mal en el chequeo, no podemos arrancar"  
        else:  
            return "El coche está parado"  
    def estado(self):  
        print("El coche tiene", self.__Ruedas, "ruedas. Un ancho de", self.__AnchoChasis, "y un  
largo de", self.__LargoChasis)  
    def __chequeo_interno(self):  
        print("Realizando chequeo interno")  
        self.gasolina = "ok"  
        self.aceite = "ok"  
        self.puertas = "cerradas"  
        if (self.gasolina == "ok" and self.aceite == "ok" and self.puertas == "cerradas"):  
            return True
```

```

else:
    return False

MiCoche = Coche()
print(MiCoche.arrancar(True))
print (MiCoche.estado())
#print(MiCoche.__chequeo_interno()) ← Se intenta modificar directamente el
print("---A continuación, creamos el segundo objeto---") método, pero como está
MiCoche2 = Coche() encapsulado, no se modifica realmente
print(MiCoche2.arrancar(False))
MiCoche2.__Ruedas = 2 ← Se intenta modificar directamente el atributo, pero
MiCoche2.estado() → como está encapsulado, no se modifica realmente

Realizando chequeo interno

El coche esta en marcha

El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250

---A continuación, creamos el segundo objeto---

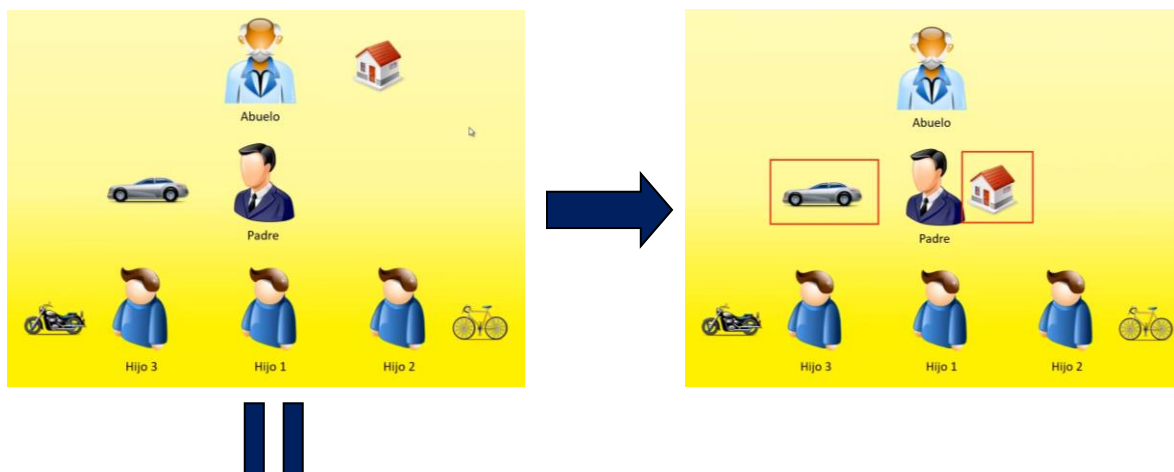
El coche está parado

El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250

```

La **herencia** permite a una clase (denominada **clase derivada o subclase**) heredar atributos y métodos de otra clase (denominada **clase base o superclase**). Esto facilita la reutilización del código, permitiendo que las subclases incorporen o extiendan la funcionalidad de las superclases sin necesidad de duplicar código.

Cabe mencionar que este concepto establece una relación jerárquica entre las clases, lo que facilita la creación de estructuras más complejas y organizadas.





Tipos de herencia:

1-. Herencia simple: Una subclase puede heredar atributos y métodos de una única superclase.

2-. Herencia múltiple: Una subclase puede heredar atributos y métodos de múltiples superclases.

Nota: En el contexto de la herencia múltiple, si una subclase hereda métodos o atributos de múltiples superclases y hay métodos o atributos con el mismo nombre en diferentes superclases, Python siempre buscará y utilizará el método de la primera superclase listada que lo contenga según el MRO (Method Resolution Order).

3-. Herencia multinivel: Las clases están organizadas en una jerarquía con múltiples niveles de abstracción.

La **sobreescritura de métodos** permite a una subclase proporcionar una implementación específica de un método que ya está definido en su superclase. Esto significa que el método en la subclase reemplaza al método con el mismo nombre en la superclase.

Nota: La capacidad de soportar herencia múltiple varía significativamente entre los lenguajes de programación. Mientras algunos lenguajes lo permiten de manera directa (como Python), otros imponen restricciones (como C++), y algunos optan por alternativas como el uso de interfaces (como Java y C#).

Ejemplo 3:

class Vehiculos:

```
def __init__(self, marca, modelo):  
    self.marca = marca  
    self.modelo = modelo  
    self.enmarcha = False  
    self.acelera = False
```

```

        self.frena = False
    def arrancar(self):
        self.enmarcha = True
    def acelerar(self):
        self.acelera = True
    def frenar(self):
        self.frena = True
    def estado(self):
        print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn marcha: ",
              self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena)
class Furgoneta(Vehiculos):
    def carga(self, cargar):
        self.cargado = cargar
        if self.cargado:
            return "La furgoneta está cargada"
        else:
            return "La furgoneta no está cargada"
class Moto(Vehiculos):
    hcaballito = ""
    def caballito(self):
        self.hcaballito = "Voy haciendo el caballito"
    def estado(self):
        print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn marcha: ",
              self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena, "\n",
              self.hcaballito)
class VElectricos(Vehiculos):
    def __init__(self, marca, modelo):
        super().__init__(marca, modelo)
        self.autonomia = 100

```

```

def cargarenergia(self):
    self.cargando = True
miMoto = Moto("Honda", "CBR")
miMoto.caballito()
miMoto.estado()
miFurgoneta = Furgoneta("Renault", "Kangoo")
miFurgoneta.arrancar()
miFurgoneta.estado()
print(miFurgoneta.carga(True))
class BicicletaElectrica(VElectricos, Vehiculos):
    pass
miBici = BicicletaElectrica("Orbea", "Kangoo") ➡

```

Marca: Honda Modelo: CBR En marcha: False Acelerando: False
 Frenando: False

Voy haciendo el caballito

Marca: Renault Modelo: Kangoo En marcha: True Acelerando: False
 Frenando: False

La furgoneta está cargada

La **función super()** se utiliza para llamar a un método de una superclase (clase padre) desde una subclase. Esto es particularmente útil en el contexto de herencia, donde una subclase quiere extender o modificar el comportamiento de un método heredado de su superclase.

Algunos propósitos y usos de esta función son:

- 1-. Acceder a Métodos de la Superclase:** Permite a la subclase llamar a métodos de la superclase sin tener que referirse explícitamente al nombre de la superclase. Esto es útil para mantener el código más limpio y evitar errores si la superclase cambia de nombre.
- 2-. Inicialización de la Superclase:** En el método `__init__`, se usa comúnmente para inicializar la superclase. Esto asegura que cualquier configuración que la superclase necesita sea realizada antes de la configuración específica de la subclase.
- 3-. Herencia Múltiple y MRO:** En casos de herencia múltiple, la función respeta el MRO y llama al siguiente método en la cadena de herencia de acuerdo con este orden, asegurando que cada clase en la jerarquía de herencia se inicialice correctamente.

El **principio de sustitución de Liskov** (LSP) es uno de los cinco principios sólidos de la programación orientada a objetos y consiste en que si “S” es una subclase de “T”, entonces los objetos de la clase “T” pueden ser reemplazados con objetos de la clase “S” (es decir, un objeto de la subclase S puede sustituir un objeto de la superclase T) sin alterar las propiedades deseables del programa (corrección, funcionalidad, etc).

La función **isinstance()** se usa para comprobar si un objeto es una instancia de una clase específica o de una tupla de clases.

Ejemplo 4:

```
class Persona():
    def __init__(self, nombre, edad, Lugar_residencia):
        self.nombre = nombre
        self.edad = edad
        self.lugar_residencia = Lugar_residencia
    def descripcion(self):
        print("Nombre: ", self.nombre, "Edad: ", self.edad, "Residencia: ",
self.lugar_residencia)

class Empleado(Persona):
    def __init__(self, salario, antigüedad, nombre_empleado, edad_empleado,
residencia_empleado):
        super().__init__(nombre_empleado, edad_empleado, residencia_empleado)
        self.salario = salario
        self.antigüedad = antigüedad
    def descripcion(self):
        super().descripcion()
        print("Salario: ", self.salario, "Antigüedad: ", self.antigüedad)

Manuel = Empleado(1500, 15, "Manuel", 55, "Colombia")
Manuel.descripcion()
print(isinstance(Manuel, Persona))————>

Nombre: Manuel Edad: 55 Residencia: Colombia

Salario: 1500 Antigüedad: 15

True
```



¿Qué características en común tienen todos los objetos?

Marca Modelo

¿Qué comportamientos en común tienen todos los objetos?

Arrancan Aceleran Frenan

CLASE PADRE O SUPERCLASE

Marca Modelo

Arrancan Aceleran Frenan

El **polimorfismo** permite que objetos de diferentes clases se traten como objetos de una clase común. Específicamente, el polimorfismo permite que una sola interfaz a múltiples implementaciones, lo que significa que se pueden usar métodos en objetos de diferentes tipos de una manera uniforme.

El polimorfismo es un concepto clave en la programación orientada a objetos (POO) que se refiere a la capacidad de una función, método u operador para tomar diferentes formas. En términos más concretos, el polimorfismo permite que una sola interfaz se use para representar diferentes tipos de objetos y que un mismo método pueda tener diferentes comportamientos según el objeto sobre el que se aplica.



Ejemplo 5:

```
class Coche():
    def desplazamiento(self):
        print("Me desplazo utilizando cuatro ruedas")

class Moto():
    def desplazamiento(self):
        print("Me desplazo utilizando dos ruedas")

class Camion():
    def desplazamiento(self):
        print("Me desplazo utilizando seis ruedas")
```

```
def desplazamientoVehiculo(vehiculo):  
    vehiculo.desplazamiento()  
  
miVehiculo = Camion()  
  
desplazamientoVehiculo(miVehiculo) —> Me desplazo utilizando seis ruedas
```

Capítulo 13: Métodos de cadenas y módulos

Los **métodos de cadenas** son funciones que se aplican a los objetos de tipo cadena (str) y permiten realizar diversas operaciones sobre ellas, como manipulación, búsqueda, reemplazo, entre otras. Estos métodos son muy útiles para trabajar con texto y realizar tareas comunes de procesamiento de cadenas.

A continuación, se presenta una lista con una breve descripción de algunos de los métodos más comunes en Python:

1-. Métodos de Manipulación de Cadenas

str.upper(): Convierte todos los caracteres de la cadena a mayúsculas.

Ejemplo:

```
texto = "hola"  
print(texto.upper()) —> HOLA
```

str.lower(): Convierte todos los caracteres de la cadena a minúsculas.

Ejemplo:

```
texto = "HOLA"  
print(texto.lower()) —> hola
```

str.capitalize(): Convierte el primer carácter de la cadena a mayúscula y el resto a minúsculas.

Ejemplo:

```
texto = "hola mundo"  
print(texto.capitalize()) —> Hola mundo
```

Nota: El método “title” convierte el primer carácter de cada palabra a mayúscula.

str.strip(): Elimina los espacios en blanco al inicio y al final de la cadena.

Ejemplo:

```
texto = " hola "  
print(texto.strip()) —> hola
```

Nota: El método “lstrip” elimina los espacios en blanco al inicio de la cadena y el método “rstrip” elimina los espacios en blanco al final de la cadena.

2-. Métodos de Búsqueda y Reemplazo

str.find(sub): Devuelve el índice de la primera aparición de la subcadena “sub” en la cadena, o “-1” si no se encuentra.

Ejemplo:

```
texto = "hola mundo"
print(texto.find("mundo")) —————> 5
```

str.replace(old, new): Reemplaza todas las ocurrencias de la subcadena “old” por “new”.

Ejemplo:

```
texto = "hola mundo"
print(texto.replace("mundo", "Python")) —————> hola Python
```

Nota: El método “startswith(prefix)” devuelve “True” si la cadena comienza con el prefijo establecido en prefix y “endswith(suffix)” devuelve “True” si la cadena termina con el sufijo establecido en suffix.

str.count (sub): Cuenta el número de ocurrencias de la subcadena “sub” en la cadena.

Ejemplo:

```
texto = "Hola mundo, hola Python"
print(texto.count("hola")) —————> 2
```

3-. Métodos de División y Unión

str.split(sep): Divide la cadena en una lista usando “sep” como delimitador.

Ejemplo:

```
texto = "hola mundo"
print(texto.split(" ")) —————> ['hola', 'mundo']
```

str.join(iterable): Une una secuencia de cadenas en una sola cadena, separadas por la cadena original.

Ejemplo:

```
lista = ["hola", "mundo"]
print(" ".join(lista)) —————> "hola mundo"
```

4-. Métodos de Validación y Comprobación

str.isdigit(): Devuelve “True” si todos los caracteres de la cadena son dígitos.

Ejemplo:

```
lista = "12345"
print(texto.isdigit()) —————> True
```

Nota: El método “`str.isalnum()`” devuelve “True” si todos los caracteres de la cadena son alfanuméricos y “`str.isalpha()`” devuelve “True” si todos los caracteres de la cadena son letras.

¿Qué es un módulo?

Un **módulo** es un archivo que contiene definiciones y declaraciones de Python, como funciones, clases y variables, que se pueden importar y reutilizar en otros programas. Los módulos permiten organizar y estructurar el código de manera más eficiente, facilitando la reutilización y el mantenimiento de este.

Nota: Cada archivo de Python (.py) puede ser considerado un módulo.

Para utilizar un módulo en Python, se emplea la instrucción `import`:

Ejemplo 1:

Imaginémonos que creamos un primer módulo llamado “`Funciones_matematicas.py`”

```
def sumar(op1, op2):  
    print("El resultado de la suma es: ", op1 + op2)  
def restar(op1, op2):  
    print("El resultado de la suma es: ", op1 - op2)  
def multiplicar(op1, op2):  
    print("El resultado de la suma es: ", op1 * op2)
```

Ahora crearemos un segundo módulo, llamado “`Uso_de_funciones`”, para poder utilizar las funciones del primer módulo.

```
import Funciones_matematicas  ← Llamada al módulo  
Funciones_matematicas.sumar(7,5) ← Se usa la notación del punto para poder  
Funciones_matematicas.restar(9,5) → utilizar las funciones del módulo
```

El resultado de la suma es: 12

El resultado de la suma es: 4

Nota: La desventaja de este enfoque es que para cada función que queremos utilizar, necesitamos escribir explícitamente el nombre del módulo.

Ejemplo 2:

```
from Funciones_matematicas import *  ← Importa todas las funciones, clases,  
sumar(7,5)                             variables y constantes definidas en el  
restar(9,5) →                             módulo
```

El resultado de la suma es: 12

El resultado de la suma es: 4

Ejemplo 3:

Para este caso, utilizaremos dos módulos: uno llamado “vehículos”, utilizado previamente en los ejemplos de programación orientada a objetos, y otro titulado “Uso_de_vehículos”.

```
from Modulo_vehiculos import *  
  
miCoche = Vehiculos("Mazda", "MX5")  
  
miCoche.estado() —————>
```

Marca: Mazda

Modelo: MX5

En marcha: False

Acelerando: False

Frenando: False

Nota: Si movemos un módulo a un directorio diferente, el programa que depende de ese módulo dejará de funcionar porque Python no podrá encontrar el módulo en su ubicación original. Por ende, una solución sería el uso de paquetes.

¿Qué es un paquete?

Un **paquete** es una forma de estructurar los módulos mediante la creación de una jerarquía de directorios. Cada directorio de un paquete contiene un archivo especial llamado `__init__.py` (puede estar vacío) que indica a Python que ese directorio debe ser tratado como un paquete.



Ejemplo 1:

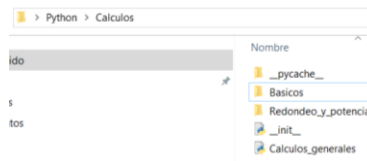
En este ejemplo, crearemos un módulo llamado “uso_modulos” en la raíz principal de la carpeta de Python. Este módulo utilizará otro módulo que estará ubicado en un paquete llamado “Calculos_generales”.

```
from Calculos.Calculos_generales import dividir ← Primero especificamos el nombre  
  
dividir(4,6) —————> de la carpeta y después el nombre
```

El resultado de la division es: 0.6666666666666666 **del módulo que queremos usar**

Nota: La carpeta `__pycache__` se crea automáticamente cuando ejecutas módulos dentro de tu paquete. Esta carpeta almacena los archivos compilados en bytecode (con extensión .pyc), lo cual mejora el rendimiento al evitar recompilar los módulos cada vez que se ejecutan.

Los **subpaquetes** son paquetes que están contenidos dentro de otros paquetes. Estos permiten organizar el código en una estructura jerárquica, lo que facilita la gestión y el mantenimiento de proyectos más grandes y complejos.



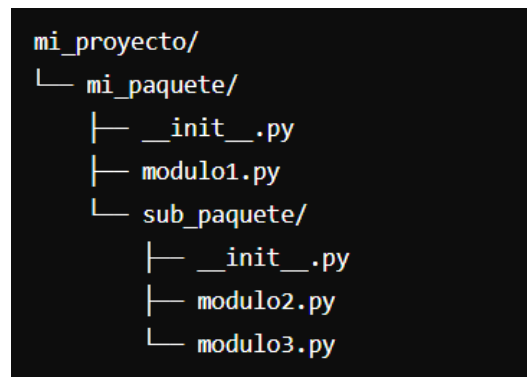
Nota: Al igual que los paquetes, los subpaquetes también deben contener un archivo `__init__.py` para ser reconocidos como tales.

Ejemplo 2:

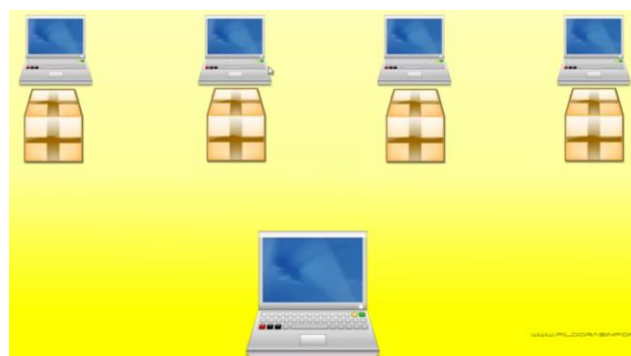
from Calculos.Basicos.Operaciones_basicas import * ← **Primero especificamos el nombre de la carpeta, después el nombre del paquete y después el nombre del módulo que queremos usar**

sumar(4,6) →

El resultado de la suma es: 10



Los **paquetes distribuibles** son colecciones de módulos y paquetes que están empaquetados de manera que puedan ser fácilmente distribuidos, instalados y utilizados por otros. Estos paquetes permiten compartir código de manera eficiente y aseguran que otros usuarios puedan instalar y utilizar el código con todas sus dependencias correctamente configuradas.



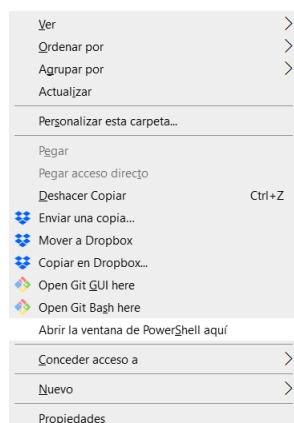
Pasos a seguir en Sublime Text para la creación de un paquete distribuible:

- 1-. View → Side Bar → Hide Open Files (Se te habilitara en la izquierda un recuadro gris)
- 2-. Arrastra la carpeta de Python al recuadro gris (Ahí podrás ver todo el contenido)
- 3-. Crearemos un nuevo archivo (New File) al darle click derecho al cuadro gris en la parte de Folders → Python
- 4-. Guardaremos dicho archivo en la raíz de la carpeta principal y empezaremos a diseñar la configuración del paquete en base a que funciones o clases queremos distribuir

```
from setuptools import setup
```

```
setup(  
    name = "Paquete de calculos",  
    version = "1.0",  
    description = "Este es un paquete que contiene las funciones de potencia y redondeo",  
    author = "Sebastian Solis Villafuerte",  
    author_email = "sebastiansolvil@gmail.com",  
    url = "https://acortar.link/CWcM3s",  
    packages = ["Calculos","Calculos.Redondeo_y_potencia"]  
)
```

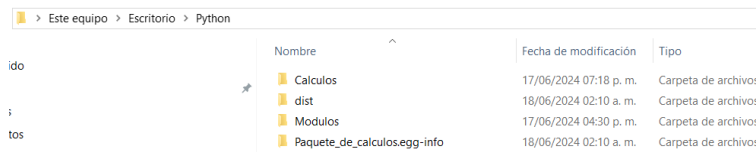
- 5-. Una vez que hayamos guardado los cambios, nos dirigiremos a la carpeta raíz del proyecto. En el módulo “setup.py”, pulsaremos Shift y haremos clic derecho para habilitar la opción de abrir una ventana de PowerShell.



- 6-. Una vez estemos en la consola, ingresaremos el siguiente comando: “python setup.py sdist” (esto convertirá nuestro módulo en un paquete distribuible).

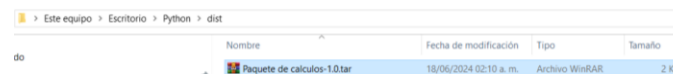

```
Windows PowerShell
PS C:\Users\52556\Desktop\Python> python setup.py sdist
running sdist
running egg_info
creating Paquete_de_calculos.egg-info
```

7-. Para comprobar que la instalación tuvo éxito, podemos verificar que en la carpeta raíz se hayan creado dos carpetas: una llamada “dist” y otra llamada “Paquete_de_calculos.egg-info”.



	Nombre	Fecha de modificación	Tipo
do	Calculos	17/06/2024 07:18 p. m.	Carpeta de archivos
i	dist	18/06/2024 02:10 a. m.	Carpeta de archivos
tos	Modulos	17/06/2024 04:30 p. m.	Carpeta de archivos
	Paquete_de_calculos.egg-info	18/06/2024 02:10 a. m.	Carpeta de archivos

8-. Si ingresamos a la carpeta “dist”, podremos ver un archivo con la extensión “.tar.gz”, que es un archivo comprimido. Este archivo podremos compartirlo con diferentes personas.



	Nombre	Fecha de modificación	Tipo	Tamaño
do	Paquete de calculos-1.0.tar	18/06/2024 02:10 a. m.	Archivo WinRAR	2 KB

9-. Para instalarlo en nuestro ordenador, ejecutaremos dos comandos en la consola de PowerShell.

a) PS C:\Users\52556\Desktop\Python> cd dist

b) PS C:\Users\52556\Desktop\Python\dist> pip install "Paquete de calculos-1.0.tar.gz"

