«Talento Tech»

Iniciación a la

Programación con Python

Clase 09

Buenos Aires Ciudad

Clase N° 9 | Funciones definidas por el usuario I

Temario:

- Definición y uso de funciones.
- Funciones con y sin parámetros.
- Alcance de variables (variables locales y globales).

Objetivos de la Clase

En esta clase, vas a dar tus primeros pasos en el uso de funciones definidas por el usuario en Python. Aprenderás a crear funciones que te permitirán dividir tu código en bloques más organizados y fáciles de entender. Esto facilitará que tus programas sean más claros, mantenibles evitando repetir el mismo código varias veces.

Vas a entender cómo se definen y utilizan funciones, tanto aquellas que no requieren parámetros como aquellas que los utilizan para personalizar su comportamiento. Además, explorarás el concepto de alcance de variables, comprendiendo la diferencia entre variables locales y globales. Esta distinción es fundamental para controlar dónde y cómo se pueden usar las variables en tu programa, ayudándote a evitar errores y conflictos de nombres.

Esta clase te preparará para profundizar en el uso de funciones en la próxima lección, donde aprenderás a devolver resultados desde las funciones y a documentarlas correctamente. ¡Estás por dar un gran paso hacia una programación más eficiente y profesional! 🚀

Excitante jornada en TalentoLab 🚀

El día comienza con el habitual bullicio en la oficina de **TalentoLab**. Mientras encendés tu computadora y revisás tus correos, un mensaje nuevo de **Mariana** aparece en tu bandeja de entrada. El asunto es claro y directo: “**Hora de organizar el código**”. Abrís el correo y leés:

Estás avanzando muy bien con las tareas del proyecto. Hasta ahora, aprendiste a manejar datos con listas y diccionarios. Sin embargo, a medida que tus programas crecen, se vuelven más largos y difíciles de mantener.

Para que tu código sea más claro y fácil de manejar, es hora de que aprendas a dividirlo en bloques más organizados. Necesitás utilizar ***funciones****, una herramienta esencial para cualquier programador o programadora.*

Sonreís mientras terminás de leer. En ese momento, **Luis**, el desarrollador senior, se asoma por tu escritorio con una taza de café en la mano.

“Hoy vas a descubrir cómo las funciones te pueden salvar de que tu código sea un caos. ¡Vas a ver que es un antes y un después! Nos vemos en la reunión”.

Con la expectativa de un nuevo desafío por delante, cerrás el correo y te preparás para sumergirte en el mundo de las funciones.

Funciónes.

En programación, una **función** es un bloque de código que se agrupa para realizar una tarea específica. Las funciones permiten encapsular una serie de instrucciones que pueden ejecutarse cuando se las necesita. En otras palabras, son como pequeñas unidades de trabajo que se encargan de hacer algo en particular, como sumar dos números, mostrar un mensaje por pantalla o realizar una operación más compleja.

Una función puede recibir datos de entrada y también puede devolver un resultado. Sin embargo, no todas las funciones requieren parámetros (los datos de entrada mencionados previamente) o devuelven un valor; algunas simplemente ejecutan una acción específica. Lo importante es que las funciones ayudan a dividir un programa en partes más pequeñas y manejables, facilitando la organización y el mantenimiento del código.

En Python, además de las funciones que el propio lenguaje nos ofrece (como **print()** o **len()**), existen otras formas de crear funciones para personalizar nuestras tareas y resolver problemas específicos. Esta flexibilidad hace que el código sea más eficiente y fácil de entender.

El concepto de Función Definida por el Usuario (UDF).

Una **función definida por el usuario o usuaria** es una función creada por vos para resolver una tarea específica dentro de tu programa. A diferencia de las funciones que ya vienen incorporadas en Python, cómo **print()** o **input()**, estas funciones las diseñás según las necesidades particulares de tu código. Básicamente, es una forma de decirle a Python qué instrucciones debe ejecutar y cuándo hacerlo.

Definir una función implica darle un nombre y especificar qué acciones debe realizar. También podés decidir si querés que la función reciba datos para trabajar con ellos.

Una vez que definís una función, la podés reutilizar en diferentes partes de tu programa sin tener que volver a escribir el mismo código. Esto no solo hace que el código sea más limpio y ordenado, sino que también facilita el mantenimiento y la corrección de errores.

