«Talento Tech»

Iniciación a la

Programación con Python

Clase 10

Buenos Aires Ciudad

Clase N° 10 | Funciones definidas por el usuario II

Temario:

- Uso de la sentencia return.
- Funciones que devuelven valores.
- Funciones que devuelven múltiples valores (tuplas).
- Documentación de funciones (docstrings).

Objetivos de la clase

En esta clase vas a explorar cómo las funciones pueden devolver resultados que se utilicen en otras partes de tu programa. Aprenderás a emplear la sentencia return, una herramienta fundamental para hacer que tus funciones sean más flexibles y útiles. Además, vamos a introducir el concepto de funciones que devuelven múltiples valores, utilizando tuplas para facilitar este tipo de operaciones.

También profundizaremos en las buenas prácticas de documentación mediante el uso de docstrings, lo que te permitirá escribir código claro y fácil de entender, tanto para vos como para cualquier persona que lo utilice en el futuro.

Al término de esta clase, podrás aplicar funciones en escenarios más complejos, integrándolas con todo lo que ya aprendiste previamente en el curso. 🚀

Hermosa mañana en TalentoLab 🚀

Es una hermosa mañana en **TalentoLab** y el equipo se encuentra reunido en la sala común, compartiendo café y algunas ideas sobre los proyectos en curso. Mientras organizás tus materiales para el día, Luis, el desarrollador senior, se acerca y te cuenta:

*Hoy vamos a llevar tus funciones al siguiente nivel. Hasta ahora, aprendiste a estructurar tu código usando funciones definidas por vos. Pero hay algo fundamental que no podés dejar pasar: el poder de devolver valores desde una función. Imaginate esto: ¿qué pasaría si cada vez que querés calcular un resultado, tenés que imprimirlo y no podés usarlo en otra parte del programa? Sería un caos, ¿no? Por eso hoy vas a aprender sobre la sentencia **return**.*

Mariana aparece por la puerta y se suma a la conversación:

Necesitamos que te familiarices con funciones que devuelvan más de un valor. También quiero que aproveches la oportunidad para incorporar documentación a tus funciones. Es una práctica esencial, no solo para trabajar en equipo, sino también para que vos puedas entender tu propio código dentro de unos meses.

Con esa introducción, sabés que el día será un desafío interesante, y te entusiasma pensar en cómo estas herramientas te van a ayudar en el desarrollo del Trabajo Final Integrador.

Introducción a la sentencia return.

En la clase anterior, aprendiste a definir y utilizar funciones para organizar tu código en bloques reutilizables que realizan tareas específicas. Hasta ahora, las funciones que viste ejecutaban acciones, pero no devolvían ningún resultado directamente. Es aquí donde entra en juego la sentencia **return**, una herramienta fundamental en Python para que una función pueda devolver un valor o un resultado a quien la llama.

La sentencia **return** marca el final de la ejecución de una función y, opcionalmente, puede devolver un valor al código que llamó a la función. Esto permite que los resultados de los

cálculos o procesos realizados dentro de la función sean utilizados fuera de ella, ya sea para mostrarlos, almacenarlos o procesarlos nuevamente.

La principal ventaja de **return** es que hace que las funciones sean más flexibles y útiles. Podés encapsular una lógica compleja dentro de una función y simplemente devolver el resultado, evitando repetir código. Además, el uso de **return** fomenta el desarrollo de código más modular y legible.

Return en una función simple.

Imaginá que queremos crear una función que calcule el cuadrado de un número y devuelva el resultado para que pueda ser utilizado más adelante en el programa. A continuación vemos una posible solución:

```
# Definimos una función que calcula el cuadrado de un número
def calcular_cuadrado(numero):
    # Elevamos el número al cuadrado
    resultado = numero ** 2
    # Devolvemos el resultado
    return resultado

# Llamamos a la función y almacenamos el valor devuelto
numero_ingresado = int(input("Ingresá un número: "))
cuadrado = calcular_cuadrado(numero_ingresado)

# Mostramos el resultado al usuario
print(f"El cuadrado de {numero_ingresado} es {cuadrado}")
```

*“En este programa, primero definimos una función llamada **calcular_cuadrado**, que toma un número como parámetro. Dentro de la función, el número se eleva al cuadrado usando el operador ******, y luego el resultado se devuelve con la sentencia **return**.”*

Cuando llamamos a la función, el valor devuelto se asigna a la variable cuadrado, que luego utilizamos para mostrar el resultado a la usuaria o usuario”, finaliza Luis.