```
Processing c:\users\52556\desktop\python\dist\paquete de calculos-1.0.tar.gz
Installing build dependencies ... done
Getting requirements to build wheel ... done
Installing backend dependencies ... done
Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: Paquete-de-calculos
Building wheel for Paquete-de-calculos (pyproject.toml) ... done
Created wheel for Paquete-de-calculos: filename=Paquete_de_calculos-1.0-py3-none-any.whl size=2217 sha256=38f639ccacf2531d77b954d641532c22b3edcf626d5b371995c1202dd71d93a8
Stored in directory: c:\users\52556\appdata\local\pip\cache\wheels\3f\60\9a\bf6c6bae4a7d31ce6cb448828e8dd151e18a54977cbf8a363
Successfully built Paquete-de-calculos
Installing collected packages: Paquete-de-calculos
Successfully installed Paquete-de-calculos-1.0
```

Capítulo 14: Archivos externos y serializaciones

Los **archivos externos** son aquellos que se almacenan fuera de un programa o sistema principal pero son utilizados por dicho programa o sistema para diversas funciones. Estos archivos pueden contener datos, configuraciones, scripts, bibliotecas o cualquier otro tipo de información que el software necesite para operar correctamente.

Algunos tipos comunes de archivos externos son:

1-. **Archivos de Datos:** Contienen información que el programa necesita procesar. Por ejemplo, bases de datos, archivos CSV, JSON, XML, etc.

Ejemplo: Un archivo CSV con una lista de clientes que un sistema de CRM utiliza para generar informes.

2-. **Archivos de Configuración:** Contienen configuraciones y parámetros que el programa utiliza para personalizar su comportamiento.

Ejemplo: Un archivo config.ini que almacena las preferencias del usuario y las opciones de configuración de una aplicación.

3-. Bibliotecas y Módulos: Archivos que contienen código reutilizable que puede ser importado y utilizado por otros programas.

Ejemplo: Una biblioteca en Python (archivo .py) que proporciona funciones matemáticas avanzadas.

4-. Archivos de Recursos: Incluyen elementos multimedia como imágenes, sonidos, videos, fuentes, etc., que el programa utiliza para su interfaz de usuario o para otros propósitos.

Ejemplo: Archivos de imagen (.jpg, .png) que un sitio web utiliza para mostrar gráficos y fotos.

5-. Scripts: Archivos que contienen secuencias de comandos para automatizar tareas.

Ejemplo: Un script de shell (.sh) que automatiza el proceso de despliegue de una aplicación en un servidor.

6-. Archivos de Registro (Logs): Archivos que registran eventos y actividades del sistema o programa, útiles para el diagnóstico y la solución de problemas.

Ejemplo: Un archivo de log que guarda los errores y eventos de una aplicación web.

7-. Archivos de Backup y Respaldo: Copias de seguridad de archivos importantes que pueden ser restaurados en caso de pérdida de datos.

Ejemplo: Un archivo .bak que es una copia de seguridad de una base de datos.

La utilización de archivos externos permite que los programas sean más modulares, flexibles y manejables, ya que pueden almacenar y acceder a grandes cantidades de datos sin sobrecargar el código principal del software.

La **persistencia de datos** se refiere a la capacidad de almacenar datos de manera que puedan ser recuperados y utilizados en el futuro, incluso después de que el programa que los creó o utilizó haya terminado de ejecutarse. En otras palabras, los datos persisten más allá del ciclo de vida de la aplicación que los generó. Este concepto es fundamental en el desarrollo de software, ya que permite que la información se mantenga disponible a lo largo del tiempo y entre diferentes sesiones de uso de un sistema.

Tipos de Persistencia de Datos

1-. Bases de Datos:

- **Relacionales:** Utilizan tablas para almacenar datos y permiten consultas complejas mediante SQL.

Ejemplo: MySQL, PostgreSQL.

- **NoSQL:** Utilizan estructuras de almacenamiento diferentes a las tablas, como documentos, gráficos o pares clave-valor.

Ejemplo: MongoDB, Cassandra.

2-. Archivos: Almacenar datos en archivos en el sistema de archivos del dispositivo.

Ejemplo: Archivos de texto, CSV, JSON, XML.

3-. Sistemas de Almacenamiento en la Nube: Servicios que permiten almacenar datos en servidores remotos accesibles a través de Internet.

Ejemplo: Amazon S3, Google Cloud Storage.

4-. Almacenamiento Local: Técnicas específicas para almacenar datos localmente en dispositivos móviles o en navegadores web.

Ejemplo: SQLite para aplicaciones móviles, LocalStorage para aplicaciones web.

Guardar información en archivos externos implica varias fases que aseguran que los datos se almacenen correctamente y estén disponibles para su uso futuro.

A continuación se describen las fases necesarias para lograr esto:

1-. Preparación de Datos: Antes de guardar la información en un archivo externo, los datos deben estar preparados y en un formato adecuado.

- **Limpieza de Datos:** Asegurarse de que los datos estén completos, sin errores y en el formato correcto.

- **Conversión de Datos:** Transformar los datos a un formato adecuado para el almacenamiento (por ejemplo, convertir un objeto de datos a JSON).

2-. Apertura del Archivo: Para guardar datos en un archivo externo, primero se debe abrir el archivo en el modo adecuado (lectura, escritura, o ambos). Esto se realiza a través de funciones específicas del lenguaje de programación que se esté utilizando.

Ejemplo: `archivo = open("datos.txt", "w")`

3-. Escritura de Datos: Una vez que el archivo está abierto, se procede a escribir los datos en él.

- **Escribir texto:** Escribir cadenas de texto directamente.

- **Serialización:** Convertir objetos complejos a un formato como JSON o XML antes de escribir.

Ejemplo: `datos = {"nombre": "Juan", "edad": 30}`

```
import json
```

```
archivo.write(json.dumps(datos))
```

4-. Cierre del Archivo: Después de escribir los datos, es crucial cerrar el archivo para asegurarse de que los datos se guarden correctamente y que los recursos del sistema se liberen.

Ejemplo: `archivo.close()`

En Python, el **módulo io** proporciona las herramientas necesarias para manejar operaciones de entrada y salida (I/O). Este módulo incluye clases y funciones para trabajar con flujos de datos (streams), que son objetos que permiten leer y escribir datos en una variedad de formatos y ubicaciones, como archivos, memoria y dispositivos.

Ejemplo 1:

`from io import open` ← **Importa la función “open” desde el módulo “io”. Esto permite abrir archivos para lectura o escritura**

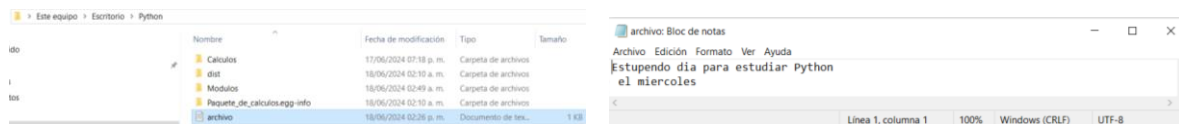
`archivo_texto = open("archivo.txt", "w")` ← **Abre (o crea si no existe) un archivo llamado archivo.txt en modo de escritura (w). Si el archivo ya existe, se borra su contenido actual.**

`frase = "Estupendo día para estudiar Python \n el miércoles"` ← **Crea una variable frase que contiene una cadena de texto. La cadena incluye un carácter de nueva línea (\n), que indica que el texto que sigue debe ir en una nueva línea cuando se escribe en el archivo.**

`archivo_texto.write(frase)` ← **Escribe el contenido de la variable frase en el archivo archivo.txt.**

`archivo_texto.close()` ← **Cierra el archivo para asegurarse de que los datos se guarden correctamente y liberar los recursos del sistema asociados al archivo.**

Resultado:



Ejemplo 2:

Si añadimos las siguientes dos líneas al código anterior, conseguiremos:

`archivo_texto = open("archivo.txt", "r")` ← **Abre el archivo llamado archivo.txt en modo lectura ("r"). Esto significa que el código leerá el contenido del archivo pero no lo modificará ni lo sobrescribirá.**

`texto = archivo_texto.read()` ← **Lee todo el contenido del archivo archivo.txt y lo guarda en la variable texto.**

`print(texto)` →

Estupendo día para estudiar Python

el miercoles

Ejemplo 3:

`lineas_texto = archivo_texto.readlines()` ← **`readlines()` es un método de los objetos de archivo en Python que lee todas las líneas del archivo y las devuelve como una lista de cadenas. Cada cadena en la lista representa una línea del archivo.**

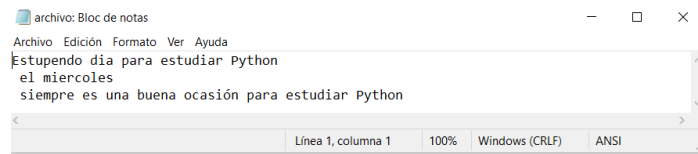
`print(lineas_texto)` → `['Estupendo día para estudiar Python \n', ' el miercoles']`

Ejemplo 4:

`archivo_texto = open("archivo.txt","a")`

`archivo_texto.write("\n siempre es una buena ocasión para estudiar Python")`

Resultado



Ejemplo 5:

`from io import open`

`archivo_texto = open("archivo.txt","r")`

`print(archivo_texto.read())`

`archivo_texto.seek(0)` ← **`seek()` es un método de los objetos de archivo en Python que mueve el puntero de lectura al principio del archivo.**

`print(archivo_texto.read())` →

Estupendo día para estudiar Python

el miercoles

siempre es una buena ocasión para estudiar Python

Estupendo día para estudiar Python

el miercoles

siempre es una buena ocasión para estudiar Python

Ejemplo 6:

```

from io import open
archivo_texto = open("archivo.txt", "r")
archivo_texto.seek(11)
print(archivo_texto.read()) —————> Estupendo día para estudiar Python
    el miércoles

    siempre es una buena ocasión para estudiar Python

```

Ejemplo 7:

```

from io import open
archivo_texto = open("archivo.txt", "r")
print(archivo_texto.read(11))
print(archivo_texto.read()) —————> Estupendo d
    ía para estudiar Python
    el miércoles

    siempre es una buena ocasión para estudiar Python

```

Ejemplo 8:

```

from io import open
archivo_texto = open("archivo.txt", "r")
archivo_texto.seek(len(archivo_texto.read())/2)
print(archivo_texto.read()) —————> siempre es una buena ocasión para estudiar Python

```

Ejemplo 9:

```

from io import open
archivo_texto = open("archivo.txt", "r")
archivo_texto.seek(len(archivo_texto.readline()))
print(archivo_texto.read()) —————> el miércoles

    siempre es una buena ocasión para estudiar Python

```

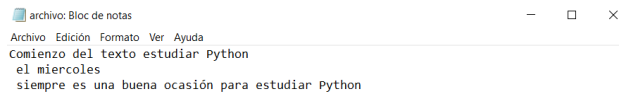
Ejemplo 10:

```

from io import open
archivo_texto = open("archivo.txt", "r+") ← Abre el archivo "archivo.txt" en modo
archivo_texto.write("Comienzo del texto") en modo lectura y escritura (r+)

```

Resultado



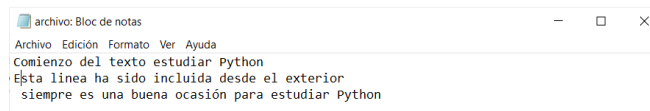
Ejemplo 11:

```
from io import open

archivo_texto = open("archivo.txt", "r+")

lista_texto = archivo_texto.readlines()

lista_texto[1] = "Esta linea ha sido incluida desde el exterior \n" ← Modifica el
archivo_texto.seek(0) ← segundo elemento de la lista agregando una línea de texto
archivo_texto.writelines(lista_texto) ← Escribe las líneas modificadas de la lista de
archivo_texto.close() ← de vuelta al archivo "archivo.txt"
```



La **serialización** es el proceso de convertir un objeto en un formato que puede ser almacenado o transmitido de manera eficiente y luego reconstruido (deserializado) para su uso posterior. En el contexto de la programación y la informática en general, este proceso es fundamental para la persistencia de datos, la comunicación entre sistemas distribuidos y el almacenamiento de datos en archivos o bases de datos.

Es decir que la serialización es el proceso de convertir un objeto en una secuencia de bytes, mientras que la deserialización es el proceso inverso.

Características Clave de la Serialización

- 1-. Transformación de Objetos en Datos Estructurados:** La serialización convierte objetos complejos de un lenguaje de programación específico en una representación que puede ser almacenada o transmitida.
- 2-. Portabilidad:** Permite que los datos serializados puedan ser transferidos a través de redes o almacenados en diferentes sistemas y plataformas sin perder su estructura o significado.
- 3-. Eficiencia:** Los datos serializados están optimizados para ocupar menos espacio en comparación con su representación en memoria o en estructuras de datos en vivo, lo que facilita su almacenamiento y transmisión rápida.

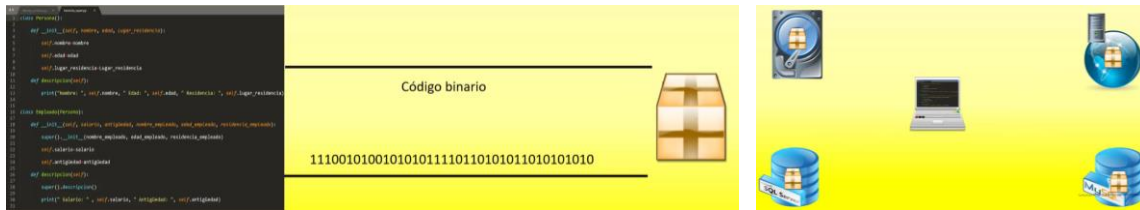
Métodos Comunes de Serialización

- **Serialización en Formato de Texto:** Utiliza formatos como JSON (JavaScript Object Notation) o XML (eXtensible Markup Language) para representar objetos como cadenas de

texto que son fáciles de leer y escribir, aunque pueden ocupar más espacio que otros formatos binarios.

- **Serialización en Formato Binario:** Emplea formatos binarios como Pickle en Python, Protocol Buffers (protobuf) de Google, o BSON (Binary JSON), que son más compactos y eficientes en términos de espacio y velocidad, pero menos legibles para los humanos.

La **biblioteca pickle** proporciona funciones para serializar y deserializar objetos Python. Esto significa que puede convertir objetos complejos en una secuencia de bytes que pueden ser almacenados en un archivo o transmitidos a través de una red, y luego reconstruidos posteriormente en su forma original.



Funciones Principales de pickle:

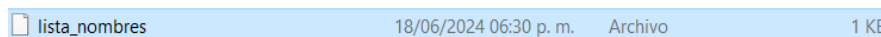
pickle.dump(objeto, archivo): Esta función serializa el objeto objeto y lo escribe en el archivo binario archivo.

```
import pickle  ← Se importa el módulo pickle, que permite la serialización y
lista_nombres = ["Pedro", "Ana", "Maria", "Isabel "]  ← deserialización de objetos
fichero_binario = open("lista_nombres", "wb")  ← Se abre un archivo llamado
                                                "lista_nombres" en modo escritura binaria (wb)
pickle.dump(lista_nombres, fichero_binario)  ← Se usa la función para serializar el
                                                objeto "lista_nombres" y escribirlo en el archivo
                                                "fichero_binario". Esto convierte la lista en una
                                                secuencia de bytes y la guarda en el archivo
```

```
fichero_binario.close()
```

```
del(fichero_binario)  ← Se elimina la referencia a la variable de la memoria. Esto es
                        opcional y no siempre necesario, pero puede ser útil para
                        liberar memoria o evitar errores posteriores
```

Resultado:



pickle.load(archivo): Esta función lee los datos serializados desde el archivo binario archivo y los deserializa para reconstruir el objeto original.


```
import pickle
fichero = open("lista_nombres", "rb")
lista = pickle.load(fichero)
print(lista)
```

← Se abre el archivo llamado lista_nombres en modo lectura binaria (rb), lo que significa que se va a leer el contenido del archivo en formato binario

← Se utiliza la función para deserializar el contenido del archivo y cargarlo en la variable lista. Esto convierte los datos binarios almacenados en el archivo de nuevo en una lista de Python

→ ['Pedro', 'Ana', 'Maria', 'Tsabel ']

Ejemplo 1:

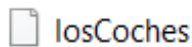
```
import pickle
class Vehiculos:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.enmarcha = False
        self.acelera = False
        self.frena = False
    def arrancar(self):
        self.enmarcha = True
    def acelerar(self):
        self.acelera = True
    def frenar(self):
        self.frena = True
    def estado(self):
        print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn marcha: ",
              self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena)
coche1 = Vehiculos("Mazda","MX5")
coche2 = Vehiculos("Seat","Leon")
coche3 = Vehiculos("Renault","Megane")
```

```

coches = [coche1, coche2, coche3]
fichero = open("losCoches", "wb")
pickle.dump(coches, fichero)
fichero.close()
del(fichero)
ficheroApertura = open("losCoches", "rb")
misCoches = pickle.load(ficheroApertura)
ficheroApertura.close()

```

Resultado:



```

[{"__main__": "Vehiculos", "marca": "Mazda", "modelo": "MX5", "enmarcha": False, "acelerando": False, "frenando": False, "estado": "Sube"}, {"__main__": "Vehiculos", "marca": "Seat", "modelo": "Leon", "enmarcha": False, "acelerando": False, "frenando": False, "estado": "Sube"}, {"__main__": "Vehiculos", "marca": "Renault", "modelo": "Megane", "enmarcha": False, "acelerando": False, "frenando": False, "estado": "Sube"}]

```

Si le agregamos un bucle for al código, obtendremos lo siguiente:

for c in misCoches:

print(c) →

<__main__.Vehiculos object at 0x000002299BF410D0>

<__main__.Vehiculos object at 0x000002299BF48EC0>

<__main__.Vehiculos object at 0x000002299BFCE570>

Nota: Al ejecutar el código, los objetos deserializados se imprimen como instancias de la clase “Vehiculos” con sus respectivas direcciones de memoria, en lugar de mostrar información detallada sobre cada vehículo.

Para mejorar el código y hacer que imprima información más legible, como los detalles de cada vehículo, podemos agregar la siguiente instrucción:

print(c.estado()) →

Marca: Mazda

Marca: Seat

Marca: Renault

Modelo: MX5

Modelo: Leon

Modelo: Megane

En marcha: False

En marcha: False

En marcha: False

Acelerando: False

Acelerando: False

Acelerando: False

Frenando: False

Frenando: False

Frenando: False

None

None

None

Nota: Si deseamos separar los métodos de serialización en archivos distintos, podemos lograrlo mediante la copia y pegado de todas las clases que hayamos creado.

Ejemplo 2:

Nota: El siguiente código define dos clases (Persona y ListaPersonas) y usa la biblioteca pickle para guardar y cargar una lista de objetos Persona desde un archivo externo llamado ficheroExterno. Al final, se crea un objeto Persona y se añade a la lista, y luego se muestra la información guardada en el archivo.

```
import pickle

class Persona:

    def __init__(self, nombre, genero, edad): ← Inicializa una nueva instancia de
        self.nombre = nombre “Persona” con “nombre”, “genero” y “edad”
        self.genero = genero También imprime un mensaje cuando se crea
        self.edad = edad una persona

        print("Se ha creado una nueva persona con el nombre de:", self.nombre)

    def __str__(self): ← Define cómo se representa el objeto “Persona” como una
        return "{} {} {}".format(self.nombre, self.genero, self.edad) cadena de texto

class ListaPersonas: ← Esta clase maneja una lista de objetos “Persona”

    personas = []

    def __init__(self):
        listaDePersonas = open("ficheroExterno", "ab+") ← Se abre el archivo
        listaDePersonas.seek(0) “ficheroExterno” en modo lectura/escritura binaria

        try:
            self.personas = pickle.load(listaDePersonas) ← Intenta cargar la lista de personas
            print("Se cargaron {} personas del fichero externo".format(len(self.personas)))
        except: ← Si el archivo está vacío o no se puede cargar, se maneja la excepción
            print("El fichero está vacío") y se imprime un mensaje

        finally:
            listaDePersonas.close()
            del(listaDePersonas)

    def agregarPersonas(self, p): ← Añade una nueva persona a la lista y guarda la lista
        self.personas.append(p) en el archivo
        self.guardarPersonasEnFicheroExterno()
```

```

def mostrarPersonas(self): ◀— Imprime cada persona en la lista
    for p in self.personas:
        print(p)

def guardarPersonasEnFicheroExterno(self): ◀— Guarda la lista de personas en el
    listaDePersonas = open("ficheroExterno", "wb")          archivo usando pickle
    pickle.dump(self.personas, listaDePersonas)
    listaDePersonas.close()
    del(listaDePersonas)

def mostrarInfoFicheroExterno(self): ◀— Imprime la información de todas las
    print("La informacion del fichero externo es la siguiente:") personas guardadas
    for p in self.personas:                                  en el archivo
        print(p)

miLista = ListaPersonas() ◀— Crea una lista de personas
persona = Persona("Ana", "Femenino", 83) ◀— Crea una persona
miLista.agregarPersonas(persona) ◀— Agrega la persona a la lista
miLista.mostrarInfoFicheroExterno() ◀— Muestra información del archivo

```

Capítulo 15: Interfaces gráficas

Las **interfaces gráficas** de usuario (GUI, por sus siglas en inglés) son sistemas interactivos que permiten a los usuarios comunicarse con las computadoras a través de elementos visuales como ventanas, iconos, botones, menús, y otros controles gráficos.

Componentes clave de las GUI:

- 1-. **Ventanas:** Áreas rectangulares en la pantalla que contienen otros elementos de la interfaz gráfica y permiten al usuario interactuar con la aplicación.
- 2-. **Iconos:** Imágenes pequeñas que representan programas, archivos, funciones u otros elementos de la interfaz gráfica.
- 3-. **Botones:** Elementos clicables que ejecutan una acción específica cuando el usuario hace clic en ellos.
- 4-. **Menús:** Listas desplegables que contienen opciones o comandos que el usuario puede seleccionar.

5-. **Barras de herramientas:** Conjuntos de botones o iconos agrupados que proporcionan acceso rápido a funciones comunes de la aplicación.

6-. **Campos de entrada:** Áreas donde los usuarios pueden introducir datos o texto.

7-. **Controles de selección:** Elementos que permiten a los usuarios seleccionar opciones de un conjunto.

8-. **Cuadros de diálogo:** Ventanas pequeñas que solicitan información o proporcionan notificaciones al usuario.

Ejemplos de herramientas y bibliotecas para crear GUI:

Ejemplo 1:

```
from tkinter import *
```

```
raiz = Tk() ← La instancia se convierte en la ventana principal de la aplicación
```

```
raiz.title("Ventana de pruebas") ← Establece el título de la ventana principal
```

```
raiz.resizable(1,1) ← Cambia el tamaño de la ventana principal
```

```
raiz.iconbitmap("shanks.ico") ← Establece el icono de la ventana raíz de la aplicación
```

```
raiz.geometry("650x350") ← Configura la forma inicial de la ventana raíz de la aplicación
```

```
raiz.config(bg = "gray") ← Configura varias opciones de la ventana principal
```

```
raiz.mainloop() ← Bucle principal de la aplicación, que espera eventos y los despacha según sea necesario
```

Resultado:



Nota: Si alguno de los parámetros del método “resizable” (width o heigth) es “True”, permitirá redimensionar la ventana en el eje respectivo.

Nota: En el parámetro del método “iconbitmap” debemos de proporcionar el nombre de archivo de la imagen que deseamos usar como icono.

Nota: En el parámetro del método “geometry” debemos de proporcionar la geometría de la ventana en forma de “anchuraxaltura+posiciónX+posiciónY”.

Nota: En el parámetro del método “config” debemos de proporcionar argumentos de clave y valor para configurar diferentes propiedades de la ventana. Algunas propiedades comunes

incluyen “bg” (color de fondo), “cursor” (cursor del ratón), “bd” (ancho del borde), entre otros.

Nota: Mientras el método “mainloop” esté en funcionamiento, el bucle principal procesará eventos como clics de botones, entradas de teclado, y otras interacciones del usuario. Sin este bucle, la ventana se abriría y cerraría instantáneamente porque el programa terminaría su ejecución.

Nota: Cabe mencionar que este método deberá ir al final del código de configuración de la GUI porque es el bucle que mantiene la aplicación (o la ventana) en ejecución y esperando interacciones del usuario.

La **extensión .pyw** se usa para scripts de Python que se ejecutan en un entorno de Windows sin mostrar una consola o terminal. Es especialmente útil para aplicaciones con interfaces gráficas de usuario creadas con bibliotecas como Tkinter, donde la ventana de la consola no es necesaria y puede ser una distracción para el usuario.

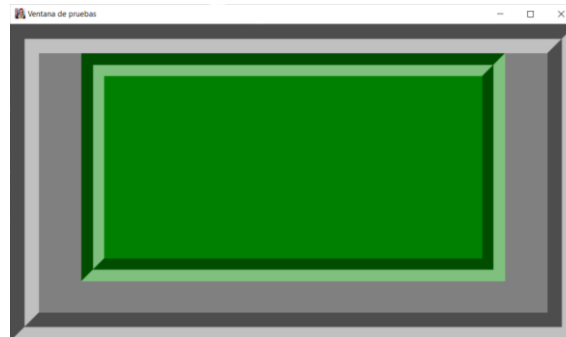
Ejemplo 2:

```
from tkinter import *  
raiz = Tk()  
raiz.title("Ventana de pruebas")  
raiz.iconbitmap("shanks.ico")  
raiz.config(bg = "gray")  
raiz.config(bd = 45)  
raiz.config(relief = "groove")  
raiz.config(cursor = "hand2")  
miFrame = Frame() ← La instancia se convierte en el “Frame” de la aplicación  
miFrame.pack() ← Empaqueta el “Frame” dentro de la ventana principal para que  
sea visible  
# miFrame.pack(side = "left", anchor = "n") ← Empaquetaría el “Frame” en el lado  
izquierdo de la ventana y lo anclaría al norte  
# miFrame.pack(fill = "y", expand = "True") ← Haría que el Frame llene el espacio  
verticalmente y permita que se expanda si la ventana cambia de tamaño  
miFrame.config(bg = "green")  
miFrame.config(width = 650, height = 350)  
miFrame.config(bd = 35)
```