Por ejemplo, si necesitás calcular el total de una compra con impuestos varias veces en un programa, en lugar de escribir el cálculo cada vez, podés definir una función y llamarla -por ejemplo- **calcular_total()** que realice esa tarea. Luego, cada vez que necesites ese cálculo, simplemente llamás a la función con los datos que corresponden.

Las funciones definidas por la usuaria o usuario permiten personalizar y modularizar el código, haciéndolo más eficiente y adaptado a los requerimientos de cada proyecto. Al usarlas, estás aplicando una de las mejores prácticas en programación, facilitando el desarrollo de soluciones más claras y profesionales.

Ventajas del enfoque funcional.

El uso de funciones en programación tiene muchas ventajas que facilitan el desarrollo y mantenimiento del código. Una de las principales ventajas es que **permite dividir el programa en bloques más pequeños y manejables**, haciendo que sea más fácil entender qué hace cada parte del código. En lugar de tener un programa largo y difícil de seguir, podés estructurarlo en funciones específicas, donde cada una realiza una tarea concreta. Esto ayuda a que el código sea más claro y lógico.

Además, las funciones promueven la **reutilización del código**. Cuando definís una función para resolver un problema específico, podés llamarla tantas veces como necesites sin tener que volver a escribir las mismas instrucciones. Esto no solo ahorra tiempo, sino que también reduce el riesgo de cometer errores, ya que si necesitás hacer una corrección, solo tenés que modificar la función una vez y el cambio se aplicará en todas partes donde la función se utilice.

El enfoque funcional también facilita el **mantenimiento y la depuración** del código. Si algo no funciona como esperabas, podés enfocarte en revisar una función específica en lugar de buscar el problema en todo el programa. Esto hace que encontrar y corregir errores sea mucho más sencillo. Además, si en el futuro necesitás agregar nuevas funcionalidades, podés hacerlo creando nuevas funciones sin modificar el resto del código.

Otra ventaja importante es que las funciones ayudan a mejorar la **legibilidad y comprensión del código**. Cuando alguien más (o vos en el futuro, después de un tiempo) lee el programa, las funciones con nombres claros indican exactamente qué hace cada parte del código. Esto hace que el programa sea más fácil de entender y mantener a largo plazo.

Por último, el uso de funciones permite implementar prácticas de **programación modular**, donde cada función actúa como un bloque independiente. Esta modularidad te permite trabajar en equipo de manera más eficiente, ya que diferentes personas pueden desarrollar y probar funciones específicas sin interferir en el trabajo de los demás.

Descomposición y abstracción.

Al profundizar en las ventajas del enfoque funcional, dos conceptos clave emergen como pilares fundamentales: **la descomposición y la abstracción**. Estos conceptos son

esenciales para entender por qué las funciones son tan poderosas y útiles en la programación.

La **descomposición** se refiere a dividir una tarea compleja en partes más pequeñas y manejables. Cada una de estas partes puede resolverse de forma independiente, lo que simplifica el proceso de desarrollo y hace que el código sea más ordenado. Las funciones son la herramienta ideal para llevar a cabo esta descomposición porque permiten encapsular comportamientos específicos en bloques de código separados. Por ejemplo, en lugar de escribir todo el código para un programa de una sola vez, podés descomponerlo en funciones individuales que realicen tareas concretas, como procesar datos, mostrar información o calcular resultados.

Por otro lado, la **abstracción** implica ocultar los detalles internos de cómo funciona una operación, mostrando solo lo necesario para utilizarla. En el contexto de las funciones, esto significa que podés usar una función sin tener que saber exactamente cómo está implementada por dentro. Lo único que necesitás conocer es su nombre, qué datos recibe (los parámetros) y qué devuelve (el resultado). Esta capacidad de abstraer detalles hace que el código sea más fácil de entender y reduce la cantidad de información que quien desarrolla el software necesita manejar a la vez. Por ejemplo, si definís una función llamada calcular_total() para sumar precios con impuestos, podés usar esa función sin preocuparte por las fórmulas específicas que contiene.

La combinación de **descomposición** y **abstracción** permite desarrollar programas más organizados y mantenibles. Dividir el código en funciones bien definidas ayuda a simplificar el trabajo y facilita agregar nuevas funcionalidades sin afectar otras partes del programa. Además, al abstraer los detalles de implementación, podés centrarte en el "qué hace" cada función en lugar del "cómo lo hace", lo que mejora la claridad y legibilidad del código.

