Intro Python

En este curso utilizaremos cuadernos Jupyter para escribir y ejecutar código Python, por lo que el primer paso será aprender a utilizarlos. Los documentos de cuaderno (o "notebooks", todo en minúsculas) son documentos creados por la aplicación Jupyter Notebook, que contienen tanto código informático (por ejemplo, python) como elementos de texto enriquecido (párrafos, ecuaciones, figuras, enlaces, etc...). Los cuadernos están formados por celdas, que son bloques de código o texto enriquecido. Cada una de estas es editable, aunque principalmente estarás modificando las celdas de código para responder a las preguntas.

Descripción técnica de un cuaderno Jupyter

La aplicación Jupyter Notebook es una aplicación servidor-cliente que permite editar y ejecutar documentos de cuaderno a través de un navegador web. La versión de IBM de la aplicación Jupyter Notebook está instalada en un servidor remoto y se accede a ella a través de internet.

Un kernel de cuaderno es un "motor computacional" que ejecuta el código contenido en un documento de cuaderno. El kernel de ipython, mencionado en esta guía, ejecuta código Python. Existen kernels para muchos otros lenguajes (consulta el menú Kernels más arriba).

Cuando abres un documento de cuaderno, el kernel asociado se inicia automáticamente. Al ejecutar el cuaderno (ya sea celda por celda o mediante el menú Celda -> Ejecutar todo), el kernel realiza los cálculos y produce los resultados. Dependiendo del tipo de cálculos, el kernel puede consumir una cantidad significativa de CPU y RAM. Ten en cuenta que la RAM no se libera hasta que el kernel se cierra.

Introducción a PEP-8 y `import this`

PEP-8

PEP-8, también conocido como "Python Enhancement Proposal 8", es la guía de estilo para escribir código en Python. Fue concebido para facilitar la lectura y comprensión del código, promoviendo la consistencia en la forma en que los programadores de Python escriben su código. Algunos de los principios clave de PEP-8 incluyen la utilización de sangrías con cuatro espacios, líneas que no excedan los 79 caracteres y la definición de funciones y variables en nombres en minúsculas separados por guiones bajos. Es altamente recomendable adherirse a PEP-8 para mantener un código Python limpio y legible.

`import this`

El comando `import this` o PEP-20 en Python es una pequeña "huevito de pascua" (easter egg) incluido en el lenguaje, que cuando se ejecuta, muestra el **"Zen de Python"**. El "Zen de Python" es una colección de 19 aforismos que expresan la filosofía de diseño del lenguaje Python. Algunos de estos aforismos hacen hincapié en la importancia de la legibilidad del código y la simplicidad sobre la complejidad. Puedes ver estos aforismos en cualquier momento mientras programas en Python ejecutando el comando `import this` en tu consola o script de Python.

Para utilizarlo, simplemente escribe el siguiente comando en tu intérprete de Python:

```
import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1. Cell types in Jupyter Notebook

`ctrl-enter`: ejecuta la celda activa y permanece en esa celda

`shift-enter`: ejecuta la celda activa y se desplaza a la siguiente celda

Pruébalo Ejecuta las siguientes celdas:

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

```
Hola Mundo
```

```
print ("Segunda línea")
```

```
Segunda línea
```

Insertando nuevas celdas: Cuando una celda está seleccionada en azul (haz clic en el margen izquierdo de la celda) o, si estás editando la celda, pulsa la tecla de escape, se muestra un cuadro azul alrededor.

Escribe:

- `a` (above/arriba) para crear una nueva celda vacía encima de la celda activa en ese momento
- `b` (below/abajo) para crear una nueva celda vacía debajo de la celda activa en ese momento

Pruébalo: Añade una celda debajo y luego una celda encima

Código de Markdown: El atajo `m` (con la selección azul) cambia la celda de computación a markdown. Esto permite crear elementos de texto enriquecido para documentar el código. Recíprocamente, puedes convertir una celda en una celda de código utilizando el atajo `y`.

Pruébalo: Convierte la primera celda de abajo en una celda de código y ejecútala, y la celda de abajo en una celda de markdown

No es una celda de codigo ahora mismo:

print("Ahora si que es una celda de codigo :)")

```
#### **Es una celda de codigo ahora mismo**
```

Ejercicio: Convertir Códigos a Markdown

En este ejercicio, tu tarea será convertir las siguientes celdas de código que contienen encabezados (headings) y texto enriquecido a celdas de markdown. De esta manera, en lugar de verse como un bloque de código, se visualizarán como texto con formato.

1. Selecciona la primera celda que contiene los encabezados y el texto enriquecido escrito como código.
2. Utiliza el atajo `m` (asegurándote de que la celda está resaltada en azul) para convertir la celda de código a una celda de markdown.
3. Presiona `Shift+Enter` o `Ctrl+Enter` para ejecutar la celda y ver el resultado.
4. Si has realizado el paso correctamente, deberías ver los encabezados y el texto enriquecido (negritas, cursivas, etc.) aplicados en lugar de como un bloque de código.

Inténtalo: Convierte las celdas con los encabezados y el texto enriquecido de código a markdown y ejecútalas para ver el resultado.

Recuerda, las celdas deben verse igual que las celdas de ejemplo proporcionadas más abajo.

¡Buena suerte!

```
'''
# Encabezado 1
## Encabezado 2
### Encabezado 3
#### Encabezado 4

**negrita**

*cursiva*

una línea vacía es un párrafo

esto es un nuevo párrafo
'''
```

Si has convertido la celda a una celda de markdown con éxito, debería mostrarse igual que la celda de abajo.

Encabezado 1

Encabezado 2

Encabezado 3

Encabezado 4

negrita

cursiva

una línea vacía es un párrafo

esto es un nuevo párrafo

Explicación de Elementos Markdown

En el ejemplo proporcionado, hay dos tipos de elementos Markdown muy útiles y comunes: enlaces y imágenes.

1. **Enlace** La primera línea `[nomre](url)` es un enlace. La sintaxis para crear un enlace es colocar el texto del enlace entre corchetes `[]` y la URL entre paréntesis `()`. Al hacer clic en el texto "Esto es Google", te llevará a la página web de Google España.

2. **Imagen** La segunda línea `![nombre](url image)` es una imagen. La sintaxis para insertar una imagen es muy similar a la de un enlace, pero lleva un signo de exclamación `!` al principio. El texto entre corchetes `[]` sirve como descripción alternativa para la imagen, que se muestra si la imagen no puede renderizarse por

cualquier motivo. En este caso, al ejecutar la celda, verás el texto "Esto es una imagen (que no se va a renderizar)" si la imagen no se puede cargar desde la URL proporcionada.

Puedes probar estos elementos por ti mismo en cualquier celda de markdown. ¡Inténtalo!

Enlace Esto es google

Image:

Shortcuts

Recordatorio Final de Atajos de Teclado

Para ejecutar la celda:

- `ctrl + enter`
- `shift + enter` # este ejecuta la celda y pasa a la siguiente

Para insertar nuevas celdas:

- `a` para nueva celda arriba
- `b` para nueva celda abajo
- `d + d` para modo de atajo (elimina la celda seleccionada)

Para alternar entre markdown y código:

- `m` o `y` para cambiar entre markdown y código

Type of data

En esta sección, exploraremos diferentes tipos de datos que puedes encontrar y utilizar en Python. Comprender los diferentes tipos de datos es fundamental para trabajar eficazmente con Python.

- **Números enteros (Integer)**: Son números sin decimales, pueden ser tanto positivos como negativos. Por ejemplo, -3, 0, 42 son todos números enteros.

- **Números reales (Float)**: Son números que contienen puntos decimales o están escritos en notación científica. Incluyen valores como 3.14, -0.001 o 2.5e2.
- **Cadenas de caracteres (Strings)**: Las cadenas son secuencias de caracteres (letras, números, símbolos, emojis) encerrados entre comillas simples (`'`) o dobles (`"`). Por ejemplo, "Hola, Mundo" o 'Python3' son cadenas.
- **Booleanos (Boolean)**: Los valores booleanos solo pueden ser `True` (verdadero) o `False` (falso), y representan el resultado de operaciones lógicas.

Recuerda que Python es un lenguaje de programación dinámico y de tipado fuerte, lo que significa que no necesitas declarar el tipo de datos de una variable explícitamente; Python lo inferirá por ti.

Integer numbers

Los números enteros, también conocidos como "integers" en inglés, son un tipo de datos que representa números enteros, es decir, sin decimales. Pueden ser tanto positivos como negativos. En Python, puedes asignar un valor entero a una variable simplemente escribiendo el número sin ningún decimal o comillas. Por ejemplo:

```
un_entero = 4
```

En esta celda, estamos declarando una variable llamada `un_entero` y le asignamos el valor 4, que es un número entero.

```
un_entero
```

```
4
```

Aquí, simplemente estamos escribiendo el nombre de la variable. Esto hará que el cuaderno Jupyter imprima su valor, que es 4.

```
print(un_entero)
```

```
4
```

En esta celda, estamos utilizando la función `print()` para imprimir el valor de la variable `un_entero`. También verás 4 como salida aquí, pero a diferencia de la celda anterior, estás utilizando específicamente una función para imprimir el valor.