```
miFrame.config(relief = "groove") ← Configura el tipo de relieve del borde del
                                   "Frame" a "groove" (surco), que es un estilo de borde tridimensional
miFrame.config(cursor = "pirate") ← Establece el tipo de cursor que se verá cuando
                                   el puntero del ratón esté sobre el "Frame"

raiz.mainloop()
```

Resultado:



Ejemplo 3:

```
from tkinter import *
raiz = Tk()
raiz.title("Ventana de pruebas")
raiz.iconbitmap("shanks.ico")
miFrame = Frame(raiz, width = 500, height = 400) ← Crea un contenedor "Frame"
                                                    dentro de "raiz" con un ancho de 500 píxeles y una
                                                    altura de 400 píxeles, y lo asigna a la variable "miFrame"

miFrame.pack()
miImagen = PhotoImage(file = "OnePieceGIF.gif") ← Carga la imagen en una
variable
miLabel = Label(miFrame, image = miImagen) ← Crea una etiqueta "Label" dentro de
                                              "miFrame" que muestra la imagen
                                              "miImagen" y la asigna a la variable "miLabel"

# miLabel = Label(miFrame, text = "One Piece", fg = "blue", font = ("Garamond", 18)) ←
                                              Crea una etiqueta con el texto "One Piece", con color de
                                              fuente azul y fuente "Garamond" de tamaño 18
```

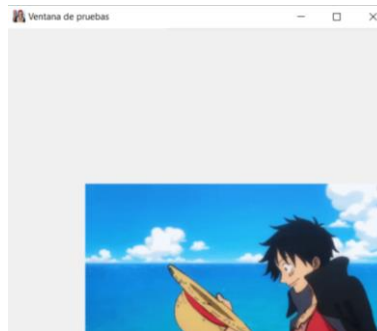
`miLabel.place(x = 100, y = 200)` ← **Posiciona la etiqueta “miLabel” en las coordenadas (100, 200) dentro de miFrame**

`raiz.mainloop()`

Una forma de simplificar el código sería combinando dos instrucciones de etiqueta, de modo que quedaría de la siguiente manera:

`Label(miFrame, text = "One Piece", fg = "blue", font = ("Garamond", 18)).place(x = 100, y = 200)`

Resultado:



Ejemplo 4:

```
from tkinter import *
```

```
raiz = Tk()
```

```
raiz.title("Ventana de pruebas")
```

```
raiz.iconbitmap("shanks.ico")
```

```
miFrame = Frame(raiz, width = 1200, height = 600)
```

```
miFrame.pack()
```

```
minombre = StringVar()
```

 ← **Crea una variable de control de tipo texto que se usará para manejar el texto de un “Entry”**

```
cuadroNombre = Entry(miFrame, textvariable = minombre)
```

 ← **Crea un campo de entrada en “miFrame” que está ligado a la variable minombre**

```
cuadroNombre.grid(row = 0, column = 1)
```

 ← **Ubica el campo de entrada en la fila 0, columna 1 del “Frame” usando el gestor de geometría “grid”**

```
cuadroNombre.config(fg = "gray", justify = "center")
```

 ← **Establece el color del texto a gris y alinea el texto en el centro**

```
cuadroApellido = Entry(miFrame)
```



```

cuadroApellido.grid(row = 1, column = 1)
cuadroApellido.config(fg = "gray", justify = "center")
cuadroDireccion = Entry(miFrame)
cuadroDireccion.grid(row = 2, column = 1)
cuadroDireccion.config(fg = "gray", justify = "center")
cuadroPass = Entry(miFrame)
cuadroPass.grid(row = 3, column = 1)
cuadroPass.config(fg = "gray", justify = "center")
cuadroPass.config(show = "*") ← Establece el carácter de ocultación a *, de modo que
                                los caracteres introducidos se muestren como asteriscos
textoComentario = Text(miFrame, width = 16, height = 5) ← Crea un campo de texto de
                                                            múltiples líneas en “miFrame”
textoComentario.grid(row = 4, column = 1)
scroll=Scrollbar(miFrame, command = textoComentario.yview) ← Crea una barra de
                                                                desplazamiento en “miFrame” que está ligada
                                                                al campo de texto “textoComentario”
scroll.grid(row = 4, column = 2, sticky = "nsew")
textoComentario.config(yscrollcommand = scroll.set) ← Configura la barra de
                                                       desplazamiento para que se ajuste al campo de texto
nombreLabel = Label(miFrame, text = "Nombre:") ← Crea una etiqueta con el texto
                                                    “Nombre:” y la asigna a “nombreLabel”
nombreLabel.grid(row = 0, column = 0, sticky = "w", padx = 10, pady = 10) ← Ubica la
                                                                                etiqueta en la fila 0, columna 0 del “Frame”, alinea el texto a
                                                                                la izquierda, y añade un relleno de 10 píxeles en los ejes x e y
apellidoLabel = Label(miFrame, text = "Apellido:")
apellidoLabel.grid(row = 1, column = 0, sticky = "w", padx = 10, pady = 10)
direccionLabel = Label(miFrame, text = "Direccion:")
direccionLabel.grid(row = 2, column = 0, sticky = "w", padx = 10, pady = 10)
passLabel = Label(miFrame, text = "Password:")

```

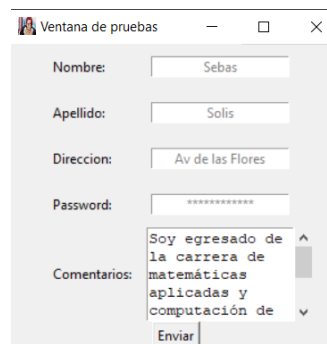
```

passLabel.grid(row = 3, column = 0, sticky = "w", padx = 10, pady = 10)
comentariosLabel = Label(miFrame, text = "Comentarios:")
comentariosLabel.grid(row = 4, column = 0, sticky = "w", padx = 10, pady = 10)
def codigoBoton(): ← Define una función “codigoBoton” que establece el valor de la
    minombre.set("Sebas") variable “minombre” a Sebas
boton = Button(raiz, text = "Enviar", command = codigoBoton) ← Crea un botón en la
ventana principal con el texto “Enviar” y asigna la función
“codigoBoton” para que se ejecute cuando se haga clic en el botón

boton.pack()
raiz.mainloop()

```

Resultado:



Ejemplo 5:

```

from tkinter import *
raiz = Tk()
VarOpcion = IntVar() ← Es una clase que crea una variable entera que se utilizará
para rastrear el estado de los botones de radio

def imprimir(): ← Esta función actualiza el texto de una etiqueta (“etiqueta”) según el
    if VarOpcion.get() == 1: valor de “VarOpcion”
        etiqueta.config(text = "Has elegido masculino")
    elif VarOpcion.get() == 2:
        etiqueta.config(text = "Has elegido femenino")
    elif VarOpcion.get() == 3:
        etiqueta.config(text = "Has elegido otras opciones")

```

Label(raiz, text = "Género:").pack() ← **Crea una etiqueta con el texto “Género:” y la coloca en la ventana principal**

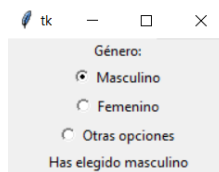
Radiobutton(raiz, text = "Masculino", variable = VarOpcion, value = 1, command = imprimir).pack() ← **Crea un botón de radio con el texto “Masculino”, asocia su valor a 1, lo vincula a la variable “VarOpcion”, y llama a la función imprimir cuando se selecciona. Luego, lo coloca en la ventana principal**

Radiobutton(raiz, text = "Femenino", variable = VarOpcion, value = 2, command = imprimir).pack()

Radiobutton(raiz, text = "Otras opciones", variable = VarOpcion, value = 3, command = imprimir).pack()

etiqueta = Label(raiz) ← **Crea una etiqueta vacía y la coloca en la ventana principal**
etiqueta.pack()
raiz.mainloop()

Resultado:



Nota: En el código, la función “imprimir” se define antes de que se cree la variable etiqueta, pero esto no causa un error inmediatamente porque la función no se ejecuta en el momento de su definición. Se ejecutará más tarde, cuando se haga clic en un Radiobutton.

Ejemplo 6:

```
from tkinter import *  
raiz = Tk()  
playa = IntVar() ← Crea 3 variables de control para los “Checkbuttons”, que se usarán  
montaña = IntVar() para almacenar el estado (seleccionado o no seleccionado)  
turismo = IntVar() de cada opción de destino  
def opcionesViaje(): ← Define una función que se ejecuta cada vez que se hace clic en  
    opcionEscogida = "" alguno de los “Checkbuttons”. Esta función verifica el  
    if(playa.get() == 1): estado de cada variable “IntVar”. Si alguna de estas  
        opcionEscogida += " Playa" variables tiene un valor de 1 (indicando que el
```

```

if(montaña.get() == 1):
    opcionEscogida += " Montaña"
if(turismo.get() == 1):
    opcionEscogida += " Turismo"
    textoFinal.config(text = "Destinos elegidos: " + opcionEscogida)
foto = PhotoImage(file = "avion.png")
Label(raiz, image = foto).pack()
frame = Frame(raiz)
frame.pack()
Label(frame, text = "Elige tu destino:", width = 50).pack()
Checkbutton(frame, text = "Playa", variable = playa, onvalue = 1, offvalue = 0, command =
opcionesViaje).pack()
Checkbutton(frame, text = "Montaña", variable = montaña, onvalue = 1, offvalue = 0,
command = opcionesViaje).pack()
Checkbutton(frame, text = "Turismo", variable = turismo, onvalue = 1, offvalue = 0,
command = opcionesViaje).pack()
textoFinal = Label(frame, text = "Destinos elegidos: ")
textoFinal.pack()
raiz.mainloop()

```

Resultado:

“Checkbutton” está marcado), concatena el nombre del destino a la cadena “opcionEscogida”.

Luego configura el texto del “textoFinal” para mostrar

los destinos elegidos

← Cada Checkbutton está asociado a una de las variables

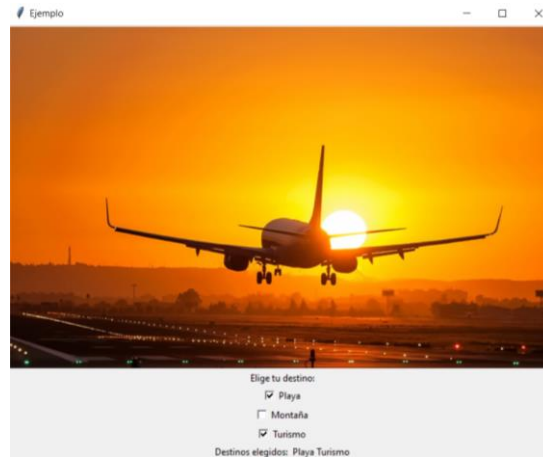
“IntVar” (playa, montaña, turismo) y ejecutará la función

“opcionesViaje” cada vez que se marque o desmarque

← Este Label se actualizará

dinámicamente por la función “opcionesViaje”

para mostrar los destinos seleccionados



Ejemplo 7:

```
from tkinter import *

from tkinter import messagebox  ← Se importa el módulo “messagebox” para mostrar
                                cuadros de diálogo

raiz = Tk()

def infoAdicional():  ← Esta función muestra un cuadro de mensaje informativo con el
                      título “Procesador de Sebas” y el mensaje “Procesador de textos versión 2024”
    messagebox.showinfo("Procesador de Sebas", "Procesador de textos versión 2024")

def avisoLicencia():  ← Esta función muestra un cuadro de advertencia con el título
                      “Licencia” y el mensaje “Producto bajo licencia GNU”
    messagebox.showwarning("Licencia", "Producto bajo licencia GNU")

def salirAplicacion():  ← Esta función muestra un cuadro de pregunta con el título
                        “Salir” y el mensaje “¿Deseas salir de la aplicación?”. Si el
                        usuario responde “yes”, se cierra la aplicación utilizando raiz.destroy()

    valor = messagebox.askquestion("Salir", "¿Deseas salir de la aplicación?")
    # valor = messagebox.askokcancel("Salir", "¿Deseas salir de la aplicación?")
    if valor == "yes":
        raiz.destroy()

def cerrarDocumento():  ← Esta función muestra un cuadro de pregunta con el título
                        “Reintentar” y el mensaje “No es posible cerrar el documento”.
                        Si el usuario elige “Cancelar”, se cierra la aplicación utilizando raiz.destroy()
```

```

valor = messagebox.askretrycancel("Reintentar", "No es posible cerrar el documento")

if valor == False:
    raiz.destroy()

barraMenu = Menu(raiz)
raiz.config(menu=barraMenu, width=300, height=300)

archivoMenu = Menu(barraMenu, tearoff=0) ← Se crea un submenú “Archivo” con
archivoMenu.add_command(label="Nuevo")      varias opciones: “Nuevo”, “Guardar”,
archivoMenu.add_command(label="Guardar")      “Guardar como”, “Cerrar” y “Salir”
archivoMenu.add_command(label="Guardar como”) “Cerrar” está asociado con
archivoMenu.add_separator()                  la función cerrarDocumento y “Salir” con la
archivoMenu.add_command(label="Cerrar", command=cerrarDocumento) función
archivoMenu.add_command(label="Salir", command=salirAplicacion) salirAplicacion

archivoEdicion = Menu(barraMenu, tearoff=0)
archivoEdicion.add_command(label="Copiar")
archivoEdicion.add_command(label="Cortar")
archivoEdicion.add_command(label="Pegar")

archivoHerramientas = Menu(barraMenu, tearoff=0) ← Se crea un submenú
                                                “Herramientas” (actualmente vacío)

archivoAyuda = Menu(barraMenu, tearoff=0)
archivoAyuda.add_command(label="Licencia", command=avisoLicencia)
archivoAyuda.add_command(label="Acercar de", command=infoAdicional)

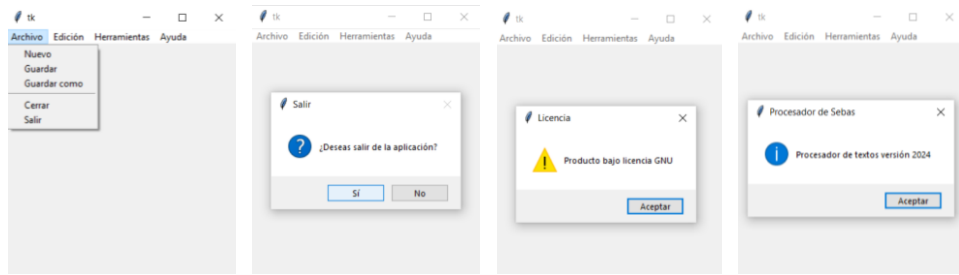
barraMenu.add_cascade(label="Archivo", menu=archivoMenu) ← Se añaden los
                                                         submenús “Archivo”, “Edición”, “Herramientas”
                                                         y “Ayuda” a la barra de menú principal

barraMenu.add_cascade(label="Edición", menu=archivoEdicion)
barraMenu.add_cascade(label="Herramientas", menu=archivoHerramientas)
barraMenu.add_cascade(label="Ayuda", menu=archivoAyuda)

raiz.mainloop()

```

Resultado:



Ejemplo 8:

```
from tkinter import *
```

```
from tkinter import filedialog
```

```
raiz = Tk()
```

```
def abreFichero():
```

```
    fichero = filedialog.askopenfilename(title = "Abrir", initialdir = "Documents", filetypes =  
    (("Ficheros de Excel", "*.xlsx"), ("Ficheros de texto", "*.txt"), ("Todos los ficheros", "*.*")))
```

Abre un cuadro de diálogo que permitirá al usuario seleccionar un archivo.

Se configura con el título “Abrir”, inicia en el directorio “Documents”, y muestra opciones para seleccionar archivos Excel, archivos de texto y

todos los archivos

```
    print(fichero)
```

```
Button(raiz, text = "Abrir fichero", command = abreFichero).pack()
```

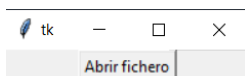
“command=abreFichero” especifica que al hacer clic

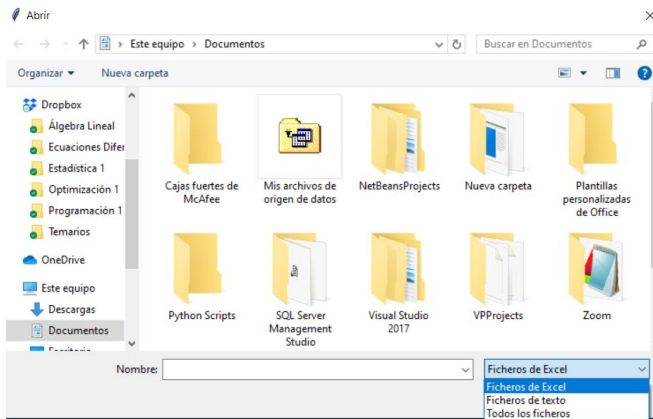
en el botón se llamará a la función abreFichero()

definida anteriormente

```
raiz.mainloop()
```

Resultado:





Capítulo 16: Bases de datos

¿Qué es SQLite?

SQLite es un sistema de gestión de bases de datos relacional pequeño, rápido y fácil de usar, contenido en una biblioteca en C. A diferencia de otras bases de datos, no necesita un servidor separado para funcionar. En lugar de eso, guarda toda la información en un solo archivo en tu computadora.

Características de SQLite:

- 1-. Integrado:** SQLite es una biblioteca que se integra directamente en el programa que la utiliza. No requiere una instalación de servidor o configuración adicional.
- 2-. Ligero:** El nombre SQLite proviene del hecho de que es "ligero" en términos de requisitos de configuración, administración y recursos.
- 3-. Compatible con ACID:** SQLite es compatible con las propiedades ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad) de las transacciones, asegurando que las operaciones de la base de datos se completen de manera segura.
- 4-. Uso común:** SQLite se usa comúnmente en aplicaciones móviles (como iOS y Android), navegadores web (como Firefox y Chrome), sistemas operativos (como Windows y MacOS) y muchas otras aplicaciones que requieren una base de datos integrada.
- 5-. Portabilidad:** Las bases de datos se almacenan en archivos individuales, lo que facilita su transporte y copia.

Desventajas de SQLite:

- 1-. Capacidad limitada:** SQLite no está diseñado para manejar grandes volúmenes de datos o alta concurrencia.
- 2-. Concurrencia limitada:** SQLite tiene limitaciones en la concurrencia de escrituras. Solo una transacción de escritura puede ocurrir a la vez, lo que puede causar cuellos de botella en aplicaciones con muchas operaciones de escritura simultáneas.

3-. Seguridad limitada: SQLite no tiene un sistema de usuarios y permisos tan completo como otros sistemas de bases de datos. No es adecuado para aplicaciones donde la seguridad de los datos es crítica.

4-. Falta de ciertas características avanzadas: SQLite carece de algunas características avanzadas que se encuentran en otros sistemas de bases de datos, como la replicación, el clustering, el almacenamiento de procedimientos y la seguridad a nivel de usuario.

Para conectar una base de datos a una aplicación, generalmente se siguen estos pasos básicos:

Ejemplo 1:

```
import sqlite3 ← Importa la librería estándar de Python sqlite3, que proporciona una
                interfaz para trabajar con bases de datos SQLite desde Python

miConexion = sqlite3.connect("Primera Base") ← Crea una conexión con una base de
                datos SQLite llamada "Primera Base". Si la base de
                datos no existe, SQLite la creará automáticamente

miCursor = miConexion.cursor() ← Crea un objeto cursor (miCursor) que se utiliza
                para ejecutar comandos SQL y manipular datos en la base de datos

# miCursor.execute("CREATE TABLE Productos (Nombre_articulo VARCHAR(50), Precio
INTEGER, Seccion VARCHAR(20))") ← Crea una tabla llamada "Productos" en la
                base de datos, con tres columnas y sus diferentes tipos de datos

miCursor.execute("INSERT INTO Productos VALUES('Balón', 15, 'Deportes')") ←
                Ejecuta una sentencia SQL para insertar un
                nuevo registro en la tabla "Productos"

miConexion.commit() ← Guarda los cambios realizados en la base de datos. En este
                caso, confirma la inserción del nuevo registro en la tabla "Productos"

miConexion.close() ← Cierra la conexión con la base de datos SQLite
```

Resultado:

Al ejecutar el programa, se creará automáticamente un archivo que será la base de datos.

Escritorio > Python > Base_de_datos			
Nombre	Fecha de modificación	Tipo	Tamaño
BDD	22/06/2024 11:16 p. m.	Python File	1 KB
Primera Base	22/06/2024 11:16 p. m.	Archivo	0 KB

Nota: Si la tabla "Productos" ya existe en la base de datos "Primera Base", ejecutar nuevamente la sentencia CREATE TABLE generaría un error porque no se pueden crear tablas con el mismo nombre más de una vez.

Ejemplo 2:

```
import sqlite3

miConexion = sqlite3.connect("Primera Base")

miCursor = miConexion.cursor()

variosProductos = [ ← Es una lista de tuplas, donde cada tupla contiene tres valores
                    que representan un producto
                    ("Camiseta", 10, "Deportes"),
                    ("Jarrón", 90, "Cerámica"),
                    ("Camión", 20, "Juguetería")
]

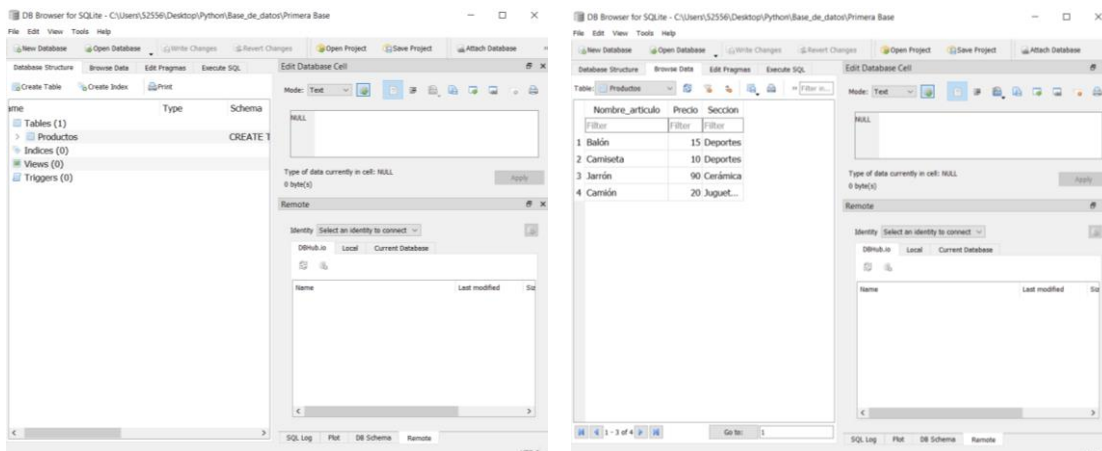
miCursor.executemany("INSERT INTO Productos VALUES(?,?,?)", variosProductos) ←
Ejecuta una sentencia SQL para insertar múltiples
registros en la tabla "Productos"

miConexion.commit()

miConexion.close()
```

Resultado:

Para poder visualizar nuestra base de datos, tendremos que instalar un visor de SQLite. Para ello, debemos ir al siguiente enlace: [Descargas - Navegador de base de datos para SQLite \(sqlitebrowser.org\)](http://sqlitebrowser.org) y descargar la opción que más nos convenga. Luego, simplemente seguimos las instrucciones de instalación haciendo clic en "Next" en cada paso de la configuración.



Ejemplo 3:

```
import sqlite3

miConexion = sqlite3.connect("Primera Base")
```

```

miCursor = miConexion.cursor()
miCursor.execute("SELECT * FROM Productos") ← Ejecuta una consulta SQL para
                                              seleccionar todos los registros de la tabla
variosProductos = miCursor.fetchall() ← Recupera todos los registros resultantes de la
                                         consulta anterior y los almacena en la variable
                                         “variosProductos” como una lista de tuplas

print(variosProductos)
miConexion.commit()
miConexion.close() →
[('Balón', 15, 'Deportes'), ('Camiseta', 10, 'Deportes'), ('Jarrón', 90, 'Cerámica'), ('Camión', 20,
'Juguetería')]

```

Ejemplo 4:

