«Talento Tech»

Iniciación a la Programación con Python

CLASE 11

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 11 | Funciones

Temario:

- Funciones en Python: definición y uso.
- Argumentos y retorno de valores.
- Funciones que llaman a funciones.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Introducción a las funciones

En programación, una función es un bloque de código que se puede reutilizar en distintas partes de un programa. Su propósito es evitar repetir las mismas líneas de código una y otra vez, además de hacer que el programa sea más claro y fácil de mantener. Las funciones te permiten dividir tu programa en pequeñas partes más manejables, cada una con una tarea específica.

¿Cómo se define una función?

En Python, para definir una función se utiliza la palabra clave **def** seguida del nombre de la función y un par de paréntesis. Todo el código que forma parte de la función se escribe indentado, como ya hemos hecho con los bucles y las estructuras condicionales. A continuación tenés un ejemplo básico:

```
def saludar():
    print("Hola, ¡bienvenido!")
```

En este caso, hemos definido una función llamada **saludar()**. Esta función, cada vez que se llame, va a imprimir el mensaje “Hola, ¡bienvenido!”. Notá que la función no se ejecuta automáticamente al definirla. Para que se ejecute, hay que llamarla.

Llamando a una función

Para usar una función, simplemente escribimos su nombre seguido de paréntesis. Siguiendo con nuestro ejemplo:

```
saludar()
```

Al escribir saludar() en tu código, Python va a ejecutar la función y va a mostrar el mensaje por pantalla. El código completo nos queda así:

```
def saludar():
   print("Hola, ¡bienvenido!")

saludar()
```

¿Por qué usar funciones?

Imaginate que estás desarrollando un sistema de inventario (como el que estás haciendo en tu Proyecto Final Integrador). Si cada vez que querés mostrar el menú de opciones o actualizar el inventario, tenés que escribir el mismo bloque de código, no solo vas a perder tiempo, sino que tu programa se va a volver complicado de entender. Usando funciones, podés agrupar todas esas acciones repetitivas en un solo lugar y simplemente llamarlas cuando las necesites.

Ventajas de usar funciones:

Una de las grandes ventajas de usar funciones es la reutilización de código. Imaginá que tenés una tarea repetitiva en tu programa, como mostrar un menú o realizar un cálculo. Si cada vez que necesitás esa tarea tenés que escribir el mismo bloque de código, no solo te va a llevar más tiempo, sino que además tu programa se va a volver más largo y desorganizado. Con las funciones, podés escribir ese bloque una sola vez, agruparlo dentro de una función, y luego llamarla cada vez que la necesites. Esto no solo te ahorra esfuerzo, sino que también hace que tu código sea más eficiente y menos propenso a errores.

Otro beneficio clave es la organización. Cuando dividís tu código en funciones, estás creando bloques independientes que realizan tareas específicas. Esto te permite abordar cada parte del programa de manera separada y te facilita pensar en el problema de manera más estructurada. Si necesitás hacer cambios en alguna parte del programa, podés modificar solo la función correspondiente sin tener que preocuparte por cómo afectará al resto del código. Además, tener el código dividido en funciones hace que sea más fácil de mantener y actualizar a lo largo del tiempo.

Por último, las funciones mejoran la claridad del programa. Un programa bien organizado, con funciones que tienen nombres descriptivos, es mucho más fácil de leer y entender. Al escribir una función, estás definiendo qué hace un bloque de código en particular, y eso permite que quien lea el programa (incluso si lo hacés vos en el futuro) entienda de inmediato qué está pasando en cada parte. Esto es fundamental no solo para trabajar en equipo, sino también para cuando quieras revisar o mejorar tu propio código una vez que lo hayas desarrollado. Una buena estructura, basada en funciones, facilita tanto la comprensión como el desarrollo de soluciones más complejas.