```
a = 10
b = 20
a
b # Notebook imprime el último valor
```

```
20
```

En este conjunto de líneas, primero asignamos 10 a `a` y 20 a `b`. Luego, escribimos `a` y `b` en líneas separadas, pero el cuaderno Jupyter solo imprimirá el valor de `b`, ya que es el último en la celda. Esto se indica con el comentario # Notebook imprime el último valor.

```
a = 10
b = 20
print(a)
b # Notebook imprime el último valor
```

```
10
```

```
20
```

Aquí, es similar a la celda anterior, pero esta vez estamos utilizando la función `print()` para imprimir el valor de `a` antes de escribir simplemente `b`. Verás tanto 10 (la salida de `print(a)`) como 20 (el valor de `b`) en la salida.

```
print(type(a))
```

```
<class 'int'>
```

Finalmente, estamos utilizando la función `type()` para imprimir el tipo de datos de la variable `a`, que en este caso es `<class 'int'>`, indicando que es un número entero.

Real numbers (floats)

En esta sección, vamos a explorar otro tipo de datos en Python: los números reales, también conocidos como "flotantes" o "floats" en inglés. Los números flotantes pueden representar números fraccionarios, es decir, números que tienen tanto una parte entera como una parte decimal. Pueden ser tanto positivos como negativos.

En Python, puedes crear un número flotante simplemente incluyendo un punto decimal en el número, o utilizando notación científica para números muy grandes o muy pequeños. Aquí hay algunos ejemplos:

x = 5.67

y = -0.23

z = 3.0e8

En este código:

- `x` es un número flotante que representa el número 5.67.
- `y` es un número flotante que representa el número -0.23.
- `z` es un número flotante que representa el número 300,000,000 (o 3.0 × 10^8, utilizando notación científica).

Al igual que con los números enteros, puedes usar la función `type()` para verificar el tipo de datos de una variable. Por ejemplo, `type(x)` devolverá `<class 'float'>`, indicando que `x` es un número flotante.

Los números flotantes son útiles cuando necesitas realizar cálculos que requieren precisión decimal. Ahora, vamos a practicar trabajando con números flotantes en Python con algunos ejercicios.

```
a = 12.34
```

En la celda anterior, hemos asignado el número float 12.34 a la variable `a`. Ahora, vamos a imprimir el valor de `a` para verificarlo.

```
print(a)
```

```
12.34
```

Ahora, vamos a usar la función `type()` para verificar el tipo de datos de la variable `a`. Esto debería confirmar que es un número flotante.

```
print(type(a))
```

```
<class 'float'>
```

```
b = 12.0
```

Ahora, intenta predecir el tipo de dato de la variable `b` antes de verificarlo con la función `type()`. ¿Crees que es:

- **int?**
- **float?**

```
type(b)
```

```
float
```

En esta última celda, verificamos el tipo de datos de `b` usando la función `type()`. Aunque `b` tiene un valor que podría ser un entero, se considera un float porque incluye un punto decimal.

Basic operations

En esta sección, exploraremos algunas operaciones básicas en Python. Aprenderemos a realizar sumas, restas, divisiones, y cómo obtener el módulo y la floor division. También veremos cómo podemos usar el módulo para determinar si un número es par o impar. ¡Empecemos con algunos ejercicios prácticos!

```
a = 10
b = 3
```

```
10 + 3
```

En la celda anterior, simplemente hemos sumado 10 y 3 directamente en una celda de código. Ahora, haremos lo mismo, pero utilizando las variables `a` y `b` que hemos definido previamente.

```
# Suma
a + b
```

Ahora procederemos a realizar una resta utilizando las mismas variables, `a` y `b`.

```
# resta

a - b
```

A continuación, exploraremos cómo hacer divisiones en Python. Primero, realizaremos una división normal y luego verificararemos el tipo de dato del resultado.

```
# divisiones

division = a / b
print(division)
```

```
1.0283333333333333
```

```
type(division)
```

```
float
```

Ahora, vamos a aprender sobre la "floor division", que redondea el resultado de la división hacia abajo al número entero más cercano. También comprobaremos el tipo de dato del resultado.

```
# divisiones: floor division: división redondeada hacia abajo

floor_division = a // b
floor_division
```

```
1.0
```

```
type(floor_division)
```

A continuación, exploraremos el operador módulo (`%`), que nos da el resto de una división. Primero, encontraremos el módulo de `a` dividido por `b`.

```
#Módulo: resto de la división

a % b
```

```
0.33999999999999986
```

Para comprender mejor cómo funcionan las divisiones y el operador módulo, imprimiremos los valores de `a` y `b`.

```
a
```

```
b
```

A continuación, utilizaremos el operador módulo para determinar si los números en una lista son pares o impares. Si un número dividido por 2 da un resto de 0, entonces es par. Vamos a crear un bucle que recorra una lista de números y nos indique si cada número es par o impar.

```
# Par / impar -> módulo (resto)
# Si el resto de una división entre dos es cero: par

list_ = [1, 2, 3, 4, 5, 6]

for i in list_:
    if i % 2 == 0:
        print(f"El numero {i} es par")
    else:
        print(f"El numero {i} NO es par")
```

```
El numero 1 NO es par
El numero 2 es par
El numero 3 NO es par
El numero 4 es par
El numero 5 NO es par
El numero 6 es par
```

Ejercicio Práctico

Ahora que hemos explorado algunas operaciones básicas en Python, es tu turno de intentarlo.

Instrucciones:

1. Crea dos nuevas variables, `x` e `y`, y asigna cualquier número entero a cada una de ellas.
2. Realiza las siguientes operaciones usando estas variables:
   - Suma
   - Resta
   - Multiplicación
   - División y comprobar el tipo de resultado
   - floor division y comprobar el tipo de resultado
   - Encuentra el módulo (resto de la división)
3. Usa un bucle `for` para iterar sobre una lista de números del 1 al 10 e imprime si cada número es par o impar.

No olvides imprimir los resultados para verificar tus soluciones.

```
# tu solución aquí

# 1. Crea dos nuevas variables, x e y, y asigna cualquier número
entero a cada una de ellas
x = 15
```

```python
y = 4

# 2. Realiza las operaciones usando estas variables

# Suma
suma = x + y
print(f'Suma: {x} + {y} = {suma}')

# Resta
resta = x - y
print(f'Resta: {x} - {y} = {resta}')

# Multiplicación
multiplicacion = x * y
print(f'Multiplicación: {x} * {y} = {multiplicacion}')

# División y comprobar el tipo de resultado
division = x / y
print(f'División: {x} / {y} = {division} (tipo: {type(division)})')

# Floor division (división entera) y comprobar el tipo de resultado
floor_division = x // y
print(f'División entera: {x} // {y} = {floor_division} (tipo:
{type(floor_division)})')

# Módulo (resto de la división)
modulo = x % y
print(f'Módulo: {x} % {y} = {modulo}')

# 3. Usa un bucle for para iterar sobre una lista de números del 1 al
10 e imprime si cada número es par o impar
print("\nNúmeros del 1 al 10: Par o Impar")
for num in range(1, 11):
    if num % 2 == 0:
        print(f'{num} es Par')
    else:
        print(f'{num} es Impar')
```

```
Suma: 15 + 4 = 19
Resta: 15 - 4 = 11
Multiplicación: 15 * 4 = 60
División: 15 / 4 = 3.75 (tipo: <class 'float'>)
División entera: 15 // 4 = 3 (tipo: <class 'int'>)
Módulo: 15 % 4 = 3

Números del 1 al 10: Par o Impar
1 es Impar
2 es Par
3 es Impar
4 es Par
```

```
5 es Impar
6 es Par
7 es Impar
8 es Par
9 es Impar
10 es Par
```

built-in and imported things

En el mundo de Python, a menudo trabajarás con diferentes métodos y funciones para realizar tareas específicas en tu código. Los "métodos" y las "funciones" son esencialmente cosas que realizan acciones (o, como decimos coloquialmente, "cosas que hacen cosas"). A continuación, vamos a explorar dos categorías principales de estos: las cosas incorporadas y las cosas importadas (*resumen methods = functions = things that do things*).

- **Métodos incorporados (built-in methods):** Son funciones o cosas que ya están incluidas en Python cuando lo instalas. No necesitas instalar nada extra para usarlos. Algunos ejemplos son los métodos `print` y `sum`.
- **Cosas importadas (imported things):** A veces, podrías necesitar usar funciones o cosas que no están incorporadas directamente en Python. En estos casos, necesitarás importarlos desde una biblioteca externa. Antes de poder usar estas funciones, deberás instalar la biblioteca correspondiente con un comando `pip install` o `conda install`, y luego importarla en tu script.

A continuación, exploraremos algunos ejemplos de ambos tipos de "cosas":

```
# Primero, vamos a explorar un método incorporado: print
# El método print nos permite imprimir mensajes en la consola.
print("Se llama built-in")
```

```
Se llama built-in
```

```
# Otro ejemplo de un método incorporado es upper.
# Este método convierte una cadena de texto (string) en mayúsculas.
"Esto es una string".upper()
```