```

import sqlite3

miConexion = sqlite3.connect("Primera Base")
miCursor = miConexion.cursor()
miCursor.execute("SELECT * FROM Productos")
variosProductos = miCursor.fetchall()

for producto in variosProductos: ← Inicia un bucle for que itera sobre cada registro en
                                   “variosProductos”

    print("Nombre del artículo: ", producto[0], "Sección: ", producto[2]) ← Esta línea
                                   imprime el nombre del artículo y la sección de cada registro.
                                   Se asume que producto[0] corresponde al nombre del
                                   artículo y producto[2] a la sección

miConexion.commit()
miConexion.close() →

```

```

Nombre del artículo: Balón Sección: Deportes
Nombre del artículo: Camiseta Sección: Deportes
Nombre del artículo: Jarrón Sección: Cerámica
Nombre del artículo: Camión Sección: Juguetería

```

Ejemplo 5:

```

import sqlite3

```

```
miConexion = sqlite3.connect("Gestión de productos")
```

```
miCursor = miConexion.cursor()
```

```
miCursor.execute(""" ← Ejecuta una consulta SQL para crear una tabla
```

```
CREATE TABLE Productos (  
ID INTEGER PRIMARY KEY AUTOINCREMENT,  
Nombre_articulo VARCHAR(50) UNIQUE,  
Precio INTEGER,  
Seccion VARCHAR(20)  
""
```

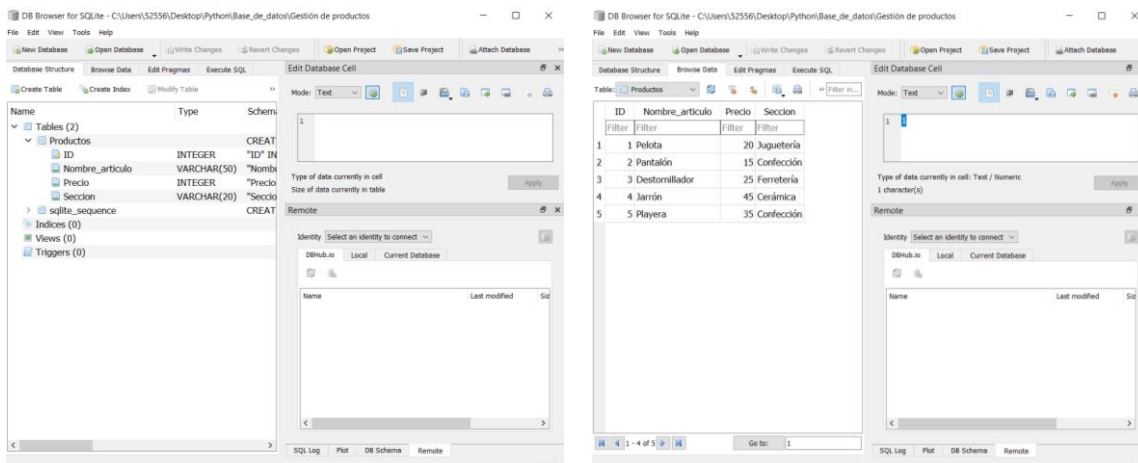
```
variosProductos = [ ← Define una lista de tuplas llamada “variosProductos”. Cada  
("Pelota", 20, "Juguetería"), tupla representa un producto con  
("Pantalón", 15, "Confección"), su nombre, precio y sección  
("Destornillador", 25, "Ferretería"),  
("Jarrón", 45, "Cerámica"),  
("Playera", 35, "Confección")  
]
```

```
miCursor.executemany("INSERT INTO Productos VALUES(NULL,?,?,?)", variosProductos) ←
```

```
miConexion.commit() Usa el método “executemany” del cursor para insertar  
varias filas en la tabla “Productos”. Los valores NULL
```

```
miConexion.close() son para la columna ID que se autoincrementa automáticamente
```

Resultado:



Ejemplo 6:

```
import sqlite3

miConexion = sqlite3.connect("Gestión de productos")

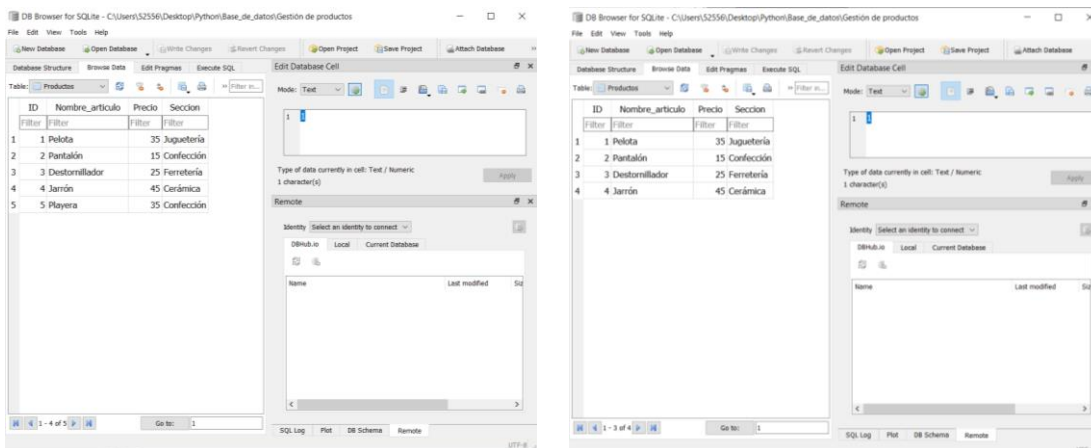
miCursor = miConexion.cursor()

miCursor.execute("UPDATE Productos SET Precio = 35 WHERE Nombre_articulo = 'Pelota'") ← Ejecutaría una consulta SQL para actualizar el precio de un producto llamado "Pelota" a 35

# miCursor.execute("DELETE FROM Productos WHERE ID = 5") ← Ejecutaría una consulta SQL para eliminar un producto con ID = 5

miConexion.commit()

miConexion.close()
```



Capítulo 17: Funciones Lambda, Filter y Map

Una **función lambda** en Python es una forma de definir funciones anónimas pequeñas. Se usan comúnmente para operaciones simples y rápidas.

La sintaxis básica de la función es:

lambda argumentos: expresión ← Solo puede tener una expresión

↑
Puede tener cualquier número de argumentos

La expresión es evaluada y devuelta cuando se llama la función.

Ejemplo 1:

```
def area_triangulo(base, altura):
    return (base * altura) / 2

area_triangulo = lambda base, altura: (base * altura) / 2
triangulo_1 = area_triangulo(5, 7)
```

```

triangulo_1 = area_triangulo(5,7)    triangulo_2 = area_triangulo(9,6)
triangulo_2 = area_triangulo(9,6)    print(triangulo_1)
print(triangulo_1)                    print(triangulo_2)
print(triangulo_2) —————> 17.5

```

27

Ejemplo 2:

```

al_cubo = lambda numero:pow(numero, 3)
print(al_cubo(13)) —————> 2197

```

Ejemplo 3:

```

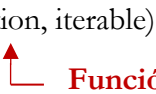
destacar_valor = lambda comision:"¡{}! $".format(comision)
comision_Ana = 15585
print(destacar_valor(comision_Ana)) —————> ¡15585! $

```

Las funciones lambda se usan comúnmente con funciones de orden superior como `map()`, `filter()`, y `sorted()`.

La **función `filter()`** en Python se utiliza para construir una nueva lista (o cualquier iterable) a partir de aquellos elementos de una secuencia (como una lista, tupla, etc.) que cumplan con una condición especificada por una función. La función que se pasa a `filter()` debe devolver `True` o `False` para cada elemento de la secuencia.

La sintaxis básica de la función es:

`filter (function, iterable)`  **Secuencia que se va a filtrar**
Función que toma un solo argumento y devuelve un valor booleano

Ejemplo 1:

```

def numero_par(num):
    if num % 2 == 0:
        return True

numeros = [17,24,7,39,8,51,92]
print(list(filter(numero_par, numeros))) —————> [24, 8, 92]

```

Usando lambda quedaría:

```

numeros = [17,24,7,39,8,51,92]
print(list(filter(lambda numero_par: numero_par % 2 == 0, numeros)))

```

Ejemplo 2:

```
class Empleado:
```

```
    def __init__(self, nombre, cargo, salario):
```

```
        self.nombre = nombre
```

```
        self.cargo = cargo
```

```
        self.salario = salario
```

```
    def __str__(self):
```

```
        return "{} que trabaja como {} tiene un salario de ${}".format(self.nombre, self.cargo, self.salario)
```

```
lista_empleados = [
```

```
Empleado("Juan", "Director", 75000),
```

```
Empleado("Ana", "Presidenta", 85000),
```

```
Empleado("Antonio", "Administrativo", 25000),
```

```
Empleado("Sara", "Secretaria", 27000),
```

```
Empleado("Mario", "Botones", 21000)
```

```
]
```

```
salarios_altos = filter(lambda Empleado:Empleado.salario > 50000, lista_empleados)
```

```
for empleado_salario in salarios_altos:
```

```
    print(empleado_salario) —————>
```

Juan que trabaja como Director tiene un salario de \$75000

Ana que trabaja como Presidenta tiene un salario de \$85000

La **función map** en Python es una función incorporada que aplica una función dada a todos los elementos de una lista (o cualquier iterable) y devuelve un mapa (map object) con los resultados.

La sintaxis básica de la función es:

map (function, iterable, ...)  **Iterable cuyos elementos se aplicaran a la función**

 **Función que se aplica a cada elemnto del iterable**

Ejemplo 1:

```
class Empleado:
```

```
    def __init__(self, nombre, cargo, salario):
```

```

        self.nombre = nombre

        self.cargo = cargo

        self.salario = salario

    def __str__(self):

        return "{} que trabaja como {} tiene un salario de ${}".format(self.nombre,
self.cargo, self.salario)

lista_empleados = [

Empleado("Juan", "Director", 6700),

Empleado("Ana", "Presidenta", 7500),

Empleado("Antonio", "Administrativo", 2100),

Empleado("Sara", "Secretaria", 2150),

Empleado("Mario", "Botones", 1800)

]

def calculo_comision(empleado):

    if(empleado.salario <= 3000):

        empleado.salario = empleado.salario * 1.03

    return empleado

lista_empleados_comision = map(calculo_comision, lista_empleados)

for empleado in lista_empleados_comision:

    print(empleado) ➡

```

Juan que trabaja como Director tiene un salario de \$6700

Ana que trabaja como Presidenta tiene un salario de \$7500

Antonio que trabaja como Administrativo tiene un salario de \$2163.0

Sara que trabaja como Secretaria tiene un salario de \$2214.5

Mario que trabaja como Botones tiene un salario de \$1854.0

Capítulo 18: Expresiones regulares

Las **expresiones regulares**, a menudo abreviadas como regex o regexp, son secuencias de caracteres que forman un patrón de búsqueda. Estas se utilizan principalmente para la

manipulación de cadenas de texto, como la búsqueda, coincidencia y reemplazo de patrones específicos.

Conceptos básicos de expresiones regulares

1-. Metacaracteres:

- Son caracteres con un significado especial. Por ejemplo, . (punto) coincide con cualquier carácter excepto una nueva línea.
- Otros metacaracteres comunes incluyen: ^, \$, *, +, ?, {}, [], (), |, \.

2-. Anclas:

^: Coincide con el inicio de una línea.

\$: Coincide con el final de una línea.

3-. Cuantificadores:

*: Coincide con cero o más repeticiones del patrón anterior.

+: Coincide con una o más repeticiones del patrón anterior.

?: Coincide con cero o una repetición del patrón anterior.

{n,m}: Coincide con al menos n y no más de m repeticiones del patrón anterior.

2-. Grupos y Rango:

(abc): Coincide con la secuencia exacta "abc".

[abc]: Coincide con cualquier carácter dentro de los corchetes, en este caso 'a', 'b' o 'c'.

[a-z]: Coincide con cualquier carácter en el rango de 'a' a 'z'.

3-. Clases de Carácter:

\d: Coincide con cualquier dígito (equivalente a [0-9]).

\w: Coincide con cualquier carácter de palabra (letras, dígitos y guiones bajos).

\s: Coincide con cualquier carácter de espacio en blanco (espacios, tabulaciones, nuevas líneas).

4-. Negación:

\D: Coincide con cualquier carácter que no sea un dígito.

\W: Coincide con cualquier carácter que no sea de palabra.

\S: Coincide con cualquier carácter que no sea de espacio en blanco.

[^abc]: Coincide con cualquier carácter que no sea 'a', 'b' o 'c'.

Ejemplo 1:

import re ← **El módulo re proporciona soporte para expresiones regulares**

```
cadena = "Vamos a aprender expresiones regulares"
```

```
print(re.search("aprender", cadena)) →
```

```
<re.Match object; span=(8, 16), match='aprender'>
```

Nota: Si la palabra no es encontrada, el programa devolverá “none”.

Ejemplo 2:

```
import re
```

```
cadena = "Vamos a aprender expresiones regulares"
```

```
textoBuscar = "aprender"
```

```
if re.search(textoBuscar, cadena) is not None:
```

```
    print("He encontrado el texto")
```

```
else:
```

```
    print("No he encontrado el texto") → He encontrado el texto
```

Ejemplo 3:

```
import re
```

```
cadena = "Vamos a aprender expresiones regulares"
```

```
textoBuscar = "aprender"
```

```
textoEncontrado = re.search(textoBuscar, cadena)
```

```
print(textoEncontrado.start())
```

```
print(textoEncontrado.end())
```

```
print(textoEncontrado.span()) → 8
```

```
16
```

```
(8, 16)
```

Ejemplo 4:

```
import re
```

```
cadena = "Vamos a aprender expresiones regulares en Python. Python es un lenguaje de  
sintaxis sencilla"
```

```
textoBuscar = "Python"
```

```
print(len(re.findall(textoBuscar, cadena))) → 2
```

Ejemplo 5:

```
import re

lista_nombres = ['Ana Gómez','María Martín','Sandra López','Santiago Martín','Sandra
Fernández']

for elemento in lista_nombres:

    if re.findall('Martín$', elemento):

        print(elemento)

#     if re.findall('^Sandra', elemento): —————> María Martín
Santiago Martín
```

Ejemplo 6:

```
import re

lista_nombres = ['http://informaticaenespaña.es','http://pildorasinformaticas.es',
'http://pildorasinformaticas.com','ftp://pildorasinformaticas.com']

for elemento in lista_nombres:

    if re.findall('[ñ]', elemento):

        print(elemento) —————> http://informaticaenespaña.es
```

Ejemplo 7:

```
import re

lista_nombres = ['hombres','mujeres','mascotas','niños','niñas']

for elemento in lista_nombres:

    if re.findall('niñ[oa]s', elemento):

        print(elemento) —————> niños

niñas
```

Ejemplo 8:

```
import re

lista_nombres = ['Ana','Pedro','María','Sandra','Celia']

for elemento in lista_nombres:

    if re.findall('^[O-T]', elemento):

        print(elemento) —————> Pedro
```

Sandra

Ejemplo 9:

```
import re

lista_nombres = ['Ma1','Se1','Ma2','Ba1','Ma3','Va1','Va2','Ma4']

for elemento in lista_nombres:
    if re.findall('Ma^[1-3 ]', elemento):
        print(elemento) —————> Ma4
```

Ejemplo 10:

```
import re

lista_nombres = ['Ma1','Se1','Ma2','Ba1','Ma3','Va1','Va2','Ma4','MaA','Ma5','MaB','MaC']

for elemento in lista_nombres:
    if re.findall('Ma[1-3A-B]', elemento):
        print(elemento) —————> Ma1
```

Ma2

Ma3

MaA

MaB

Ejemplo 11:

```
import re

Nombre1 = "Sandra López"
Nombre2 = "Antonio Gómez"
Nombre3 = "sandra López"
Nombre4 = "Jara López"
Nombre5 = "Lara López"

if re.match(".ara", Nombre4, re.IGNORECASE):
    print("Hemos encontrado el nombre")
else:
    print("No he encontrado el nombre") —————> Hemos encontrado el nombre
```

Ejemplo 12:

```
import re

Cadena1 = "Jara López"
Cadena2 = "546546546"
Cadena3 = "a54654654"

# if re.match("\d", Cadena2):
    print("Hemos encontrado el valor")
else:
    print("No he encontrado el valor") —————> Hemos encontrado el valor
```

Capítulo 19: Decoradores y funciones decoradoras

¿Qué son los decoradores?

Los **decoradores** son una forma poderosa y elegante de modificar el comportamiento de funciones o métodos. Un decorador es esencialmente una función que toma otra función y extiende su comportamiento sin modificarla explícitamente. Los decoradores se utilizan a menudo para añadir funcionalidades adicionales a una función existente de una manera modular y reutilizable.

En otras palabras, es una función que recibe otra función como argumento y devuelve una nueva función que usualmente extiende o altera el comportamiento de la función original. Los decoradores se aplican utilizando la sintaxis `@decorador` justo antes de la definición de la función que deseas decorar.

La sintaxis básica de la función es:

```
def decorador(funcion): ← Función A (definida) y función B (parámetro)

    def nueva_funcion(*args, **kwargs): ← Función C (interna)

        resultado = funcion(*args, **kwargs) ← Código que se ejecuta antes de la función

        return resultado ← Código que se ejecuta después de la función

    return nueva_funcion
```

Ejemplo 1:

```
def funcion_decoradora(funcion_parametro):

    def funcion_interior():

        print("Vamos a realizar un cálculo: ")

        funcion_parametro()
```

```

        print("Hemos terminado el cálculo.")

    return funcion_interior

@funcion_decoradora
def suma():
    print(15 + 20)

@funcion_decoradora
def resta():
    print(30 - 10)

suma()
resta() ➡ Vamos a realizar un cálculo:
35

Hemos terminado el cálculo.
Vamos a realizar un cálculo:
20

Hemos terminado el cálculo.

```

Ejemplo 2:

```

def funcion_decoradora(funcion_parametro):

    def funcion_interior(*args, **kwargs):

        print("Vamos a realizar un cálculo:")

        funcion_parametro(*args, **kwargs)

        print("Hemos terminado el cálculo.")

    return funcion_interior

@funcion_decoradora
def suma(num1, num2, num3):
    print(num1 + num2 + num3)

@funcion_decoradora
def resta(num1, num2):
    print(num1 - num2)

def potencia(base, exponente):

```

```
print(pow(base, exponente))
```

suma(7, 5, 7)

resta(12, 10)

potencia(base = 5,exponente = 3) —————> Vamos a realizar un cálculo:

19

Hemos terminado el cálculo.

Vamos a realizar un cálculo:

2

Hemos terminado el cálculo.

125

Capítulo 20: Documentación

La **documentación** en el contexto de la programación y el desarrollo de software se refiere a la información escrita que explica cómo funciona el software, cómo usarlo y cómo mantenerlo. Esta documentación facilita la comprensión y el uso del código tanto para los desarrolladores como para los usuarios finales.

Tipos de Documentación

1-. Documentación del código:

- **Comentarios:** Anotaciones dentro del código fuente que explican qué hace el código. Ayudan a otros desarrolladores (y a ti mismo en el futuro) a entender la lógica y el propósito del código.
- **Docstrings:** Comentarios especiales en Python que se utilizan para describir el propósito de una función, clase o módulo. Se colocan al principio de la definición.

2-. Documentación Técnica:

- **Manual de referencia:** Explica en detalle cómo utilizar cada parte del software. Incluye descripciones de funciones, métodos, clases, parámetros y ejemplos de uso.
- **Guías del usuario:** Documentos que ayudan a los usuarios finales a entender cómo utilizar el software. Incluyen instrucciones paso a paso, tutoriales, y ejemplos prácticos.
- **Guías de instalación:** Instrucciones sobre cómo instalar y configurar el software en diferentes entornos.
- **Especificaciones técnicas:** Detalles técnicos sobre cómo está construido el software, incluyendo diagramas de arquitectura, especificaciones de API, y protocolos de comunicación.

3-. Documentación de Desarrollo:

- **Guías de contribución:** Instrucciones para desarrolladores que quieren contribuir al proyecto. Incluyen estándares de codificación, procedimientos para enviar contribuciones, y pautas de revisión de código.
- **Registro de cambios (changelog):** Lista de cambios y actualizaciones realizadas en cada versión del software. Ayuda a los usuarios y desarrolladores a seguir el desarrollo del software y conocer las novedades y correcciones de errores.
- **Planificación y diseño:** Documentos que detallan la planificación y el diseño del software, incluyendo hojas de ruta, especificaciones de características, y decisiones de diseño.

Importancia de la Documentación

- **Facilita la colaboración:** La documentación clara y completa permite a múltiples desarrolladores trabajar juntos en un proyecto sin malentendidos.
- **Mejora la mantenibilidad:** Ayuda a los desarrolladores a entender el código existente y a realizar cambios y mejoras sin introducir errores.
- **Ayuda en la formación y capacitación:** Permite a los nuevos miembros del equipo ponerse al día rápidamente con el proyecto.
- **Soporte al usuario:** Los usuarios finales pueden aprender a utilizar el software de manera efectiva y resolver problemas comunes por sí mismos.
- **Comunicación:** Actúa como una fuente de verdad sobre cómo funciona el software, evitando confusiones y malentendidos.

Buenas Prácticas

- 1-. **Ser claro y conciso:** Escribir de manera clara y directa, evitando jerga técnica innecesaria.
- 2-. **Mantenerla actualizada:** Asegurarse de que la documentación refleje siempre el estado actual del software.
- 3-. **Usar ejemplos:** Incluir ejemplos prácticos que demuestren cómo utilizar las funciones y características del software.
- 4-. **Estructurarla bien:** Organizar la documentación de manera lógica y fácil de navegar.
- 5-. **Automatizar la generación de documentación:** Utilizar herramientas que generen documentación a partir del código, como Sphinx para Python, para mantener la documentación sincronizada con el código.

Ejemplo 1:

```
def areaCuadrado(lado):  
    """Calcula el área de un cuadrado elevando al  
    cuadrado el lado pasado por parámetro """
```



```

    return "El área del cuadrado es: " + str(lado * lado)

def areaTriangulo(base, altura):

    return "El área del triángulo es: " + str((base * altura) / 2)

print(areaCuadrado.__doc__) ➡

```

```

Calcula el área de un cuadrado elevando al
cuadrado el lado pasado por parámetro

```

Ejemplo 2:

```

def areaCuadrado(lado):

    """Calcula el área de un cuadrado elevando al
cuadrado el lado pasado por parámetro """

    return "El área del cuadrado es: " + str(lado * lado)

def areaTriangulo(base, altura):

    """Calcula el área de un triángulo utilizando los
parámetros base y altura """

    return "El área del triángulo es: " + str((base * altura) / 2)

help(areaCuadrado)

help(areaTriangulo) ➡

```

```

Help on function areaCuadrado in module __main__:
areaCuadrado(lado)
  Calcula el área de un cuadrado elevando al
  cuadrado el lado pasado por parámetro

Help on function areaTriangulo in module __main__:
areaTriangulo(base, altura)
  Calcula el área de un triángulo utilizando los
  parámetros base y altura

```

Ejemplo 3:

```

class Areas:

    """Esta clase calcula las área de diferentes
figuras geométricas """

    def areaCuadrado(lado):

        """Calcula el área de un cuadrado elevando al
cuadrado el lado pasado por parámetro """

        return "El área del cuadrado es: " + str(lado * lado)


```

```
def areaTriangulo(base, altura):

    """Calcula el área de un triángulo utilizando los

    parámetros base y altura """

    return "El área del triángulo es: " + str((base * altura) / 2)
```

help(Areas.areaCuadrado) 

```
Help on function areaCuadrado in module __main__:

areaCuadrado(lado)
    Calcula el área de un cuadrado elevando al
    cuadrado el lado pasado por parámetro
```

Si en lugar de la última instrucción, pusieramos “help(Areas)”, obtendríamos información mucho más amplia y explícita sobre todas y cada una de las funciones pertenecientes a esa clase.