Funciones definidas por el usuario en Python

En Python, una función es un bloque de código que realiza una tarea específica y puede reutilizarse en diferentes partes del programa. Para definir una función en Python, se usa la palabra clave **def**, seguida del nombre de la función y un par de paréntesis. Después de los paréntesis, se coloca un dos puntos **:** y se escribe el cuerpo de la función, que es el conjunto de instrucciones que se ejecutarán cuando se llame a la función.

Es importante recordar que el cuerpo de la función debe estar **indentado**, es decir, con una **sangría** de (generalmente) 4 espacios o un tabulador.

Ejemplo 1: Una función simple

```
def saludo():
   print("¡Hola, mundo!")

# Llamada a la función
saludo()
```

En este ejemplo, saludo es una función que no acepta argumentos y su única acción es imprimir el mensaje "¡Hola, mundo!" en la consola. Para llamar o ejecutar esta función, simplemente usas su nombre seguido de paréntesis.

```
# Llamada a la función
saludo()
```

Ejemplo 2: Función para mostrar un mensaje predefinido

```
def mostrar_mensaje():
   mensaje = "Este es un mensaje predefinido."
   print(mensaje)

# Llamada a la función
mostrar_mensaje()
```

En este caso, **mostrar_mensaje** es otra función que no recibe datos externos. Dentro de su cuerpo, se define una variable llamada mensaje con un texto predefinido y luego se muestra ese mensaje en la consola. Al llamar a esta función, se ejecuta el código contenido en ella.

Estas funciones son una buena manera de empezar a organizar tu código en bloques reutilizables, incluso si todavía no necesitás trabajar con datos externos ni devolver resultados.

Funciones que reciben valores.

En Python, una función puede definirse para recibir datos externos. Estos datos se especifican entre los paréntesis al definir la función y se llaman **parámetros**. Cuando llamás a la función, debés pasarle valores específicos, conocidos como **argumentos**, que serán utilizados dentro de la función para realizar alguna tarea.

Diferencia entre parámetro y argumento

Un **parámetro** es una variable que se define en la función y representa un valor que la función espera recibir. Por otro lado, un **argumento** es el valor real que se le pasa a la función cuando se la llama. Es importante que el número y el orden de los argumentos coincidan con los parámetros definidos en la función.

Para definir una función que recibe datos, se escribe el nombre de la función seguido de paréntesis que contienen los parámetros. Estos parámetros se comportan como variables dentro de la función. Veamos un ejemplo:

```
def saludar(nombre):
   print(f"¡Hola, {nombre}!")

# Llamada a la función con un argumento
saludar("Lucía")
```

En este ejemplo, nombre es un parámetro de la función saludar. Al llamar a la función y pasar el argumento "Lucía", la función muestra el mensaje "¡Hola, Lucía!".

```
¡Hola, Lucía!
```

Podemos llamar a la función usando diferentes argumentos:

```
# Llamada a la función con distintos argumentos
saludar("Ana")
saludar("Martín")
```

Esta es la salida en la terminal:

```
¡Hola, Ana!
¡Hola, Martín!
```

Luis, que está viendo tus avances, te propone un ejemplo más interesante:

"Esta función recibe números, los suma y almacena su valor en la variable resultado y, por último, lo muestra junto con un mensaje.¿Te animás a probarla?"

```
def sumar(numero1, numero2):
    resultado = numero1 + numero2
    print(f"El resultado de la suma es: {resultado}")

# Llamada a la función con argumentos
sumar(10, 5)
```

En este caso, la función que te muestra Luis tiene dos parámetros: **numero1** y **numero2**.

Acá tenés una tabla comparativa que muestra las diferencias entre funciones con parámetros y funciones sin parámetros, destacando su utilidad en distintos contextos:

Aspecto	Funciones sin parámetros	Funciones con parámetros
Definición	No reciben valores al ser llamadas.	Reciben valores (argumentos) al ser llamadas.

Ejemplo	`def saludar():`
	`  print("¡Hola!")`
	
	`saludar()`
		`def saludar(nombre):`
		`  print(f"¡Hola, {nombre}!")`
		
		`saludar("Ana")`

Uso típico	Tareas que no requieren datos externos para ejecutarse.	Tareas que dependen de datos específicos para su ejecución.