```
'ESTO ES UNA STRING'
```

```
# Ahora, vamos a explorar cómo importar y usar cosas de una biblioteca
externa.
# Primero, necesitamos importar la biblioteca. En este caso, estamos
importando la biblioteca math.
import math

# A continuación, usamos una función de la biblioteca math: floor.
# La función floor redondea un número hacia abajo al entero más
cercano.
math.floor(8789.98767)
```

```
8789
```

Strings (character strings)

Las cadenas de caracteres, también conocidas como "strings", son una secuencia de caracteres, que pueden incluir letras, números, símbolos, y hasta emojis. Las cadenas pueden estar encerradas en comillas dobles o simples, y podemos incluso definir cadenas de varias líneas utilizando comillas triples. A continuación, vamos a explorar varios ejemplos y características de las cadenas en Python.

Ejemplos de cómo definir una cadena con diferentes tipos de comillas:

```
"Esto es una string"
```

```
'Esto es una string'
```

```
'Esto es una string con comillas simples'
```

```
'Esto es una string con comillas simples'
```

```
"Esto es una string con comillas simples" #End Of Line
```

```
'Esto es una string con comillas simples'
```

Podemos asignar un `string` a una variable, como se muestra aquí:

```
esto_es_string_tambien = "4"
esto_es_string_tambien
```

```
'4'
```

Podemos verificar el tipo de una variable utilizando la función `type`:

```
print(type(esto_es_string_tambien))
```

```
<class 'str'>
```

Intentando crear un `string` de varias líneas sin comillas triples resultará en un error:

```
"Esto es una string
con varias lineas"
```

Para crear un `string` de varias líneas correctamente, debemos usar comillas triples:

```
"""
Esto es una string
con varias lineas"""
```

```
'\nEsto es una string\ncon varias lineas'
```

También podemos imprimir un `string` de varias líneas usando la función print y comillas triples:

```
print("""
Esto es una string
con varias lineas""")


Esto es una string
con varias lineas
```

icons Laos strings pueden contener emojis, como se muestra aquí:

```
"😍" #emojis -> son strings

cara_corazones = "😍"

cara_corazones

print(type(cara_corazones))

<class 'str'>
```

Casting in Python

La conversión de tipos, también conocida como "casting", se refiere al proceso de convertir un tipo de dato a otro. Previamente hemos visto tipos de datos como `int`, `string` o `float`. Bueno, resulta que es posible convertir de un tipo a otro. Pero antes que nada, veamos los diferentes tipos de conversiones o transformaciones de tipo que se pueden realizar. Existen dos:

Conversión implícita: Esto es realizado automáticamente por Python. Ocurre cuando realizamos ciertas operaciones con dos tipos diferentes, Python hace la conversión en el fondo sin necesidad de que el programador lo indique explícitamente.

Conversión explícita: Nosotros realizamos esto de manera explícita, como convertir un `str` a `int` con `int()` o a `float` con `float()`. Este tipo de conversión es realizado mediante el uso de funciones predefinidas de Python.

Es importante tener en cuenta que no todos los tipos de datos pueden convertirse entre sí de forma segura. Por ejemplo, intentar convertir una cadena de texto que contiene letras a un entero producirá un error. Por lo tanto, siempre es una buena práctica manejar posibles errores usando estructuras de control de excepciones, lo que veremos más adelante en el curso.

¡Vamos a ver algunos ejemplos de cada uno para entender mejor cómo funcionan!

Implicit conversion

Este tipo de conversión es realizado automáticamente por Python, prácticamente sin que nos demos cuenta. Aún así, es importante conocer lo que está sucediendo debajo del capó para evitar problemas futuros.

El ejemplo más sencillo donde podemos ver este comportamiento es el siguiente:

- `a` es un `int`
- `b` es un `float`

Pero si sumamos `a` y `b` y almacenamos el resultado en `a`, podemos ver cómo internamente Python ha convertido el `int` a `float` para realizar la operación, y la variable resultante es `float`. Sin embargo, hay otros casos donde Python no es tan inteligente y no es capaz de realizar la conversión. Si intentamos añadir un `int` a una `string`, obtendremos un `TypeError`.

```
a = 5       # Esto es un int
b = 4.5     # Esto es un float

c = a + b   # Python convierte automáticamente a a float para realizar la operación
print(c)    # El resultado, 9.5, es un float
```

```
9.5
```

```
# Pero, si intentamos hacer una operación entre un string y un entero:
d = "Hola" + a  # Esto provocará un TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[43], line 2
      1 # Pero, si intentamos hacer una operación entre un string y un entero:
----> 2 d = "Hola" + a  # Esto provocará un TypeError

TypeError: can only concatenate str (not "int") to str
```

explicit conversion

Por otro lado, podemos realizar conversiones entre tipos o castings de manera explícita utilizando diferentes funciones que Python proporciona. Las más utilizadas son las siguientes:

`float()`, `str()`, `int()`, `list()`, `set()`

Convert float to int

Para convertir de float a int debemos usar `int()`. Pero cuidado, porque el tipo entero no puede almacenar decimales, así que perderemos lo que esté después del punto decimal.

```
# Ejemplo de conversión explícita
numero_float = 5.7
numero_int = int(numero_float)

print(numero_int)  # El resultado será 5, perdiendo la parte decimal.
```

```
5
```

Convert a float to a string

Podemos convertir un float a un string de texto utilizando `str()`. Podemos ver en el siguiente código cómo cambia el tipo de `a` después de la conversión.

```
# Ejemplo de convertir un float a un string
a = 3.14159
print(type(a))  # Esto imprimirá: <class 'float'>

<class 'float'>

a = str(a)
print(type(a))  # Esto imprimirá: <class 'str'>
print(a)        # Esto imprimirá: 3.14159, pero ahora como una cadena
de texto.

<class 'str'>
3.14159
```

Convert int to str

Al igual que la conversión a float que vimos anteriormente, podemos convertir de int a str utilizando `str()`. Veamos un ejemplo a continuación:

```
# Ejemplo de convertir un entero a una cadena de texto
a = 42
print(type(a))  # Esto imprimirá: <class 'int'>

<class 'int'>

a = str(a)
print(type(a))  # Esto imprimirá: <class 'str'>
print(a)        # Esto imprimirá: "42", pero ahora como una cadena de
texto.

<class 'str'>
42
```

Ejercicio

Ahora es tu turno de probar la conversión de int a str. Realiza las siguientes tareas:

1. Crea una variable `edad` y asígnale tu edad como un entero.
2. Convierte la variable `edad` a un string utilizando la función `str()`.
3. Concatena la cadena "Mi edad es: " con la variable `edad` (ahora una cadena) y almacena el resultado en una nueva variable llamada `mensaje`.
4. Imprime la variable `mensaje` en la consola.

A continuación, te dejo un esquema básico que puedes utilizar:

```
# Paso 1: Crea una variable edad con tu edad como un entero
edad = 25  # Cambia este valor por tu edad actual

# Paso 2: Convierte la variable edad a un string
edad_str = str(edad)

# Paso 3: Concatena "Mi edad es: " con la variable edad y almacena el resultado en una nueva variable llamada mensaje
mensaje = "Mi edad es: " + edad_str

# Paso 4: Imprime la variable mensaje
print(mensaje)
```

```
Mi edad es: 25
```

Input and output data

En el desarrollo de programas, a menudo necesitamos interactuar con los usuarios permitiéndoles introducir datos (input) y mostrándoles resultados o mensajes (output). Python proporciona métodos sencillos y directos para lograr esto, facilitando la creación de scripts interactivos y amigables. A continuación, exploraremos cómo podemos lograr esto utilizando Python.

Input

Para asignar una variable a un valor ingresado por el usuario desde la consola, utilizamos la función `input()`. Esta función puede llevar un argumento opcional: el mensaje o la indicación que se desea mostrar en pantalla para guiar al usuario sobre qué tipo de información se espera que ingrese. Es fundamental tener en cuenta que, independientemente del tipo de datos que el usuario ingrese, la función `input()` siempre devolverá un string. Aquí está el esquema básico de cómo funciona:

`input(message)` : Muestra el mensaje en la terminal y devuelve una cadena con la entrada del usuario.

A menudo, puede ser necesario convertir esta cadena a un tipo de datos diferente, dependiendo de cómo planeas usar el valor ingresado en tu script. Esto lo podríamos hacer utilizando técnicas de conversión de tipos, o "casting", que discutimos en secciones anteriores.

```
saludo = input()
saludo
```

```
 Hola
```

```
'Hola'
```

Ahora, vamos a personalizar el mensaje que aparece cuando pedimos una entrada al usuario utilizando el argumento `prompt` de la función `input()`.

```
saludo = input(prompt = "Esto es el prompt")
```

```
Esto es el prompt Hola
```

Continuemos solicitando más información al usuario, como su nombre y un número. Luego, experimentaremos con las operaciones que podemos realizar con estas entradas.

```
nombre = input("Escribe aquí tu nombre: ")
nombre
```

```
Escribe aquí tu nombre:  sdfkñlfss
```

```
'sdfkñlfss'
```

```
number = input("Escribe aquí tu número: ")
number
```

```
Escribe aquí tu número:  zxfdsfsdfs
```

```
'zxfdsfsdfs'
```

Al intentar operar con un string que representa un número, veremos que no se comporta como un número real. Por ejemplo, si intentamos multiplicarlo por 10 o dividirlo por 2, obtendremos errores o comportamientos no deseados.

```
number * 10
```

```
'zxfdsfsdfszxfdsfsdfszxfdsfsdfszxfdsfsdfszxfdsfsdfszxfdsfsdfszxfdsfsdf
szxfdsfsdfszxfdsfsdfszxfdsfsdfs'
```

```
# Esto generará un error porque estamos intentando dividir un string
number / 2
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
Cell In[56], line 2
      1 # Esto generará un error porque estamos intentando dividir un
string
----> 2 number / 2

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Para evitar estos problemas, podemos convertir la entrada a un número utilizando la función `int()`. Una vez que la entrada es un entero, podemos realizar operaciones numéricas con ella.

```
numero = int(input("Escribe aquí tu número: "))
numero
```

```
Escribe aquí tu número:  34
```

```
34
```

```
type(numero)
```

```
int
```

```
int(numero / 2)
```

```
17
```

Reflexiones Finales

Es importante anticipar posibles problemas en las entradas de los usuarios y gestionarlos adecuadamente para evitar errores durante la ejecución. Una forma de hacerlo es a través de la programación defensiva, donde se verifica la validez de las entradas antes de proceder con las operaciones. Además, podemos utilizar estructuras de control de flujo, como `if/else`, y gestión de errores con `try/except` para manejar situaciones inesperadas de manera más elegante.

A Considerar:

- Anticipar un posible problema -> Programación defensiva.
- Asegurarse de que el número es un entero antes de realizar operaciones que requieren enteros.
- Utilizar lógica condicional (`if/else`) para gestionar diferentes casos.
- Manejar errores de forma proactiva utilizando `try/except` para prevenir fallos durante la ejecución.

Print

La función `print()` se utiliza para escribir el mensaje especificado en la pantalla o en otro dispositivo de salida estándar.

El mensaje puede ser una cadena (string) o cualquier otro objeto; el objeto se convertirá en un string antes de ser escrito en la pantalla. A continuación, vamos a explorar algunas de las funcionalidades y particularidades de la función `print()`.

Observaciones:

- **NUESTRO AMIGO**: La función `print()` es una herramienta fundamental para la depuración de código. Nos permite visualizar los valores de diferentes variables en varios puntos de nuestro código, lo que facilita la identificación de errores o "bugs".
- **Depuración**: Es el proceso de identificar y corregir errores en el código. Utilizar `print()` es una de las formas más simples de "debuggear", es decir, buscar y corregir errores en el código.

Vamos a ver cómo funciona en el siguiente fragmento de código:

```
greeting = "Hellooooo"
type(greeting)
```

```
str
```

```
type(print(greeting))
```

```
Hellooooo
```

```
NoneType
```

```
type(greeting)
```

```
str
```

Reflexiones

Notarás que cuando utilizamos `print()`, el tipo que devuelve es `NoneType`. Esto es porque `print()` es una función que realiza una acción (imprimir algo en la consola) pero no devuelve ningún valor (su retorno es `None`). Esta es una distinción importante, especialmente cuando se compara con funciones que sí devuelven valores.

Advertencia

- Es fundamental recordar que `print()` imprime en la consola pero no retorna un valor que pueda ser utilizado en operaciones subsecuentes en el código.

Format

En esta sección, exploraremos diferentes formas de formatear strings en Python, lo que puede ser especialmente útil cuando queremos incluir valores variables dentro de un string. Existen varias formas de hacer esto en Python, incluyendo la concatenación de strings usando `+`, usando una coma `,`, utilizando el método `.format()` o a través de las f-strings. A continuación, veremos ejemplos de cada uno de estos métodos y analizaremos los tipos de datos resultantes.

Format - 1

```
# Primer caso: esto podría ser un caso de uso para las funciones input y output
name = "Santi"
age = 24

# Usando concatenación con '+'
greeting = "Hello my name is " + name + " and my age is " + str(age)
greeting
```

```
'Hello my name is Santi and my age is 24'
```

```
# Usando coma para concatenar
greeting = "Hello my name is ", name, " and my age is ", str(age)
greeting
```

```
('Hello my name is ', 'Santi', ' and my age is ', '24')
```

```
# Verificando el tipo de la variable greeting
type(greeting)
```

```
tuple
```

```
# Añadiendo algunas opciones para reflexionar sobre qué tipo de datos tenemos aquí:
# 1. String
# 2. List
# 3. Tuple
# 4. CSV: comma separated values?

# Utilizando f-strings para una formateación más limpia
greeting = f"Hello my name is {name} and my age is {age}"
greeting
```

```
'Hello my name is Santi and my age is 24'
```

Format - 2

```
# Segundo caso:
name = "Laura"
age = 30

# Usando el método .format() para insertar valores en un string
greeting = "Hello my name is {} and my age is {}".format(name, age)
greeting
```

```
'Hello my name is Santi and my age is 24'
```

Ejercicio

Ahora que hemos aprendido varias formas de formatear strings en Python, es momento de poner en práctica lo aprendido. En este ejercicio, te pediremos que utilices la función `input()` para solicitar cierta información al usuario y luego formatea esa información utilizando al menos dos de los métodos que hemos discutido anteriormente (concatenación con '+', usando comas, f-strings o el método `.format()`).

Instrucciones:

1. Pide al usuario que ingrese su nombre y su edad utilizando la función `input()`.
2. Crea un mensaje de saludo que incluya el nombre y la edad del usuario, utilizando dos métodos diferentes de formateo de strings.
3. Imprime ambos mensajes en la consola para verificar tu trabajo.

```
# tu codigo aquí

# Paso 1: Pide al usuario que ingrese su nombre y su edad
nombre = input("Ingresa tu nombre: ")
```

```
edad = input("Ingresa tu edad: ")

# Paso 2: Crea un mensaje de saludo utilizando dos métodos diferentes
de formateo de strings

# Método 1: Usando concatenación con "+"
mensaje_concatenacion = "Hola, " + nombre + ". Tienes " + edad + "
años."

# Método 2: Usando f-strings
mensaje_fstring = f"Hola, {nombre}. Tienes {edad} años."

# Paso 3: Imprime ambos mensajes
print(mensaje_concatenacion)
print("-----------------------")
print(mensaje_fstring)
```

Strings

En Python, como ya hemos visto antes, un string, es una secuencia de caracteres encerrada entre comillas simples (`'`), dobles (`"`) o triples (`'''` o `"""`). Los strings son inmutables, lo que significa que una vez creadas, no podemos modificar su contenido directamente, aunque sí podemos crear nuevas cadenas a partir de manipulaciones de la original mediante varios métodos y operaciones. Estas pueden contener letras, números, caracteres especiales, espacios o una combinación de todos ellos.

String methods

Los métodos son acciones o funciones que un objeto puede realizar. Al igual que Python nos ofrece una serie de funciones "integradas", también nos brinda una serie de métodos ya creados. Estos métodos dependen del tipo de objeto con el que estemos trabajando, y en el caso de las cadenas, nos permiten realizar una variedad de operaciones para manipular e inspeccionarlas.

En esta sección, vamos a explorar algunos de los métodos de cadena más utilizados durante el bootcamp, utilizando un`sample_string` para demostrar su funcionalidad. A medida que avanzamos en el bootcamp, es esencial familiarizarse con estos métodos, ya que pueden facilitar considerablemente tu proceso de codigo.

Para obtener una visión más completa de los métodos para string, no dudes en consultar la documentación de Python. Yo siempre uso Google :)

```
sample_string = "this is a string"
```

- `capitalize` Devuelve una copia de la cadena con su primer carácter en mayúsculas y el resto en minúsculas. Es ideal para cuando queremos que una oración o título comience con un toque más formal. ¡Probémoslo con el ejemplo!

```
sample_string.capitalize()

'This is a string'
```

- `upper` Nos devuelve una copia de la cadena pero con todos los caracteres en mayúsculas. Es perfecto para destacar algo con fuerza o simplemente para igualar el formato de diferentes textos. ¡Vamos a ponerlo a prueba con una cadena que tenemos por aquí!

```
sample_string.upper()
```

```
'THIS IS A STRING'
```

```
# También podemos comprobar si el string está en formato upper (letras mayúsculas)
sample_string.upper().isupper()
```

```
True
```

- `lower` Te devuelve una copia de la string pero con todos los caracteres en minúsculas. Este método es genial para cuando queremos mantener la uniformidad en nuestro texto o simplemente para evitar "GRITAR" en una conversación digital. ¡Veamos cómo funciona con un ejemplo práctico!

```
sample_string.lower()
```

```
'this is a string'
```

```
sample_string.lower().islower()
```

```
True
```

- `swapcase` Este método es como el intercambio de ropas en una fiesta de disfraces: convierte todos los caracteres en mayúsculas a minúsculas y viceversa en la string. Es especialmente útil si queremos invertir la capitalización de un string de texto rápidamente. ¡Probémoslo con un ejemplo!

```
sample_string.swapcase()
```

```
'THIS IS A STRING'
```