```
Help on class Areas in module __main__:

class Areas(builtins.object)
    Esta clase calcula las área de diferentes
    figuras geométricas

    Methods defined here:

    areaCuadrado(lado)
        Calcula el área de un cuadrado elevando al
        cuadrado el lado pasado por parámetro

    areaTriangulo(base, altura)
        Calcula el área de un triángulo utilizando los
        parámetros base y altura

    -----
    Data descriptors defined here:

    __dict__
        dictionary for instance variables (if defined)

    __weakref__
        list of weak references to the object (if defined)
```

Ejemplo 4:

```
from Modulos import Funciones_matematicas
```

```
class Areas:
```

```
    """Esta clase calcula las área de diferentes

    figuras geométricas """
```

```
    def areaCuadrado(lado):
```

```
        """Calcula el área de un cuadrado elevando al

        cuadrado el lado pasado por parámetro """
```

```
        return "El área del cuadrado es: " + str(lado * lado)
```

```
    def areaTriangulo(base, altura):
```

```
        """Calcula el área de un triángulo utilizando los
```

parámetros base y altura """

return "El área del triángulo es: " + str((base * altura) / 2)

help(Funciones_matematicas) —→

```
Help on module Modulos.Funciones_matematicas in Modulos:

NAME
  Modulos.Funciones_matematicas - Este módulo permite realizar operaciones matemáticas básicas

FUNCTIONS
  multiplicar(op1, op2)
  restar(op1, op2)
  sumar(op1, op2)

FILE
  c:\users\52556\desktop\python\modulos\funciones_matematicas.py
```

Ejemplo 5:

def area_Triangulo(base, altura):

"""

Calcula el área de un triángulo dado

>>> area_Triangulo(3,6)

8.0

"""

return (base * altura) / 2

import doctest

doctest.testmod() —→

```
*****
File "C:\Users\52556\Desktop\Python\Expresiones_regulares.py", line 4, in __main__.area_Triangulo
Failed example:
  area_Triangulo(3,6)
Expected:
  8.0
Got:
  9.0
*****
1 items had failures:
  1 of 1 in __main__.area_Triangulo
***Test Failed*** 1 failures.
```

Ejemplo 6:

def area_Triangulo(base, altura):

"""

Calcula el área de un triángulo dado

>>> area_Triangulo(3,6)

```

    "El área del triángulo es: 9.0"

    """

    return "El área del triángulo es: " + str((base * altura) / 2)

import doctest

doctest.testmod() ➡

```

```

*****
File "C:\Users\52556\Desktop\Python\Expresiones_regulares.py", line 4, in __main__.area_Triangulo
Failed example:
    area_Triangulo(3,6)
Expected:
    "El área del triángulo es: 9.0"
Got:
    'El área del triángulo es: 9.0'
*****
1 items had failures:
  1 of 1 in __main__.area_Triangulo
***Test Failed*** 1 failures.

```

Ejemplo 7:

```

def area_Triangulo(base, altura):

    """

    Calcula el área de un triángulo dado

    >>> area_Triangulo(3,6)

    'El área del triángulo es: 9.0'

    >>> area_Triangulo(4,5)

    'El área del triángulo es: 11.0'

    >>> area_Triangulo(9,3)

    'El área del triángulo es: 13.5'

    """

    return "El área del triángulo es: " + str((base * altura) / 2)

import doctest

doctest.testmod() ➡

```

```

*****
File "C:\Users\52556\Desktop\Python\Expresiones_regulares.py", line 6, in __main__.area_Triangulo
Failed example:
    area_Triangulo(4,5)
Expected:
    'El área del triángulo es: 11.0'
Got:
    'El área del triángulo es: 10.0'
*****
1 items had failures:
  1 of 3 in __main__.area_Triangulo
***Test Failed*** 1 failures.

```

Ejemplo 8:

```
def comprobacion(email):
```

```
    """
```

La función comprobación evalúa un email recibido buscando el carácter '@'. Si contiene una '@', el email es considerado correcto. Si contiene más de una '@' o si la '@' está al final, el email es considerado incorrecto.

```
    >>> comprobacion('sebastiansolvil@gmail.com')
```

```
    True
```

```
    >>> comprobacion('sebastiansolvilgmail.com@')
```

```
    False
```

```
    >>> comprobacion('sebastiansolvilgmail.com')
```

```
    False
```

```
    >>> comprobacion('sebastiansolvil@gmail@.com')
```

```
    False
```

```
    """
```

```
    arroba = email.count('@')
```

```
    if(arroba != 1 or email.rfind('@') == (len(email) - 1) or email.find('@') == 0):
```

```
        return False
```

```
    else:
```

```
        return True
```

```
import doctest
```

```
doctest.testmod()
```

Ejemplo 9:

```
import math
```

```
def RaizCuadrada(lista):
```

```
    """
```

La función devuelve una lista con la raíz cuadrada de los elementos numéricos

pasados por parámetro en otra lista.

```
>>> fila = []
```

```
>>> for i in [4, 9, 16]:
```

```
...     fila.append(i)
```

```
>>> RaizCuadrada(fila)
```

```
[2.0, 3.0, 4.0]
```

```
"""
```

```
    return [math.sqrt(n) for n in lista]
```

```
#print(RaizCuadrada([9, 16, 25, 36]))
```

```
import doctest
```

```
doctest.testmod()
```

Ejemplo 10:

```
import math
```

```
def RaizCuadrada(lista):
```

```
    """
```

La función devuelve una lista con la raíz

cuadrada de los elementos numéricos

pasados por parámetro en otra lista.

```
>>> fila = []
```

```
>>> for i in [4, 9, 16]:
```

```
...     fila.append(i)
```

```
>>> RaizCuadrada(fila)
```

```
[2.0, 3.0, 4.0]
```

```
>>> fila = []
```

```
>>> for i in [4, 9, -16, 50]:
```

```
...     fila.append(i)
```

```
>>> RaizCuadrada(fila)
```

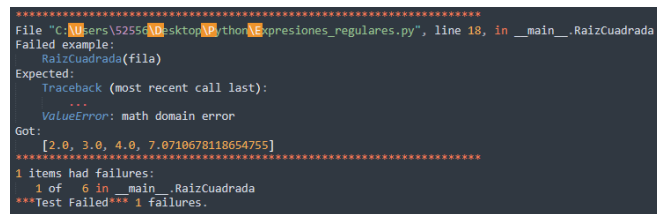
```
Traceback (most recent call last):
```

```

...
ValueError: math domain error
"""

return [math.sqrt(n) for n in lista]
# print(RaizCuadrada([9, 16, 25, 36]))
import doctest
doctest.testmod()

```



```

File "C:\Users\52556\Desktop\Python\Práctica\expresiones_regulares.py", line 18, in __main__.RaizCuadrada
Failed example:
RaizCuadrada(9)
Expected:
[3.0, 4.0, 5.0, 6.0]
Got:
[3.0, 4.0, 5.0, 6.0, 7.0710678118654755]
1 items had failures:
1 of 6 in __main__.RaizCuadrada
***Test Failed*** 1 failures.

```

¿Qué es un ejecutable ?

Un **ejecutable** es un tipo de archivo que contiene un programa en un formato que un sistema operativo puede ejecutar directamente. Estos archivos contienen código binario que la computadora puede leer y ejecutar sin necesidad de interpretación o compilación adicional en tiempo de ejecución. Los ejecutables suelen ser el producto final de un proceso de compilación que transforma el código fuente escrito por los desarrolladores en código binario que puede ser ejecutado por la máquina.

Pasos a realizar para crear un ejecutable

1-. Instalar **pyinstaller** en nuestra consola mediante el siguiente comando

```
>pip install pyinstaller
```

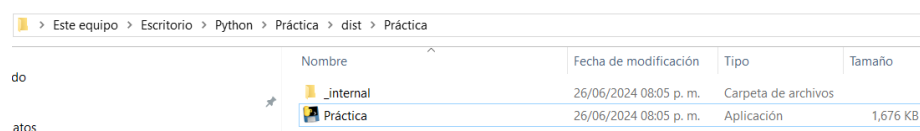
2-. Copiar y pegar la ruta de la carpeta en donde tenga el ejecutable

```
>cd C:\Users\52556\Desktop\Python\Práctica
```

3-. Luego, tendremos que ingresar la palabra del programa seguida del nombre del archivo que queremos convertir a ejecutable.

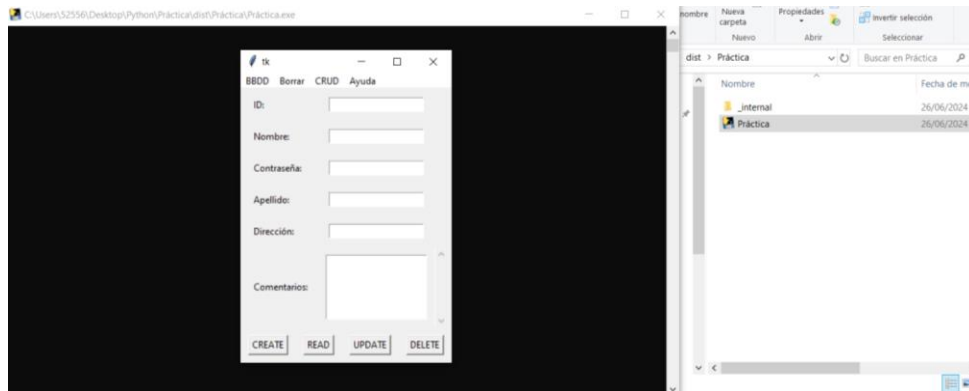
```
>pyinstaller Práctica.py
```

4-. Al finalizar la instalación, nos dirigiremos a la carpeta “dist”, un directorio comúnmente utilizado en proyectos de software para almacenar los archivos de distribución final del proyecto.



Nombre	Fecha de modificación	Tipo	Tamaño
_internal	26/06/2024 08:05 p. m.	Carpeta de archivos	
Práctica	26/06/2024 08:05 p. m.	Aplicación	1,676 KB

5-. Si queremos ver el funcionamiento de nuestra aplicación, basta con hacer doble clic en el ejecutable.



6-. Si queremos que la consola desaparezca al abrir la aplicación, tendremos que agregar un comando.

```
>pyinstaller --windowed Práctica.py
```

7-. Si queremos incluir todos los archivos que acompañan al ejecutable, utilizaremos el siguiente comando.

```
>pyinstaller --windowed --onefile Práctica.py
```

8-. Si queremos que nuestro ejecutable final tenga un logotipo, debemos colocar una imagen en formato .ico en la carpeta del ejecutable y utilizar el siguiente comando.

```
>pyinstaller --windowed --onefile --icon=../nombre del icono y su extensión Práctica.py
```

Capítulo 21: Pandas

Pandas es una biblioteca de Python muy popular para el análisis y la manipulación de datos. Su nombre proviene de “Panel Data”, que se refiere a conjuntos de datos estructurados en tablas bidimensionales, similares a las hojas de cálculo de Excel o a las bases de datos SQL.

Características principales:

1-. Ofrece dos estructuras de datos principales:

- **Series:** Son arreglos unidimensionales que se asemejan a una columna en una tabla y pueden contener datos de cualquier tipo, como enteros o cadenas.
- **DataFrames:** Son tablas bidimensionales compuestas por filas y columnas, similares a una hoja de cálculo de Excel o a una tabla en una base de datos, donde se pueden almacenar datos de diferentes tipos en cada columna.

2-. **Manejo de datos faltantes:** Proporciona herramientas para detectar, eliminar y llenar datos faltantes, facilitando el manejo de conjuntos de datos incompletos.

3-. Filtrado y selección: Permite realizar operaciones de filtrado y selección de datos de manera eficiente, usando condiciones y etiquetas.

4-. Agrupación y resumen: Facilita la agrupación de datos y la realización de operaciones de resumen, como promedios y conteos.

5-. Lectura y escritura de datos: Soporta la lectura y escritura de datos en múltiples formatos, como CSV, Excel, SQL, y más.

6-. Integración con otras bibliotecas: Se integra bien con otras bibliotecas de Python, como NumPy, Matplotlib y Scikit-learn, lo que la hace muy poderosa para el análisis de datos.

Historia de pandas:

Pandas fue creado por **Wes McKinney** en 2008. En ese momento, McKinney trabajaba en una firma de inversión llamada AQR Capital Management y desarrolló pandas para ayudar a analizar grandes volúmenes de datos financieros de una manera eficiente y flexible, ya que no había herramientas adecuadas en Python para esa tarea.

Más tarde, Wes McKinney publicó la biblioteca como código abierto, lo que permitió que la comunidad de desarrolladores contribuyera a su crecimiento y mejora.

Ejercicios prácticos:

1-. Importar la biblioteca de pandas →

```
import pandas as pd
```

2-. Crear una serie a partir de una lista →

```
edades = [22,13,8,33,25,44,38,22]
```

```
serie = pd.Series(edades)
```

```
serie
```

```
0    22
1    13
2     8
3    33
4    25
5    44
6    38
7    22
dtype: int64
```

Nota: También se puede trabajar con un diccionario o un arreglo.

3-. Extraer los primeros 5 registros →

```
serie.head()
```

```
0    22
1    13
2     8
3    33
4    25
dtype: int64
```

Nota: El proceso es el mismo con los DataFrames.

4-. Extraer los últimos 5 registros →

`serie.tail()`

```
3    33
4    25
5    44
6    38
7    22
dtype: int64
```

Nota: El proceso es el mismo con los DataFrames.

5-. Ordenar los valores de la serie →

`serie.sort_values()`

```
2     8
1    13
0    22
7    22
4    25
3    33
6    38
5    44
dtype: int64
```

Nota: El proceso es el mismo con los DataFrames.

6-. Determinar el número de elementos de la serie →

`serie.size`

8

Nota: El proceso es el mismo con los DataFrames.

7-. Extraer el valor mínimo de la serie →

`serie.min()`

8

Nota: En el caso de una columna de un DataFrame, agregaremos el nombre de esta en medio.

`Nombre_DataFrame.Población.min()`

8-. Extraer el valor máximo de la serie →

```
serie.max()
```

44

Nota: En el caso de una columna de un DataFrame, agregaremos el nombre de esta en medio.

```
Nombre_DataFrame.Población.max()
```

9-. Sumar todos los valores de la serie →

```
serie.sum()
```

205

10-. Extraer el valor promedio de la serie →

```
serie.mean()
```

25.625

11-. Obtener una descripción general de los valores de la serie →

```
serie.describe()
```

```
count      8.000000
mean       25.625000
std        12.199971
min         8.000000
25%        19.750000
50%        23.500000
75%        34.250000
max        44.000000
dtype: float64
```

Nota: El proceso es el mismo con los DataFrames.

12-. Extraer el segundo valor de la serie →

```
serie[1]
```

13

Nota: Otra opción es usar `serie.get(1)`.

13-. Filtrar los registros con un valor igual a 22 →

```
serie[serie == 22]
```

```
0    22
7    22
dtype: int64
```

14-. Suma dos o más series →

```
edades2 = [23,15,12,25,20,50,31,2]
```

```
serie2 = pd.Series(edades2)
```

serie2

serie + serie2

```
0    45
1    28
2    20
3    58
4    45
5    94
6    69
7    24
dtype: int64
```

15-. Reordenar la serie anterior invirtiendo los índices →

serie2 = serie2.sort_index(ascending = False)

serie2

```
7     2
6    31
5    50
4    20
3    25
2    12
1    15
0    23
dtype: int64
```

16-. Restar 5 unidades a todos los elementos de una serie →

serie2 - 5

```
7    -3
6    26
5    45
4    15
3    20
2     7
1    10
0    18
dtype: int64
```

17-. Calcular el cuadrado de todos los valores en una serie →

def cuadrado(x):

return x * x

serie2.map(cuadrado)

```
7      4
6   961
5  2500
4   400
3   625
2   144
1   225
0   529
dtype: int64
```

18-. Crear un DataFrame llamado “países” a partir de un diccionario de datos →

```

datos = {"Nombre":["Argentina","Brasil","Chile","Colombia","Ecuador","Paraguay","Peru"],
        "Capital":["Buenos Aires","Brasilia","Santiago de Chile","Bogotá","Quito","Asunción","Lima"],
        "Población":[45380000,212600000,19120000,50880000,17640000,7133000,32970000],
        "Sup. en km2":[2780000,8516000,756950,1143000,283560, 406752,1285216]}

```

```

países = pd.DataFrame(datos)

```

países

	Nombre	Capital	Población	Sup. en km2
0	Argentina	Buenos Aires	45380000	2780000
1	Brasil	Brasilia	212600000	8516000
2	Chile	Santiago de Chile	19120000	756950
3	Colombia	Bogotá	50880000	1143000
4	Ecuador	Quito	17640000	283560
5	Paraguay	Asunción	7133000	406752
6	Perú	Lima	32970000	1285216

19-. Extraer un rango determinado de registros en un DataFrame →

```

países[0:5]

```

	Nombre	Capital	Población	Sup. en km2
0	Argentina	Buenos Aires	45380000	2780000
1	Brasil	Brasilia	212600000	8516000
2	Chile	Santiago de Chile	19120000	756950
3	Colombia	Bogotá	50880000	1143000
4	Ecuador	Quito	17640000	283560

Nota: El proceso es el mismo con las series.

20-. Crear un DataFrame utilizando el diccionario de datos anterior con las columnas “nombre” y “población” →

```

países2 = pd.DataFrame(datos, columns = ["Nombre", "Población"])

```

países2

	Nombre	Población
0	Argentina	45380000
1	Brasil	212600000
2	Chile	19120000
3	Colombia	50880000
4	Ecuador	17640000
5	Paraguay	7133000
6	Perú	32970000

Nota: Si queremos seleccionar columnas específicas sin necesidad de crear un nuevo DataFrame, utilizaríamos la siguiente sintaxis, `países2[["Nombre", "Población"]]`.

21.- Crear un DataFrame utilizando el diccionario de datos, estableciendo la columna “Nombre” como índice →

```
países3 = pd.DataFrame(datos, index = datos["Nombre"], columns = ["Capital", "Población", "Sup. en km2"])
```

países3

	Capital	Población	Sup. en km2
Argentina	Buenos Aires	45380000	2780000
Brasil	Brasília	212600000	8516000
Chile	Santiago de Chile	19120000	756950
Colombia	Bogotá	50880000	1143000
Ecuador	Quito	17640000	283560
Paraguay	Asunción	7133000	406752
Perú	Lima	32970000	1285216

22.- Seleccionar las capitales que comienzan con “B” →

```
países3[países3["Capital"].str.startswith("B")]
```

	Capital	Población	Sup. en km2
Argentina	Buenos Aires	45380000	2780000
Brasil	Brasília	212600000	8516000
Colombia	Bogotá	50880000	1143000

23.- Extraer los países cuya población se encuentra entre los 20 y 60 millones →

```
países3[países3["Población"].between(20000000, 60000000)]
```

	Capital	Población	Sup. en km2
Argentina	Buenos Aires	45380000	2780000
Colombia	Bogotá	50880000	1143000
Perú	Lima	32970000	1285216

24-. Filtrar países con poblaciones específicas →

```
países3[países3["Población"].isin([19120000, 32970000])]
```

	Capital	Población	Sup. en km2
Chile	Santiago de Chile	19120000	756950
Perú	Lima	32970000	1285216

25-. Obtener los datos de la fila con el índice “Argentina” →

```
países3.loc[["Argentina"]]
```

	Capital	Población	Sup. en km2
Argentina	Buenos Aires	45380000	2780000

26-. Obtener la población y capital de “Argentina” y “Brasil” →

```
países3.loc[["Argentina", "Brasil"], ["Capital", "Población"]]
```

	Capital	Población
Argentina	Buenos Aires	45380000
Brasil	Brasília	212600000

Otras formas de trabajar con la instrucción .loc son las siguientes:

```
países3.loc[:,["Sup. en km2 "]]
```

	Sup. en km2
Argentina	2780000
Brasil	8516000
Chile	756950
Colombia	1143000
Ecuador	283560
Paraguay	406752
Perú	1285216

Utilizando la notación del punto (países3.Población)

```
Argentina    45380000
Brasil       212600000
Chile        19120000
Colombia     50880000
Ecuador     17640000
Paraguay     7133000
Peru         32970000
Name: Población, dtype: int64
```

O colocando el nombre de una o más columnas, separados por comas, dentro del segundo par de corchetes países3[["Sup. en km2 "]]

	Sup. en km2
Argentina	2780000
Brasil	8516000
Chile	756950
Colombia	1143000
Ecuador	283560
Paraguay	406752
Perú	1285216

27-. Obtener toda la información de los países con una superficie mayor a 1,000,000 de kilómetros cuadrados →

```
países3.loc[países3["Sup. en km2"] > 1000000]
```

	Capital	Población	Sup. en km2
Argentina	Buenos Aires	45380000	2780000
Brasil	Brasilia	212600000	8516000
Colombia	Bogotá	50880000	1143000
Perú	Lima	32970000	1285216

Nota: En el caso de que el nombre de la columna sea sencillo y práctico, no pondremos los corchetes en dicha columna; basta con usar un punto entre el nombre del DataFrame y el de la columna.

Nota: Otra forma de realizar el mismo ejercicio es utilizando el método query, por ejemplo, se puede escribir de la siguiente manera, países3.query("`Sup. en km2` > 1000000").

Nota: Para filtrar múltiples condiciones, debemos encerrar cada condición entre paréntesis dentro de los corchetes y unirlos con un ampersand (&).

```
países3.loc[(países3["Sup. en km2"] > 1000000) & (países3["Población"] > 50000000)]
```

	Capital	Población	Sup. en km2
Brasil	Brasilia	212600000	8516000
Colombia	Bogotá	50880000	1143000

28-. Cambiar el nombre de la capital de Argentina a “Buenos Aires” →

```
países3.loc["Argentina", "Capital"] = "Buenos Aires"
```

```
países3
```


	Capital	Población	Sup. en km2
Argentina	Buenos Aires	45380000	2780000
Brasil	Brasilia	212600000	8516000
Chile	Santiago de Chile	19120000	756950
Colombia	Bogotá	50880000	1143000
Ecuador	Quito	17640000	283560
Paraguay	Asunción	7133000	406752
Perú	Lima	32970000	1285216

29-. Obtener todos los datos de la tercera fila →

`países3.iloc[[2]]`

	Capital	Población	Sup. en km2
Chile	Santiago de Chile	19120000	756950

Nota: Si queremos introducir más filas, basta con agregar una coma dentro del segundo corchete.

30-. Obtener todos los datos de la segunda columna →

`países3.iloc[:, [1]]`

	Población
Argentina	45380000
Brasil	212600000
Chile	19120000
Colombia	50880000
Ecuador	17640000
Paraguay	7133000
Perú	32970000

31-. Obtener el nombre de todas las columnas del DataFrame →

`países3.columns`

`Index(['Capital', 'Población', 'Sup. en km2'], dtype='object')`

32-. Obtener todos los valores de la columna “Población” →

`países3.Población.values`

`array([45380000, 212600000, 19120000, 50880000, 17640000, 7133000, 32970000])`

33-. Obtener la desviación estándar de la columna “Población” →

`países3.Población.std()`

71188621.18043464

34-. Contar valores no nulos en el DataFrame →

```
países3.count()
```

```
Capital      7
Población    7
Sup. en km2   7
dtype: int64
```

Nota: El proceso es el mismo con las series.

35-. Agregar una columna al DataFrame “países” que contenga la población en millones de habitantes →

```
países3["Pob. en millones"] = países3.Población / 1000000
```

```
países3
```

	Capital	Población	Sup. en km2	Pob. en millones
Argentina	Buenos Aires	45380000	2780000	45.380
Brasil	Brasilia	212600000	8516000	212.600
Chile	Santiago de Chile	19120000	756950	19.120
Colombia	Bogotá	50880000	1143000	50.880
Ecuador	Quito	17640000	283560	17.640
Paraguay	Asunción	7133000	406752	7.133
Perú	Lima	32970000	1285216	32.970

Nota: Otra forma es mediante la función `apply()`

```
def millones(x):
```

```
    return x / 1000000
```

```
países3["Pob. en millones"] = países3.Población.apply(millones)
```

```
países3
```

36-. Agregar una columna al DataFrame “países” que contenga la población por cada km² de superficie, utilizando la función `apply()` →

```
def densidad(df):
```

```
    return df["Población"] / df["Sup. en km2"]
```

```
países3["Densidad poblacional"] = países3.apply(densidad, axis = 1)
```

```
países3
```

	Capital	Población	Sup. en km2	Pob. en millones	Densidad poblacional
Argentina	Buenos Aires	45380000	2780000	45.380	16.323741
Brasil	Brasilia	212600000	8516000	212.600	24.964772
Chile	Santiago de Chile	19120000	756950	19.120	25.259264
Colombia	Bogotá	50880000	1143000	50.880	44.514436
Ecuador	Quito	17640000	283560	17.640	62.209056
Paraguay	Asunción	7133000	406752	7.133	17.536484
Perú	Lima	32970000	1285216	32.970	25.653275

37-. Guardar y cargar el DataFrame en formato pickle →