Por ejemplo, si querés crear una función para mostrar el menú de tu inventario:

```
def mostrar_menu():
    print("1. Agregar producto")
    print("2. Mostrar inventario")
    print("3. Salir")
```

Cada vez que quieras mostrar el menú, en lugar de escribir todas esas líneas de nuevo, solo llamás a la función:

```
mostrar_menu()
```

El resultado será el siguiente:

```
1. Agregar producto
2. Mostrar inventario
3. Salir
```

Argumentos en las funciones de Python

Cuando hablamos de argumentos en Python, nos referimos a la información que le pasamos a una función para que esta pueda realizar su tarea. Es como darle instrucciones adicionales a la función para que trabaje sobre ciertos valores o realice operaciones específicas. Los argumentos le permiten a una función ser flexible y adaptable, ya que en lugar de depender de datos fijos, puede operar sobre los valores que le pases cada vez que la llamás.

Una función que acepta argumentos se declara especificando una lista de variables dentro de los paréntesis que siguen al nombre de la función. Estas variables funcionan como "placeholders", es decir, reciben los valores que le pases a la función cuando la llames.

Por ejemplo, imaginá que tenés una función que suma dos números. Podrías definirla de la siguiente manera:

```
def sumar(a, b):
   resultado = a + b
   print("El resultado es:", resultado)
```

En este caso, **a y b son los argumentos** de la función *sumar*. Cuando llamás a la función, le pasás dos valores que ocuparán esos lugares:

```
sumar(5, 3)
# Salida: El resultado es: 8
```

Cuando llamás a sumar(5, 3), Python asigna 5 a **a** y 3 a **b**, y luego realiza la operación de suma dentro de la función.

¿Por qué son importantes los argumentos?

El uso de argumentos permite que las funciones sean mucho más dinámicas y reutilizables. En lugar de escribir una función diferente para cada conjunto de valores, podés escribir una función general que trabaje con cualquier par de números o cualquier otro tipo de dato que necesites.

Además, los argumentos no tienen que limitarse a números. Podés pasarle cadenas, listas, diccionarios, o cualquier tipo de dato que Python soporte. Por ejemplo, una función que trabaja con cadenas:

```
def saludar(nombre):
   print("Hola", nombre)

saludar("Ana")
saludar("Carlos")
```

En este ejemplo, **nombre** es el argumento y, dependiendo del valor que le pases al llamar a la función, te va a saludar de manera diferente:

```
Hola Ana
Hola Carlos
```

Argumentos por defecto

En ocasiones, puede ser útil definir valores por defecto para los argumentos de una función. Esto es útil cuando querés que la función tenga un comportamiento estándar, pero que también pueda cambiar cuando le pases un valor específico. Si no se proporciona un valor para un argumento que tiene un valor por defecto, Python usará el valor por defecto.

Veamos un ejemplo:

```
def saludar(nombre="invitado"):
   print("Hola", nombre)

saludar()  # Usa el valor por defecto
saludar("Lucía")  # Sobrescribe el valor por defecto
```

En este caso, si llamás a saludar() sin pasarle ningún argumento, Python va a utilizar el valor por defecto "invitado". Pero si le pasás un nombre, ese valor será utilizado en lugar del valor por defecto. Esta es la salida en la pantalla:

```
Hola invitado
Hola Lucía
```

Argumentos posicionales y nombrados

Cuando llamás a una función, podés pasarle argumentos de dos maneras: posicionales o nombrados.

Argumentos posicionales: El valor que le pases a la función será asignado a los argumentos en el orden en que aparecen. Por ejemplo, en sumar(3, 5), 3 va a **a** y 5 a **b**.

Argumentos nombrados: También podés pasar los argumentos especificando el nombre del parámetro. Esto te permite alterar el orden y hace que el código sea más claro:

```
def sumar(a, b):
   resultado = a + b
   print("El resultado es:", resultado)

sumar(b=7, a=2)
# Salida: El resultado es: 9
```

Acá le estás diciendo explícitamente a Python que a debe valer 2 y b debe valer 7, sin importar el orden en que están escritos en la llamada.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Con esto tus funciones pueden volverse mucho más versátiles y adaptarse a diferentes situaciones. Cuando estés diseñando tu Proyecto Final Integrador, este tipo de flexibilidad te va a permitir que tu código sea mucho más potente y reutilizable.

