



ACTIVIDAD: INTRODUCCIÓN A LAS FUNCIONES

Kevin Meléndez Juárez

26 de febrero de 2026

"Actúa como tutor de Python para Ciencia de Datos. Explicame detalladamente en qué se diferencian los distintos tipos de funciones en Python (por ejemplo: built-in, definidas por el usuario con def, lambda/anónimas, métodos (instancia/clase/estático), generadoras con yield y funciones async). Para cada tipo: (1) definición breve, (2) un ejemplo corto en Python, y (3) un caso real de cuándo usarla."

Hoy  

Pensó durante 9 s >

¡Claro que sí! Con gusto seré tu tutor. Entender bien los tipos de funciones es clave en Python, especialmente en Ciencia de Datos, donde constantemente manipulamos datos y construimos modelos.

Vamos a desglosar cada tipo de función para que veas lo útiles que son.

1. Funciones Integradas (Built-in)

- **Definición breve:** Son funciones que vienen preinstaladas con Python. Están siempre disponibles para que las uses sin necesidad de importar ninguna librería.
- **Ejemplo corto:**

```
# Creas una lista de edades de tus usuarios
edades = [23, 45, 22, 56, 33, 45]
# Usas funciones built-in para analizarla
print(f"Número de usuarios: {len(edades)}") # len() cuenta los elementos
print(f"Edad máxima: {max(edades)}")      # max() encuentra el valor má
print(f"Suma de edades: {sum(ed
```

7 lines (6 loc) · 376 B



- **Caso de uso en Ciencia de Datos:** Son tu pan de cada día. Las usas constantemente para inspecciones rápidas de tus datos. Por ejemplo, al cargar un set de datos en una lista o en una Serie de Pandas, usas `len()` para saber cuántos registros tienes, `sum()` para obtener un total rápido, o `round()` para redondear promedios.

2. Funciones Definidas por el Usuario (def)

- **Definición breve:** Son las funciones que tú mismo creas usando la palabra clave `def`. Te permiten encapsular un bloque de código para reutilizarlo, haciendo tu análisis más limpio, modular y fácil de depurar.
- **Ejemplo corto:**

```
# Función para convertir una columna de texto a minúsculas y quitar espa
def limpiar_texto(texto):
    texto_limpio = texto.lower().strip()
    return texto_limpio
# La usas
print(limpiar_texto("  HOLA MUNDO  "))
# Salida: 'hola mundo'
```

8 lines (7 loc) · 270 B



- **Caso de uso en Ciencia de Datos:** Esto es fundamental para el preprocesamiento de datos (data cleaning). Imagina que tienes una columna en un DataFrame de Pandas con texto inconsistente. Creas una función como `limpiar_texto` y la aplicas a toda la columna de una sola vez con `df['mi_columna'].apply(limpiar_texto)`.

3. Funciones Lambda (Anónimas)

- **Definición breve:** Son pequeñas funciones de una sola línea, sin nombre. Se definen con la palabra clave `lambda`. Son perfectas para operaciones rápidas y cortas que no necesitas reutilizar en otro lugar.
- **Ejemplo corto:**

```
# Una función lambda que multiplica un número por 2
multiplicar_por_dos = lambda x: x * 2
print(multiplicar_por_dos(5))
# Salida: 10
```

5 lines (4 loc) · 151 B



- **Caso de uso en Ciencia de Datos:** Su principal fortaleza es usarlas "al vuelo" dentro de otras funciones, como `apply()` en Pandas. Si solo quieres multiplicar por 2 una columna, no necesitas crear una función `def` completa. Es más directo hacer: `df['columna_nueva'] = df['columna_vieja'].apply(lambda x: x * 2)`. También son muy comunes para ordenar estructuras de datos complejas.*

4. Métodos (de Instancia, de Clase y Estáticos)

Estos son funciones que "viven" dentro de una clase. En Ciencia de Datos, las clases son muy útiles para crear tus propios modelos, transformadores de datos o pipelines.