```
países3.to_pickle("./archivo_pickle")
```

```
pd.read_pickle("./archivo_pickle")
```

	Capital	Población	Sup. en km2	Pob. en millones	Densidad poblacional
Argentina	Buenos Aires	45380000	2780000	45.380	16.323741
Brasil	Brasilia	212600000	8516000	212.600	24.964772
Chile	Santiago de Chile	19120000	756950	19.120	25.259264
Colombia	Bogotá	50880000	1143000	50.880	44.514436
Ecuador	Quito	17640000	283560	17.640	62.209056
Paraguay	Asunción	7133000	406752	7.133	17.536484
Perú	Lima	32970000	1285216	32.970	25.653275

38-. Guardar y cargar el DataFrame en formato CSV →

```
países3.to_csv("./archivo_csv")
```

```
pd.read_csv("./archivo_csv")
```

	Sin nombre: 0	Capital	Población	Sup. en km2	Pob. en millones	Densidad poblacional
0	Argentina	Buenos Aires	45380000	2780000	45.380	16.323741
1	Brasil	Brasilia	212600000	8516000	212.600	24.964772
2	Chile	Santiago de Chile	19120000	756950	19.120	25.259264
3	Colombia	Bogotá	50880000	1143000	50.880	44.514436
4	Ecuador	Quito	17640000	283560	17.640	62.209056
5	Paraguay	Asunción	7133000	406752	7.133	17.536484
6	Perú	Lima	32970000	1285216	32.970	25.653275

Nota: Cuando se utiliza `index_col=0` al leer un archivo de este tipo, se le indica a pandas que la primera columna debe ser tratada como el índice del DataFrame. Esto es útil en situaciones donde la primera columna contiene identificadores que son únicos y relevantes para las filas, evitando así la inclusión de una columna adicional que sería redundante.

Nota: Al usar `nombre_del_archivo.to_csv("nombre.csv", index = True, header = True)` incluirás una columna de índice numerado y los nombres de las columnas en la primera fila del archivo CSV, lo que ayuda a identificar filas y entender la estructura de los datos.

39-. Guardar y cargar el DataFrame en formato .xlsx →

```
países3.to_excel("./archivo_excel.xlsx", sheet_name = "Países", index_label = "País")
```

```
pd.read_excel("./archivo_excel.xlsx", index_col = "País")
```

	Capital	Población	Sup. en km2	Pob. en millones	Densidad poblacional
Argentina	Buenos Aires	45380000	2780000	45.380	16.323741
Brasil	Brasília	212600000	8516000	212.600	24.964772
Chile	Santiago de Chile	19120000	756950	19.120	25.259264
Colombia	Bogotá	50880000	1143000	50.880	44.514436
Ecuador	Quito	17640000	283560	17.640	62.209056
Paraguay	Asunción	7133000	406752	7.133	17.536484
Perú	Lima	32970000	1285216	32.970	25.653275

40-. Combinar dos DataFrames utilizando la columna “Nombre” como clave (Joins) →

```
estudiantes = pd.DataFrame({  
    "Nombre": ["Andrea", "Luis", "Carlos", "Marta", "Pedro"],  
    "Edad": [20, 21, 19, 22, 23],  
    "Carrera": ["Economía", "Arquitectura", "Medicina", "Ingeniería", "Derecho"],  
    "Asignatura": ["Matemáticas", "Física", "Química", "Biología", "Historia"],  
    "Inscripción": ["2021-09-01", "2019-08-15", "2020-07-10", "2024-01-05", "2023-03-12"]  
})
```

estudiantes

	Nombre	Edad	Carrera	Asignatura	Inscripción
0	Andrea	20	Economía	Matemáticas	2021-09-01
1	Luis	21	Arquitectura	Física	2019-08-15
2	Carlos	19	Medicina	Química	2020-07-10
3	Marta	22	Ingeniería	Biología	2024-01-05
4	Pedro	23	Derecho	Historia	2023-03-12

```
calificaciones = pd.DataFrame({  
    "Nombre": ["Andrea", "Luis", "Marta", "Carlos", "Sandra"],  
    "Género": ["Femenino", "Masculino", "Femenino", "Masculino", "Femenino"],  
    "Calificación": [85, 90, 88, 92, 95]  
})
```

calificaciones

	Nombre	Género	Calificación
0	Andrea	Femenino	85
1	Luis	Masculino	90
2	Marta	Femenino	88
3	Carlos	Masculino	92
4	Sandra	Femenino	95

INNER JOIN

```
inner_join = pd.merge(estudiantes, calificaciones, on = "Nombre", how = "inner")
```

inner_join

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20	Economía	Matemáticas	2021-09-01	Femenino	85
1	Luis	21	Arquitectura	Física	2019-08-15	Masculino	90
2	Carlos	19	Medicina	Química	2020-07-10	Masculino	92
3	Marta	22	Ingeniería	Biología	2024-01-05	Femenino	88

LEFT JOIN

```
left_join = pd.merge(estudiantes, calificaciones, on = "Nombre", how = "left")
```

left_join

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20	Economía	Matemáticas	2021-09-01	Femenino	85.0
1	Luis	21	Arquitectura	Física	2019-08-15	Masculino	90.0
2	Carlos	19	Medicina	Química	2020-07-10	Masculino	92.0
3	Marta	22	Ingeniería	Biología	2024-01-05	Femenino	88.0
4	Pedro	23	Derecho	Historia	2023-03-12	Nan	Nan

Nota: Si agregamos estas líneas extra, manejaremos de mejor manera los valores NaN.

```
left_join["Género"] = left_join["Género"].fillna("Desconocido")
```

```
left_join["Calificación"] = left_join["Calificación"].fillna(left_join["Calificación"].mean())
```

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20	Economía	Matemáticas	2021-09-01	Femenino	85.00
1	Luis	21	Arquitectura	Física	2019-08-15	Masculino	90.00
2	Carlos	19	Medicina	Química	2020-07-10	Masculino	92.00
3	Marta	22	Ingeniería	Biología	2024-01-05	Femenino	88.00
4	Pedro	23	Derecho	Historia	2023-03-12	Desconocido	88.75

Nota: En la columna “Calificación”, se utiliza el promedio para imputar los valores NaN, asegurando que las calificaciones se representen de manera adecuada.

RIGHT JOIN

```
right_join = pd.merge(estudiantes, calificaciones, on = "Nombre", how = "right")
```

right_join

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20.0	Economía	Matemáticas	2021-09-01	Femenino	85
1	Luis	21.0	Arquitectura	Física	2019-08-15	Masculino	90
2	Marta	22.0	Ingeniería	Biología	2024-01-05	Femenino	88
3	Carlos	19.0	Medicina	Química	2020-07-10	Masculino	92
4	Sandra	Nan	Nan	Nan	Nan	Femenino	95

Nota: Si agregamos estas líneas extra, manejaremos de mejor manera los valores NaN.

```
right_join["Edad"] = right_join["Edad"].fillna(right_join["Edad"].median())
```

```
right_join["Carrera"] = right_join["Carrera"].fillna("Desconocida")
```

```
right_join["Asignatura"] = right_join["Asignatura"].fillna("Desconocida")
```

```
right_join["Inscripción"] = right_join["Inscripción"].fillna("Desconocida")
```

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20.0	Economía	Matemáticas	2021-09-01	Femenino	85
1	Luis	21.0	Arquitectura	Física	2019-08-15	Masculino	90
2	Marta	22.0	Ingeniería	Biología	2024-01-05	Femenino	88
3	Carlos	19.0	Medicina	Química	2020-07-10	Masculino	92
4	Sandra	20.5	Desconocida	Desconocida	Desconocida	Femenino	95

Nota: Para la columna “Edad”, se utiliza la mediana como método de imputación, lo que permite minimizar el impacto de valores atípicos.

Nota: Las columnas “Carrera”, “Asignatura”, “Inscripción” y “Género” se rellenan con un valor predeterminado, “Desconocido”, garantizando que no haya valores faltantes.

OUTER JOIN

```
outer_join = pd.merge(estudiantes, calificaciones, on = "Nombre", how = "outer")
```

outer_join

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20.0	Economía	Matemáticas	2021-09-01	Femenino	85.0
1	Carlos	19.0	Medicina	Química	2020-07-10	Masculino	92.0
2	Luis	21.0	Arquitectura	Física	2019-08-15	Masculino	90.0
3	Marta	22.0	Ingeniería	Biología	2024-01-05	Femenino	88.0
4	Pedro	23.0	Derecho	Historia	2023-03-12	Nan	Nan
5	Sandra	Nan	Nan	Nan	Nan	Femenino	95.0

Nota: Al combinar todas las filas de un left join y un right join, podremos manejar de forma más eficiente los valores NaN.

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20.0	Economía	Matemáticas	2021-09-01	Femenino	85.0
1	Carlos	19.0	Medicina	Química	2020-07-10	Masculino	92.0
2	Luis	21.0	Arquitectura	Física	2019-08-15	Masculino	90.0
3	Marta	22.0	Ingeniería	Biología	2024-01-05	Femenino	88.0
4	Pedro	23.0	Derecho	Historia	2023-03-12	Desconocido	90.0
5	Sandra	21.0	Desconocida	Desconocida	Desconocida	Femenino	95.0

Formas alternativas de manejar estos valores incluyen el uso de diccionarios y la interpolación (aplicable solo a columnas con valores numéricos).

Diccionario:

```
outer_join.fillna({
    "Edad": outer_join["Edad"].median(),
    "Carrera": "Desconocida",
    "Asignatura": "Desconocida",
    "Inscripción": "Desconocida",
    "Género": "Desconocido",
    "Calificación": outer_join["Calificación"].mean()
})
```

Interpolación:

```
outer_join["Edad"] = outer_join["Edad"].interpolate()
outer_join["Calificación"] = outer_join["Calificación"].interpolate()
```

41-. Convertir los nombres a mayúsculas en la columna “Nombre” de un DataFrame →

```
inner_join["Nombre"] = inner_join ["Nombre"].str.upper()
inner_join
```

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	ANDREA	20	Economía	Matemáticas	2021-09-01	Femenino	85
1	LUIS	21	Arquitectura	Física	2019-08-15	Masculino	90
2	CARLOS	19	Medicina	Química	2020-07-10	Masculino	92
3	MARTA	22	Ingeniería	Biología	2024-01-05	Femenino	88

Nota: Si deseas cambiar la primera letra de cada palabra a mayúscula en cualquier columna del DataFrame, puedes usar el método `.str.title()`.

42-. Incorporar la columna “Año” en el DataFrame para reflejar el año de inscripción de cada estudiante →

```
inner_join ['Año'] = pd.to_datetime(inner_join['Inscripción']).dt.year
```

```
inner_join
```

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación	Año
0	ANDREA	20	Economía	Matemáticas	2021-09-01	Femenino	85	2021
1	LUIS	21	Arquitectura	Física	2019-08-15	Masculino	90	2019
2	CARLOS	19	Medicina	Química	2020-07-10	Masculino	92	2020
3	MARTA	22	Ingeniería	Biología	2024-01-05	Femenino	88	2024

Nota: Para extraer el mes y el año de una columna de fechas, utiliza `.dt.to_period('M')`. Si deseas obtener el mes y el día, usa `.dt.strftime("%m-%d")`. Para obtener solo el día, aplica `.dt.day`, y para obtener solo el mes, utiliza `.dt.month`.

43-. Filtrar estudiantes por carrera →

```
filtrados = inner_join [inner_join ['Carrera'].str.contains('Ingeniería')]
```

```
filtrados
```

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación	Año
3	MARTA	22	Ingeniería	Biología	2024-01-05	Femenino	88	2024

44-. Contar la cantidad de estudiantes inscritos por año →

```
conteo_por_año = inner_join.groupby('Año').size()
```

```
conteo_por_año
```

```
Año
2019    1
2020    1
2021    1
2024    1
dtype: int64
```

45-. Contar la cantidad de alumnos inscritos por año y carrera utilizando una tabla dinámica →

```
tabla_resumen = inner_join.pivot_table(index='Año', columns='Carrera', values='Nombre',
aggfunc='count', fill_value=0)
```

```
tabla_resumen
```

Carrera	Arquitectura	Economía	Ingeniería	Medicina
Año				
2019	1	0	0	0
2020	0	0	0	1
2021	0	1	0	0
2024	0	0	1	0

46-. Combinar dos DataFrames utilizando la columna “Nombre” como clave (Concatenación horizontal y vertical) →


```
concat_horizontal = pd.concat([estudiantes.set_index('Nombre'),
calificaciones.set_index('Nombre')], axis=1).reset_index()
```

concat_horizontal

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20.0	Economía	Matemáticas	2021-09-01	Femenino	85.0
1	Luis	21.0	Arquitectura	Física	2019-08-15	Masculino	90.0
2	Carlos	19.0	Medicina	Química	2020-07-10	Masculino	92.0
3	Marta	22.0	Ingeniería	Biología	2024-01-05	Femenino	88.0
4	Pedro	23.0	Derecho	Historia	2023-03-12	Nan	Nan
5	Sandra	Nan	Nan	Nan	Nan	Femenino	95.0

```
concat_vertical = pd.concat([estudiantes, calificaciones], axis=0, ignore_index=True)
```

concat_vertical

	Nombre	Edad	Carrera	Asignatura	Inscripción	Género	Calificación
0	Andrea	20.0	Economía	Matemáticas	2021-09-01	Nan	Nan
1	Luis	21.0	Arquitectura	Física	2019-08-15	Nan	Nan
2	Carlos	19.0	Medicina	Química	2020-07-10	Nan	Nan
3	Marta	22.0	Ingeniería	Biología	2024-01-05	Nan	Nan
4	Pedro	23.0	Derecho	Historia	2023-03-12	Nan	Nan
5	Andrea	Nan	Nan	Nan	Nan	Femenino	85.0
6	Luis	Nan	Nan	Nan	Nan	Masculino	90.0
7	Marta	Nan	Nan	Nan	Nan	Femenino	88.0
8	Carlos	Nan	Nan	Nan	Nan	Masculino	92.0
9	Sandra	Nan	Nan	Nan	Nan	Femenino	95.0

47-. Insertar un nuevo registro en el DataFrame ➔

```
import numpy as np
```

```
participantes = pd.DataFrame({
    "Nombre": ["Antonio", "Juan", "Pedro", "Maria", "Luis", "Antonio", "Juan", "Carlos",
"Lucia"],
    "Edad": [22, 25, np.nan, 23, 20, 22, 25, 28, 30],
    "Ciudad": ["Barcelona", "Madrid", "Valencia", None, "Bilbao", "Barcelona", "Madrid",
"Sevilla", "Zaragoza"]
})
participantes
```

	Nombre	Edad	Ciudad
0	Antonio	22.0	Barcelona
1	Juan	25.0	Madrid
2	Pedro	Nan	Valencia
3	Maria	23.0	Ninguno
4	Luis	20.0	Bilbao
5	Antonio	22.0	Barcelona
6	Juan	25.0	Madrid
7	Carlos	28.0	Sevilla
8	Lucia	30.0	Zaragoza

`participantes.loc[len(participantes)] = ["Sofia", 16, "Granada"]`

`participantes`

	Nombre	Edad	Ciudad
0	Antonio	22.0	Barcelona
1	Juan	25.0	Madrid
2	Pedro	Nan	Valencia
3	Maria	23.0	Ninguno
4	Luis	20.0	Bilbao
5	Antonio	22.0	Barcelona
6	Juan	25.0	Madrid
7	Carlos	28.0	Sevilla
8	Lucia	30.0	Zaragoza
9	Sofía	16.0	Granada

48-. Eliminar todas las filas con al menos un valor nulo en cualquier columna →

`participantes.dropna()`

	Nombre	Edad	Ciudad
0	Antonio	22.0	Barcelona
1	Juan	25.0	Madrid
4	Luis	20.0	Bilbao
5	Antonio	22.0	Barcelona
6	Juan	25.0	Madrid
7	Carlos	28.0	Sevilla
8	Lucia	30.0	Zaragoza
9	Sofía	16.0	Granada

Nota: Si queremos hacer lo mismo, pero en determinadas columnas, utilizaremos la siguiente sintaxis, `participantes.dropna(subset="Edad", "Ciudad")`

49-. Eliminar filas duplicadas →

```
participantes.drop_duplicates()
```

	Nombre	Edad	Ciudad
0	Antonio	22.0	Barcelona
1	Juan	25.0	Madrid
2	Pedro	Nan	Valencia
3	Maria	23.0	Ninguno
4	Luis	20.0	Bilbao
7	Carlos	28.0	Sevilla
8	Lucia	30.0	Zaragoza
9	Sofía	16.0	Granada

50-. Reemplazar valores nulos en las columnas “Ciudad” y “Edad” usando un diccionario →

```
participantes.replace({  
    "Edad": {np.nan: "Desconocido"},  
    "Ciudad": {None: "Desconocido"}  
})
```

	Nombre	Edad	Ciudad
0	Antonio	22.0	Barcelona
1	Juan	25.0	Madrid
2	Pedro	Desconocido	Valencia
3	Maria	23.0	Desconocido
4	Luis	20.0	Bilbao
5	Antonio	22.0	Barcelona
6	Juan	25.0	Madrid
7	Carlos	28.0	Sevilla
8	Lucia	30.0	Zaragoza
9	Sofía	16.0	Granada

Nota: Otra alternativa para obtener el mismo resultado sería la siguiente

```
participantes["Ciudad"].replace(to_replace = [None], value = "Desconocido")  
participantes["Edad"].replace(to_replace = [np.nan], value = "Desconocido")  
participantes
```

Nota: No es posible calcular funciones directamente dentro del diccionario en el método replace. Para realizar reemplazos que dependen de cálculos, como la mediana, primero debes calcular el valor por separado y luego usar ese valor en el diccionario de reemplazo.

51-. Ordenar las columnas en el orden deseado →

```
columnas_ordenadas = ["Ciudad", "Edad", "Nombre"]
```

```
participantes = participantes[columnas_ordenadas]
```

```
participantes
```

	Ciudad	Edad	Nombre
0	Barcelona	22.0	Antonio
1	Madrid	25.0	Juan
2	Valencia	Nan	Pedro
3	Ninguno	23.0	Maria
4	Bilbao	20.0	Luis
5	Barcelona	22.0	Antonio
6	Madrid	25.0	Juan
7	Sevilla	28.0	Carlos
8	Zaragoza	30.0	Lucia
9	Granada	16.0	Sofía

52-. Renombrar la columna “Ciudad” a “Localidad” →

```
participantes.rename(columns={"Ciudad": "Localidad"})
```

	Localidad	Edad	Nombre
0	Barcelona	22.0	Antonio
1	Madrid	25.0	Juan
2	Valencia	Nan	Pedro
3	Ninguno	23.0	Maria
4	Bilbao	20.0	Luis
5	Barcelona	22.0	Antonio
6	Madrid	25.0	Juan
7	Sevilla	28.0	Carlos
8	Zaragoza	30.0	Lucia
9	Granada	16.0	Sofía

Nota: Si queremos introducir más columnas, basta con agregar una coma dentro del diccionario.

53-. Eliminar la columna “Ciudad” del DataFrame →

```
ciudad_eliminada = participantes.pop("Ciudad")
```

```
participantes
```

	Edad	Nombre
0	22.0	Antonio
1	25.0	Juan
2	Nan	Pedro
3	23.0	Maria
4	20.0	Luis
5	22.0	Antonio
6	25.0	Juan
7	28.0	Carlos
8	30.0	Lucia
9	16.0	Sofía

54-. Eliminar la filas con los índices 2 y 3 →

`participantes.drop(index = [2, 3])`

	Edad	Nombre
0	22.0	Antonio
1	25.0	Juan
4	20.0	Luis
5	22.0	Antonio
6	25.0	Juan
7	28.0	Carlos
8	30.0	Lucia
9	16.0	Sofía

55-. Agrupar por ciudad y calcular la media de las columnas “Puntuación” y “Edad” →

```
examen = pd.DataFrame ({
    "Nombre":["Miguel","Sebas","Luis","Germain","Paola", "Sandra"],
    "Ciudad":["Madrid","Barcelona","Madrid", "Valencia", "Barcelona", "Madrid"],
    "Edad":[25, 33, 30, 28, 45, 38],
    "Puntuación":[80, 90, 85, 88, 75, 91],
    "Categoría": ["C", "B", "C", "B", "C", "B"]
})
examen
```

	Nombre	Ciudad	Edad	Puntuación	Categoría
0	Miguel	Madrid	25	80	C
1	Sebas	Barcelona	33	90	B
2	Luis	Madrid	30	85	C
3	Germain	Valencia	28	88	B
4	Paola	Barcelona	45	75	C
5	Sandra	Madrid	38	91	B

```
agregacion_ciudad = examen.groupby("Ciudad")[["Edad", "Puntuación"]].mean()
```

```
agregacion_ciudad
```

	Edad	Puntuación
Ciudad		
Barcelona	39.0	82.500000
Madrid	31.0	85.333333
Valencia	28.0	88.000000

56-. Agrupar por ciudad y aplicar funciones de agregación personalizadas para cada una de las columnas anteriores →

```
agregacion_personalizada = examen.groupby("Ciudad").agg({
    "Edad": ["mean", "max"],
    "Puntuación": ["mean", "min", "max"]
})
```

```
agregacion_personalizada
```

	Edad		Puntuación		
	significar	máximo	significar	min	máximo
Ciudad					
Barcelona	39.0	45	82.500000	75	90
Madrid	31.0	38	85.333333	80	91
Valencia	28.0	28	88.000000	88	88

57-. Agrupar por ciudad y categoría con agregaciones multinivel →

```
agregacion_multinivel = examen.groupby(["Ciudad", "Categoría"]).agg({
    "Puntuación": "mean",
    "Edad": "count"
})
```

		Puntuación	Edad
Ciudad	Categoría		
Barcelona	B	90.0	1
	C	75.0	1
Madrid	B	91.0	1
	C	82.5	2
Valencia	B	88.0	1

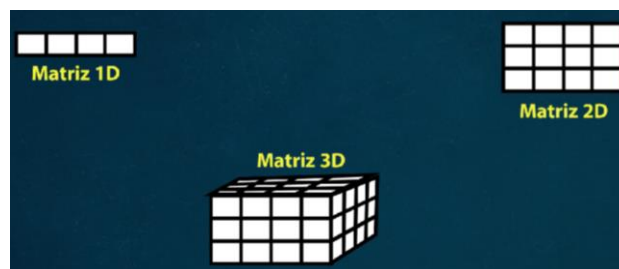
Capítulo 22: NumPy

NumPy (abreviatura de Numerical Python) es una biblioteca de Python que proporciona soporte para arreglos multidimensionales y funciones matemáticas optimizadas para realizar operaciones con grandes cantidades de datos numéricos de manera rápida y eficiente.



Características principales:

1-. Arreglos multidimensionales (ndarray): El arreglo ndarray es el núcleo de NumPy, similar a las listas pero más eficiente y optimizado para manejar datos en múltiples dimensiones (1D, 2D, 3D, etc.), permitiendo trabajar fácilmente con vectores, matrices y tensores.



2-. Operaciones vectorizadas: Permite realizar operaciones matemáticas en todo el arreglo sin necesidad de utilizar bucles explícitos. Esto se conoce como vectorización, lo que mejora drásticamente el rendimiento y hace que el código sea más claro y conciso.

3-. Funciones matemáticas y estadísticas: Incluye un conjunto robusto de funciones para realizar cálculos avanzados como sumas, productos, operaciones trigonométricas, logaritmos y más. Además, proporciona herramientas para el análisis estadístico, permitiendo calcular promedios, desviaciones estándar y otros indicadores clave. También incluye funciones de álgebra lineal, como la resolución de sistemas de ecuaciones, el cálculo de determinantes, valores y vectores propios.

Asimismo, soporta transformadas de Fourier para análisis de señales y cuenta con un módulo especializado en la generación de números aleatorios, útil para simulaciones, creación de distribuciones estadísticas y pruebas aleatorias.

4-. Manipulación y transformación de arreglos: Ofrece una amplia gama de funciones para reorganizar, redimensionar, cortar, dividir, unir y transponer arreglos.

5-. Compatibilidad con otras bibliotecas: Es la base de muchas otras bibliotecas populares en el ecosistema de Python, como Pandas (para análisis de datos), Matplotlib (para visualización de datos), SciPy (para cálculo científico), y TensorFlow (para machine learning).

Historia de NumPy:

NumPy fue creado en 2005 por **Travis Olliphant**, quien combinó y mejoró dos bibliotecas anteriores: Numeric y Numarray. Olliphant, un científico de datos, se dio cuenta de la necesidad de una herramienta más eficiente para realizar cálculos numéricos y operaciones sobre arreglos multidimensionales en Python. NumPy fue diseñado para ofrecer un soporte sólido para arrays grandes y multidimensionales, así como una colección de funciones matemáticas para operar con ellos.

Más tarde, NumPy se convirtió en la base de muchas otras bibliotecas científicas en Python, como SciPy y Pandas, y fue publicado como código abierto, lo que permitió a la comunidad de desarrolladores contribuir a su evolución y mejora continua.

Ejercicios prácticos:

1-. Importar la biblioteca de numpy →