Flexibilidad	Limitada, siempre realiza la misma acción.	Flexible, se adapta a distintos datos según los argumentos.
Ventaja	Simplicidad y claridad. Útil para operaciones estáticas o predeterminadas.	Permite reutilizar la función con diferentes datos y contextos.
Ejemplo de aplicación	Imprimir un mensaje fijo o una rutina de inicialización.	Calcular el área de una figura o mostrar información personalizada.

Parámetros predeterminados.

Las funciones en muchos lenguajes de programación, incluido Python, pueden diseñarse para aceptar parámetros opcionales, también conocidos como **parámetros predeterminados u opcionales**. Estos permiten que una función sea llamada con menos argumentos de los que se especifican en su definición. Además, les podes pasar los parámetros por su nombre, lo que aumenta la claridad y flexibilidad del código.

Los parámetros opcionales se definen asignando un valor predeterminado al parámetro en la definición de la función. Si al llamar a la función no se proporciona un argumento para ese parámetro, se utiliza el valor predeterminado.

Veamos un ejemplo:

```
def saludar(nombre, mensaje="Hola"):
    print(f"{mensaje}, {nombre}!")

saludar("Ana")
saludar("Luis", "Buenos días")
```

En este ejemplo, el parámetro mensaje tiene un valor predeterminado de "Hola". Esto significa que si al llamar a saludar() no se proporciona un argumento para mensaje, se usará "Hola" por defecto. La salida del código anterior es la siguiente:

```
Hola, Ana!
Buenos días, Luis!
```

En este otro ejemplo, vamos a definir una función que construya una dirección completa a partir de diferentes partes de una dirección, como la calle, el número, la ciudad, y el código postal. Haremos opcionales el número y el código postal, proporcionándoles valores predeterminados.

```
def crear_direccion(calle, numero='S/N', ciudad='Sin especificar',
codigo_postal='N/A'):
   return f"Dirección: {calle} {numero}, {ciudad}. C.P.:
{codigo_postal}"

# Llamada con todos los parámetros especificados
print(crear_direccion("Avenida Corrientes", 1234, "Buenos Aires",
"1043"))

# Llamada sin especificar el número ni el código postal
print(crear_direccion("Calle Florida", ciudad="CABA"))
```

Luis te explica la función que tiene cada parámetro:

calle: Parámetro obligatorio que especifica el nombre de la calle.
numero (opcional): Número de la vivienda, con un valor predeterminado de 'S/N' (sin número).
ciudad (opcional): Ciudad de la dirección, con un valor predeterminado de 'Sin especificar'.
codigo_postal (opcional): Código postal de la dirección, con un valor predeterminado de 'N/A' (no aplicable).

El código anterior produce la siguiente salida:

```
Dirección: Avenida Corrientes 1234, Buenos Aires. C.P.: 1043
Dirección: Calle Florida S/N, CABA. C.P.: N/A
```

En el primer caso, sólo se proporciona el nombre de la calle, y se usan los valores predeterminados para el número, la ciudad y el código postal. En el segundo caso, se especifican la calle, el número y la ciudad, pero se omite el código postal, por lo que se utiliza el valor predeterminado.

Estos ejemplos te muestran cómo podés utilizar parámetros con valores por defecto para flexibilizar las llamadas a funciones, facilitando su uso en diferentes situaciones sin necesidad de proporcionar todos los datos cada vez.

Parámetros pasados por nombre.

Python te permite pasar argumentos a funciones especificando explícitamente el nombre del parámetro, lo que es muy útil cuando se necesita ignorar el orden en que se definen los parámetros en la función. Esto es particularmente importante en funciones con muchos parámetros, ya que mejora la legibilidad del código:

```
def registrar_usuario(nombre, edad, ciudad):
   print(f"Nombre: {nombre}, Edad: {edad}, Ciudad: {ciudad}")

registrar_usuario(edad=30, nombre="Carlos", ciudad="Madrid")
```

En este caso, los argumentos se pasan a la función **registrar_usuario** por su nombre, lo que te permite cambiar el orden en que se especifican sin afectar el comportamiento de la función. Esto es lo que ves en la terminal:

```
Nombre: Mateo, Edad: 30, Ciudad: Madrid
```

Algunas de las ventajas que tiene el uso de parámetros pasados por nombre incluyen:

Claridad	El uso de parámetros opcionales y el paso de argumentos por nombre hacen que el código sea más claro y fácil de entender, especialmente cuando se trabaja con funciones que tienen muchos parámetros.
Flexibilidad	Permiten que una función sea más versátil y se adapte a diferentes situaciones sin necesidad de definir múltiples funciones para cada variante de comportamiento.

Simplicidad	Reducen la necesidad de sobrecargar funciones o definir múltiples funciones para casos ligeramente diferentes.

El uso de parámetros opcionales y el paso de argumentos por nombre son características muy poderosas que proporcionan flexibilidad y claridad al definir y llamar funciones. Te permiten escribir código más limpio y mantenible, haciéndote más fácil la gestión de la lógica de las funciones y la interacción con ellas.

¿Cómo nombrar las funciones definidas por el usuario?

A la hora de nombrar funciones en Python, seguir convenciones y buenas prácticas es clave para que el código sea fácil de leer, mantener y entender. Esto no solo ayuda a que puedas organizarte mejor, sino que también facilita el trabajo en equipo y el mantenimiento del código a futuro. A continuación, te detallo algunas pautas que te van a ayudar a elegir nombres adecuados para tus funciones.

Los nombres de las funciones tienen que ser **claros y descriptivos**, reflejando de manera precisa lo que hace la función. Por ejemplo, calcular_area() es mucho más entendible que func1(). Elegir nombres que describan bien la acción de la función evita confusiones y hace que el código sea más intuitivo.

En Python, lo más común es **usar letras minúsculas** y separar las palabras con **guiones bajos** (underscores). Este estilo se llama snake_case. Por ejemplo, calcular_area_circulo() es preferible a CalcularAreaCirculo o calcularAreaCirculo.

Es una buena idea empezar los nombres de las funciones con un **verbo** que indique claramente qué acción realiza. Ejemplos de verbos comunes son: obtener, calcular, mostrar, crear o imprimir. Por ejemplo, **obtener_nombre_completo()** deja claro que la función se encarga de obtener un nombre completo.

Tratemos de **evitar abreviaturas y acrónimos** que puedan ser confusos. Aunque a veces los acrónimos son aceptables si son muy conocidos (como IMC para "Índice de Masa Corporal"), siempre es mejor priorizar la claridad. Por ejemplo, calcular_imc() puede ser entendible, pero calcular_indice_masa_corporal() es aún más claro. Pero si bien es importante que los nombres sean descriptivos, no te excedas con la longitud. Nombres excesivamente largos pueden dificultar la lectura. Encontrá un equilibrio entre ser claro y ser breve.

Los nombres deben ser lo más **específicos** posible. En lugar de usar algo genérico como procesar_datos(), un nombre más preciso sería filtrar_usuarios_activos(). Esto ayuda a entender rápidamente qué hace exactamente la función sin necesidad de revisar su código.

Es importante evitar nombres que puedan confundirse con funciones integradas de Python, como **list()**, **str()** o **input()**, ya que esto puede generar errores y comportamientos inesperados. Elegí nombres que no entren en conflicto con estos identificadores.

El alcance de las variables en Python.

Cuando trabajamos con funciones en Python, es fundamental entender el concepto de alcance de las variables. El alcance determina en qué partes del programa una variable es accesible y dónde no lo es. En otras palabras, define el "ámbito" en el que una variable existe y se puede usar. Existen dos tipos principales de alcance: **local** y **global**.

Variables locales

Las variables locales son aquellas que se definen dentro de una función. Su alcance se limita exclusivamente a esa función, lo que significa que sólo pueden ser usadas dentro de ella. No se puede acceder a una variable local desde fuera de la función en la que fue declarada. Estas variables se crean cuando se llama a la función y se eliminan automáticamente cuando la función termina de ejecutarse.

Veamos un ejemplo claro:

```
def saludar():
    mensaje = "¡Hola, mundo!"  # 'mensaje' es una variable local
    print(mensaje)

saludar()
print(mensaje)
# Esto da un error porque 'mensaje' no existe fuera de la función
```

En este caso, la variable mensaje sólo existe dentro de la función saludar(). Si intentás acceder a mensaje fuera de esa función, obtenés un error porque su alcance es local.

Variables globales

Por otro lado, una **variable global** es aquella que se declara fuera de cualquier función. Esto significa que puede ser utilizada tanto dentro como fuera de las funciones. Sin

embargo, si queremos modificar una variable global desde dentro de una función, debemos declararla explícitamente como global usando la palabra clave **global**. De lo contrario, Python interpretará que estamos creando una nueva variable local dentro de esa función.

Veamos un ejemplo de variable global:

```
# 'nombre' es una variable global
nombre = "Ana"

def saludar():
    # Declaramos que queremos usar la variable global 'nombre'
    global nombre
    nombre = "María"
    print(f"Hola, {nombre}")

saludar()
print(nombre)
# Esto imprime 'María' porque modificamos la variable global
```

En este caso, la variable nombre se define fuera de la función y puede ser accedida y modificada dentro de saludar() gracias a la declaración global nombre. Si no hubiéramos usado la declaración global, se hubiese creado una variable local dentro de la función, su nombre fuera de la función, donde está el print(nombre), mostraria el mensaje “Ana”. ¡Probalo!

Mutabilidad y referencia en listas y diccionarios.

Cuando pasamos una lista o un diccionario a una función como parámetro, lo que realmente se pasa es una **referencia al objeto**. Esto significa que si modificas el contenido de esa lista o diccionario dentro de la función, esos cambios se reflejarán fuera de la función porque estamos operando sobre el mismo objeto en memoria.

Ejemplo sin **global**:

```
def agregar_item(lista):
    lista.append("nuevo elemento")

frutas = ["manzana", "banana"]
agregar_item(frutas)
print(frutas)  # Salida: ['manzana', 'banana', 'nuevo elemento']
```

En este caso, se modifica la lista original porque estamos trabajando con su referencia directa.

La cláusula global en funciones con listas y diccionarios.

La cláusula **global** permite **declarar una variable global dentro de una función**, lo que significa que cualquier asignación hecha a esa variable dentro de la función afectará a la variable global, no a una copia local.

Sin embargo, en el caso de **listas y diccionarios**, es importante distinguir entre:

1. **Modificar el contenido del objeto (mutación):** Esto **no requiere global** porque la función está trabajando con la referencia al objeto.
2. **Reasignar una lista o diccionario completo:** Esto **sí requiere global** si se quiere afectar la variable global.

Ejemplo con **global** para reasignar una lista:

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

def reemplazar_lista():
    # Declaramos que queremos modificar la variable global
    global frutas
    frutas = ["naranja", "kiwi"]

reemplazar_lista()
print(frutas)  # Salida: ['naranja', 'kiwi']
```

En este caso, sin global, se crearía una nueva variable local frutas dentro de la función y la lista global no se vería afectada.

Ejemplo sin global para modificar el contenido:

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

def agregar_item():
    # Modifica el contenido sin necesidad de 'global'
    frutas.append("naranja")

agregar_item()
print(frutas)  # Salida: ['manzana', 'banana', 'naranja']
```

```

```

Aquí no es necesario declarar **global** porque sólo estamos modificando el contenido de la lista global, no reasignándola.

Tuplas.

Las tuplas son **estructuras inmutables**, es decir, no se pueden modificar una vez creadas. Si intentás cambiar una tupla dentro de una función, Python generará un error. Sin embargo, si reasignás una tupla dentro de una función, la reasignación no afectará a la tupla original fuera de la función.

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

def agregar_item():
    # Modifica el contenido sin necesidad de 'global'
    frutas.append("naranja")

agregar_item()
print(frutas)  # Salida: ['manzana', 'banana', 'naranja']
```

En este caso, la tupla **numeros** no cambia fuera de la función porque, aunque se crea una nueva tupla dentro de la función, no se modifica la original.

Recomendación

Cuando trabajes con funciones y necesites pasar una **lista** o un **diccionario**, recordá que las modificaciones dentro de la función afectarán al objeto original. Si no querés que esto ocurra, podés crear una **copia** del objeto antes de pasarlo a la función:

```
def agregar_elemento(lista):
    lista.append("nuevo elemento")

frutas = ["manzana", "banana"]

# Pasamos una copia de la lista
```

```
agregar_elemento(frutas.copy())

print(frutas)  # Salida: ['manzana', 'banana']
```

De esta manera, preservás el estado original de la lista. En cambio, con las **tuplas**, como son **inmutables**, no hace falta preocuparse por este tipo de efectos secundarios.

La importancia del alcance.

Entender el alcance de las variables ayuda a evitar errores y a escribir código más organizado y eficiente. Las variables locales permiten encapsular datos dentro de funciones sin interferir con otras partes del programa. Por otro lado, las variables globales pueden ser útiles para datos que necesitan ser compartidos en varias funciones, pero deben usarse con cuidado para no generar confusión o errores inesperados.

Siempre que sea posible, usá **variables locales** dentro de tus funciones. Las variables locales ayudan a que el código sea más modular, fácil de entender y mantener. Al limitar el alcance de una variable a una función específica, evitás interferencias con otras partes del programa y reducís la posibilidad de errores inesperados.

Además, el uso de variables locales facilita el proceso de depuración y hace que tus funciones sean más reutilizables, ya que no dependen del estado global del programa. Si necesitás que una función devuelva datos, es mejor hacerlo a través de un **valor de retorno** en lugar de modificar variables globales. ¡Aprenderemos a hacer eso pronto!

De esta manera, combinando el uso de funciones con una buena gestión del alcance de las variables, podrás desarrollar programas más organizados, claros y fáciles de mantener.

Implementación de un menú para gestionar una lista de frutas con funciones.

Vamos a implementar un pequeño programa que utilice funciones para agregar, consultar y borrar frutas de una lista. El programa presentará un menú que le permitirá al usuario elegir qué acción desea realizar. Utilizaremos funciones definidas por el usuario para organizar el código de manera modular, lo que facilita su lectura y mantenimiento.

En primer lugar, definimos una lista vacía llamada **frutas** que almacenará los nombres de las frutas. Luego, crearemos tres funciones: una para **agregar frutas**, otra para **consultar la lista de frutas**, y una más para **borrar una fruta específica**. Finalmente,

implementaremos una función para mostrar el menú y permitir que se seleccione una opción.

Aquí está el código completo con los comentarios necesarios para entender cada parte:

```
# Lista vacía para almacenar las frutas
frutas = []

# Función para agregar una fruta a la lista
def agregar_fruta():
    fruta = input("Ingresá el nombre de la fruta que querés agregar: ").capitalize()
    frutas.append(fruta)
    print(f"Fruta '{fruta}' agregada con éxito.\n")

# Función para consultar (mostrar) todas las frutas en la lista
def consultar_frutas():
    if frutas:
        print("Lista de frutas:")
        for i, fruta in enumerate(frutas, start=1):
            print(f"{i}. {fruta}")
    else:
        print("La lista de frutas está vacía.")
    print()

# Función para borrar una fruta de la lista
def borrar_fruta():
    if frutas:
        fruta = input("Ingresá el nombre de la fruta que querés borrar: ").capitalize()
        if fruta in frutas:
            frutas.remove(fruta)
            print(f"Fruta '{fruta}' eliminada con éxito.\n")
        else:
            print(f"La fruta '{fruta}' no está en la lista.\n")
    else:
        print("La lista de frutas está vacía. No hay nada para borrar.\n")
```

```
# Función para mostrar el menú y manejar las opciones del usuario
def mostrar_menu():
   while True:
       print("Menú de gestión de frutas:")
       print("1. Agregar una fruta")
       print("2. Consultar la lista de frutas")
       print("3. Borrar una fruta")
       print("4. Salir")

       opcion = input("Elegí una opción (1-4): ")

       if opcion == "1":
           agregar_fruta()
       elif opcion == "2":
           consultar_frutas()
       elif opcion == "3":
           borrar_fruta()
       elif opcion == "4":
           print("¡Gracias por usar el programa! ¡Chau!")
           break
       else:
           print("Opción no válida. Por favor, ingresá un número
del 1 al 4.\n")

# Llamada inicial para ejecutar el menú
mostrar_menu()
```

Luis ve la forma en la que mirás el código y te ayuda a interpretar cómo funciona:

"*El programa comienza definiendo una lista vacía llamada **frutas** que servirá para almacenar los nombres de las frutas ingresadas por quién usa el programa.*

1. ***Función agregar_fruta**: Esta función pide que se ingrese el nombre de una fruta, lo convierte a mayúscula inicial con .capitalize() y la agrega a la lista con el método .append(). Luego, muestra un mensaje confirmando que la fruta fue agregada.*
2. ***Función consultar_frutas**: Muestra todas las frutas almacenadas en la lista. Si la lista está vacía, informa que no hay frutas para mostrar. Utiliza un bucle for con enumerate para mostrar cada fruta con un número de posición.*
3. ***Función borrar_fruta**: Pide el nombre de una fruta para eliminarla. Si la fruta está en la lista, la elimina con el método .remove(). Si no se encuentra en la lista, muestra*

un mensaje indicando que la fruta no existe. Si la lista está vacía, avisa que no hay nada para borrar.

4. ***Función mostrar_menu**: Muestra un menú con opciones numeradas del 1 al 4. Dependiendo de la opción ingresada, llama a la función correspondiente. La opción 4 permite salir del programa, mientras que cualquier otra entrada no válida muestra un mensaje de error*."

Este ejemplo te muestra cómo podés usar **funciones** para organizar tu código y aplicar conceptos como **listas** y **bucles while**. Además, muestra cómo estructurar un pequeño programa de manera modular y reutilizable.

Ejercicio práctico:

Mariana te ha asignado una tarea clave para el proyecto de **TalentoLab**: desarrollar un pequeño programa que permita gestionar una lista de productos utilizando funciones. El objetivo es que la usuaria o usuario pueda agregar, consultar y eliminar productos, aplicando lo que aprendiste sobre funciones y listas. Este ejercicio te ayudará a modularizar el código y a practicar la interacción con el usuario. Así te lo ha comunicado Mariana:

¡Hola!

Tu tarea es crear un programa en Python con las siguientes características:

- **Agregar productos**: Permite a la usuaria o usuario agregar productos a una lista. Cada producto debe tener un nombre y un precio.
- **Consultar productos**: Muestra todos los productos en la lista junto con sus precios.
- **Eliminar productos**: Elimina un producto de la lista a partir de su nombre.
- **Menú interactivo**: El programa debe ofrecer un menú para que el usuario o usuaria pueda elegir qué acción realizar. Debe incluirse una opción para salir del programa.

¡Ponete a prueba y completá el desafío!

Materiales y recursos adicionales:

Artículos:
Escuela Superior Politécnica del Litoral: Funciones en Python
Pablo Londoño: Guía básica de funciones en Python

Videos:
Tutoriales sobre Ciencia y Tecnología: Funciones en Python
Píldoras informáticas: Funciones en Python (I) y Funciones en Python (II)

Preguntas para reflexionar:

1. ¿Por qué es útil dividir el código en funciones en lugar de escribir todo en un solo bloque? Reflexioná sobre cómo las funciones ayudan a organizar y simplificar tareas complejas en tu programa. ¿Qué ventajas notás al estructurar el código de esta manera?
2. ¿Cómo podría mejorar el programa de gestión de productos que desarrollaste? Considerá qué funcionalidades adicionales podrías agregar (por ejemplo, modificar el precio de un producto o buscar productos específicos). ¿Cómo cambiaría el código si implementaras estas mejoras?
3. ¿Cómo podrías aplicar lo aprendido sobre funciones en tu Trabajo Final Integrador (TFI)? Pensá en qué partes del TFI podrías simplificar o mejorar usando funciones. ¿Qué beneficios tendría modularizar tu código para el proyecto final?

Próximos pasos:

En la próxima clase, vas a profundizar en el uso de funciones para llevar tu código al siguiente nivel. Aprenderás a trabajar con la sentencia **return**, una herramienta fundamental que permite a las funciones devolver resultados para ser utilizados en otras partes del programa. Esto te permitirá crear funciones más flexibles y reutilizables, que no solo ejecuten una tarea sino que también devuelvan valores útiles.

Además, explorarás cómo las funciones pueden devolver **múltiples valores** utilizando tuplas, lo que te permitirá manejar varios datos de manera sencilla y eficiente. También incorporarás buenas prácticas de programación con **docstrings**, una manera de documentar tus funciones para que sean más fáciles de entender y mantener, tanto para vos como para otros miembros del equipo de TechLab.


Buenos Aires
aprende
Buenos
Aires
Ciudad