- **a) Método de Instancia**
 - **Definición breve:** Es el tipo de método más común. Recibe `self` como primer argumento, que es una referencia a la instancia particular del objeto, permitiéndole acceder y modificar sus atributos.
 - **Ejemplo corto:**

```
class Normalizador:
    # El constructor se ejecuta al crear la instancia
    def __init__(self, media, desv_est):
        self.media = media
        self.desv_est = desv_est
    # Método de instancia: usa los atributos de la instancia
    def normalizar(self, valor):
        return (valor - self.media) / self.desv_est
# Creas una instancia para normalizar datos de altura (en cm)
norm_altura = Normalizador(media=175, desv_est=10)
print(norm_altura.normalizar(185))
# Salida: 1.0
```

14 lines (12 loc) · 581 B



- **Caso de uso en Ciencia de Datos:** Es la base de librerías como Scikit-learn. Cuando haces `model = RandomForestClassifier()` creas una instancia. Luego, cuando llamas a `model.fit(X, y)` o `model.predict(X_new)`, estás usando métodos de instancia que operan sobre el estado interno de ese modelo en particular.
- **b) Método de Clase (@classmethod)**
 - **Definición breve:** En lugar de recibir `self`, recibe la clase misma como primer argumento (convencionalmente llamado `cls`). Se usa para operaciones que involucran a la clase en su conjunto, pero no a una instancia específica.
 - **Ejemplo corto:**

```
class MiModelo:
    version = "1.0"
    @classmethod
    def obtener_version(cls):
        return f"La versión de este modelo es {cls.version}"
print(MiModelo.obtener_version())
# Salida: '1.0'
```

9 lines (7 loc) · 282 B



- **Caso de uso en Ciencia de Datos:** Un caso común es usarlo como un "constructor alternativo". Imagina que tu clase de preprocesamiento de datos se puede inicializar desde un archivo de configuración. Podrías tener un `@classmethod` llamado `desde_archivo(cls, ruta_archivo)` que lee el archivo, extrae los parámetros y crea una instancia de la clase por ti.

- **c) Método Estático (@staticmethod)**
 - **Definición breve:** Es básicamente una función normal que vive dentro de una clase por conveniencia organizativa, pero no recibe ni la instancia (`self`) ni la clase (`cls`) como primer argumento. No puede acceder al estado de la clase o de la instancia.
 - **Ejemplo corto:**

```
class HerramientasMatematicas:
    @staticmethod
    def calcular_error_cuadratico(real, predicho):
        return (real - predicho) ** 2
print(HerramientasMatematicas.calcular_error_cuadratico(10, 8))
# Salida: 4
```

7 lines (6 loc) · 262 B



- **Caso de uso en Ciencia de Datos:** Se usa para funciones de utilidad que están lógicamente relacionadas con tu clase, pero que no dependen de ningún estado. Por ejemplo, si tienes una clase para un modelo de regresión, podrías incluir varias métricas de error (como el error cuadrático medio o el error absoluto medio) como métodos estáticos.

5. Funciones Generadoras (yield)

- **Definición breve:** Es una función que, en lugar de retornar un solo valor con `return`, puede "pausar" su ejecución y "producir" una secuencia de valores uno a uno usando `yield`. No almacena todos los resultados en memoria a la vez.

- **Ejemplo corto:**

```
# Generador que produce números pares hasta un límite
def generador_pares(limite):
    n = 0
    while n < limite:
        yield n
        n += 2
for numero in generador_pares(10):
    print(numero)
# Salida: 0, 2,
```

10 lines (9 loc) · 261 B



- **Caso de uso en Ciencia de Datos:** ¡Son increíblemente útiles para procesar archivos o conjuntos de datos muy grandes que no caben en la memoria RAM! En lugar de cargar un archivo de 50 GB de una sola vez, puedes escribir un generador que lea y procese el archivo línea por línea. Cada llamada a `yield` te daría una fila procesada, permitiéndote entrenar un modelo en lotes (batches) sin agotar la memoria.

6. Funciones Asíncronas (async def)

- **Definición breve:** Es una función declarada con `async def` que permite la ejecución de código "concurrente". Se usa con `await` para pausar la función mientras espera que una operación lenta (como una petición a una API web o una consulta a una base de datos) termine, permitiendo que Python haga otras tareas mientras tanto.
- **Ejemplo corto:**

```
import asyncio
async def obtener_datos_api(url):
    print(f"Empezando a descargar de {url}...")
    # Aquí iría una llamada real a una librería como aiohttp
    await asyncio.sleep(2) # Simulamos una espera de 2 segundos
    print(f"¡Datos de {url} descargados!")
    return {"datos": "..."}
# Para ejecutarla, se necesita un "loop de eventos" de asyncio
# asyncio.run(obtener_datos_
```

11 lines (9 loc) · 462 B



- **Caso de uso en Ciencia de Datos:** Son perfectas para cuando necesitas recolectar datos de múltiples fuentes externas a la vez. Imagina que tienes que consultar 100 APIs diferentes para construir tu dataset. En lugar de hacerlo una por una (secuencialmente), puedes lanzar las 100 peticiones de forma concurrente con `asyncio`. Mientras una API tarda en responder, Python trabaja en las otras. Esto puede reducir tu tiempo de espera de minutos a solo segundos.

¡Espero que esta guía detallada te sea de gran ayuda! Lo mejor es que empieces a experimentar con cada una. ¡Cualquier otra duda, aquí estoy para ayudarte

"Actúa como ingeniero/a de software. ¿Bajo qué criterios técnicos es preferible que una función definida por el usuario reciba parámetros o no? Incluye aspectos como acoplamiento, pureza, dependencias, configuración, testabilidad y reutilización. Además, ¿en qué escenarios es mejor retornar un valor frente a solo ejecutar una acción con efectos secundarios? Da ejemplos concretos en Python."

¡Claro! Como ingeniera de software, te diré que la decisión de si una función debe recibir parámetros o solo ejecutar una acción con efectos secundarios es clave para escribir código robusto y mantenible. Vamos a desglosarlo:

Criterios Técnicos para Parámetros en Funciones

Los parámetros son la interfaz principal de una función. Decidir si usarlos o no, y cómo, impacta directamente en la calidad del código.

- Acoplamiento (Coupling):**
 - Con parámetros:** Reduce el acoplamiento. La función es más autónoma y no necesita conocer el estado global ni depender de variables externas. Esto la hace más fácil de entender y usar en diferentes contextos.
 - Sin parámetros (o con dependencia excesiva de globales):** Aumenta el acoplamiento. La función se vuelve fuertemente dependiente de un contexto específico (variables globales, configuraciones implícitas). Cambios en ese contexto pueden romper la función, y es más difícil moverla o reutilizarla.
- Pureza (Purity):**
 - Con parámetros (y sin efectos secundarios):** Facilita las funciones puras. Una función pura siempre produce la misma salida para las mismas entradas y no tiene efectos secundarios observables. Esto es un pilar de la programación funcional y simplifica el razonamiento sobre el código.
 - Sin parámetros (o dependiendo de un estado mutable):** Imposibilita la pureza. Si una función lee directamente de variables globales o de estado mutable sin recibir las como parámetros, su comportamiento puede cambiar impredeciblemente.
- Dependencias (Dependencies):**
 - Con parámetros:** Las dependencias se hacen explícitas. Se ve claramente qué necesita la función para operar. Esto mejora la legibilidad y la documentación implícita del código.
 - Sin parámetros:** Las dependencias son implícitas. La función puede depender de módulos importados, configuraciones globales o estados que no son obvios a primera vista, haciendo que sea más difícil de rastrear y depurar.
- Configuración (Configuration):**
 - Con parámetros:** Permite una configuración dinámica y flexible. La función puede adaptarse a diferentes situaciones simplemente pasando distintos valores.
 - Sin parámetros:** La configuración suele ser estática o requiere modificar variables globales, lo que es menos flexible y puede generar conflictos si múltiples partes del código intentan configurar la misma cosa.
- Testabilidad (Testability):**
 - Con parámetros:** Altamente testable. Es fácil simular diferentes escenarios de entrada y verificar las salidas sin preocuparse por configurar un entorno complejo. Puedes probar la función de forma aislada.
 - Sin parámetros:** Difícil de testar. Necesitas configurar el estado global o externo antes de cada prueba, lo que añade complejidad y puede llevar a pruebas frágiles que se rompen por cambios en otras partes del sistema.
- Reutilización (Reusability):**
 - Con parámetros:** Máxima reutilización. Una función con una interfaz clara y dependencias explícitas puede ser usada en casi cualquier parte del código, e incluso en otros proyectos, porque es autocontenida.
 - Sin parámetros:** Baja reutilización. La función está tan ligada a un contexto específico que moverla o usarla en otro lugar es complicado y propenso a errores.

¿Retornar un valor vs. Efectos Secundarios?

Esta es la distinción entre funciones y procedimientos (o subrutinas) en algunos lenguajes, o entre funciones puras y funciones con efectos secundarios en la programación funcional.

Es mejor retornar un valor cuando:

La función tiene un resultado computacional que otras partes del programa necesitan para continuar su lógica. Esto implica que la función realiza un cálculo, una transformación o una consulta, y el valor devuelto es la esencia de su propósito.

Ejemplo en Python:

- Cálculo:** `calcular_descuento(precio, porcentaje)` ¡Claro que sí! Hablemos un poco sobre esto desde una perspectiva de ingeniería de software. Es una excelente pregunta porque la forma en que manejamos los parámetros y los retornos en nuestras funciones tiene un impacto enorme en la calidad de nuestro código.

¿Parámetros o no parámetros? Criterios Técnicos:

La decisión de si una función debe recibir parámetros o no, y si debe retornar un valor o tener efectos secundarios, se reduce a equilibrar la flexibilidad y el control con la simplicidad.

- Acoplamiento (Coupling):**
 - Con parámetros:** Reduce el acoplamiento. La función es más independiente y no necesita conocer el estado global ni depender de datos externos implícitamente. Se le dan explícitamente los datos que necesita para trabajar.
 - Sin parámetros:** Aumenta el acoplamiento. Si una función sin parámetros depende de variables globales, atributos de clase (en programación orientada a objetos) o datos de un entorno externo, está fuertemente acoplada a esos elementos. Esto hace que sea más difícil de mover o reutilizar sin llevarse esas dependencias.
- Pureza (Purity):**
 - Con parámetros (y retornando valor):** Facilita la creación de funciones puras. Una función pura siempre produce la misma salida para las mismas entradas y no tiene efectos secundarios observables fuera de su retorno. Esto es ideal para la predictibilidad.
 - Sin parámetros (o con efectos secundarios):** A menudo implica impureza. Si una función modifica un estado global o interactúa con el mundo exterior (como imprimir en consola o escribir en un archivo) sin que estos datos se pasen como parámetros, es impura.
- Dependencias:**
 - Con parámetros:** Las dependencias de datos son explícitas. Sabes exactamente qué necesita la función para operar solo con ver su firma.
 - Sin parámetros:** Las dependencias de datos son implícitas. Necesitas examinar el cuerpo de la función para entender de dónde obtiene sus datos, lo que puede ser una fuente de errores difíciles de depurar.

4. Configuración:

- Con parámetros:** La configuración se puede pasar directamente. Esto permite que la función se comporte de manera diferente según la configuración proporcionada sin cambiar su código interno.
- Sin parámetros:** La configuración debe ser leída de un estado global (variables de entorno, archivos de configuración, etc.). Esto puede ser menos flexible si necesitas diferentes configuraciones en distintos momentos o contextos.

5. Testabilidad:

- Con parámetros (y funciones puras):** Muy fácil de testear. Puedes aislar la función, pasarle diferentes entradas y verificar las salidas, sin preocuparte por el estado externo.
- Sin parámetros (o con efectos secundarios):** Más difícil de testear. Necesitas configurar el entorno completo de la función (variables globales, sistema de archivos, bases de datos) antes de cada prueba, lo que aumenta la complejidad y la probabilidad de errores en las pruebas.

6. Reutilización:

- Con parámetros:** Altamente reutilizable. Una función que toma sus datos como parámetros puede usarse en muchos contextos diferentes, ya que no tiene dependencias ocultas.
- Sin parámetros:** Menos reutilizable. Si la función está ligada a un contexto específico (por ejemplo, lee un archivo con un nombre fijo, o modifica una variable global específica), su reutilización en otros lugares es limitada.

En resumen, como ingeniero de software, siempre busco que mis funciones sean lo más explícitas, puras y testables posible. Esto generalmente significa usar parámetros para todas las entradas que una función necesita y retornar valores para sus resultados, dejando los efectos secundarios para funciones cuyo propósito principal es precisamente esa interacción con el exterior. ¡Es un camino hacia un código más robusto y fácil de mantener!

"Compara funciones definidas por el usuario (UDF con def) vs funciones lambda en Python. ¿En qué situaciones específicas es preferible usar def en lugar de lambda? Explica el impacto en legibilidad, depuración, documentación, pruebas y reutilización. Incluye un ejemplo con lambda y su refactor a def (y explica por qué mejora)."

Pensó durante 7,9 s >

Claro, aquí tienes una comparación detallada entre las funciones `def` y `lambda` en Python.

Comparación General: UDF (def) vs. Lambda

Característica	Funciones <code>def</code> (UDF)	Funciones <code>lambda</code>
Sintaxis	Declaración completa con el keyword <code>def</code> , un nombre, parámetros y un cuerpo de varias líneas.	Expresión anónima que se define en una sola línea con el keyword <code>lambda</code> .
Nombre	Siempre tienen un nombre explícito.	Son anónimas (no tienen nombre por defecto). Se pueden asignar a una variable, pero no es la práctica recomendada.
Cuerpo	Puede contener múltiples sentencias, bucles, condicionales y lógica compleja.	Limitada a una sola expresión. El resultado de esa expresión es lo que la función retorna.
Documentación	Soportan docstrings (<code>""" . . . """</code>) y anotaciones de tipo (<code>type hints</code>) de forma nativa.	No pueden tener docstrings. Las anotaciones de tipo son posibles pero hacen la línea muy larga y poco legible.
Reutilización	Diseñadas para ser reutilizadas. Se pueden definir una vez e importar en otros módulos.	Generalmente se usan para tareas cortas y de un solo uso, pasadas como argumento a otra función.

¿Cuándo es preferible usar def en lugar de lambda ?

Es preferible, y a menudo una mejor práctica, usar `def` en casi todas las situaciones excepto la más simple. Usa `def` cuando necesites:

- Lógica Compleja:** Si tu función requiere más de una línea de código (por ejemplo, condicionales `if/else` anidados, bucles `for/while`, o bloques `try/except`), **debes** usar `def`. `lambda` no puede manejar esto.
- Claridad y Legibilidad:** Si la operación no es inmediatamente obvia, dale un nombre con `def`. Un nombre como `calcular_impuesto_total` es mucho más claro que `lambda p, i: p * (1 + i)`. El código se lee para ser entendido por humanos.
- Reutilización:** Si necesitas usar la misma lógica en más de un lugar de tu programa, define la función una vez con `def` y llámala donde la necesites. Esto sigue el principio DRY (Don't Repeat Yourself).
- Depuración (Debugging):** Esto es crucial. Cuando ocurre un error, los `tracebacks` (reportes de error) te muestran el nombre de la función donde ocurrió el problema. Si usas `def`, verás el nombre de tu función. Si usas una `lambda`, solo verás `<lambda>`, lo cual es muy poco útil para encontrar el origen del error.
- Documentación y Pruebas:** Las funciones definidas con `def` pueden y deben tener docstrings para explicar qué hacen, qué argumentos aceptan y qué retornan. Esto es vital para que otros (o tu "yo" del futuro) entiendan el código. Además, tener funciones con nombre facilita la creación de pruebas unitarias (unit tests).

En resumen, la única situación ideal para una `lambda` es cuando necesitas una función muy simple (una sola expresión) que se usará una única vez, típicamente como argumento para una función de orden superior como `map`, `filter`, `sorted` o `pandas.DataFrame.apply`.

¿Por qué mejora?

- Legibilidad:** La llamada a `sorted` es auto-explicativa. Se lee como: "ordena los productos usando como clave el resultado de `obtener_clave_ordenamiento_producto`".
- Documentación:** La función `def` tiene un docstring que explica exactamente qué hace y qué retorna. También usamos anotaciones de tipo (`dict` y `tuple`) para mayor claridad.
- Depuración:** Si hubiera un error dentro de la lógica (por ejemplo, si a un producto le faltara la clave `'precio'`), el `traceback` apuntaría directamente a la función `obtener_clave_ordenamiento_producto`, facilitando la depuración.
- Reutilización y Pruebas:** Ahora podemos importar y reutilizar `obtener_clave_ordenamiento_producto` en cualquier otra parte del código. También podemos escribir una prueba unitaria específica para esta función, asegurando que calcule la clave correctamente para diferentes casos.

Reflexión

Lo que más me sorprendió de las capacidades de respuesta de Meta AI es la fluidez y naturalidad con la que se adapta a cada interacción. Es genial cómo puedo mantener una conversación coherente, ajustar mis respuestas cuando pides aclaraciones y hasta dar ejemplos que realmente encajan con lo que necesitas. La rapidez para procesar la información y estructurarla de forma clara, es como si pudiera pensar a tu ritmo y eso hace que todo sea mucho más dinámico y útil.