```
import numpy as np
```

2-. Crear un arreglo ndarray a partir de una lista →

```
lista= [31,28,29,19]
```

```
np.array(lista)
```

```
array([31, 28, 29, 19])
```

3-. Crear un arreglo ndarray a partir de una lista bidimensional →

```
lista_2d=[[2,3,4],[3,4,5],[3,4,5],[3,4,5]]
```

```
np.array(lista_2d)
```

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

4-. Utilizar la propiedad shape para obtener las dimensiones de un arreglo →

```
arreglo.shape
```


(4, 3)

5-. Utilizar la propiedad `ndim` para determinar el número de dimensiones de un arreglo →

`arreglo.ndim`

2

6-. Utilizar la propiedad `dtype` para obtener el tipo de dato de un arreglo →

`arreglo.dtype`

`dtype('int64')`

7-. Utilizar la propiedad `size` para obtener el número total de elementos de un arreglo →

`arreglo.size`

12

8-. Crear un arreglo a partir de una lista, especificando un tipo de dato en particular →

`np.array(lista_2d, dtype = np.int16)`

```
array([[2, 3, 4],
       [3, 4, 5],
       [3, 4, 5],
       [3, 4, 5]], dtype=int16)
```

9-. Utilizar la función `np.zeros` para crear un arreglo lleno de ceros con una forma determinada →

`np.zeros((3,2))`

```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

10-. Utilizar la función `np.ones` para crear un arreglo lleno de unos con una forma específica →

`np.ones((3,3))`

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

11-. Utilizar la función `np.eye` para crear una matriz identidad →

`np.eye(3)`

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

12-. Utilizar la función `np.empty` para crear un arreglo con una forma fija, sin inicializar sus valores →

`np.empty((2,4))`

```
array([[0.00e+000, 0.00e+000, 0.00e+000, 0.00e+000],
       [0.00e+000, 4.09e-321, 0.00e+000, 0.00e+000]])
```

13-. Crear un arreglo unidimensional con valores específicos →

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

14-. Crear un arreglo bidimensional con una forma y valores específicos →

```
np.arange(10).reshape(2,5)
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

15-. Crear y visualizar una matriz tridimensional →

```
matriz_3d = np.arange(24).reshape(2,3,4)
```

```
matriz_3d
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

16-. Crear un arreglo con una secuencia y valores específicos →

```
np.arange(10,21,2)
```

```
array([10, 12, 14, 16, 18, 20])
```

17-. Utilizar la función np.linspace para crear un arreglo unidimensional que contenga un número específico de valores igualmente espaciados entre dos límites determinados →

```
np.linspace(10,20,9)
```

```
array([10. , 11.25, 12.5 , 13.75, 15. , 16.25, 17.5 , 18.75, 20. ])
```

18-. Utilizar la función np.linspace para crear un arreglo unidimensional que contenga un número específico de valores igualmente espaciados entre 0 y 2π →

```
from numpy import pi
```

```
radian = np.linspace(0,2*pi,100)
```

```
radian
```

```
array([0.          , 0.06346652, 0.12693304, 0.19039955, 0.25386607,
       0.31733259, 0.38079911, 0.44426563, 0.50773215, 0.57119866,
       0.63466518, 0.6981317 , 0.76159822, 0.82506474, 0.88853126,
       0.95199777, 1.01546429, 1.07893081, 1.14239733, 1.20586385,
       1.26933037, 1.33279688, 1.3962634 , 1.45972992, 1.52319644,
       1.58666296, 1.65012947, 1.71359599, 1.77706251, 1.84052903,
       1.90399555, 1.96746207, 2.03092858, 2.0943951 , 2.15786162,
       2.22132814, 2.28479466, 2.34826118, 2.41172769, 2.47519421,
       2.53866073, 2.60212725, 2.66559377, 2.72906028, 2.7925268 ,
       2.85599332, 2.91945984, 2.98292636, 3.04639288, 3.10985939,
       3.17332591, 3.23679243, 3.30025895, 3.36372547, 3.42719199,
       3.4906585 , 3.55412502, 3.61759154, 3.68105806, 3.74452458,
       3.8079911 , 3.87145761, 3.93492413, 3.99839065, 4.06185717,
       4.12532369, 4.1887902 , 4.25225672, 4.31572324, 4.37918976,
       4.44265628, 4.5061228 , 4.56958931, 4.63305583, 4.69652235,
       4.75998887, 4.82345539, 4.88692191, 4.95038842, 5.01385494,
       5.07732146, 5.14078798, 5.2042545 , 5.26772102, 5.33118753,
       5.39465405, 5.45812057, 5.52158709, 5.58505361, 5.64852012,
       5.71198664, 5.77545316, 5.83891968, 5.9023862 , 5.96585272,
       6.02931923, 6.09278575, 6.15625227, 6.21971879, 6.28318531])
```

19-. Emplear la función `np.sin` para calcular el seno de cada elemento de un arreglo →

`np.sin(radian)`

```
array([ 0.00000000e+00,  6.34239197e-02,  1.26592454e-01,  1.89251244e-01,
        2.51147987e-01,  3.12033446e-01,  3.71662456e-01,  4.29794912e-01,
        4.86196736e-01,  5.40640817e-01,  5.92907929e-01,  6.42787610e-01,
        6.90079011e-01,  7.34591709e-01,  7.76146464e-01,  8.14575952e-01,
        8.49725430e-01,  8.81453363e-01,  9.09631995e-01,  9.34147860e-01,
        9.54902241e-01,  9.71811568e-01,  9.84807753e-01,  9.93838464e-01,
        9.98867339e-01,  9.99874128e-01,  9.96854776e-01,  9.89821442e-01,
        9.78802446e-01,  9.63842159e-01,  9.45000819e-01,  9.2254294e-01,
        8.9593774e-01,  8.66025404e-01,  8.32569855e-01,  7.95761841e-01,
        7.55749574e-01,  7.12694171e-01,  6.66769001e-01,  6.18158986e-01,
        5.67059864e-01,  5.13677392e-01,  4.58226522e-01,  4.00930535e-01,
        3.42020143e-01,  2.81732557e-01,  2.20310533e-01,  1.58001396e-01,
        9.50560433e-02,  3.17279335e-02, -3.17279335e-02, -9.50560433e-02,
       -1.58001396e-01, -2.20310533e-01, -2.81732557e-01, -3.42020143e-01,
       -4.00930535e-01, -4.58226522e-01, -5.13677392e-01, -5.67059864e-01,
       -6.18158986e-01, -6.66769001e-01, -7.12694171e-01, -7.55749574e-01,
       -7.95761841e-01, -8.32569855e-01, -8.66025404e-01, -8.9593774e-01,
       -9.2254294e-01, -9.45000819e-01, -9.63842159e-01, -9.78802446e-01,
       -9.89821442e-01, -9.96854776e-01, -9.99874128e-01, -9.98867339e-01,
       -9.93838464e-01, -9.84807753e-01, -9.71811568e-01, -9.54902241e-01,
       -9.34147860e-01, -9.09631995e-01, -8.81453363e-01, -8.49725430e-01,
       -8.14575952e-01, -7.76146464e-01, -7.34591709e-01, -6.90079011e-01,
       -6.42787610e-01, -5.92907929e-01, -5.40640817e-01, -4.86196736e-01,
       -4.29794912e-01, -3.71662456e-01, -3.12033446e-01, -2.51147987e-01,
       -1.89251244e-01, -1.26592454e-01, -6.34239197e-02, -2.44929360e-16])
```

20-. Emplear la función `np.cos` para calcular el coseno de cada elemento de un arreglo →

`np.cos(radian)`

```
array([ 1.          ,  0.99798668,  0.99195481,  0.9819287 ,  0.9679487 ,
        0.95007112,  0.92836793,  0.90292654,  0.87384938,  0.84125353,
        0.80527026,  0.76604444,  0.72373404,  0.67850941,  0.63055267,
        0.58005691,  0.52722547,  0.47227107,  0.41541501,  0.35688622,
        0.29692038,  0.23575894,  0.17364818,  0.1108382 ,  0.04758192,
       -0.01586596, -0.07924996, -0.14231484, -0.20480667, -0.26647381,
       -0.32706796, -0.38634513, -0.44406661, -0.5          , -0.55392006,
       -0.60560969, -0.65486073, -0.70147489, -0.74526445, -0.78605309,
       -0.82367658, -0.85798341, -0.88883545, -0.91610846, -0.93969262,
       -0.95949297, -0.97542979, -0.98743889, -0.99547192, -0.99949654,
       -0.99949654, -0.99547192, -0.98743889, -0.97542979, -0.95949297,
       -0.93969262, -0.91610846, -0.88883545, -0.85798341, -0.82367658,
       -0.78605309, -0.74526445, -0.70147489, -0.65486073, -0.60560969,
       -0.55392006, -0.5          , -0.44406661, -0.38634513, -0.32706796,
       -0.26647381, -0.20480667, -0.14231484, -0.07924996, -0.01586596,
        0.04758192,  0.1108382 ,  0.17364818,  0.23575894,  0.29692038,
        0.35688622,  0.41541501,  0.47227107,  0.52722547,  0.58005691,
        0.63055267,  0.67850941,  0.72373404,  0.76604444,  0.80527026,
        0.84125353,  0.87384938,  0.90292654,  0.92836793,  0.95007112,
        0.9679487 ,  0.9819287 ,  0.99195481,  0.99798668,  1.          ])
```

21-. Sumar arreglos de igual longitud →

`a= np.linspace(10,20,6)`

`b= np.linspace(5,25,6)`

`a + b`

```
array([15., 21., 27., 33., 39., 45.])
```

22-. Concatenar múltiples arreglos →

```
c = np.concatenate((a,b))
```

c

```
array([10., 12., 14., 16., 18., 20., 5., 9., 13., 17., 21., 25.])
```

23-. Ordenar un arreglo en orden ascendente →

```
c.sort()
```

c

```
array([ 5., 9., 10., 12., 13., 14., 16., 17., 18., 20., 21., 25.])
```

24-. Generar un arreglo de 5 números aleatorios entre 0 y 1 usando np.random.default_rng →

```
ga = np.random.default_rng(2)
```

```
aleatorio = ga.random(5)
```

aleatorio

```
array([0.26161213, 0.29849114, 0.81422574, 0.09191594, 0.60010053])
```

Nota: Este método se usa más que np.random.rand() porque ofrece mejor control sobre la generación de números aleatorios y es más seguro y moderno para aplicaciones actuales.

25-. Generar un arreglo de 5 números aleatorios de una distribución normal con media 10 y desviación estándar 5 →

```
normal = ga.normal(10,5,5)
```

normal

```
array([13.99155475, 7.25570843, 15.73920226, 3.73078936, 7.23629579])
```

26-. Generar un arreglo de 2000 números enteros aleatorios entre 0 y 19 →

```
enteros = ga.integers(20, size = 2000)
```

enteros

```
array([ 4, 3, 9, ..., 7, 9, 10])
```

27-. Seleccionar 10 números aleatorios del 0 al 25 sin reemplazo →

```
choice = ga.choice(26, size = 10, replace = False)
```

choice

```
array([10, 12, 21, 15, 22, 4, 3, 25, 13, 16])
```

28-. Determinar el valor mínimo en un arreglo →

`choice.min()`

3

29- Determinar el valor máximo en un arreglo →

`choice.max()`

25

30- Calcular la media de los valores seleccionados en un arreglo →

`choice.mean()`

14.1

31- Calcular la suma de los elementos en un arreglo →

`choice.std()`

6.934695379034323

32- Calcular la desviación estándar de los elementos en un arreglo →

`choice.sum()`

141

Nota: Si necesitas sumar los valores de un array de manera acumulativa, utiliza la función `cumsum()`, que genera una secuencia donde cada valor es la suma del actual y los anteriores.

33- Generar un arreglo bidimensional de números enteros aleatorios →

`estadisticos = ga.integers(20,size=(5,4))`

`estadisticos`

```
array([[ 8,  5,  7,  1],
       [ 9,  9,  3,  7],
       [ 2,  9, 13, 11],
       [12, 18, 16, 19],
       [ 2,  5,  8, 11]])
```

34- Encontrar el valor mínimo de cada columna en un arreglo bidimensional →

`estadisticos.min(axis = 0)`

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

35- Encontrar el valor máximo de cada fila en un arreglo bidimensional →

`estadisticos.max(axis = 1)`

```
array([ 8,  9, 13, 19, 11])
```

36- Filtrar valores menores a 11 en un arreglo bidimensional →

```
estadisticos[estadisticos < 11]
```

```
array([8, 5, 7, 1, 9, 9, 3, 7, 2, 9, 2, 5, 8])
```

37- Unir arreglos verticalmente →

```
np1 = ga.integers(20,size=(3,3))
```

```
np2 = ga.integers(20,size=(3,3))
```

```
print(np1,"\n\n",np2)
```

```
[[ 5 10  1]
 [ 2 18 16]
 [14  2 19]]
```

```
[[19 15 19]
 [15 10 15]
 [16  5  9]]
```

```
np.vstack((np1, np2))
```

```
array([[ 5, 10,  1],
       [ 2, 18, 16],
       [14,  2, 19],
       [19, 15, 19],
       [15, 10, 15],
       [16,  5,  9]])
```

38- Unir arreglos horizontalmente →

```
np.hstack((np1, np2))
```

```
array([[ 5, 10,  1, 19, 15, 19],
       [ 2, 18, 16, 15, 10, 15],
       [14,  2, 19, 16,  5,  9]])
```

39- Seleccionar los primeros 6 elementos del arreglo →

```
enteros = ga.integers(20,size=(10))
```

```
array([10, 15, 15,  0,  8,  2,  6,  3,  4, 12])
```

```
enteros[0:6]
```

```
array([10, 15, 15,  0,  8,  2])
```

40- Seleccionar elementos de 2 en 2 desde los primeros 6 →

```
enteros[0:6:2]
```

```
array([10, 15,  8])
```

41- Seleccionar todos los elementos con un paso de 2 →

```
enteros[:,2]
```

```
array([10, 15,  8,  6,  4])
```

42- Seleccionar el elemento en la segunda fila y cuarta columna →

```
enteros_2d = ga.integers(20,size=(8,5))
```

```
enteros_2d
```

```
array([[17,  5,  4, 11, 10],
       [11,  4, 17,  5, 19],
       [ 3, 10, 12,  4,  7],
       [19,  4, 14, 15,  3],
       [ 6, 19, 14, 11, 10],
       [18,  6,  5, 17,  7],
       [ 9,  1, 16,  9,  8],
       [ 2, 10, 10, 15,  2]])
```

```
enteros_2d[1, 3]
```

```
5
```

43- Seleccionar elementos de la cuarta a la sexta fila en la segunda columna →

```
enteros_2d[3:6, 1]
```

```
array([ 4, 19,  6])
```

44- Seleccionar un subarreglo de filas de la cuarta a la sexta y columnas de la primera a la segunda →

```
enteros_2d[3:6, 0:2]
```

```
array([[ 6, 19],
       [18,  6],
       [ 9,  1]])
```

45- Elevar al cuadrado los elementos de una lista bidimensional usando np.power →

```
edades_1 = np.array([[2,3], [4,5]])
```

```
edades_2 = np.array([[6,7],[8,9]])
```

```
np.power(edades_1,2)
```

```
array([[ 4,  9],
       [16, 25]])
```

46- Sumar un valor constante a cada elemento de una lista →

```
edades + 2
```

```
array([[ 4,  9],
       [16, 25]])
```

47- Multiplicación de matrices utilizando el producto punto →

```
edades_1.dot(edades_2)
```

```
array([[36, 41],  
       [64, 73]])
```

48- Calcular el rango de temperaturas semanales en tres ciudades →

```
temperaturas = np.array([  
    [30, 32, 29, 35, 33, 31, 30],  
    [25, 27, 26, 28, 29, 24, 26],  
    [20, 21, 23, 22, 24, 25, 20]  
)
```

```
np.ptp(temperaturas, axis=1)
```

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

49- Calcular la mediana de ingresos anuales en una población →

```
ingresos = np.array([30, 25, 45, 50, 29, 40, 35])
```

```
np.median(ingresos)
```

35

50- Calcular el percentil 90 de las calificaciones de un examen →

```
calificaciones = np.array([85, 90, 78, 92, 88, 76, 95, 89, 84])
```

```
np.percentile(calificaciones, 90)
```

92.6

51- Cálculo del promedio ponderado de calificaciones →

```
calificaciones = np.array([85, 90, 78, 92])
```

```
pesos = np.array([0.3, 0.5, 0.1, 0.1])
```

```
np.average(calificaciones, weights = pesos)
```

87.5

52- Transposición de una matriz →

```
matriz = np.array([[1, 2, 3],[4, 5, 6]])
```

```
np.transpose(matriz)
```



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

Nota: Otra forma de hacerlo es utilizando `matriz.T` (atajo de transposición).

53- Resolución y verificación de un sistema de ecuaciones lineales $Ax=b$ con una matriz de 3×3 →

```
A = np.array([[2,1,-2], [3,0,1], [1,1,-1]])
```

```
b = np.array([-3],[5],[-2])
```

```
x = np.linalg.solve(A,b)
```

```
x
```

```
array([[ 1.],
       [-1.],
       [ 2.]])
```

```
np.allclose(np.dot(A,x),b)
```

```
True
```

54- Convertir un DataFrame en un array →

```
edad = pd.DataFrame({
    "Nombre": ["Andrea", "Juan", "Pedro"],
    "Edad": [22, 25, 30]
})
```

```
edad
```

	Nombre	Edad
0	Andrea	22
1	Juan	25
2	Pedro	30

```
array = edad.to_numpy()
```

```
array
```

```
array([[ 'Andrea', 22],
       [ 'Juan', 25],
       [ 'Pedro', 30]], dtype=object)
```

Nota: Para hacer el proceso inverso, asignamos el array a una variable y creamos el DataFrame usando esa variable, especificando los encabezados.

55- Modificar el valor en la posición $[1, 1]$ del array a 35 →

```
array[1, 1] = 35
```

array

```
array([[ 'Andrea', 22],  
       [ 'Juan', 35],  
       [ 'Pedro', 30]], dtype=object)
```

Nota: En matrices unidimensionales, solo se aplica `axis=0`, ya que tienen una única dimensión. En matrices bidimensionales, `axis=0` se refiere a las filas y `axis=1` a las columnas. En arreglos tridimensionales, `axis=0` se refiere a los bloques o matrices 2D, `axis=1` a las filas de cada matriz 2D, y `axis=2` a las columnas de cada matriz 2D.

Para arreglos de cuatro dimensiones, `axis=0` se refiere a los bloques 3D, `axis=1` a las matrices 2D en cada bloque, `axis=2` a las filas, y `axis=3` a las columnas.

Nota: Tanto los operadores aritméticos como las funciones equivalentes de NumPy permiten realizar operaciones de suma, resta, multiplicación y división entre arrays o valores escalares de manera eficiente.

Nota: En una matriz 4D con forma (2, 2, 3, 4), el primer número indica que hay 2 bloques 3D, el segundo número significa que cada bloque 3D contiene 2 matrices 2D, el tercer número representa que cada matriz 2D tiene 3 filas y el cuarto número indica que cada matriz 2D tiene 4 columnas.

Nota: Cuando utilizas `print()` en un ndarray, se muestra únicamente el contenido del array de forma limpia y sin información adicional sobre su tipo. Por otro lado, si mencionas el nombre de la variable que contiene el ndarray sin usar `print()`, se mostrará una representación más detallada que incluye su tipo, dimensiones y el contenido, lo que ayuda a entender su estructura.

Capítulo 23: Matplotlib y Seaborn

Matplotlib es una biblioteca de Python utilizada para la creación de gráficos y visualizaciones de datos. Es muy popular en la comunidad de ciencia de datos y programación debido a su versatilidad y facilidad de uso. Con Matplotlib, puedes generar una amplia variedad de gráficos, como gráficos de líneas, barras, histogramas, gráficos de dispersión, gráficos de pastel, entre otros.

Características principales:

- 1-. **Personalización:** Permite ajustar casi todos los aspectos de un gráfico, desde los colores, títulos, etiquetas de ejes, hasta el estilo de las líneas y los puntos.
- 2-. **Compatibilidad:** Se integra bien con otras bibliotecas de Python, como NumPy y pandas, lo que facilita la manipulación y visualización de grandes conjuntos de datos.
- 3-. **Interactividad:** Los gráficos pueden ser interactivos, permitiendo acercar, alejar o moverse por las visualizaciones.
- 4-. **Publicación de calidad:** Produce gráficos de alta calidad que pueden ser exportados a formatos como PNG, PDF o SVG para ser utilizados en publicaciones.

Historia de Matplotlib:

Matplotlib fue creada en 2003 por **John D. Hunter**, un neurocientífico que buscaba una herramienta flexible para crear gráficos científicos en Python. Inspirado en MATLAB, Hunter diseñó Matplotlib para proporcionar una interfaz similar, pero más accesible y personalizable. La biblioteca se centró en generar gráficos en 2D de alta calidad, con soporte para una amplia variedad de formatos de salida.

Con el tiempo, Matplotlib se convirtió en una herramienta clave en la visualización de datos en Python y fue adoptada por la comunidad científica. Es de código abierto, lo que ha permitido su crecimiento y mejoras constantes gracias a las contribuciones de numerosos desarrolladores.