Este enfoque te permite reutilizar la lógica de calcular el cuadrado en diferentes partes de tu programa, simplemente llamando a la función con un nuevo número como argumento.

Ejemplo práctico: calculadora simple.

En este ejemplo, crearemos una función para cada operación matemática básica (suma, resta, multiplicación y división). Estas funciones serán llamadas desde un menú que permite elegir la operación que desea realizar.

```
# Definimos funciones para cada operación matemática
def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

def multiplicar(a, b):
    return a * b

def dividir(a, b):
    if b != 0:
        return a / b
    else:
        return "Error: No se puede dividir por cero."

# Función para mostrar el menú y ejecutar la calculadora
def calculadora():
    print("Bienvenido/a a la calculadora simple.")
    print("Elegí una operación:")
    print("1. Suma")
    print("2. Resta")
    print("3. Multiplicación")
    print("4. División")
    print("5. Salir")

    while True:
        opcion = input("Ingresá el número de la operación que querés
realizar: ")

        if opcion == "5":
```

```
            print("Gracias por usar la calculadora. ¡Hasta luego!")
            break

        if opcion in ["1", "2", "3", "4"]:
            num1 = float(input("Ingresá el primer número: "))
            num2 = float(input("Ingresá el segundo número: "))

            if opcion == "1":
                print(f"Resultado: {sumar(num1, num2)}")
            elif opcion == "2":
                print(f"Resultado: {restar(num1, num2)}")
            elif opcion == "3":
                print(f"Resultado: {multiplicar(num1, num2)}")
            elif opcion == "4":
                print(f"Resultado: {dividir(num1, num2)}")
        else:
            print("Opción no válida. Por favor, elegí una opción entre 1
y 5.")
```

Luis te explica que *"el corazón de esta calculadora son las funciones que definimos para cada operación matemática. Cada función toma dos argumentos (a y b) y devuelve el resultado correspondiente. Por ejemplo, la función sumar simplemente retorna la suma de sus dos parámetros.*

Lo interesante acá es cómo integramos estas funciones dentro de un bucle para crear una experiencia interactiva. En el menú principal, otorgamos la opción de elegir qué operación realizar o salir del programa. Cuando se elige una operación válida (del 1 al 4), pedimos los dos números necesarios, llamamos a la función correspondiente y mostramos el resultado.

El manejo de la división tiene una validación extra para evitar dividir por cero, que es un caso común de error en programación. Este tipo de detalle demuestra la importancia de anticiparse a posibles problemas y solucionarlos de antemano.

Por último, el uso del bucle while nos permite que la calculadora siga funcionando hasta que se seleccione la opción salir. Este enfoque modular y reutilizable no solo hace el código más legible, sino que también sienta una base sólida para futuras mejoras (¡y para implementar en el TFI!), como agregar nuevas operaciones o implementar funciones más avanzadas.

Ejemplo práctico:

Veamos cómo en este caso podemos crear una función que calcule el precio total de un producto en base a su precio unitario y cantidad, pero con un parámetro opcional que permita aplicar un descuento.

```
# Definimos una función para calcular el precio total
def calcular_precio_total(precio_unitario, cantidad, descuento=0):
    subtotal = precio_unitario * cantidad  # Calculamos el subtotal
    descuento_aplicado = subtotal * (descuento / 100)  # Calculamos el
descuento
    total = subtotal - descuento_aplicado  # Calculamos el total con
descuento
    return total  # Devolvemos el precio total

# Solicitamos datos al usuario
precio = float(input("Ingresá el precio unitario del producto: "))
cantidad = int(input("Ingresá la cantidad de unidades: "))
descuento = input("Ingresá el descuento (en %), o presioná Enter para
no aplicar descuento: ")

# Convertimos el descuento si se ingresó algún valor
if descuento.strip() == "":
    descuento = 0  # Sin descuento
else:
    descuento = float(descuento)

# Llamamos a la función y mostramos el resultado
total = calcular_precio_total(precio, cantidad, descuento)
print(f"El precio total con un descuento de {descuento}% es:
${total:.2f}")
```

Esta función llamada **calcular_precio_total** tiene tres parámetros: **precio_unitario**, **cantidad** y un parámetro opcional llamado **descuento**, que por defecto tiene un valor de **0**.