Funciones que devuelven valores

Ahora, vamos a hablar sobre la capacidad que tienen las funciones de retornar valores, que es una de las herramientas más poderosas en Python. Esto les da a las funciones un nivel adicional de utilidad y flexibilidad.

Cuando una función retorna un valor, lo que está haciendo es devolver un resultado al lugar desde donde fue llamada. Hasta ahora, hemos visto funciones que simplemente ejecutan un conjunto de instrucciones y muestran un resultado en pantalla con print(). Sin embargo, hay momentos en los que solo mostrar un valor no es suficiente: tal vez necesites realizar cálculos adicionales con ese resultado, almacenarlo en una variable, o utilizarlo más adelante en tu programa. Es aquí donde entra en juego el uso de **return.**

Al utilizar **return,** le estás diciendo a la función que una vez que haya hecho su trabajo, debe *"regresar"* un valor al código que la llamó, para que se pueda hacer algo más con él. Este proceso hace que las funciones sean mucho más flexibles, ya que no están limitadas a ejecutar una sola tarea sin seguimiento.

Por ejemplo, imaginá que querés sumar dos números en una función. Con print(), la función simplemente mostraría el resultado, pero no podrías hacer nada más con él. Ahora, si usás return, podés usar ese resultado de manera más amplia en el resto de tu programa. Es una forma de mantener el flujo de información y acciones dentro del código.

Veamos un ejemplo básico de una función que devuelve un valor:

```
def sumar(a, b):
    resultado = a + b
    return resultado
```

Aquí, la función sumar no sólo calcula la suma de **a y b**, sino que también **retorna** el resultado de esa operación. Al retornar el valor, ahora podés almacenarlo en una variable, usarlo en otra operación o pasarlo a otra función. Esto te da un nivel mucho mayor de control.

```
total = sumar(5, 3)
print("El total es:", total)
```

Este es el aspecto que tiene la terminal luego de ejecutar ese código:

```
El total es: 8
```

En este caso, el resultado de la suma de 5 y 3 se almacena en la variable total, y luego podés hacer con ese valor lo que necesites. Es mucho más flexible que simplemente mostrarlo.

¿Por qué es tan importante retornar valores?

El hecho de que una función pueda devolver un valor hace que tu código sea mucho más modular y fácil de mantener. En lugar de depender de variables globales o de tener que repetir el mismo código una y otra vez, podés escribir funciones que resuelvan problemas específicos, retornen los resultados y luego utilizarlos en cualquier parte de tu programa. De esta manera, las funciones se convierten en bloques reutilizables que podés llamar en diferentes lugares y para diferentes propósitos.

Otra gran ventaja es que te permite hacer cálculos complejos dividiendo el problema en partes más pequeñas. Por ejemplo, podrías tener varias funciones que calculan diferentes partes de un problema y cada una devuelve su propio resultado. Luego, podés combinarlos para obtener la solución final. Esto se relaciona directamente con el tema que desarrollaremos a continuación, que es cómo una función puede llamar a otra.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejemplos de funciones que retornan valores

Vamos a ver un par de ejemplos para ilustrar cómo funciona esto en la práctica.

El primer ejemplo es bastante sencillo: imaginemos que querés calcular el cuadrado de un número y usar ese resultado más adelante en el programa.

```
def cuadrado(n):
    return n * n
```

Con esta función, podés calcular el cuadrado de cualquier número y luego usar ese valor en otras partes del código.

```
def cuadrado(n):
    return n * n

resultado = cuadrado(4)
print("El cuadrado de 4 es:", resultado)
# El cuadrado de 4 es: 16
```

El return permite que el resultado de la función cuadrado se almacene en la variable resultado, que después es utilizada para mostrar el valor en pantalla. Podrías reutilizar esta información más adelante para otros cálculos, si fuera necesario.