Ejercicios prácticos:

1-. Importar la biblioteca de matplotlib →

```
import matplotlib.pyplot as plt
```

2-. Creación de un gráfico de línea y dispersión →

```
x = [0, 1, 2, 3, 4, 5]
```

```
y = [0, 1, 2, 3, 4, 5]
```

```
plt.figure(figsize=(8, 5))
```

```
plt.plot(x, y, color="black", label="Línea")
```

```
plt.scatter(x, y, color="red", label="Puntos")
```

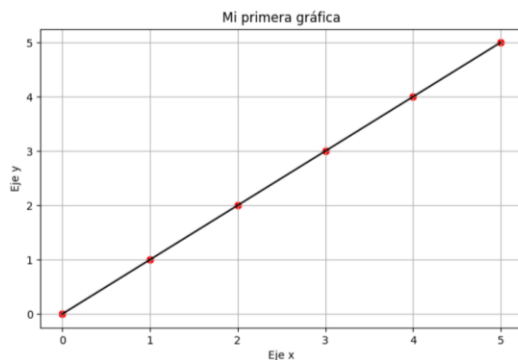
```
plt.title("Mi primera gráfica")
```

```
plt.xlabel("Eje x")
```

```
plt.ylabel("Eje y")
```

```
plt.grid(True)
```

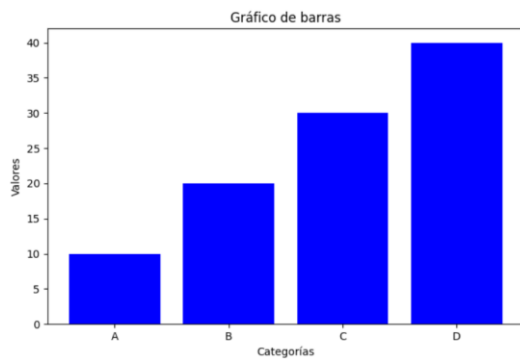
```
plt.show()
```



3-. Generación de un gráfico de barras →

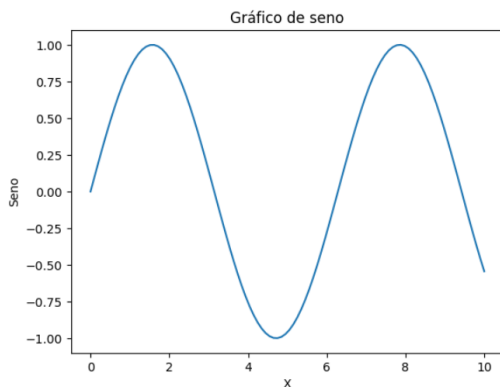
```
categorias = ["A", "B", "C", "D"]
```

```
valores = [10, 20, 30, 40]
plt.figure(figsize=(8, 5))
plt.bar(categorias, valores, color = "blue")
plt.title("Gráfico de barras")
plt.xlabel("Categorías")
plt.ylabel("Valores")
plt.show()
```



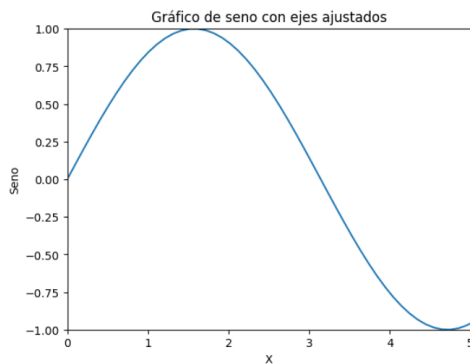
4-. Creación de un gráfico de la función seno →

```
x = np.linspace(0, 10, 100)
y = np.sin(x)
fig, ax = plt.subplots()
ax.plot(x,y)
ax.set_title("Gráfico de seno")
plt.xlabel("X")
plt.ylabel("Seno")
plt.show()
```



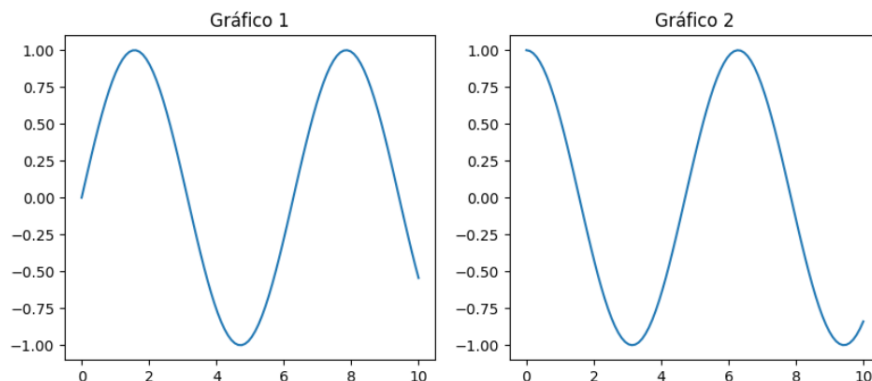
5-. Gráfico de seno con rango de ejes personalizado →

```
fig, ax = plt.subplots()
ax.plot(x,y)
ax.set_xlim([0,5])
ax.set_ylim([-1,1])
ax.set_title("Gráfico de seno con ejes ajustados")
plt.xlabel("X")
plt.ylabel("Seno")
plt.show()
```



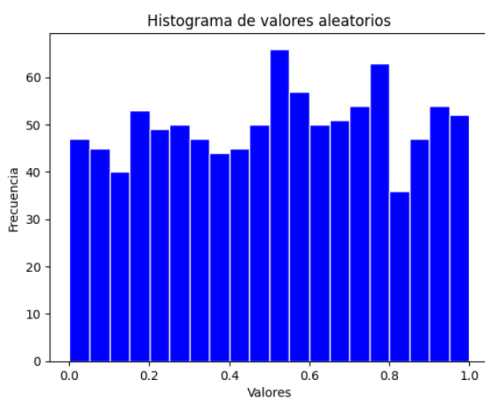
6-. Creación de subgráficos con funciones seno y coseno →

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10,4))
ax1.set_title("Gráfico 1")
ax1.plot(x, y)
ax2.set_title("Gráfico 2")
ax2.plot(x, np.cos(x))
plt.show()
```



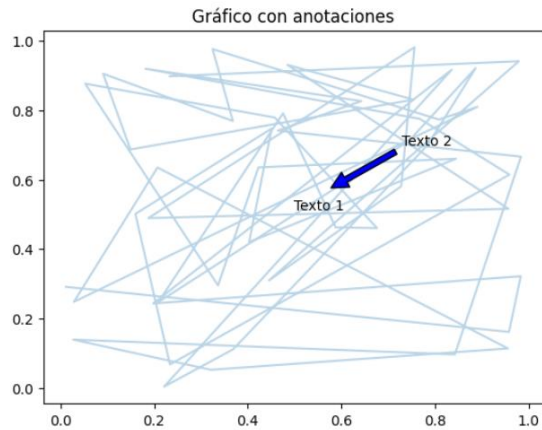
7-. Creación de un histograma de datos aleatorios →

```
data = np.random.rand(1000)
fig, ax = plt.subplots()
ax.hist(data, bins = 20, color = "blue", edgecolor = "white")
ax.set_title("Histograma de valores aleatorios")
plt.xlabel("Valores")
plt.ylabel("Frecuencia")
plt.show()
```



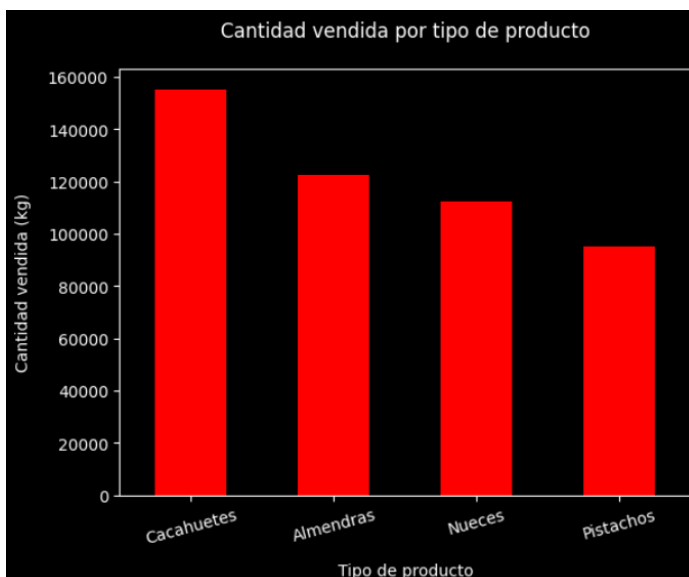
8-. Creación de un histograma de datos aleatorios →

```
x = np.random.rand(50)
y = np.random.rand(50)
fig, ax = plt.subplots()
ax.plot(x, y, alpha = 0.3)
ax.text(0.5, 0.52, "Texto 1", transform = ax.transAxes)
ax.annotate("Texto 2", xy = (0.57, 0.57), xytext = (0.73, 0.7), arrowprops = dict(facecolor =
"blue", shrink = 0.05))
ax.set_title("Gráfico con anotaciones")
plt.show()
```



9-. Visualización de ventas de frutos secos por tipo de producto en un gráfico de barras ➔

```
data = pd.read_csv("Frutos_Secos.csv", delimiter = ";")
ventas_por_producto = data.groupby("Tipo Producto")["Cantidad Vendida (kg)"].sum().sort_values(ascending = False)
plt.style.use("dark_background")
fig, ax = plt.subplots()
ventas_por_producto.plot(kind = "bar", color = "red", ax = ax)
ax.set_title("Cantidad vendida por tipo de producto", fontsize = 12, pad = 20)
ax.set_xlabel("Tipo de producto", fontsize = 10, labelpad = 10)
ax.set_ylabel("Cantidad vendida (kg)", fontsize = 10, labelpad = 10)
plt.xticks(rotation = 15)
plt.show()
```



10-. Visualización de estilos disponibles→

```
estilos = plt.style.available
```

```
for estilo in estilos:
```

```
    print(estilo)
```

```
Solarize_Light2
_classic_test_patch
_mpl-gallery
_mpl-gallery-nogrid
bmh
classic
dark_background
fast
fivethirtyeight
ggplot
grayscale
seaborn-v0_8
seaborn-v0_8-bright
seaborn-v0_8-colorblind
seaborn-v0_8-dark
seaborn-v0_8-dark-palette
seaborn-v0_8-darkgrid
seaborn-v0_8-deep
seaborn-v0_8-muted
seaborn-v0_8-notebook
seaborn-v0_8-paper
seaborn-v0_8-pastel
seaborn-v0_8-poster
seaborn-v0_8-talk
seaborn-v0_8-ticks
seaborn-v0_8-white
seaborn-v0_8-whitegrid
tableau-colorblind10
```

11-. Aplicación de estilos y creación de un gráfico personalizado→

```
plt.style.use("ggplot")
```

```
x = [0, 1, 2, 3, 4, 5]
```

```
y = [0, 1, 4, 9, 16, 25]
```

```
plt.figure(figsize=(8, 5))
```

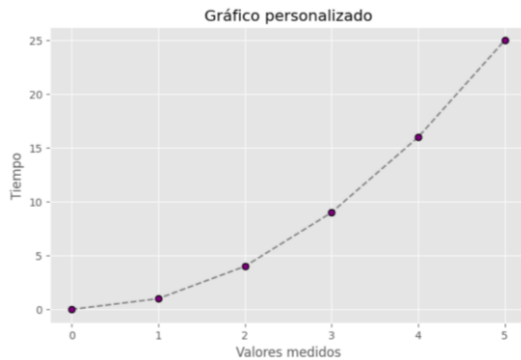
```
plt.plot(x, y, color="gray", linestyle="--", marker="o", markerfacecolor="purple",
markeredgecolor="black")
```

```
plt.xlabel("Valores medidos")
```

```
plt.ylabel("Tiempo")
```

```
plt.title("Gráfico personalizado")
```

```
plt.show()
```

Seaborn es una biblioteca de Python para la visualización de datos, construida sobre Matplotlib. Está diseñada para hacer que la creación de gráficos sea más sencilla y estéticamente agradable, proporcionando gráficos estadísticos de alto nivel con menos código. Seaborn es especialmente útil para trabajar con datos de tipo tabular, como los que se encuentran en DataFrames de pandas, y está orientada a mostrar relaciones entre variables, distribuciones y patrones complejos.

Características principales:

- 1-. Gráficos estadísticos:** Facilita la creación de gráficos como diagramas de dispersión, gráficos de barras, gráficos de cajas (box plots), gráficos de violín y mapas de calor (heatmaps), entre otros.
- 2-. Estilos predefinidos:** Proporciona estilos visuales atractivos por defecto, eliminando la necesidad de configurar cada detalle de los gráficos, como colores o tipos de ejes.
- 3-. Integración con pandas:** Funciona de manera natural con DataFrames de pandas, permitiendo la visualización directa de datos tabulares.
- 4-. Facilidad de uso:** Requiere menos líneas de código que Matplotlib para generar visualizaciones complejas y es ideal para explorar datos rápidamente.

Historia de Seaborn:

Seaborn fue creada por **Michael Waskom** en 2014 como una extensión de Matplotlib, con el objetivo de simplificar la creación de gráficos estadísticos y mejorar la estética visual de las visualizaciones en Python. Waskom, un neurocientífico, diseñó la biblioteca para facilitar el análisis exploratorio de datos, proporcionando gráficos de alto nivel que permiten mostrar relaciones y distribuciones con menos código.

Seaborn se integra perfectamente con pandas y Matplotlib, lo que la hace popular en el ámbito de la ciencia de datos. Su enfoque en gráficos estadísticos y su capacidad para generar visualizaciones atractivas la han convertido en una herramienta indispensable. Como proyecto de código abierto, ha crecido gracias a las contribuciones de la comunidad.

Ejercicios prácticos:

- 1-. Importar la biblioteca de seaborn →**

```
import seaborn as sns
```

2-. Carga del conjunto de datos “Iris” →

```
datos_1 = sns.load_dataset("Iris")
```

3-. Creación de un gráfico de dispersión →

```
plt.figure(figsize=(8, 5))
```

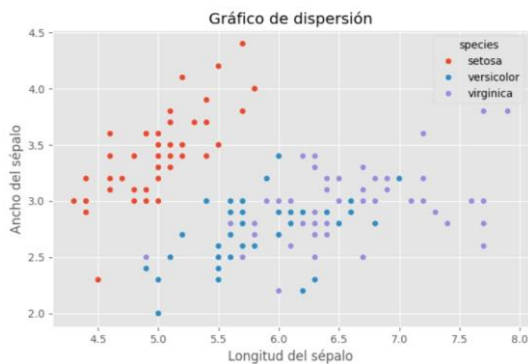
```
sns.scatterplot(x="sepal_length", y="sepal_width", hue="species", data = datos_1)
```

```
plt.xlabel("Longitud del sépalo")
```

```
plt.ylabel("Ancho del sépalo")
```

```
plt.title("Gráfico de dispersión")
```

```
plt.show()
```



Nota: La instrucción `sns.set_style` permite personalizar el fondo, las cuadrículas y otros elementos visuales del gráfico, ayudando a mejorar la claridad y la legibilidad de los datos.

4-. Generación de un ridgeplot (o gráfico de crestas) para longitudes de pétalo →

```
setosa = datos_1[datos_1 ["species"] == "setosa"]
```

```
versicolor = datos_1 [datos_1 ["species"] == "versicolor"]
```

```
virginica = datos_1 [datos_1["species"] == "virginica"]
```

```
plt.figure(figsize=(8, 6))
```

```
plt.xlabel("Longitud del pétalo")
```

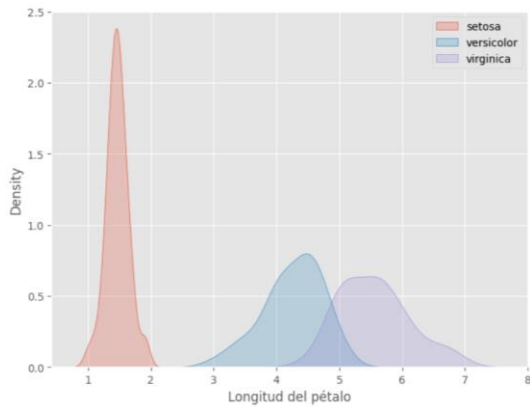
```
sns.kdeplot(setosa["petal_length"], label="setosa", fill=True)
```

```
sns.kdeplot(versicolor["petal_length"], label="versicolor", fill=True)
```

```
sns.kdeplot(virginica["petal_length"], label="virginica", fill=True)
```

```
plt.legend(loc="upper right")
```

```
plt.show()
```



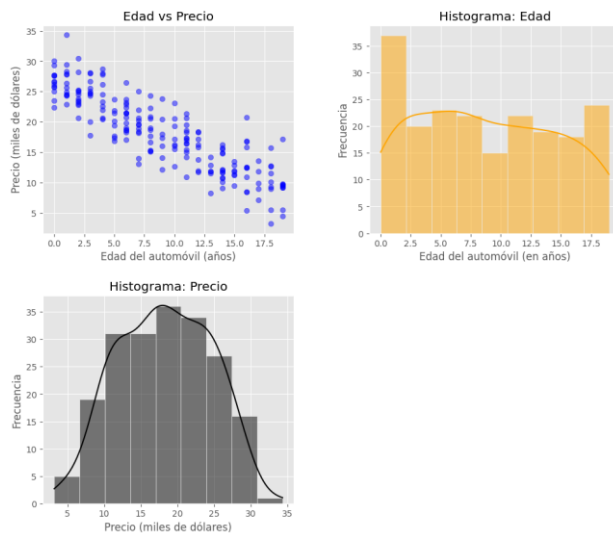
Nota: Para este gráfico de densidad (kdeplot), el eje Y generalmente representa la “densidad” o “probabilidad”. Esto se debe a que estás trazando gráficos de densidad kernel, que miden cuán frecuente o probable es un rango de valores en el conjunto de datos.

Nota: El parámetro `fill=True` se utiliza para rellenar el área bajo la curva de densidad. Cuando activas esta opción, Seaborn colorea la región debajo de cada curva de densidad, lo que facilita visualizar la distribución y comparar entre los diferentes grupos (en este caso, las especies de flores).

5-. Creación de gráficos compuestos con matplotlib y seaborn →

```
import pandas as pd
import numpy as np
np.random.seed(42)
edad_autos = np.random.randint(0, 20, size=200)
precio_autos = 30 - edad_autos + np.random.normal(-3, 3, size=200)
data = pd.DataFrame({ "Edad": edad_autos, "Precio": precio_autos })
fig, ax = plt.subplots(2, 2, figsize = (12, 10))
ax[0, 0].scatter(data["Edad"], data["Precio"], color = "blue", alpha = 0.5)
ax[0, 0].set_xlabel("Edad del automóvil (años)")
ax[0, 0].set_ylabel("Precio (miles de dólares)")
ax[0, 0].set_title("Edad vs Precio")
sns.histplot(data["Edad"], ax = ax[0, 1], kde = True, color = "orange")
ax[0, 1].set_xlabel("Edad del automóvil (en años)")
ax[0, 1].set_ylabel("Frecuencia")
ax[0, 1].set_title("Histograma: Edad")
```

```
sns.histplot(data["Precio"], ax = ax[1, 0], kde = True, color = "black")
ax[1, 0].set_xlabel("Precio (miles de dólares)")
ax[1, 0].set_ylabel("Frecuencia")
ax[1, 0].set_title("Histograma: Precio")
ax[1, 1].axis("off")
plt.subplots_adjust(wspace=0.3, hspace=0.3)
plt.show()
```



Nota: La variable ax es una matriz (array) que almacena cada subgráfico en las posiciones correspondientes a la cuadrícula de 2x2 que se creó.

Nota: alpha=0.5 ajusta la transparencia de los puntos para evitar la saturación visual si hay demasiados puntos superpuestos.

Nota: El parámetro kde=True añade una curva de densidad (KDE - Kernel Density Estimate) que suaviza la distribución, facilitando la comprensión visual.

Capítulo 24: Análisis exploratorio de datos

El **análisis exploratorio de datos (EDA)** es un enfoque inicial en el proceso de análisis de datos que se utiliza para comprender mejor sus características antes de aplicar modelos estadísticos o técnicas más complejas. Su objetivo es identificar patrones, detectar valores atípicos, probar hipótesis preliminares y resumir las principales características de los datos.

Elementos clave del proceso:

1-. Resumen descriptivo: Calcular estadísticas básicas como la media, mediana, moda, varianza y percentiles para entender la distribución de los datos.

2-. Visualización: Utilizar gráficos como histogramas, diagramas de dispersión (scatter plots), diagramas de caja (boxplots) y diagramas de barras para observar patrones y relaciones entre variables.

3-. Detección de valores atípicos: Identificar datos que se alejan mucho del resto y que pueden influir en los resultados o revelar información interesante.

4-. Relaciones entre variables: Examinar la correlación y otras relaciones entre variables para entender cómo interactúan entre sí.

5-. Distribución de datos: Observar cómo se distribuyen las variables (si son normales, sesgadas, etc.), lo que puede ayudar a seleccionar los métodos de análisis adecuados.

Dentro del EDA, es fundamental comprender cómo se comportan las variables y cómo se relacionan entre sí.

Por ende, existen varias variaciones de análisis que son esenciales en este proceso, entre las cuales están:

Análisis univariado	Análisis bivariado	Análisis multivariado
Técnica estadística que se centra en el estudio de una sola variable a la vez. Su objetivo principal es describir y resumir las características de esa variable, proporcionando una comprensión clara de su distribución, tendencia central y dispersión.	Técnica estadística que se utiliza para analizar la relación entre dos variables. Su objetivo principal es determinar si existe una asociación entre las variables y cómo se comportan en conjunto.	Técnica estadística que examina la relación entre tres o más variables al mismo tiempo. Su objetivo es entender cómo estas variables interactúan entre sí, identificar patrones complejos, y obtener información más detallada sobre los datos en escenarios donde múltiples factores están involucrados.

Medidas más utilizadas	Principales enfoques	Métodos más comunes
Medidas de tendencia central - Media: El promedio de los valores. - Mediana: El valor que divide los datos en dos mitades. - Moda: El valor que ocurre con mayor frecuencia.	Correlación: Mide la fuerza y dirección de la relación entre dos variables. Se utilizan métricas como el coeficiente de correlación de Pearson para variables numéricas, que varía entre -1 y 1, donde 1 indica una correlación positiva perfecta, -1 indica una correlación negativa perfecta y 0 indica que no hay correlación.	Análisis de regresión múltiple: Permite predecir una variable dependiente a partir de varias variables independientes.

<p>Medidas de dispersión</p> <ul style="list-style-type: none"> - Varianza: Qué tan dispersos están los valores respecto a la media. - Desviación estándar: Raíz cuadrada de la varianza, que muestra la dispersión en las mismas unidades que los datos. - Rango: Diferencia entre el valor máximo y el mínimo. 	<p>Tablas de contingencia: Utilizadas para variables categóricas, permiten analizar la frecuencia con la que ocurren combinaciones específicas de dos variables.</p>	<p>Análisis de componentes principales (PCA): Utilizado para reducir la dimensionalidad de los datos manteniendo la mayor variabilidad posible, lo que facilita la visualización y el análisis.</p>
<p>Distribución</p> <ul style="list-style-type: none"> - Forma de la distribución: Examina si los datos están sesgados (asimetría) o si tienen colas largas (curtosis). - Visualización: Gráficos como histogramas o diagramas de caja permiten ver cómo se distribuyen los datos. 	<p>Gráficos de dispersión para variables numéricas: Muestran visualmente la relación entre dos variables, donde cada punto en el gráfico representa un par de valores</p>	<p>Análisis factorial: Agrupa variables en factores latentes o subyacentes para explicar patrones comunes y reducir el número de variables.</p>
	<p>Prueba chi-cuadrado: Se utiliza para evaluar la independencia entre dos variables categóricas</p>	<p>Análisis discriminante: Se utiliza para predecir la categoría a la que pertenece una observación en función de varias variables predictoras.</p>

	Regresión lineal simple: Modelo utilizado para analizar la relación entre una variable dependiente (respuesta) y una variable independiente (predictora), cuando ambas son numéricas.	Análisis de conglomerados (clustering): Agrupa las observaciones en subconjuntos (clusters) según su similitud, permitiendo segmentar datos en grupos que comparten características comunes.
--	---	---

Capítulo 25: Minería de datos

La **minería de datos**, o data mining, es el proceso de descubrir patrones, tendencias y conocimientos útiles a partir de grandes conjuntos de datos. Utiliza técnicas de estadística, aprendizaje automático y análisis de bases de datos para identificar información valiosa que puede ayudar a tomar decisiones informadas.

Algunos aspectos clave de este proceso son:

- 1-. **Recolección de datos:** Se recopilan datos de diversas fuentes, como bases de datos, archivos, redes sociales y sensores.
- 2-. **Preprocesamiento:** Los datos se limpian y preparan para el análisis, lo que puede incluir la eliminación de duplicados, el manejo de valores faltantes y la normalización.
- 3-. **Análisis:** Se aplican algoritmos y técnicas de análisis, como la clasificación, la regresión, el agrupamiento y la detección de anomalías, para extraer patrones y conocimientos.
- 4-. **Interpretación:** Los resultados se interpretan para obtener información útil que pueda aplicarse a problemas específicos o a la toma de decisiones estratégicas.
- 5-. **Visualización:** Los hallazgos se presentan de manera visual, facilitando la comprensión y la comunicación de los resultados a las partes interesadas.

¿Qué es CRISP-DM?

CRISP-DM (Cross-Industry Standard Process for Data Mining) es un modelo de proceso estándar que guía la ejecución de proyectos de minería de datos. Fue desarrollado en los años 90 por un consorcio de empresas para proporcionar una metodología estructurada y flexible para la minería de datos, aplicable en diferentes industrias.

Fases principales:

- 1-. **Comprensión del negocio:** Implica identificar los objetivos y requisitos del negocio, comprender el contexto del problema, y traducirlo en un problema de minería de datos.

2-. Comprensión de los datos: Se recogen los datos disponibles, se exploran, y se evalúa su calidad. También se identifican características importantes y relaciones en los datos que podrían ser relevantes para el análisis.

3-. Preparación de los datos: En esta fase, los datos se limpian y preparan para el análisis. Esto puede incluir tareas como la selección de variables, la transformación de datos y la creación de nuevas variables derivadas.

4-. Modelado: Se seleccionan y aplican técnicas de modelado (algoritmos de minería de datos) que sean apropiadas para el problema. Puede ser necesario ajustar los parámetros de los modelos para obtener los mejores resultados.

5-. Evaluación: Se evalúan los modelos para verificar que cumplen los objetivos definidos en la fase de comprensión del negocio. A menudo, se comparan varios modelos para elegir el más adecuado.

6-. Despliegue: Una vez que se ha seleccionado el modelo más adecuado, se implementa en el entorno de producción. Esto puede implicar el uso del modelo para generar informes, hacer predicciones o mejorar los procesos del negocio.