- `title` Este método devuelve una versión de la string donde cada palabra comienza con un carácter en mayúscula, y el resto de los caracteres están en minúscula. Es como si convirtiera la string en un título de un libro o una película, otorgándole un aspecto más formal y cuidado. ¡Vamos a ver cómo funciona con un ejemplo práctico!

```
sample_string.title()
```

```
'This Is A String'
```

- `join(iterable)` Este método devuelve una string que es la concatenación de las strings en el iterable. Cabe señalar que se generará un TypeError si hay valores no string en el iterable, incluidos los objetos bytes. El separador entre los elementos es la string proporcionada por este método. Es una forma efectiva de unir múltiples strings en una sola, utilizando un separador específico que define la propia string. ¡Observemos cómo lo podemos utilizar con algunos ejemplos!

```
# Nuevo ejemplo
list_of_strings = ["Santi", "Clara", "Laura", "Albert"]
" □ ".join(list_of_strings)
```

```
'Santi □ Clara □ Laura □ Albert'
```

- `startswith` Este método devuelve `True` si la string comienza con el prefijo especificado; de lo contrario, devuelve `False`. Es interesante mencionar que el prefijo también puede ser una tupla de prefijos para buscar. Además, cuenta con dos parámetros opcionales: `start`, que permite especificar desde qué posición de la string empezar a comprobar, y `end`, que indica dónde detener la comprobación. Veamos algunos ejemplos para entender mejor su funcionamiento:

```
number = "3434567"

number.startswith("+")
```

```
False
```

```
number.startswith("34")
```

```
True
```

- `endswith` Este método funciona de manera similar al método `startswith`, pero en este caso verifica si la string termina con el sufijo especificado. Si es así, devuelve `True`; de lo contrario, devuelve `False`. Al igual que `startswith`, este método permite especificar los parámetros opcionales `start` y `end` para definir el rango de la string donde realizar la comprobación. A continuación, te presentamos algunos ejemplos para ilustrar cómo funciona:

```
number.endswith("67")
```

```
True
```

- `str.lstrip([chars])` Este método retorna una copia de la string original pero sin los caracteres especificados en el argumento `chars` que se encuentren al inicio de la misma. Si no se especifica ningún argumento (o sea, se omite o no hay ninguno presente), se eliminarán los espacios en blanco por defecto. Es importante tener en cuenta que el argumento `chars` no actúa como un prefijo; en lugar de ello, elimina todas las combinaciones posibles de los valores que se encuentren dentro de él. Aquí te mostramos algunos ejemplos para entender mejor cómo funciona este método:

```
# Definimos una string con algunos espacios y caracteres al inicio
cadena_original = "   ##Este es un ejemplo. ##"

# Usamos el método lstrip para remover los espacios en blanco al
inicio
cadena_modificada = cadena_original.lstrip()
print(cadena_modificada)
# Salida: "##Este es un ejemplo."
```

```
##Este es un ejemplo. ##

# También podemos utilizar lstrip para remover otros caracteres especificando un argumento
cadena_modificada2 = cadena_original.lstrip(" #")
print(cadena_modificada2)
# Salida: "Este es un ejemplo."
```

```
Este es un ejemplo.
```

- `lstrip` El método `lstrip` en Python es utilizado para eliminar caracteres no deseados que se encuentran al principio de una string. Puedes usarlo de dos maneras: sin argumentos, lo que eliminará todos los espacios en blanco al inicio de la string; o con un conjunto de caracteres específicos como argumento (indicados entre paréntesis), lo que eliminará todas las instancias de esos caracteres que encuentre al principio de la string.
- `rstrip` De forma análoga, el método `rstrip` se utiliza para eliminar caracteres al final de una string. Funciona de la misma manera que `lstrip`, pero afecta al final de la string en lugar del principio. Si no se especifica ningún conjunto de caracteres, eliminará todos los espacios en blanco que se encuentren al final de la string.
- `replace` El método `replace` es utilizado para reemplazar todas las ocurrencias de una subcadena específica (`old`) por una nueva subcadena (`new`). Puedes usarlo de dos maneras: sin el argumento opcional `count`, lo que reemplazará todas las ocurrencias encontradas en la string; o especificando el argumento `count`, lo que limitará el número de reemplazos a la cantidad indicada. Este método devuelve una copia de la string original con las sustituciones realizadas, dejando la string original intacta.

```
cadena_original.replace("#", "")
```

```
'   Este es un ejemplo. '
```

- `split` El método `split` se utiliza para dividir una string en una lista de palabras basándose en un delimitador especificado (el parámetro `sep`). Si no se especifica ningún delimitador (o se establece como `None`), se utilizarán espacios en blanco (espacios, tabulaciones, nuevas líneas, etc.) como delimitadores predeterminados. Este método es especialmente útil cuando deseas descomponer una string en componentes más pequeños para realizar operaciones adicionales o análisis en cada fragmento individual.

```
sentence = "Hello my name is Santo"
sentence.split(" ")
```

```
['Hello', 'my', 'name', 'is', 'Santo']
```

```
sentence.split("e")
```

```
['H', 'llo my nam', ' is Santo']
```

Data structures

En Python, disponemos de cuatro estructuras principales para almacenar colecciones de datos, cada una con sus propias características y utilidades. Estas estructuras nos facilitan la organización y manipulación de los datos de una manera más eficiente. A continuación, te presentamos las cuatro estructuras de datos fundamentales en Python:

- **Listas (Lists)**: Colecciones ordenadas y modificables que pueden almacenar una variedad de tipos de datos, incluyendo otras listas. **usamos -> []**
- **Tuplas (Tuples)**: Colecciones ordenadas e inmutables, similares a las listas, pero que no pueden ser modificadas una vez creadas. **usamos -> ()**
- **Conjuntos (Sets)**: Colecciones desordenadas y sin índices, que no permiten elementos duplicados, lo que las hace ideales para almacenar conjuntos únicos de datos. **usamos -> {}**
- **Diccionarios (Dictionaries)**: Colecciones desordenadas, modificables e indexadas, donde los datos se almacenan en pares clave-valor, facilitando la organización y recuperación de información compleja. **usamos -> {key:values, key2:value2}**

A lo largo de esta sección, exploraremos cada una de estas estructuras de datos en detalle, descubriendo cómo pueden ayudarnos a trabajar con datos de manera más efectiva en Python.

Lists

Las listas en Python son estructuras de datos que pueden contener diferentes tipos de datos, desde números hasta cadenas de texto, e incluso otras listas. Esto las convierte en herramientas extremadamente versátiles y útiles en la programación. Los elementos en una lista están ordenados y tienen un índice específico, lo que nos permite acceder, modificar, añadir o eliminar elementos de manera sencilla. A continuación, presentamos varios ejemplos de listas junto con la sintaxis para acceder a los datos dentro de ellas.

```
# 1. Lista de números enteros
numeros = [1, 2, 3, 4, 5]

# 2. Lista de cadenas de texto (strings)
frutas = ["manzana", "banana", "cereza"]

# 3. Lista mixta (conteniendo diferentes tipos de datos)
mixta = [1, "Hola", 3.14, True]

# 4. Lista anidada (una lista dentro de otra lista)
anidada = [[1, 2, 3], ["a", "b", "c"]]

# Acceder a los datos dentro de una lista
# Accediendo al primer elemento de la lista de números
numeros[0]

1

# Accediendo al último elemento de la lista de frutas
frutas[2]
```

```
'cereza'
```

```
# Accediendo a un elemento de una lista dentro de otra lista (lista anidada)
anidada[1][1]
```

```
'b'
```

```
# Para saber el tamaño de una lista
len(anidada)
```

```
2
```

list methods

A continuación, describiremos algunos de los métodos más comunes que puedes utilizar para manipular listas. Además, te invitamos a consultar la documentación oficial para una guía completa de todos los métodos disponibles.

Existen varios métodos que facilitan la gestión de las listas, aquí te presentamos algunos de los más utilizados:

- `append()`: Añade un elemento al final de la lista.

```
# 1. append()
lista = []
lista.append('A')
print(lista)
```

```
['A']
```

- `extend()`: Extiende la lista agregando todos los elementos de la lista dada.

```
# 2. extend()
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
lista1.extend(lista2)
print(lista1)
```

```
[1, 2, 3, 4, 5, 6]
```

- `insert()`: Inserta un elemento en la lista en el índice especificado.

```
# 3. insert()
lista = [1, 2, 3]
lista.insert(20, 'B')
print(lista)
```

```
[1, 2, 3, 'B']
```

- `remove()`: Elimina el primer elemento de la lista cuyo valor sea igual al valor especificado.

```
# 4. remove()
lista = [1, 2, 3, 2]
lista.remove(2)
print(lista)
```

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

- `pop()`: Elimina el elemento en la posición dada de la lista, y lo devuelve.

```
# 5. pop()
lista = [1, 2, 3]
lista.pop(1)
print(lista)
```

```
[1, 3]
```

- `clear()`: Elimina todos los elementos de la lista.

```
# 6. clear()
lista = [1, 2, 3]
lista.clear()
print(lista)
```

```
[]
```

- `index()`: Devuelve el índice del primer elemento con el valor especificado.

```
# 7. index()
lista = [1, 2, 3, 2]
print(lista.index(3))
```

```
2
```

- `count()`: Devuelve el número de veces que el valor especificado aparece en la lista.

```
# 8. count()
lista = [1, 2, 3, 2, 2]
print(lista.count(2))
```

```
3
```

- `sort()`: Ordena los elementos de la lista.

```
# 9. sort()
lista = [3, 1, 2]
lista.sort()
print(lista)
```

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

- `reverse()`: Invierte el orden de los elementos de la lista.

```python
# 10. reverse()
lista = [1, 2, 3]
lista.reverse()
print(lista)
```

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

- `copy()`: Devuelve una copia de la lista.

```python
# 11. copy()
lista1 = [1, 2, 3]
lista2 = lista1.copy()
print(lista2)
```

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

Slicing y Start, Stop, Step en Listas

El "slicing", no es un método como tal, pero nos permite jugar con los elementos de las listas y sus posiciones. Esencialmente, nos permite seleccionar una "rebanada" de la lista utilizando tres parámetros: inicio (start), fin (stop) y paso (step). La sintaxis para esto es `lista[start:stop:step]`, donde:

- **start**: representa el índice del primer elemento que queremos incluir en nuestra selección. Es importante recordar que los índices en Python comienzan en 0.
- **stop**: representa el índice del primer elemento que NO queremos incluir en nuestra selección. La selección incluirá elementos hasta el índice `stop`-1.
- **step**: define el incremento entre los índices seleccionados. Si se omite, el valor predeterminado será 1, lo que significa que se seleccionarán todos los elementos desde `start` hasta `stop`-1.

A continuación, veremos ejemplos prácticos de cómo utilizar estos parámetros para seleccionar diferentes segmentos de una lista en Python.

```python
nombres = ["Ana", "Beto", "Carla", "David", "Elena", "Fernando"]

# Seleccionar elementos desde el índice 2 hasta el final
print(nombres[2:])
```

```
['Carla', 'David', 'Elena', 'Fernando']
```

```python
# Seleccionar elementos desde el índice 4 hasta el final
print(nombres[4:])
```

```
['Elena', 'Fernando']
```

```python
# Invertir el orden de los elementos en la lista
print(nombres[::-1])
```

```
['Fernando', 'Elena', 'David', 'Carla', 'Beto', 'Ana']
```

```
lista_numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Seleccionar elementos desde el índice 0 hasta el 10, saltando 2 elementos cada vez
print(lista_numeros[0:10:2])
```

```
[0, 2, 4, 6, 8]
```

Ejercicio: Trabajando con Listas

1. Crea una lista llamada `meses` que contenga los nombres de todos los meses del año.
2. Utiliza el método `append` para añadir un elemento extra en la lista que sea "Fin de año".
3. Utiliza el método `remove` para eliminar este último elemento que has añadido.
4. Usando `slicing`, crea una nueva lista que contenga solo los meses del segundo trimestre (abril, mayo y junio).
5. Usa el método `reverse` para invertir el orden de los elementos en la lista original de meses.
6. Encuentra el método apropiado para ordenar la lista de meses en orden alfabético y aplícalo.
7. Utiliza el método `index` para encontrar la posición de tu mes de nacimiento en la lista ordenada alfabéticamente.

Extra:

- Crea una lista de listas, donde cada sublista contenga los meses de cada trimestre.
- Utiliza un bucle `for` para imprimir cada mes de cada trimestre, formateando la salida de la siguiente manera: "El {número de mes}° mes del año es {nombre del mes}".

Recuerda revisar la documentación de Python o usar la función `help()` para obtener detalles sobre cómo usar cada uno de los métodos de la lista.

```
# Aquí tu codigo

# Paso 1: Crea una lista llamada 'meses' que contenga los nombres de todos los meses del año
meses = ["Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
         "Julio", "Agosto", "Septiembre", "Octubre", "Noviembre",
"Diciembre"]

# Paso 2: Utiliza el método 'append' para añadir un elemento extra en la lista que sea "Fin de año"
meses.append("Fin de año")
print(f"Lista después de añadir 'Fin de año': {meses}")
```

```
Lista después de añadir 'Fin de año': ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio', 'Agosto', 'Septiembre', 'Octubre', 'Noviembre', 'Diciembre', 'Fin de año']
```

```python
# Paso 3: Utiliza el método 'remove' para eliminar este último elemento que has añadido
meses.remove("Fin de año")
print(f"Lista después de eliminar 'Fin de año': {meses}")

# Paso 4: Usando 'slicing', crea una nueva lista que contenga solo los meses del segundo trimestre (abril, mayo y junio)
segundo_trimestre = meses[3:6]
print(f"Segundo trimestre: {segundo_trimestre}")
```

```
Lista después de eliminar 'Fin de año': ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio', 'Agosto', 'Septiembre', 'Octubre', 'Noviembre', 'Diciembre']
Segundo trimestre: ['Abril', 'Mayo', 'Junio']
```

```python
# Paso 5: Usa el método 'reverse' para invertir el orden de los elementos en la lista original de meses
meses.reverse()
print(f"Lista de meses en orden inverso: {meses}")

# Paso 6: Encuentra el método apropiado para ordenar la lista de meses en orden alfabético y aplícalo
meses.sort()
print(f"Lista de meses en orden alfabético: {meses}")
```

```
Lista de meses en orden inverso: ['Diciembre', 'Noviembre', 'Octubre', 'Septiembre', 'Agosto', 'Julio', 'Junio', 'Mayo', 'Abril', 'Marzo', 'Febrero', 'Enero']
Lista de meses en orden alfabético: ['Abril', 'Agosto', 'Diciembre', 'Enero', 'Febrero', 'Julio', 'Junio', 'Marzo', 'Mayo', 'Noviembre', 'Octubre', 'Septiembre']
```

```python
# Paso 7: Utiliza el método 'index' para encontrar la posición de tu mes de nacimiento en la lista ordenada alfabéticamente
# Cambia 'Enero' por el mes de tu nacimiento
mes_nacimiento = "Enero"  # Ajusta según el mes de tu nacimiento
posicion_mes = meses.index(mes_nacimiento) + 1  # Sumamos 1 para indicar posición desde 1
print(f"El mes de {mes_nacimiento} está en la posición {posicion_mes} en la lista alfabética.")
```

```
El mes de Enero está en la posición 4 en la lista alfabética.
```

```python
# Extra: Crea una lista de listas, donde cada sublista contenga los meses de cada trimestre
trimestres = [
    ["Enero", "Febrero", "Marzo"],  # Primer trimestre
    ["Abril", "Mayo", "Junio"],     # Segundo trimestre
    ["Julio", "Agosto", "Septiembre"],  # Tercer trimestre
    ["Octubre", "Noviembre", "Diciembre"]  # Cuarto trimestre
]
```

```python
# Utiliza un bucle 'for' para imprimir cada mes de cada trimestre
for i, trimestre in enumerate(trimestres, start=1):
    for j, mes in enumerate(trimestre, start=1):
        numero_mes = (i - 1) * 3 + j
        print(f"El {numero_mes}° mes del año es {mes}.")
```

```
El 1° mes del año es Enero.
El 2° mes del año es Febrero.
El 3° mes del año es Marzo.
El 4° mes del año es Abril.
El 5° mes del año es Mayo.
El 6° mes del año es Junio.
El 7° mes del año es Julio.
El 8° mes del año es Agosto.
El 9° mes del año es Septiembre.
El 10° mes del año es Octubre.
El 11° mes del año es Noviembre.
El 12° mes del año es Diciembre.
```

Tuples

Las tuplas son una estructura de datos muy similar a las listas, con la principal diferencia de que son inmutables. Lo que significa que no puedes cambiar los elementos de una tupla una vez que ha sido creada. A pesar de esta característica, las tuplas son bastante flexibles y pueden almacenar diferentes tipos de datos, incluyendo otros contenedores como listas o diccionarios. Al igual que las listas, las tuplas permiten la indexación y el desempaquetado, facilitando así el acceso y la manipulación de los datos contenidos en ellas.

Las tuplas se definen utilizando paréntesis `()` y los elementos se separan por comas. Vamos a explorar algunos ejemplos y métodos asociados con las tuplas.

```python
# Crear una tupla
mi_tupla = (1, 2, 3, "Hola", True,2)

# Acceder a elementos de una tupla
mi_tupla[0]
```

```
1
```

```python
# Acceder al ultimo elemento
mi_tupla[-1]
```

```
True
```

```python
# Desempaquetar una tupla
a, b, c, d, e = mi_tupla
c
```

```
3
```

```
# Métodos disponibles en una tupla
# Contar la cantidad de veces que aparece un elemento
mi_tupla.count(2)
```

```
2
```

```
# Encontrar el índice de un elemento
mi_tupla.index("Hola")
```

```
3
```

```
mi_tupla
```

```
(1, 2, 3, 'Hola', True, 2)
```

```
mi_tupla[1] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[150], line 1
----> 1 mi_tupla[1] = 10

TypeError: 'tuple' object does not support item assignment
```

```
# Intentando modificar un elemento de la tupla (esto generará un error, porque las tuplas son inmutables)
try:
    mi_tupla[1] = 10
except TypeError as e:
    print(f"Error es una tupla: {e}")

# Mostrando que la tupla no ha cambiado
print(mi_tupla)
```

```
Error es una tupla: 'tuple' object does not support item assignment
(1, 2, 3, 'Hola', True, 2)
```

Métodos de las Tuplas: Las tuplas, a diferencia de las listas, son inmutables, lo que significa que no podemos agregar, modificar o eliminar elementos una vez que la tupla ha sido definida. Sin embargo, las tuplas cuentan con varios métodos que pueden resultar muy útiles. A continuación, te presento algunos de ellos:

- `tuple.index(x)`: Este método devuelve el índice del primer elemento igual a x.

```
# Creación de una tupla
mi_tupla = (1, 2, 3, 4, 3, 2, 1, 2)

# Uso del método index
indice = mi_tupla.index(4)
print(f"El índice del primer elemento igual a 3 es: {indice}")
```

```
El índice del primer elemento igual a 3 es: 3
```

- `tuple.count(x)`: Este método cuenta el número de veces que x aparece en la tupla.

```
# Uso del método count
conteo = mi_tupla.count(2)
print(f"El número 2 aparece {conteo} veces en la tupla")
```

- `tuple.__len__()`: Este método devuelve la longitud de la tupla.

```
# Uso del método len para obtener la longitud
longitud = mi_tupla.__len__()
print(f"La longitud de la tupla es: {longitud}")

La longitud de la tupla es: 8
```

- `tuple.__contains__(x)`: Este método verifica si un elemento x está presente en la tupla.

```
# Verificar si un elemento está en la tupla
if mi_tupla.__contains__(22):
    print("El número está en la tupla.")
else:
    print("El número no está en la tupla.")

El número no está en la tupla.
```

- `tuple.__getitem__(i)`: Este método permite acceder a un elemento de la tupla mediante su índice i.

```
# Acceder a un elemento por índice
elemento = mi_tupla.__getitem__(3)
print(f"El elemento en el índice 3 es: {elemento}")

El elemento en el índice 3 es: 4
```

- `tuple.__reversed__()`: Este método devuelve una versión invertida de la tupla.

```
# Obtener una versión invertida de la tupla
tupla_invertida = tuple(reversed(mi_tupla))
print(f"Tupla invertida: {tupla_invertida}")

Tupla invertida: (2, 1, 2, 3, 4, 3, 2, 1)
```

Puedes aprender más sobre los métodos de tuplas en la documentación oficial.

sets

En Python, un conjunto (set) es una colección desordenada de elementos únicos. A diferencia de las listas y las tuplas, los conjuntos no permiten elementos duplicados. Los conjuntos son útiles

cuando necesitas almacenar elementos en los que el orden no es importante y quieres asegurarte de que no haya duplicados.

Los conjuntos se definen utilizando llaves `{}` o la función `set()`, y los elementos se separan por comas. A lo largo de esta sección, exploraremos cómo trabajar con conjuntos en Python y algunos de los métodos disponibles para ellos.

```
# Creación de un conjunto
mi_conjunto = {1, 2, 3, 4, 5,2,4,5}

# Mostrar el conjunto
print(mi_conjunto)

# Los conjuntos no permiten elementos duplicados
mi_conjunto = {1, 2, 2, 3, 3, 4, 5}
print(mi_conjunto)  # Salida: {1, 2, 3, 4, 5}
```

El constructor `set()` en Python se utiliza para crear un conjunto vacío o para convertir otros objetos iterables (como listas o tuplas) en conjuntos. Aquí tienes algunos ejemplos de cómo funciona:

```
# También puedes crear un conjunto vacío con set()
conjunto_vacio = set()
print(conjunto_vacio)  # Salida: set()

# Convertir una lista en un conjunto:
mi_lista = [1, 2, 2, 3, 4, 4]
mi_conjunto = set(mi_lista)
print(mi_conjunto)  # Salida: {1, 2, 3, 4}

# Convertir una tupla en un conjunto:
mi_tupla = (1, 2, 3, 3, 4, 5)
mi_conjunto = set(mi_tupla)
print(mi_conjunto)  # Salida: {1, 2, 3, 4, 5}
```

Operaciones de Conjunto en Python: Los conjuntos (sets) en Python no solo son útiles para almacenar elementos únicos, sino que también permiten realizar diversas operaciones de conjunto, como la unión, la intersección y la diferencia. Estas operaciones son muy útiles para trabajar con conjuntos de datos y realizar análisis.

A continuación, exploraremos tres de las operaciones de conjunto más comunes en Python: la unión, la intersección y la diferencia. A través de ejemplos prácticos, veremos cómo realizar estas operaciones y cómo pueden ser beneficiosas en diferentes situaciones.

```python
# Operaciones de conjunto: unión, intersección y diferencia
conjunto1 = {1, 2, 3, 4, 5}
conjunto2 = {3, 4, 5, 6, 7}

# Unión
union = conjunto1 | conjunto2
print(union)

# Intersección
interseccion = conjunto1 & conjunto2
print(interseccion)

# Diferencia
diferencia = conjunto1 - conjunto2
print(diferencia)  # Salida: {1, 2}
```

Métodos Disponibles para Conjuntos en Python :En Python, los conjuntos (sets) son una estructura de datos útil que proporciona una serie de métodos incorporados para realizar operaciones y manipulaciones. Aquí hay algunos de los métodos más comunes que puedes utilizar con conjuntos:

- `add(elemento)`: Agrega un elemento al conjunto.

```python
mi_conjunto = {1, 2, 3}
mi_conjunto.add(4)
print(mi_conjunto)
```

- `remove(elemento)`: Elimina un elemento específico del conjunto. Genera un error si el elemento no está presente.

```python
mi_conjunto = {1, 2, 3}
mi_conjunto.remove(2)
print(mi_conjunto)
```

- `discard(elemento)`: Elimina un elemento del conjunto si está presente, pero no genera un error si el elemento no existe.

```
mi_conjunto = {1, 2, 3}
mi_conjunto.discard(4)
print(mi_conjunto)
```

- `pop()`: Elimina y devuelve un elemento aleatorio del conjunto.

```
mi_conjunto = {1, 2, 3}
elemento = mi_conjunto.pop()
print(elemento)
```

-`clear()`: Elimina todos los elementos del conjunto, dejándolo vacío.

```
mi_conjunto = {1, 2, 3}
mi_conjunto.clear()
print(mi_conjunto)
```

- `union(otro_conjunto)`: Devuelve un nuevo conjunto que es la unión de dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
union = conjunto1.union(conjunto2)
print(union)
```

- `intersection(otro_conjunto)`: Devuelve un nuevo conjunto que es la intersección de dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
interseccion = conjunto1.intersection(conjunto2)
print(interseccion)
```

- `difference(otro_conjunto)`: Devuelve un nuevo conjunto que es la diferencia entre dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
diferencia = conjunto1.difference(conjunto2)
print(diferencia)
```

- `issubset(otro_conjunto)`: Verifica si el conjunto es un subconjunto de otro conjunto.

```
conjunto1 = {1, 2}
conjunto2 = {1, 2, 3, 4}
es_subconjunto = conjunto1.issubset(conjunto2)
print(es_subconjunto)
```

- `issuperset(otro_conjunto)`: Verifica si el conjunto es un superconjunto de otro conjunto.

```
conjunto1 = {1, 2, 3, 4}
conjunto2 = {1, 2}
es_superconjunto = conjunto1.issuperset(conjunto2)
print(es_superconjunto)  # Salida: True
```

Estos son solo algunos de los métodos disponibles para trabajar con conjuntos en Python. Puedes utilizarlos para realizar una variedad de operaciones y manipulaciones en tus datos.

Dictionaries

En Python, los diccionarios son una estructura de datos que permite almacenar pares clave-valor. Cada elemento en un diccionario consiste en una clave única asociada a un valor correspondiente. Los diccionarios son extremadamente flexibles y versátiles, y se utilizan para representar datos estructurados en forma de tabla de búsqueda.

En un diccionario:

- Las claves son únicas y no pueden repetirse.
- Los valores pueden ser de cualquier tipo de datos, como enteros, cadenas, listas u otros diccionarios.
- Los diccionarios son desordenados, lo que significa que no mantienen un orden específico de los elementos.

Los diccionarios se definen utilizando llaves `{}` y cada par clave-valor se separa con `:`. Por ejemplo:

```
# Creación de un diccionario
informacion_estudiante = {
    "nombre": "Juan",
    "edad": 22,
    "asignaturas": ["Matemáticas", "Ciencias", "Lengua"],
}
informacion_estudiante

# Accediendo a elementos en el diccionario
print(informacion_estudiante["nombre"])
print(informacion_estudiante["asignaturas"])

# Modificando un valor en el diccionario
informacion_estudiante["edad"] = 23
informacion_estudiante

# Añadiendo un nuevo par clave-valor al diccionario
informacion_estudiante["graduacion"] = 2023
informacion_estudiante

# Eliminando un par clave-valor del diccionario
del informacion_estudiante["asignaturas"]
informacion_estudiante
```

- El método `get()` se utiliza para obtener el valor asociado con una clave específica en el diccionario. Si la clave no existe, devuelve un valor predeterminado opcional.

```
diccionario = {'nombre': 'Ana', 'edad': 25}
valor = diccionario.get('nombre')
print(valor)
```

- El método `keys()` devuelve una vista de todas las claves presentes en el diccionario.

```
diccionario = {'nombre': 'Ana', 'edad': 25}
claves = diccionario.keys()
print(claves)
```

- El método `values()` devuelve una vista de todos los valores presentes en el diccionario.

```
diccionario = {'nombre': 'Ana', 'edad': 25}
valores = diccionario.values()
print(valores)
```

- El método `items()` devuelve una vista de todos los pares clave-valor presentes en el diccionario.

```
diccionario = {'nombre': 'Ana', 'edad': 25}
items = diccionario.items()
print(items)
```

- El método `update()` se utiliza para actualizar el diccionario con los pares clave-valor de otro diccionario o con pares clave-valor especificados.

```
diccionario = {'nombre': 'Ana', 'edad': 25}
diccionario.update({'edad': 26})
print(diccionario)
```

Puedes leer más sobre estos y otros métodos de diccionarios en la documentación oficial de Python.

Comparativa de Estructuras de Datos en Python

En Python, tienes varias estructuras de datos disponibles para almacenar y manipular información. A continuación, haremos una comparativa entre las listas, las tuplas, los conjuntos y los diccionarios, destacando sus diferencias y cuándo es apropiado usar cada uno:

Listas (Lists):

- **Uso**: Utiliza una lista cuando necesites una colección ordenada y mutable de elementos.
- **Sintaxis**: Se definen con corchetes `[]`.
- **Características Principales**:
  - Pueden contener elementos de diferentes tipos.
  - Los elementos se pueden cambiar (mutable).
  - Se accede a los elementos por índice.

  - Pueden contener duplicados.

Tuplas (Tuples):

- **Uso**: Utiliza una tupla cuando necesites una colección ordenada e inmutable de elementos.
- **Sintaxis**: Se definen con paréntesis `()`.
- **Características Principales**:
  - Pueden contener elementos de diferentes tipos.
  - Los elementos no se pueden cambiar (inmutable).
  - Se accede a los elementos por índice.
  - Pueden contener duplicados.

Conjuntos (Sets):

- **Uso**: Utiliza un conjunto cuando necesites una colección no ordenada y no duplicada de elementos.
- **Sintaxis**: Se definen con llaves `{}`.
- **Características Principales**:
  - Contienen elementos únicos (sin duplicados).
  - No son indexables ni ordenados.
  - Son útiles para realizar operaciones de conjunto como unión e intersección.

Diccionarios (Dictionaries):

- **Uso**: Utiliza un diccionario cuando necesites una colección de pares clave-valor.
- **Sintaxis**: Se definen con llaves `{}` y cada par clave-valor se separa por `:`. Ejemplo: `{"clave": valor}`.
- **Características Principales**:
  - Almacenan datos en forma de pares clave-valor.
  - Las claves son únicas y no pueden duplicarse.
  - Los valores pueden ser de cualquier tipo.
  - Son eficientes para búsquedas basadas en claves.

Ejercicio final

Vas a crear un programa que simule un sistema de inventario simple para una tienda. Deberás utilizar variables, tipos de datos, operaciones básicas, listas, tuplas, conjuntos, diccionarios, métodos de cadenas, y operaciones de conjuntos para desarrollar este programa. Aquí están las tareas específicas que debes realizar:

Tareas

- **Paso 1**: Crea un diccionario que represente el inventario de la tienda. El diccionario debe contener productos como claves y tuplas como valores, donde cada tupla contiene la cantidad de unidades disponibles y el precio por unidad. Por ejemplo:

```
# Ejemplo
inventario = {
```

```
    "Producto A": (30, 20.50),
    "Producto B": (20, 30.00)
}
```

- **Paso 2**: Usa la función input() para solicitar al usuario que introduzca el nombre de un producto, la cantidad de unidades vendidas y el precio de venta. Utiliza un metodo string para el input().
- **Paso 3:** Usa operaciones básicas para actualizar el inventario después de una venta, y calcula el total de ingresos generados por la venta.
- **Paso 4:** Utiliza métodos string para formatear y mostrar un recibo de venta que incluya el nombre del producto, la cantidad vendida, el precio por unidad y el total de la venta.
- **Paso 5:** Crea una lista que contenga los nombres de todos los productos en el inventario y utiliza operaciones de conjuntos para identificar cualquier producto nuevo que no estaba previamente en el inventario.

Instrucciones Adicionales:

- Utiliza comentarios para documentar tu código de forma clara.
- Asegúrate de que tu programa puede manejar múltiples tipos de datos (como strings y números) e implementa conversiones de tipos cuando sea necesario.
- Trata de incorporar al menos un ejemplo de cada uno de los métodos de cadenas mencionados en la sección de resumen.

```
# tu codigo aquí

# PASO 1: Crear un diccionario que represente el inventario inicial de la tienda
# Cada clave es un nombre de producto y cada valor es una tupla que contiene la
# cantidad de unidades disponibles y el precio por unidad.
inventario = {
    "Producto A": (30, 20.50),
    "Producto B": (20, 30.00)
}

# PASO 2: Solicitar al usuario que introduzca los detalles de la venta
# Utilizamos el método title() para asegurar que la primera letra de cada palabra en el nombre del producto esté en mayúscula.
nombre_producto = input("Por favor, introduce el nombre del producto: ").title()

# Convertimos la entrada de la cantidad vendida a un entero.
cantidad_vendida = int(input("Por favor, introduce la cantidad de unidades vendidas: "))

# Convertimos la entrada del precio de venta a un número flotante.
```

```
precio_venta = float(input("Por favor, introduce el precio de venta
por unidad: "))

# PASO 3: Actualizar el inventario después de una venta y calcular los
ingresos generados por la venta
if nombre_producto in inventario:
    # Obtén la cantidad actual y el precio por unidad del producto.
    cantidad_actual, precio_unitario = inventario[nombre_producto]

    # Ajusta la cantidad de unidades disponibles.
    nueva_cantidad = cantidad_actual - cantidad_vendida

    # Calcula el total de ingresos generados por la venta.
    ingresos_generados = cantidad_vendida * precio_venta

    # Actualiza el inventario con la nueva cantidad. Utilizamos una
tupla para los valores.
    inventario[nombre_producto] = (nueva_cantidad, precio_unitario)
else:
    print(f"El producto '{nombre_producto}' no está en el
inventario.")
    ingresos_generados = 0

# PASO 4: Formatear y mostrar un recibo de venta
# Creamos un recibo formateado con los detalles de la venta.
recibo = f"""
Recibo de Venta
Producto: {nombre_producto}
Cantidad Vendida: {cantidad_vendida}
Precio por Unidad: €{precio_venta:.2f}
Total Venta: €{ingresos_generados:.2f}
"""

# Mostramos el recibo.
print(recibo)

# PASO 5: Crear una lista con los nombres de todos los productos en el
inventario y identificar productos nuevos
# Creamos una lista con los nombres de todos los productos en el
inventario.
lista_productos = list(inventario.keys())

# Supongamos que estos son los nuevos productos que podrían ser
añadidos al inventario
nuevos_productos = {"Producto C", "Producto D"}

# Conjunto de productos actuales
conjunto_productos_actuales = set(lista_productos)

# Productos nuevos que no estaban en el inventario
```

```
productos_nuevos = nuevos_productos - conjunto_productos_actuales

# Mostrar la lista actualizada de productos en el inventario
print("Inventario actualizado:", inventario)
print("Productos nuevos que no estaban en el inventario:",
productos_nuevos)
```

Summary

En este cuaderno de Jupyter, hemos explorado los conceptos fundamentales de Python para principiantes. Aquí hay un resumen de lo que hemos aprendido:

Variables y Tipos de Datos

- Aprendimos cómo declarar variables y explorar tipos de datos como enteros, flotantes, cadenas y booleanos.
- Conocimos las conversiones de tipos implícitas y explícitas.

Operaciones Básicas

- Realizamos operaciones aritméticas básicas como suma, resta, multiplicación, división y módulo.
- Comprendimos la diferencia entre la división normal y la división entera.

Entrada y Salida de Datos

- Utilizamos `input()` para recibir datos del usuario y `print()` para mostrar información en la consola.

Listas, Tuplas y Conjuntos

- Exploramos listas, tuplas y conjuntos como estructuras de datos para almacenar colecciones de elementos.
- Aprendimos a acceder a elementos dentro de estas estructuras y a realizar operaciones comunes.

Diccionarios

- Introducimos los diccionarios como estructuras de datos clave-valor y cómo usarlos para almacenar y recuperar información relacionada.

Operaciones de Conjuntos

- Aprendimos sobre operaciones de conjuntos como unión, intersección y diferencia.

Métodos string

- Exploramos varios métodos para manipular cadenas de texto, incluyendo `capitalize()`, `upper()`, `lower()`, `swapcase()`, `title()`, `join()`, `startswith()`, `endswith()`, `lstrip()`, `rstrip()`, `replace()`, y `split()`.

Este cuaderno proporciona una base sólida para principiantes en Python y servirá como referencia útil a medida que continúes aprendiendo y trabajando con el lenguaje.