Otro ejemplo, un poco más relacionado con el manejo de inventarios para tu Proyecto Final Integrador, podría ser calcular el total de ventas de un producto en base a su precio y la cantidad vendida:

```
def calcular_total(precio, cantidad):
    return precio * cantidad

total_venta = calcular_total(100, 3)
print("El total de la venta es:", total_venta)
# El total de la venta es: 300
```

Acá, la función toma dos argumentos: el precio del producto y la cantidad vendida. Luego, retorna el total de la venta, multiplicando ambos valores.

El uso de return en el último ejemplo permite que el cálculo del total de la venta pueda ser almacenado en una variable, total_venta, que luego podés utilizar para mostrar el total o realizar otros cálculos como sumar el total de ventas del día.

Retornar valores en las funciones te permite mantener el código organizado y reutilizable. En lugar de que una función realice una sola acción y termine, podés obtener resultados que se pueden seguir utilizando en otros espacios de tu programa. Esto hace que tu código sea mucho más dinámico y apropiado para resolver problemas complejos de forma sencilla y, sobre todo, modular.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Funciones que llaman a otras funciones

Cuando hablamos de funciones que llaman a otras funciones, nos referimos a la posibilidad de que, dentro de una función, podés invocar a otra función. Esto es algo súper útil porque permite dividir tareas complejas en partes más pequeñas y fáciles de manejar, además de reutilizar código. En lugar de escribir todo de nuevo, podés aprovechar funciones ya definidas para resolver partes de un problema mayor.

¿Cómo funciona esto en la práctica?

Imaginá que tenés un programa que gestiona un inventario de productos y dentro de ese programa necesitás realizar varias tareas diferentes: agregar productos, calcular el total de los productos, mostrar los productos, etc. En lugar de meter todo ese código dentro de una sola función gigante (lo cual sería muy difícil de leer y mantener), podés dividirlo en varias funciones pequeñas y hacer que una función principal llame a las demás según sea necesario.

Ejemplo 1: Llamar a funciones desde otra función
Supongamos que queremos crear un pequeño sistema que calcule el total de ventas de dos productos, usando dos funciones separadas para sumar las ventas de cada uno.

```
# Función para calcular las ventas de un producto
def calcular_ventas_producto1():
    ventas = 10  # Ejemplo de ventas
    return ventas

# Función para calcular las ventas de otro producto
def calcular_ventas_producto2():
    ventas = 15  # Ejemplo de ventas
    return ventas

# Función que llama a las anteriores para calcular el total
```

```
def calcular_ventas_totales():
    total = calcular_ventas_producto1() + calcular_ventas_producto2()
    return total

# Llamamos a la función principal
ventas_totales = calcular_ventas_totales()
print("Las ventas totales son:", ventas_totales)
```

¿Qué está pasando aquí? Tenemos dos funciones, calcular_ventas_producto1 y calcular_ventas_producto2, que calculan las ventas de dos productos diferentes. Después, tenemos una tercera función, calcular_ventas_totales, que llama a las dos funciones anteriores y suma sus resultados. Finalmente, llamamos a calcular_ventas_totales para obtener el total.

Este es un ejemplo sencillo, pero muestra el concepto clave: una función puede utilizar el resultado de otra función para hacer cálculos o realizar tareas más complejas.

Ejemplo 2: Organizar el flujo del programa
Otro escenario en el que las funciones que llaman a otras funciones son útiles es cuando querés organizar mejor el flujo de tu programa. Por ejemplo, si tenés un menú interactivo, podés tener una función principal que controle el menú y, dependiendo de la opción seleccionada, llame a diferentes funciones.

```
def mostrar_menu():
    print("1. Agregar producto")
    print("2. Ver productos")
    print("3. Salir")

def agregar_producto():
    print("Producto agregado.")

def ver_productos():
```

```
    print("Aquí están los productos.")

def iniciar_programa():
    while True:
        mostrar_menu()
        opcion = input("Elegí una opción: ")

        if opcion == "1":
            agregar_producto()
        elif opcion == "2":
            ver_productos()
        elif opcion == "3":
            print("Saliendo del programa...")
            break
        else:
            print("Opción inválida, intentá de nuevo.")

# Llamamos a la función que inicia el programa
iniciar_programa()
```

La función mostrar_menu simplemente muestra las opciones disponibles.
Tenemos dos funciones adicionales: agregar_producto y ver_productos, que realizan acciones específicas. La función iniciar_programa es la encargada de controlar el flujo del programa: muestra el menú, recibe la opción seleccionada, y según esa elección, llama a la función correspondiente.

Este enfoque hace que el código sea mucho más ordenado y fácil de mantener. Si necesitás agregar nuevas opciones al menú, solo tenés que crear nuevas funciones y llamarlas desde iniciar_programa. Así, cada parte del programa está bien separada y no hay necesidad de escribir el mismo código varias veces.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Alcance de las variables

Cuando hablamos de alcance de las variables nos referimos a dónde y cómo podés usar una variable dentro de tu código. En Python, las variables que se crean dentro de una función tienen un comportamiento especial: solo existen mientras esa función se está ejecutando. Eso quiere decir que, si creás una variable dentro de una función, no podés usarla afuera de la misma, y una vez que la función termina, esa variable desaparece.

Por ejemplo, mirá este código:

```
def saludar():
    mensaje = "Hola, ¿cómo estás?"
    print(mensaje)

saludar()
print(mensaje)   # Esto va a dar un error
```

En este caso, la variable mensaje solo existe dentro de la función saludar. Cuando tratamos de usarla fuera de la función, Python nos va a dar un error, porque no sabe qué es mensaje fuera de la función. Esto es lo que llamamos **alcance local:** las variables dentro de las funciones son exclusivas de esa función.

Ahora, si una variable se crea fuera de una función, esa variable puede usarse tanto fuera como dentro de la función. Fijate en este ejemplo:

```
mensaje_global = "Hola, ¿cómo estás?"

def saludar():
    print(mensaje_global)

saludar()
print(mensaje_global)   # Esto funciona bien
```

En este caso, la variable mensaje_global se creó fuera de la función, por lo que tanto el programa como la función saludar la pueden usar sin problemas. Las variables que se crean fuera de una función se denominan **variables globales** y son accesibles desde cualquier parte del código, incluso desde dentro de las funciones.

En pocas palabras:

- Si creás una variable dentro de una función, solo podés usarla dentro de esa función. Cuando la función termina, esa variable desaparece.
- Si una variable se crea fuera de la función, podés usarla tanto dentro como fuera de la función.

Este comportamiento es muy útil para que las variables dentro de las funciones no interfieran con otras partes de tu código. Ayuda a mantener todo ordenado y fácil de manejar, evitando confusiones.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicios prácticos:

Gestión de descuentos

Imaginá que en tu tienda querés implementar un sistema de descuentos automáticos. Vas a desarrollar un programa que permita calcular el precio final de un producto después de aplicar un descuento. Para hacerlo:

1. Crea una función que reciba como parámetros el precio original del producto y el porcentaje de descuento, y que retorne el precio final con el descuento aplicado.
2. Luego, solicitá que se ingrese el precio y el porcentaje de descuento. Mostrá el precio final después de aplicar el descuento.

Cálculo de promedio de ventas

Desarrollá un programa que permita calcular el promedio de ventas de la tienda. Para esto:

1. Creá una función que reciba como parámetro una lista de ventas diarias y devuelva el promedio de esas ventas.
2. Solicitá a la persona que ingrese las ventas de cada día durante una semana (7 días). Usá la función para calcular y mostrar el promedio de ventas al finalizar.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad