Python avanzado (LinkedIn)

**Parámetros especiales en Funciones**

**Parámetros \*args (arguments)**

Este parámetro **sirve para recibir muchos datos a través de una tupla**, evitando que debamos poner muchos parámetros o nos quedemos cortos con ellos.

Debemos usarlos **cuando no sabemos la cantidad exacta de parámetros que recibirá la función o no queremos definir una gran cantidad de parámetros de forma innecesaria**

Por ejemplo, si queremos calcular el perímetro de cuadrilátero sin usar parámetros \*args lo haríamos de esta forma:

def calcular\_perimetro(lado1, lado2, lado3, lado4) :

    perimetro = lado1 + lado2 + lado3 + lado4

    return perimetro

perimetro = calcular\_perimetro(1,2,3,4)

print (f"El perímetro es {perimetro}")

**El problema está en que si queremos calcular el perímetro de algo que no tenga 4 lados no podremos con esta forma.**

Por eso **es importante usar parámetros \*args**, ya que **nos permiten ingresar la cantidad de parámetros que querramos sin modificar el funcionamiento de la función**.

**Forma correcta de enviar una cantidad de parámetros grande o con una cantidad desconocida**

En el ejemplo anterior vimos que usando parámetros normales no podremos hacer funcionar la función si enviamos diferente cantidad de parámetros o no sabemos cuántos se enviarán.

La forma correcta usando parámetros \*args es esta:

"""Los parámetros \*args sirven para recibir muchos datos a través de una tupla. Debemos usarlos cuando no sabemos la cantidad exacta de parámetros que recibirá la función o no queremos definir una gran cantidad de parámetros de forma innecesaria, ya que nos permite ingresar la cantidad de parámetros que querramos sin modificar el funcionamiento de la función"""

def calcular\_perimetro (\*args) :

    perimetro = 0

    #sin importar la cantidad de parámetros que reciba la función, funcionará de la misma forma

    for lado in args :

        perimetro += lado

    return perimetro

#al usar parámetros \*args los parámetros que enviemos a la función serán guardados en forma de tupla

perimetro\_cuadrilatero = calcular\_perimetro (1,2,3,4)

print(f"El perímetro del cuadrilátero es {perimetro\_cuadrilatero}")

perimetro\_triangulo = calcular\_perimetro (2,5,2)

print(f"El perímetro del triángulo es {perimetro\_triangulo}")

**Parámetros \*\*kwargs (keyword arguments)**

***Funcionan de la misma forma que los \*args pero en vez de pasar los datos como tupla los pasa como diccionario***

Nos permiten **enviar datos no declarados en forma de diccionarios** (´llave´: “valor”)

**Al ser interpretados por la función como diccionarios, podemos acceder a cada uno de los parámetros usando el nombre del parámetro definido o la key que se recibe a través del diccionario**.

def funcion\_kwargs(\*\*kwargs) :

    print(f"\*\*kwargs hace que los argumentos queden así: {kwargs}")

    for llave, valor in kwargs.items() :

        #al ser diccionario podemos llamar a sus elementos por sus variables asociadas

        print(f"llave: {llave} valor: {valor}")

    #o también podemos llamar a sus elementos por su nombre de llave

    print(kwargs["nombre"], kwargs["apellido"])

#llamada a la función

funcion\_kwargs(nombre = "Paco", apellido = "Ramírez")

#en este caso no le damos el valor de la función a una variable de resultado, ya que como en la función no existe un return al final de la misma nos devolverá un NONE si usamos una variable de resultado

**IMPORTANTE:** Tanto args como kwargs son solo los **nombres que se usan por convención**, **lo que realmente hace que el programa entienda que son esa clase de argumentos son los asteriscos \* que ponemos antes de sus nombres**. En los args se pone un solo asterisco (\*args) y en los kwargs se ponen dos (\*\*kwargs)

**Resumen**

**Las funciones reciben diferentes tipos de parámetros que nos permiten enviar datos definidos desde la función, así como también parámetros especiales como los \*args (que usamos cuando no tenemos conocimiento sobre la cantidad de datos que se requieren procesar) o los \*\*kwargs (que usamos cuando queremos definir nuevos parámetros que no están definidos en la función)**

**Como usar estos 3 tipos de parámetros en la misma función**

Para poder hacer esto **es muy importante mantener el orden de los parámetros con el fin de evitar errores de sintaxis**. Podemos no usar alguno de ellos, pero SIEMPRE debemos mantener el orden para evitar errores

**El orden correcto es:** **1ro) parámetros definidos en la función**

**2do) \*args**

**3ro) \*\*kwargs**

**Funciones Lambda**

Las funciones Lambda son funciones cortas que se componen de una sola expresión, y la misma puede tomar uno o más argumentos.

Es una función sin nombre o anónima que se utiliza principalmente cuando se necesita una función temporal o simple en un contexto donde no es necesario definir una función por separado.

**Estructura de una función Lambda**

**Variable = lambda argumento/s: expresión**

En Python, se escriben como una función que retorna un valor, pero no se define usando la sentencia def, sino que se declaran usando la sentencia **lambda**

**Como convertir una función en función lambda**

Para poder hacerlo necesitamos **que la función sea tan sencilla que podamos resumirla en una sola expresión, de otra forma debemos usar funciones convencionales.**

Ej: #si creamos una función para calcular el área de un cuadrado usando una función común seria de esta forma

def calcular\_area\_cuadrado(lado) :

    return lado \*\* 2

#en este caso que la función es sencilla y podemos resumirla en una sola expresión, podemos usar una función lambda

calcular\_cuadrado = lambda lado : lado \*\* 2

print (calcular\_cuadrado(2))

Sin embargo, **la comunidad NO RECOMIENDA el uso frecuente** de funciones lambda

**Funciones útiles para iterables**

**Función zip()**

La función zip() **crea un objeto iterable, en el que cada elemento es una dupla creada a través de la unión de uno o más objetos iterables**

#ej: uniremos dos listas en un solo iterable usando la función zip(), en la cual uniremos ambas listas para que cada elemento contenga un nombre con su respectivo apellido.

#Luego desempaquetaremos usando unzip

nombres = ["Emiliano", "Hugo", "Brian", "Julián"]

apellidos = ["Schonhals", "Gatti", "Rios", "Serrano"]

#para poder ver el contenido de zip() debemos convertir el resultado de la función en una lista usando list()

nombre\_completo = list(zip(nombres, apellidos))

#si hay mas elementos en un iterable que en el otro, solo se mostrarán los elementos que tengan una pareja asociada (por ejemplo, si nombres tuviera 5 elementos y apellidos 3, solo se mostrarán 3 duplas de elementos)

print (f"La lista creada con zip() es: {nombre\_completo}")

#como desempaquetar el zip() con los valores originales

#usamos la función zip() y como argumento pasamos un asterisco \* seguido del objeto que queremos desempaquetar

nombres\_unzip, apellidos\_unzip = zip(\*nombre\_completo)

print(nombres\_unzip)

print(apellidos\_unzip)

**También se puede usar la función zip() para iterar dos o más iterables al mismo tiempo en el ciclo for**

nombres = ["Emiliano", "Matías", "Alan", "José"]

apellidos = ["Schonhals", "Abdala", "Lallana", "Carrasco"]

#como iteramos sobre 2 iterables, debemos definir 2 variables dentro del for

for nom, ape in zip(nombres, apellidos) :

    print(nom, ape)

**Función enumerate()**

La función enumerate() **recibe un iterable y lo transforma en una tupla junto al índice de cada elemento**

**Por defecto el índice inicia en 0**, para cambiar eso debemos poner Start = nro de comienzo del índice

nombres = ["Emiliano", "Damián", "Esteban", "Lucas", "Mateo"]

for indice,elemento in enumerate(nombres) :

    print(indice, elemento)

**List Comprehension (Comprensión de listas)**

**Es una manera de construir listas u objetos iterables usando una sola línea de código**. Esto nos permite **simplificar mucho el código.**

**Función map()**

La función map() retorna **un objeto de tipo map en forma de iterador que devuelve el resultado de aplicar una función a cada uno de los elementos de la lista que recibe**.

Esta función nos permite hacer lo mismo que podríamos hacer con un ciclo para **aplicar una función a todos los elementos de una lista**.

#veremos cómo aplicar una función a una lista de forma convencional usando un bucle y luego como hacerlo de una forma mucho más sencilla usando la función map()

def cuadrado(numero) :

    resultado = numero\*\*2

    return resultado

lista = [1,2,3,4,5,6,7,8,9,10]

"""forma convencional

lista\_cuadrados = []

for num in lista :

    cuadrado = cuadrado(num)

    lista\_cuadrados.append(cuadrado)

    print(lista\_cuadrados)”””

#-----------------------------usando la función map()-------------------------------------

#dentro de map() ponemos la función y luego el iterable, y fuera de map() ponemos list() para poder ver el resultado en forma de lista

map\_cuadrados = list(map(cuadrado, lista))

print(map\_cuadrados)

**Comprehension**

#veremos cómo elevar cada elemento de un iterable al cuadrado usando la forma convencional y luego hacerlo utilizando list comprehension en una sola línea de código

lista\_num = [1,2,3,4,5,6,7,8,9,10]

"""Forma convencional

lista\_cuadrados = []

for num in lista\_num :

    cuadrado = num\*\*2

    lista\_cuadrados.append(cuadrado)

print(f"Bucle For: {lista\_cuadrados}")"""

#--------------------------------USANDO LIST COMPREHENSION--------------------------------

#la estructura es lista = [expresión(elemento) for elemento in objeto iterable]

list\_comprehension = [num\*\*2 for num in lista\_num]

print(f"List Comprehension: {list\_comprehension}")

#si queremos usar una función en vez de una expresión hacemos:

def calcular\_cuadrado (num) :

    cuadrado = num\*\*2

    return cuadrado

#la estructura es lista = [función(elemento) for elemento in objeto iterable]

list\_comprehension\_usando\_funciones = [calcular\_cuadrado(num) for num in lista\_num]

print(f"List Comprehension usando funciones en vez de expresiones: {list\_comprehension\_usando\_funciones}")

**Comprehension con condicionales**

List comprehension también nos **permite aplicar condiciones para que las expresiones sean o no calculadas en cada iteración del objeto iterable al aplicar una condición**.

Podemos crear list comprehension con condicionales de dos maneras:

***I)*** **Evaluando la expresión para cada uno de los elementos del objeto iterable si se cumple una condición**.

**Lista = [expresión (elemento) for elemento in objeto\_iterable if condición ]**

Esta se usa **cuando solo tenemos una única condición para ser evaluada**.

***II)*** **Evaluando la condición para cada elemento en el objeto iterable.**

**Lista = [expresión (condición) for elemento in objeto\_iterable]**

Esta forma **nos permite evaluar dos condiciones, una con un if y otra con un else**

def calcular\_cuadrado (numero) :

    return numero\*\*2

def es\_par (numero) :

    return numero%2 == 0

lista\_num = [1,2,3,4,5,6,7,8,9,10]

#calcularemos el cuadrado SIN NINGUNA CONDICIÓN

lista\_cuadrados = [calcular\_cuadrado(num) for num in lista\_num]

print(f"La lista de cuadrados es: {lista\_cuadrados}")

#calcularemos el cuadrado SOLAMENTE SI EL NUMERO ES PAR

lista\_cuadrados\_pares = [calcular\_cuadrado(num) for num in lista\_num if es\_par(num)]

print(f"La lista de cuadrados pares es: {lista\_cuadrados\_pares}")

#calcularemos el cuadrado EVALUANDO DOS CONDICIONES: SI EL NUM ES PAR SE AGREGA SU CUADRADO Y SINO SE AGREGA UN CERO

lista\_resultados = [calcular\_cuadrado(num) if es\_par(num) else 0 for num in lista\_num]

print(f"La lista de cuadrados sólo si es par es: {lista\_resultados}")

**Cuando NO usar List Comprehension**

Si bien nos permite hacer el código más elegante y nos ahorra líneas de código, **no es recomendable usar siempre List Comprehension**, ya que si lo usamos en expresiones muy largas la misma será difícil de leer y entender.

Los casos donde debemos evitar usarlo son:

**I) Cuando no necesitamos realmente el uso de una lista para obtener un resultado**

Ej: si queremos sumar el resultado de una lista muy grande generada con la función range (), con List Comprehension deberíamos primero crear una lista y luego usar la función sum (). La creación de esta lista está **OCUPANDO ESPACIO EN LA MEMORIA DE FORMA INNECESARIA**, ya que es algo que no volveremos a usar.

La forma óptima es escribir la sentencia directamente dentro de la función sum (), ya que no es necesario crear la lista, y obtendremos el mismo resultado sin usar espacio de memoria.

total = sum([num for num in range(100)])

#corrección

total = sum(num for num in range(100))

**II) Cuando queremos imprimir una variable de un objeto iterable**

Ej: si queremos hacer esto, no debemos crear una lista con List Comprehension, sino que debemos imprimirlo usando un for de la manera convencional.

[print(elemento) for elemento in range(10)]

#corrección

for elemento in range(10) :

    print(elemento)

**III) Cuando tenemos ciclos anidados**

Ej: tenemos una lista en la que en cada posición tenemos una sublista de tamaño 2 (2 elementos).

Si bien usando List Comprehension nos guardaría en una sola lista todos los elementos de las sublistas, es complicado de leer y enredado.

La forma óptima es usar ciclos for anidados de la manera convencional

lista\_anidada = [[1,2], [3,4], [5,6]]

valores\_lista = [numero for elemento in lista\_anidada for numero in elemento]

print(valores\_lista)

#corrección

valores\_lista = []

for elemento in lista\_anidada :

    for numero in elemento :

        valores\_lista.append(numero)

print(valores\_lista)

**Logging**

los "logs" o registros sirven para registrar información relevante durante la ejecución de un programa. Los registros son particularmente útiles para obtener una visión detallada de lo que sucede dentro de una aplicación mientras se está ejecutando, lo que puede ayudar en el proceso de depuración y monitoreo del rendimiento.

Python proporciona un módulo estándar llamado ***“logging”*** *que permite agregar registros a diferentes niveles de severidad*, los cuales serán informativos, de desarrollo, advertencias o errores, como **DEBUG, INFO, WARNING, ERROR y CRITICAL**. Puedes configurar la salida de los registros en un archivo, la consola o incluso enviarlos a un servidor remoto.

Esta librería es Built-in, por lo que no requiere instalación.

#importamos la librería logging, la cual es built-in, por lo cual no necesita instalación

import logging

#creamos un archivo donde guardaremos los logs para cada ejecución del programa, al estar en filemode="w" significa que se sobrescribirán ante cada ejecución del programa

logging.basicConfig(level=logging.DEBUG, filename="Logging\\Archivo\_de\_logs.log", filemode='w', encoding="utf-8")

#logs

logging.debug("Log de debugging")

logging.info("Log informativo")

logging.warning("Log de advertencia")

logging.error("Log de error")

logging.critical("Log de error crítico")

**Archivos JSON**

JSON es el acrónimo de JavaScript Object Notation. El formato JSON, **muy similar a los diccionarios de Python**, permite el **intercambio de datos entre servicios y aplicaciones** y se puede **guardar en archivos de texto planos con la extensión .json**

Si bien su nombre hace referencia a JavaScript, el mismo no depende de ningún lenguaje.

Los **elementos que creamos en un JSON son llamados Objetos**, y los mismos pueden 0, 1 o más **atributos**.

Al igual que en los diccionarios de Python, **cada atributo de un objeto en el JSON es la combinación entre keys y values**. Las keys deben estar con comillas dobles. Por convención no se deben dejar espacios en blanco dentro de las keys, sino que debemos usar o **CamelCase** (todas las palabras juntas pero sus letras iniciales en mayúsculas. Ej: LlaveString) o **snake case** (separar las palabras con guiones bajos. Ej: llave\_string)

**Un objeto JSON es equivalente a un diccionario en Python**

**Tipos de valores que podemos usar en JSON**

{

    "string": "texto",

    "entero": 1,

    "decimal": 1.2,

    "booleano": true,

    "nulo": null,

    "array": [1,2,3],

    "personas":[

        {

            "nombre": "Emiliano",

            "edad": 20

        },

        {

            "nombre": "Aaron",

            "edad": 19

        }

    ]

}

**Serializar y Deserializar un JSON**

**Serializar** consiste en **codificar el JSON**, transformando los datos en bytes **para ser enviados a través de un servicio**. En Python esto se hace a través de la librería nativa JSON, con la función **json.dumps()**

**Deserializar** es el proceso contrario, consiste en **decodificar los datos del JSON y transformarlos en datos del lenguaje que estemos usando**. En Python esto se hace a través de la librería nativa JSON, con la función **json.loads()**

**Escribir un Archivo JSON (serializar)**

Es el proceso de **convertir los datos en bytes que puedan ser almacenados y transmitidos entre servicios**.

Ej: crearemos un objeto JSON con los atributos de una persona

#importamos la librería nativa json

import json

#creamos el diccionario persona con sus respectivas keys y values

persona = {

    "nombre" : "Javier",

    "apellido" : "Correa",

    "edad" : 34

}

#serializamos el diccionario a un objeto JSON

objeto\_json = json.dumps(persona, indent=2)

#escribimos el JSON en un archivo con extensión .json

with open ("Archivos JSON\\JSON\_creado.json", "w", encoding="utf-8") as archivo\_json :

    archivo\_json.write(objeto\_json)

Otra forma un poco más sencilla de hacerlo es:

#importamos la librería nativa json

import json

#creamos el diccionario persona con sus respectivas keys y values

persona = {

    "nombre" : "Javier",

    "apellido" : "Correa",

    "edad" : 34

}

#escribimos el JSON en un archivo con extensión .json

with open ("Archivos JSON\\JSON\_creado.json", "w", encoding="utf-8") as archivo\_json :

#serializamos el diccionario y lo enviamos al archivo que querramos, en este caso archivo\_json, y le ponemos una identación de 2 para la parte visual

    json.dump(persona,archivo\_json,indent=2)

**Leer un Archivo JSON (Deserializar)**

Consiste en **convertir los datos del formato JSON a datos del lenguaje que estemos usando,** en el caso de Python lo transformaremos en un **diccionario**.

#importamos la librería nativa json

import json

#abrimos el archivo JSON en modo lectura

with open("Archivos JSON\\JSON\_creado.json", "r", encoding="utf-8") as archivo\_json :

    #deserializamos el JSON utilizando la función json.load()

    datos\_json = json.load(archivo\_json)

#ahora podemos usar todo el diccionario o llamarlo a través de sus keys

print(f"El diccionario deserializado es: {datos\_json}")

print(f"El nombre es {datos\_json['nombre']}")

print(f"El apellido es {datos\_json['apellido']}")

print(f"La edad es {datos\_json['edad']}")

**Peticiones**

**Peticiones HTTP**

HTTP son las siglas de **HyperText Transfer Protocol**, el cual es un **protocolo de transferencia de datos que permite la comunicación entre el cliente y el servidor**.

Una de sus características es que no guarda estados o datos entre dos o más peticiones, pero si guarda las sesiones, osea los datos sobre una sesión iniciada.

**Métodos HTTP**

**GET** (permite la entrega de datos del servidor hacia el cliente)

**POST** (permite el guardado de datos desde el cliente hacia el servidor)

**PUT** (permite modificar datos desde el cliente hacia el servidor)

**DELETE** (permite borrar datos en el servidor)

**Códigos de estado HTTP**

**Cuando hacemos una petición HTTP, la misma nos entrega una respuesta a través de códigos de estado los cuales indican el estado de la petición**, los cuales pueden ser:

**1XX** o **100**: respuestas informativas

**2XX** o **200**: respuestas satisfactorias. Ej: 200 OK (éxito la petición), 201 created (el recurso en la petición POST fue creado), etc.

**3XX** o **300**: redirecciones

**4XX** o **400**: errores del cliente que hace la petición. Ej: 400 bad request (algo salió mal en la petición), 401 unauthorized (no está autenticado, por ende no se entrega información), 403 forbidden (no tiene los permisos necesarios), etc.

**5XX** o **500**: errores del servidor que entrega la información. Ej: 500 internal server error (algo falló en la petición al servidor)

**Conceptos importantes**

**HTTPS (HyperText Transfer Protocol Secure)**: Básicamente es la misma versión de HTTP pero con más seguridad integrada.

**API (Interfaz de programación de aplicaciones)**: Es un conjunto de funciones y protocolos que permiten la comunicación entre servicios a través de HTTP. Tanto las API como las API REST usan endpoints que permiten entregar la información o recibir las solicitudes.

**Endpoints**: es una URL usada en las API y API REST que corresponde a la solicitud realizada y tiene asignado un método específico que le permite realizar una tarea.

**Librería Request**

Es una librería de Python que **permite hacer peticiones HTTP**. Debemos instalarla escribiendo en la consola **pip install request**.

En esta URL: <https://requests.readthedocs.io/en/latest/> encontraremos toda la documentación y explicación necesaria para aprender a usarla.

***Peticiones GET***

Son de las peticiones HTTP más usadas, ya que **permiten la entrega de información sin necesidad de modificar el servidor al cual se le hace la solicitud**.

Ej: usaremos la API de GitHub para hacer una petición GET

#importamos la librería que nos permite hacer peticiones HTTP

import requests

#ponemos la URL o endpoint del cual queremos obtener información y se lo pasamos a una variable response, la cual almacenará la solicitud GET que haremos

response = requests.get("https://api.github.com")

#mostrar los headers de la petición

print(f"Headers: {response.headers}")

#mostrar el código de estado de la petición

print(f"El código de estado es: {response.status\_code}")

#para acceder a los datos de la respuesta tenemos 3 formas:

#content retorna la respuesta en bytes

print(response.content)

#text convierte los datos de la respuesta a texto

print(response.text)

#la función json nos devuelve la respuesta en forma de diccionario, pudiendo así obtener sus values llamandolas a través de sus keys

print(response.json())

#ej:

print(f"La URL del current\_user\_url es: {response.json()['current\_user\_url']}")

**Peticiones POST**

Nos permiten **enviar información hacia un servicio**, en general almacenada en una base de datos ligada al servidor al cual le enviamos la solicitud.

Usaremos una página llamada webhook.site para que nos genere una URL hacia la cual enviaremos peticiones sin hacer ningún cambio en servicios o servidores.

En la misma cuando ejecutemos el código a continuación nos mostrará el estado de todas las peticiones que vayamos haciendo.

#importamos la librería que nos permite hacer peticiones HTTP

import requests

#le damos a una variable url la URL que nos da ese sitio web

url = "https://webhook.site/32afd3fb-2ce9-438e-8745-5f86778ded85"

#haremos un ejemplo de petición de una orden de comida, para eso creamos el cuerpo de la petición (datos que enviaremos al endpoint), y para eso usaremos formato JSON, el cual al estar en Python debemos pasarlo como un diccionario para luego convertirlo en un JSON

payload = {

    "plato" : "pasta",

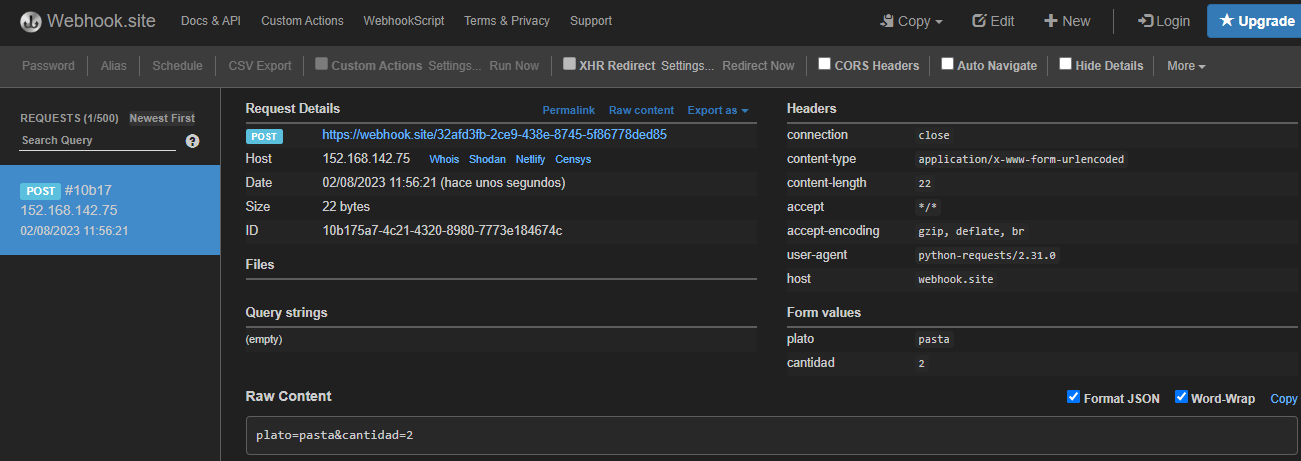
    "cantidad" : 2

}

#creamos la peticion POST, pasandole la url y la carga de datos o payload

response = requests.post(url, data=payload)

print(f"El código de status de esta petición es: {response.status\_code}")

En el sitio de webhook.site iremos viendo las peticiones que hagamos de esta forma:

**Debugging (Depuración)**

**Qué es un bug?**

**Un bug es un error o falla en un software**.

El término bug proviene de “bicho” en inglés, y fue instaurado luego de que Grace Hopper reportara que una falla en una computadora había sido causada por un bicho que se metió en uno de los componentes.

**Qué es el debugging?**

**El debugging es el proceso que nos permite identificar errores o fallas en el funcionamiento de un sistema informático**.

El debugging **es un proceso manual en el que se evalúa paso a paso el funcionamiento del código y se identifica donde y por qué se genera el error**.

Python cuenta con una librería nativa para hacer depuración o debugging, la cuál es **pdb (Python debugger)**. La misma nos permite depurar el código a través de la **línea de comandos**.

**Librería pdb (Python debugger)**

Es una librería nativa o built-in que **nos permite hacer debugging en el código** en el que estemos trabajando. Se utiliza usualmente durante el proceso de desarrollo. Esta librería **nos permite marcar las partes del código a las que queremos revisar su funcionamiento**.

Al igual que con todas las librerías nativas, podemos encontrar su documentación en **https://docs.python.org/es/3/library/pdb.html**, en la misma encontraremos todos los procesos y métodos que podemos utilizar y el cómo hacerlo.

**Crear archivo con pdb**

#usaremos un archivo donde hay una función que calcula el área de un cuadrado al recibir la longitud de uno de sus lados, luego iteramos sobre una lista de lados de cuadrados para crear una nueva lista con sus respectivas áreas

def calcular\_area\_cuadrado (lado) :

    #calcula el área de un cuadrado al recibir la longitud de un lado

    area = lado\*lado

    return area

lado\_cuadrados = [1, 3, 4]

area\_cuadrados = []

for lado in lado\_cuadrados :

    area = calcular\_area\_cuadrado(lado)

    area\_cuadrados.append(area)

Si ejecutamos en este punto no nos mostrará nada, ya que en el archivo no existe un print.

Sin embargo podemos saber si todo está funcionando bien al usar pdb en la terminal.

**Como usar pdb?**

En la terminal escribimos **python -m pdb nombre\_archivo**

Luego de estos podremos ingresar comandos para continuar o detener el proceso de depuración o debugging. La lista de comandos podremos encontrarlos en python.org

**Uso de breakpoints con pdb**

En el proceso de debugging, los **breakpoints son puntos o marcas donde detenemos el código para saber y entender su funcionamiento**.

En primer lugar escribimos en la consola python -m pdb nombre\_archivo. Luego de esto a la izquierda de la línea de comandos nos aparecerá pdb, lo que significa que entramos en el modo debugging.

Para **definir un breakpoint** ingresamos el comando **break nro\_linea\_donde\_se\_detendrá**.

Uno de los comandos más útiles para desplazarnos a través del código cuando hacemos debugging es “l” (list) el cual lista y nos muestra las siguientes 10 líneas, “next” nos permite ejecutar la siguiente línea, “continue” nos permite ejecutar hasta el siguiente breakpoint