- Calculamos el subtotal multiplicando el precio unitario por la cantidad.
- Si se aplica un descuento, lo calculamos como un porcentaje del subtotal.
- Luego, restamos el descuento al subtotal para obtener el total final.

Cuando llamamos a la función, el usuario o usuaria puede optar por no ingresar un descuento, en cuyo caso se utiliza el valor predeterminado (**0%**). Esto hace que el programa sea más flexible y permita resolver casos específicos del Trabajo Final Integrador.

Funciones que devuelven múltiples valores.

En Python, las funciones no están limitadas a devolver un solo valor. Podés devolver múltiples valores usando **tuplas**. Esto es especialmente útil cuando una operación o cálculo genera más de un resultado que necesitás utilizar después. ¡Veamos cómo!

Cuando una función necesita devolver varios resultados, Python agrupa automáticamente esos valores en una tupla únicamente **si separás por comas en la sentencia *return***. Luego, al llamar a la función, podés asignar esos valores a variables individuales o procesarlos directamente como una tupla.

Ejemplo práctico: cálculo combinado.

Imaginemos que necesitás una función para calcular el área y el perímetro de un rectángulo y querés devolver ambos resultados al mismo tiempo. Podrías escribir un script como el siguiente:

```
# Función que calcula el área y el perímetro de un rectángulo
def calcular_area_y_perimetro(base, altura):
    area = base * altura
    perimetro = 2 * (base + altura)
    # Devolvemos ambos valores como una tupla
    return area, perimetro


# Solicitamos al usuario los datos del rectángulo
print("Calculadora de área y perímetro de un rectángulo.")
base = float(input("Ingresá la base del rectángulo: "))
altura = float(input("Ingresá la altura del rectángulo: "))

# Llamamos a la función y asignamos los valores retornados
area, perimetro = calcular_area_y_perimetro(base, altura)
```

```
# Mostramos los resultados
print(f"El área del rectángulo es: {area}")
print(f"El perímetro del rectángulo es: {perimetro}")
```

En este ejemplo, la función **calcular_area_y_perimetro** toma dos parámetros: **base** y **altura**. Calcula el área y el perímetro del rectángulo y devuelve ambos resultados utilizando la sentencia **return**.

Cuando llamamos a la función, los valores devueltos se asignan a las variables **area** y **perimetro**.

Esto es posible gracias a que Python empaqueta los valores en una tupla y luego los descomprime al asignarlos. Este enfoque te permite manejar múltiples resultados de manera clara y sencilla.

El uso de tuplas es especialmente útil en aplicaciones más complejas, donde una función necesita devolver un conjunto de resultados relacionados sin requerir estructuras más avanzadas o personalizadas como clases o diccionarios.

Documentación de funciones (docstrings).

En Python, los **docstrings** son una herramienta esencial para documentar tus funciones. Se trata de cadenas de texto que se colocan justo al comienzo del cuerpo de una función y explican de manera breve y clara qué hace la función, qué parámetros recibe (si los tiene) y qué valores devuelve (si aplica). Esta documentación no solo es útil para otras personas que puedan leer tu código, sino también para cuando necesites revisarlo en el futuro.

¿Qué son los docstrings y para qué sirven?

Un **docstring** es una cadena de texto delimitada por comillas triples **(""")** que se usa para describir el propósito y uso de una función. Python lo reconoce automáticamente como documentación asociada a esa función. Los docstrings también pueden ser utilizados en clases y módulos para describir su propósito general.

La principal ventaja de los docstrings es que se pueden acceder programáticamente mediante la función **help()** de Python. Esto facilita obtener información sobre cómo usar una función sin necesidad de leer el código fuente.

Documentar las funciones con docstrings es una buena práctica que ayuda a que tu código sea más legible, comprensible y profesional.

Ejemplo básico de docstring.

```
def calcular_area_y_perimetro(base, altura):
    """
    Calcula el área y el perímetro de un rectángulo.

    Parámetros:
        base (float): La base del rectángulo.
        altura (float): La altura del rectángulo.

    Retorna:
        tuple: Una tupla que contiene el área y el perímetro del
rectángulo.
    """
    area = base * altura
    perimetro = 2 * (base + altura)
    return area, perimetro
```

¿Cómo usar los docstrings?

Para consultar el docstring de una función en Python podés usar la función **help()**:

```
help(calcular_area_y_perimetro)
```

Al ejecutar este código, Python mostrará la descripción contenida en el docstring, incluyendo información sobre los parámetros y el valor de retorno:

```
Help on function calcular_area_y_perimetro in module __main__:
```

```
calcular_area_y_perimetro(base, altura)
    Calcula el área y el perímetro de un rectángulo.

    Parámetros:
        base (float): La base del rectángulo.
        altura (float): La altura del rectángulo.

    Retorna:
        tuple: Una tupla que contiene el área y el perímetro del
rectángulo.
(END)
```

Beneficios de los docstrings.

1. **Claridad**: Ayudan a que el propósito de la función sea claro para cualquier persona que lea el código.
2. **Accesibilidad**: Podés acceder a la documentación directamente desde el intérprete usando **help()**.
3. **Estándar de la comunidad**: Los docstrings son una convención ampliamente aceptada en Python, lo que facilita la colaboración en proyectos.
4. **Mantenimiento**: Cuando el código evoluciona, los docstrings permiten actualizar la documentación sin tener que crear archivos separados.

Comparación entre funciones con y sin return.

En Python, las funciones pueden definirse para realizar tareas sin devolver un resultado explícito, o para devolver un valor (o varios) mediante la sentencia **return**. Saber cuándo usar una función con o sin **return** es muy importante para escribir código eficiente, modular y reutilizable.

¿Cuándo usar funciones con return?

Las funciones con **return** son ideales cuando se necesita que una operación devuelva un resultado que pueda ser usado en otras partes del programa. Por ejemplo, calcular un valor, realizar una transformación de datos o extraer información. Estas funciones son especialmente útiles en casos donde se requiere modularizar el código y permitir que las funciones sean reutilizables en diferentes contextos.

Un ejemplo típico sería una función que calcula el precio final de un producto con impuestos. Al devolver el resultado, este puede ser usado directamente en otras operaciones:

```
def calcular_precio_final(precio, impuesto):
    return precio + (precio * impuesto / 100)
```

Las funciones con **return** aportan modularidad y flexibilidad, ya que pueden integrarse en flujos más complejos de lógica. Por otro lado, las funciones sin **return** son útiles para ejecutar acciones específicas, como imprimir datos o modificar estructuras existentes, sin necesidad de devolver un valor. Estas últimas son más adecuadas para tareas donde no se espera que la función entregue un resultado para su reutilización.

Por ejemplo, una función sin **return** podría ser usada para mostrar un mensaje a quien usa el programa:

```
def mostrar_bienvenida(nombre):
    print(f"¡Bienvenido, {nombre}!")
```

Sin embargo, este tipo de funciones no permite que el resultado sea almacenado o procesado más adelante, limitando su flexibilidad en flujos complejos.

Característica	Funciones con return	Funciones sin return
Propósito	Devuelven un resultado que puede ser reutilizado.	Ejecutan una acción específica sin devolver datos.
Uso común	Cálculos, transformaciones y extracción de información.	Mostrar mensajes, modificar estructuras in-place.
Reutilización	Pueden ser llamadas en otros contextos con diferentes datos.	Limitadas a la acción específica para la que se diseñaron.
Flexibilidad	Alta: pueden integrarse en flujos de lógica complejos.	Baja: no permiten almacenar o procesar resultados.

“Esta comparativa te ayuda a identificar cuándo elegir un enfoque sobre el otro según las necesidades de tu programa. Ambos tipos de funciones tienen su lugar en la programación y, al combinarlos de manera adecuada, podés escribir código más claro, modular y eficiente.”

Ejemplo práctico:

Como parte del proyecto de **TalentoLab**, parece que necesitamos hacer una mejora: queremos que este código no sólo ejecute las acciones, sino que también devuelva resultados que puedan ser utilizados en otras partes del programa.

“Queremos aprovechar este ajuste para incorporar las nuevas habilidades que estás adquiriendo. Esto nos ayudará a seguir construyendo programas cada vez más organizados y modulares."

Aquí está el código mejorado, adaptado para que las funciones devuelvan valores:

```
# Lista inicial de frutas
frutas = []

# Función para agregar una fruta
def agregar_fruta():
    fruta = input("Ingresá el nombre de la fruta que querés agregar:
").capitalize()
    frutas.append(fruta)
    return fruta

# Función para consultar frutas
def consultar_frutas():
    if frutas:
        return frutas
    else:
        return []

# Función para borrar una fruta
def borrar_fruta():
```

```
    fruta = input("Ingresá el nombre de la fruta que querés borrar:
").capitalize()
    if fruta in frutas:
        frutas.remove(fruta)
        return fruta
    else:
        return None

# Función principal para manejar el menú
def mostrar_menu():
    while True:
        print("\nMenú de gestión de frutas:")
        print("1. Agregar una fruta")
        print("2. Consultar frutas")
        print("3. Borrar una fruta")
        print("4. Salir")
        opcion = input("Elegí una opción (1-4): ")
        if opcion == "1":
            fruta = agregar_fruta()
            print(f"Fruta '{fruta}' agregada con éxito.")
        elif opcion == "2":
            lista = consultar_frutas()
            if lista:
                print("Lista de frutas:")
                for i, fruta in enumerate(lista, start=1):
                    print(f"{i}. {fruta}")
            else:
                print("La lista de frutas está vacía.")
        elif opcion == "3":
            fruta_eliminada = borrar_fruta()
            if fruta_eliminada:
                print(f"Fruta '{fruta_eliminada}' eliminada con éxito.")
            else:
                print("La fruta no está en la lista.")
        elif opcion == "4":
            print("¡Gracias por usar el programa! ¡Hasta luego!")
            break
        else:
            print("Opción no válida. Por favor, ingresá un número del 1
al 4.")
mostrar_menu()
```

"Este nuevo enfoque te permite obtener resultados de las funciones que escribís. Por ejemplo, ***agregar_fruta*** *devuelve el nombre de la fruta recién agregada, lo que facilita confirmar al usuario su acción.* ***consultar_frutas*** *devuelve una lista de frutas o una lista vacía, dependiendo del estado actual de la lista. En tanto,* ***borrar_fruta*** *devuelve el nombre de la fruta eliminada o* ***None*** *si no encuentra la fruta. Esto te da flexibilidad para manejar estos datos en otras partes del programa, manteniendo la lógica encapsulada y el código modular."*

Con este diseño, no sólo estás aplicando buenas prácticas de programación, sino que también estás construyendo un código más robusto, escalable y reutilizable. Estas mejoras son muy importantes para que tu Trabajo Final Integrador sea un éxito.

Ejercicio práctico:

El día finaliza con otra reunión en la oficina de Mariana. Con su energía habitual, te felicita por el progreso que estás logrando. "Luis me mostró el programa de gestión de frutas que desarrollaste, y estoy muy impresionada con lo bien que aplicaste las funciones para organizar el código. Sin embargo, quiero proponerte un desafío adicional:

Tu tarea es modificar el programa que escribiste la clase anterior, para que realice las mismas tareas, pero utilizando funciones que devuelvan valores cuando sea posible. Recordá que el programa te permitía:

- **Agregar productos**: Cada producto debe tener un nombre y un precio.
- **Consultar productos**: Muestra todos los productos en la lista junto con sus precios.
- **Eliminar productos**: A partir de su nombre.
- **Menú interactivo**: Debe ofrecer un menú para que se pueda elegir qué acción realizar.

¡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. ¿De qué manera las funciones que devuelven valores pueden hacer que tu código sea más modular y reutilizable? Pensá en cómo podrías aplicar este concepto en proyectos más grandes.
2. ¿Cómo te ayuda la posibilidad de devolver múltiples valores con tuplas a resolver problemas de programación de forma más eficiente? ¿Podés pensar en un ejemplo práctico donde esto sea útil?
3. ¿Qué importancia tiene documentar tus funciones con docstrings? Reflexioná sobre cómo esto puede facilitar la colaboración en proyectos de equipo.

Próximos pasos

En la próxima clase, vas a dar un paso clave en la organización y ampliación de tus proyectos de programación. Exploraremos cómo dividir el código en módulos para hacerlo más ordenado, fácil de mantener y reutilizar en diferentes aplicaciones. Aprenderás a importar módulos propios y externos, lo que te permitirá trabajar de manera más eficiente y profesional.

Además, vamos a conocer algunos de los módulos estándar más útiles de Python, como **random** y **datetime**, para resolver problemas comunes de manera práctica. También haremos una introducción a módulos de terceros, como **colorama**, que aportan herramientas interesantes para mejorar la experiencia del usuario en tus programas.

Asegurate de repasar lo aprendido sobre funciones y diccionarios, ya que estos conceptos serán esenciales para los ejemplos y ejercicios de la próxima clase. ¡Nos vemos pronto para seguir avanzando juntos!

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad