Curso de Python Avanzado

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Bienvenida

Bienvenidos al curso de **Python Avanzado** para Scotiabank, brindado por Belatrix.

En este curso aprenderás a desarrollar una programación avanzada con Python.

El curso está dirigido a profesionales relacionados al área de desarrollo, soporte y análisis de datos.

Se requieren conocimientos previos de Python Intermedio para poder cubrir satisfactoriamente este curso.

Temario

Módulo 1: Programación Funcional en Python

1. Introducción a la programación funcional
2. Funciones de orden superior
3. Lambdas y expresiones generadoras

Módulo 2: Diseño de Patrones Avanzados

1. Patrones de diseño comunes en Python
2. Aplicación práctica de patrones en el desarrollo de software

Módulo 3: Manipulación Eficiente de Datos

1. Uso avanzado de NumPy y Pandas
2. Operaciones vectorizadas y broadcasting

Módulo 4: Concurrencia y Paralelismo en Python

1. Hilos y procesos en Python
2. Multiprocessing y Asyncio

Módulo 5: Optimización de Código

1. Estrategias para mejorar el rendimiento
2. Perfilado de código y herramientas de optimización

Módulo 6: Seguridad en Desarrollo Python

1. Principios básicos de seguridad
2. Mejores prácticas de codificación segura

Módulo 7: Despliegue y Escalabilidad

1. Configuración de entornos de producción
2. Estrategias para escalabilidad horizontal y vertical

Módulo 6: Seguridad en Desarrollo Python

1. Principios básicos de seguridad
2. Mejores prácticas de codificación segura

1. Principios básicos de seguridad

La seguridad en el desarrollo de software es esencial para evitar vulnerabilidades que puedan ser explotadas maliciosamente. Aquí algunos principios básicos de seguridad en Python:

- **Principio de menor privilegio**: Cada módulo, usuario o proceso del sistema debe operar usando el conjunto más limitado de privilegios necesario para completar el trabajo. Esto minimiza las vías de ataque disponibles si ocurre una brecha de seguridad.
- **Defensa en profundidad**: Aplica múltiples capas de seguridad para proteger el software. Si una capa es vulnerada, las adicionales están en su lugar para mantener la protección.
- **Seguridad desde el diseño**: Incorpora la seguridad en todas las etapas del desarrollo del software, no solo en las fases finales o después de que se han encontrado problemas.

- **Principio de fallo seguro**: Configura tus sistemas para que, en caso de un fallo, se cierren o restrinjan el acceso en lugar de permitirlo. Por ejemplo, un sistema de autenticación debería fallar a no permitir el acceso si algo sale mal.
- **Validación de entrada**: Nunca confíes en los datos de entrada. Valida y sanitiza todas las entradas para evitar inyecciones SQL, ataques de scripts entre sitios (XSS), y otros vectores de ataque.

2. Mejores prácticas de codificación segura

Implementar buenas prácticas de codificación es crucial para la seguridad. Algunas de las más importantes en Python son:

- **Uso de bibliotecas actualizadas y seguras**: Asegúrate de utilizar versiones de bibliotecas que sean soportadas y reciban actualizaciones de seguridad regularmente. Utiliza herramientas como `pip-audit` para identificar vulnerabilidades conocidas en las dependencias.
- **Manejo de excepciones robusto**: Captura las excepciones de manera específica y controla los errores de forma segura para evitar revelar detalles sensibles del sistema interno a través de mensajes de error.
- **Seguridad en la gestión de sesiones**: Utiliza tokens de sesión seguros y mecanismos de almacenamiento que defiendan contra robos de sesión, como HttpOnly y Secure en cookies.
- **Cifrado de datos sensibles**: Usa bibliotecas criptográficas fuertes para cifrar datos sensibles en reposo y en tránsito. Python ofrece bibliotecas como `cryptography` para manejar cifrado y descifrado de manera segura.
- **No hardcodificar credenciales**: Nunca almacenes credenciales directamente en el código. Utiliza variables de entorno o servicios de gestión de secretos para almacenarlos de manera segura.
- **Pruebas de seguridad**: Incorpora pruebas de seguridad en el proceso de desarrollo. Herramientas como Bandit o PyLint pueden ayudar a identificar problemas de seguridad en el código.

Implementar estos principios y prácticas no solo mejora la seguridad de tus aplicaciones, sino también su calidad y confiabilidad general. ¿Hay algún aspecto específico de la seguridad en Python sobre el que te gustaría profundizar más?

Casos prácticos de seguridad en Python

Cifrado de archivos con AES

Para cifrar un archivo utilizando AES-256 en Python, necesitas utilizar una biblioteca de criptografía que soporte AES. Una de las bibliotecas más populares y robustas para esto es `cryptography`. Aquí se muestra cómo puedes hacerlo paso a paso.

Pasos para cifrar un archivo con AES-256

1. **Instalar la biblioteca `cryptography`**: Primero, necesitas instalar la biblioteca si aún no lo has hecho. Puedes instalarla utilizando pip:

   ```
   pip install cryptography
   ```

2. **Escribir el código para cifrar el archivo**: Vamos a utilizar el modo AES GCM para el cifrado, ya que proporciona tanto confidencialidad como autenticidad de los datos.

   A continuación, te muestro un ejemplo de cómo podrías cifrar un archivo:

   ```
   from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
   from cryptography.hazmat.primitives import hashes
   from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
   from cryptography.hazmat.backends import default_backend
   from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
   from cryptography.hazmat.primitives.ciphers.aead import AESGCM
   import os

   def encrypt_file(input_file_path, output_file_path, password):
       # Generar una sal segura
       salt = os.urandom(16)

       # Derivar la clave usando PBKDF2HMAC
       kdf = PBKDF2HMAC(
           algorithm=hashes.SHA256(),
           length=32,
           salt=salt,
           iterations=100000,
           backend=default_backend()
       )
       key = kdf.derive(password.encode())

       # Leer los datos del archivo
       with open(input_file_path, 'rb') as file:
   ```

```
        data = file.read()

    # Cifrar los datos
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)  # GCM necesita un nonce de 12 bytes
    encrypted_data = aesgcm.encrypt(nonce, data, None)

    # Escribir los datos cifrados en un archivo, incluyendo el nonce y la sal
    with open(output_file_path, 'wb') as file:
        file.write(salt + nonce + encrypted_data)

# Uso de la función
password = 'tu_contraseña_segura'
encrypt_file('path_a_tu_archivo_original', 'path_a_tu_archivo_cifrado', password)
```

En este código:

- Se genera una sal aleatoria que se utiliza para derivar la clave a partir de la contraseña usando PBKDF2HMAC.
- Se utiliza AES en modo GCM para cifrar los datos, lo cual es recomendado porque ofrece autenticidad además de cifrado.
- Se guarda la sal y el nonce junto con los datos cifrados para que puedan ser usados en el proceso de descifrado.

Consideraciones

- **Seguridad de la contraseña**: Asegúrate de utilizar una contraseña fuerte, ya que una contraseña débil puede ser vulnerable a ataques de fuerza bruta.
- **Manejo de la sal y nonce**: Guarda la sal y el nonce de forma segura y junto con el mensaje cifrado, como en el ejemplo, ya que son necesarios para el descifrado.
- **Copias de seguridad**: Asegúrate de tener copias de seguridad de la información necesaria para descifrar tus datos.

Este código proporciona una base sólida para cifrar archivos con AES-256 en Python de manera segura. ¿Necesitas ayuda con algo más, como el código para descifrar el archivo, o algún otro detalle?

```
import os

os.urandom(16)
```

```
b'\xae\xc36\xcd\xee\xb6\xf3N\x93X?\xb2\x7f \xbf\xd7'
```

```
with open("/content/candado.jpg", "rb") as inputFile:
  dataInput = inputFile.read()

print("Bytes:", len(dataInput))
```

```
Bytes: 14993
```

```
with open("/content/candado-copia.jpg", "wb") as outputFile:
  outputFile.write(dataInput)

print("Archivo copiado")
```

```
Archivo copiado
```

```
leftSize = int(len(dataInput) / 2)
rightSize = len(dataInput) - leftSize

with open("/content/candado-p1.jpg", "wb") as outputFile:
  outputFile.write(dataInput[:leftSize])

with open("/content/candado-p2.jpg", "wb") as outputFile:
  outputFile.write(dataInput[leftSize:])

print("Se guardó candado-p1.jpg con", leftSize, "bytes")
print("Se guardó candado-p2.jpg con", rightSize, "bytes")
```

```
Se guardó candado-p1.jpg con 7496 bytes
Se guardó candado-p2.jpg con 7497 bytes
```

```
with open("/content/candado-p1.jpg", "rb") as inputFile:
  dataInputLeft = inputFile.read()

with open("/content/candado-p2.jpg", "rb") as inputFile:
  dataInputRight = inputFile.read()

data = dataInputLeft + dataInputRight

with open("/content/candado-reconstruido.jpg", "wb") as outputFile:
  outputFile.write(data)

print("Se reconstruyó el archivo con", len(data), "bytes")
```

```
Se reconstruyó el archivo con 14993 bytes
```

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import os

def encrypt_file(input_file_path, output_file_path, password):
    # Generar una sal segura
    salt = os.urandom(16)

    # Derivar la clave usando PBKDF2HMAC
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    key = kdf.derive(password.encode())

    # Leer los datos del archivo
    with open(input_file_path, 'rb') as file:
        data = file.read()

    # Cifrar los datos
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)  # GCM necesita un nonce de 12 bytes
    encrypted_data = aesgcm.encrypt(nonce, data, None)

    # Escribir los datos cifrados en un archivo, incluyendo el nonce y la sal
    with open(output_file_path, 'wb') as file:
        file.write(salt + nonce + encrypted_data)
```

```
encrypt_file("/content/candado.jpg", "candado-cifrado.jpg", "HolaMundo$123")
```

Descifrado de archivos con AES

Para descifrar un archivo cifrado con AES en modo GCM utilizando Python, necesitas tener acceso a la misma clave utilizada para el cifrado, así como al nonce y a los datos cifrados, los cuales deben haber sido almacenados o transmitidos de manera segura. El proceso de descifrado debe también verificar la integridad de los datos antes de considerar que el descifrado fue exitoso.

Veamos cómo implementar el descifrado para el ejemplo de cifrado que se proporcionó anteriormente utilizando AES en modo GCM:

Código para descifrar un archivo cifrado con AES-GCM

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import os

def decrypt_file(encrypted_file_path, output_file_path, password):
    # Leer los datos cifrados desde el archivo
    with open(encrypted_file_path, 'rb') as file:
        file_contents = file.read()

    # Extraer la sal, el nonce y los datos cifrados
    salt = file_contents[:16]
    nonce = file_contents[16:28]
    encrypted_data = file_contents[28:]

    # Derivar la clave usando la misma sal y contraseña
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    key = kdf.derive(password.encode())

    # Descifrar los datos
    aesgcm = AESGCM(key)
    decrypted_data = aesgcm.decrypt(nonce, encrypted_data, None)

    # Escribir los datos descifrados en un archivo
    with open(output_file_path, 'wb') as file:
        file.write(decrypted_data)
```

```
# Uso de la función
password = 'tu_contraseña_segura'
decrypt_file('path_a_tu_archivo_cifrado', 'path_a_tu_archivo_descifrado', password)
```

Explicación del código

- **Lectura de archivo cifrado**: Se leen los datos cifrados del archivo, incluyendo la sal, el nonce y el texto cifrado propiamente dicho.
- **Extracción de componentes**: La sal y el nonce se extraen de la parte inicial del archivo cifrado. La sal y el nonce deben ser los mismos que se usaron para cifrar los datos.
- **Derivación de la clave**: Se utiliza el mismo método y parámetros de derivación de clave (KDF) que se usaron durante el cifrado para generar la misma clave a partir de la contraseña.
- **Descifrado**: Utilizando `AESGCM`, los datos se descifran verificando también la autenticidad e integridad del mensaje. Si la verificación falla (por ejemplo, debido a una modificación de los datos cifrados), se lanzará una excepción.
- **Almacenamiento de los datos descifrados**: Finalmente, los datos descifrados se escriben en un nuevo archivo.

Consideraciones

- **Seguridad**: Si se modifica cualquier parte del nonce, de la sal o de los datos cifrados, el proceso de descifrado fallará, lanzando una excepción, dado que GCM verifica la integridad de los datos.
- **Manejo de errores**: Es importante implementar un adecuado manejo de errores para gestionar los casos en los que el descifrado falle debido a una clave incorrecta o modificaciones en los datos.

Este código proporciona un método robusto para descifrar archivos cifrados con AES-GCM, asegurando tanto la confidencialidad como la integridad de los datos. ¿Hay algo más en lo que te gustaría profundizar o alguna otra pregunta que tengas sobre la criptografía en Python?

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import os

def decrypt_file(encrypted_file_path, output_file_path, password):
    # Leer los datos cifrados desde el archivo
    with open(encrypted_file_path, 'rb') as file:
        file_contents = file.read()

    # Extraer la sal, el nonce y los datos cifrados
    salt = file_contents[:16]
    nonce = file_contents[16:28]
    encrypted_data = file_contents[28:]

    # Derivar la clave usando la misma sal y contraseña
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    key = kdf.derive(password.encode())

    # Descifrar los datos
    aesgcm = AESGCM(key)
    decrypted_data = aesgcm.decrypt(nonce, encrypted_data, None)

    # Escribir los datos descifrados en un archivo
    with open(output_file_path, 'wb') as file:
        file.write(decrypted_data)
```

```
decrypt_file("/content/candado-cifrado.jpg", "/content/candado-descifrado.jpg", "HolaMundo$123")
```

Crifrado y Descifrado de archivos con RSA

Para crear un par de claves pública y privada en Python y utilizarlas para cifrar y descifrar un archivo, puedes utilizar la biblioteca `cryptography`, que proporciona herramientas robustas para manejar cifrado asimétrico. A continuación, veremos paso a paso en el proceso utilizando RSA, uno de los algoritmos de cifrado asimétrico más comunes.

Paso 1: Instalar la biblioteca `cryptography`

Si aún no tienes instalada la biblioteca `cryptography`, puedes instalarla usando pip:

```
pip install cryptography
```

Paso 2: Crear el par de claves (pública y privada)

Primero, vamos a generar el par de claves RSA:

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

def generate_keys():
    # Generar la clave privada
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
    )

    # Generar la clave pública a partir de la clave privada
    public_key = private_key.public_key()

    # Serializar la clave privada
    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    # Serializar la clave pública
    public_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    return private_pem, public_pem

private_pem, public_pem = generate_keys()

# Guardar las claves en archivos
with open('private_key.pem', 'wb') as f:
    f.write(private_pem)

with open('public_key.pem', 'wb') as f:
    f.write(public_pem)
```

Paso 3: Cifrar un archivo utilizando la clave pública

Para cifrar datos con RSA, generalmente se utiliza la clave pública:

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import serialization

def encrypt_file(input_file_path, output_file_path, public_key_path):
    # Cargar la clave pública
    with open(public_key_path, 'rb') as key_file:
        public_key = serialization.load_pem_public_key(
            key_file.read(),
        )

    # Leer los datos del archivo
    with open(input_file_path, 'rb') as file:
        plaintext = file.read()

    # Cifrar los datos
    ciphertext = public_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # Guardar los datos cifrados
    with open(output_file_path, 'wb') as file:
        file.write(ciphertext)

encrypt_file('path_to_your_plaintext_file', 'encrypted_file_path', 'public_key.pem')
```

Paso 4: Descifrar el archivo utilizando la clave privada

Para descifrar los datos cifrados, utilizamos la clave privada:

```
def decrypt_file(encrypted_file_path, output_file_path, private_key_path):
    # Cargar la clave privada
    with open(private_key_path, 'rb') as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
        )

    # Leer los datos cifrados
    with open(encrypted_file_path, 'rb') as file:
        ciphertext = file.read()

    # Descifrar los datos
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # Guardar los datos descifrados
    with open(output_file_path, 'wb') as file:
        file.write(plaintext)

decrypt_file('encrypted_file_path', 'path_to_decrypted_output_file', 'private_key.pem')
```

Este proceso te permite manejar cifrado y descifrado de manera segura y efectiva utilizando claves públicas y privadas. Es importante tener en cuenta que RSA es adecuado para cifrar pequeñas cantidades de datos debido a las limitaciones de tamaño de bloque del algoritmo. Si necesitas cifrar archivos grandes, es común usar RSA para cifrar una clave simétrica, y luego usar esa clave para cifrar los datos utilizando un cifrado simétrico como AES.

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

def generate_keys():
    # Generar la clave privada
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
    )

    # Generar la clave pública a partir de la clave privada
    public_key = private_key.public_key()

    # Serializar la clave privada
    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    # Serializar la clave pública
    public_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    return private_pem, public_pem
```

```
priv_pem, pub_pem = generate_keys()

print("La llave privada es:")
print(priv_pem)

print("La llave pública es:")
print(pub_pem)
```

```
La llave privada es:
b'-----BEGIN PRIVATE KEY-----\nMIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQCorjV7b0wmcK/D\nILPl8mktxdgOkGnCs0plIDBKyFNv3Nrf6exIogcM91BK5ZvmvO
La llave pública es:
b'-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqK41e29MJnCvwyCz5fJp\nLcXYDpBpwrNKZSAwSshTb9za3+nsSKIHDPdQSuWb5rzgxYEnQlT
```

```
with open("user1_priv_key.pem", "wb") as outputFile:
  outputFile.write(priv_pem)

with open("user1_pub_key.pem", "wb") as outputFile:
  outputFile.write(pub_pem)
```

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import serialization

def encrypt_file_with_public_key(input_file_path, output_file_path, public_key_path):
    # Cargar la clave pública
    with open(public_key_path, 'rb') as key_file:
        public_key = serialization.load_pem_public_key(
            key_file.read(),
        )

    # Leer los datos del archivo
    with open(input_file_path, 'rb') as file:
        plaintext = file.read()

    # Cifrar los datos
    ciphertext = public_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # Guardar los datos cifrados
    with open(output_file_path, 'wb') as file:
        file.write(ciphertext)
```

```
def decrypt_file_with_private_key(encrypted_file_path, output_file_path, private_key_path):
    # Cargar la clave privada
    with open(private_key_path, 'rb') as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
        )

    # Leer los datos cifrados
    with open(encrypted_file_path, 'rb') as file:
        ciphertext = file.read()

    # Descifrar los datos
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # Guardar los datos descifrados
    with open(output_file_path, 'wb') as file:
        file.write(plaintext)
```

```
encrypt_file_with_public_key("/content/demo_datos_confidenciales.txt", "/content/demo_datos_confidenciales_enc.txt", "/content/user1_pub_key.pem")
```

```
decrypt_file_with_private_key("/content/demo_datos_confidenciales_enc.txt", "/content/demo_datos_confidenciales_dec.txt", "/content/user1_priv_key.pem"
```

Uso de Web Tokens (JWT)

Los JSON Web Tokens (JWT) son una forma estándar de pasar información de manera segura entre dos partes en forma de objeto JSON. Un JWT típicamente consiste en tres partes: el encabezado, el cuerpo (payload) y la firma.

- **Encabezado**: Contiene metadatos sobre el token, generalmente incluyendo el tipo de token (`typ`) y el algoritmo de hashing usado (`alg`).
- **Cuerpo (Payload)**: Contiene las declaraciones (claims) del token, que son los datos almacenados dentro del JWT. Esto puede incluir información del usuario como el nombre y correo electrónico, así como datos de expiración y emisión.
- **Firma**: Se utiliza para verificar que el mensaje no ha sido alterado. Se genera tomando el encabezado codificado, el cuerpo codificado, una clave secreta y el algoritmo especificado en el encabezado.

Para trabajar con JWT en Python, una de las bibliotecas más populares es `PyJWT`. A continuación, veremos cómo instalar esta biblioteca y crear un ejemplo sencillo para generar un JWT y luego verificarlo para extraer la información contenida en él.

Paso 1: Instalar PyJWT

Primero, necesitas instalar la biblioteca `PyJWT`:

```
pip install PyJWT
```

Paso 2: Crear y verificar un JWT

Este un ejemplo de cómo generar un JWT y luego validar ese token para obtener los metadatos almacenados:

```
import jwt
import datetime

# Función para generar un JWT
def create_jwt():
    payload = {
        "name": "Juan Pérez",
        "email": "juan.perez@example.com",
        "exp": datetime.datetime.utcnow() + datetime.timedelta(seconds=600),  # Expira en 10 minutos
        "iat": datetime.datetime.utcnow()  # Tiempo de emisión
    }
    secret_key = "tu_clave_secreta_super_secreta"
    encoded_jwt = jwt.encode(payload, secret_key, algorithm="HS256")
    return encoded_jwt

# Función para verificar y decodificar un JWT
def verify_jwt(encoded_jwt):
    secret_key = "tu_clave_secreta_super_secreta"
    try:
        # Decodificar el JWT
        decoded_jwt = jwt.decode(encoded_jwt, secret_key, algorithms=["HS256"])
        return decoded_jwt
    except jwt.ExpiredSignatureError:
        return "Token expirado"
    except jwt.InvalidTokenError:
        return "Token inválido"

# Crear un JWT
token = create_jwt()
print("JWT Creado:", token)

# Verificar el JWT
decoded_info = verify_jwt(token)
print("Información Decodificada:", decoded_info)
```

Explicación del código

1. **Generación del JWT**: La función `create_jwt` crea un token con un payload que contiene el nombre y el correo del usuario, además de la fecha de expiración y emisión. Utiliza una clave secreta para firmar el token.
2. **Validación del JWT**: La función `verify_jwt` verifica y decodifica el token. Maneja dos tipos de errores comunes: si el token ha expirado y si el token es inválido por cualquier otra razón.

Consideraciones de seguridad

- **Mantener la clave secreta segura**: Es fundamental que la clave secreta no se exponga y se mantenga protegida para prevenir que terceros creen tokens falsos.
- **Validación de errores**: Al decodificar un JWT, es importante manejar adecuadamente los errores para prevenir vulnerabilidades y malas prácticas de manejo de errores.

Este ejemplo muestra cómo puedes trabajar con JWT en Python para manejar autenticación simple sin necesidad de implementar un servidor completo. ¿Te gustaría explorar algún otro aspecto de JWT o tienes alguna otra pregunta?

```
!pip install PyJWT
```

```
Requirement already satisfied: PyJWT in /usr/lib/python3/dist-packages (2.3.0)
```

```
import jwt
import datetime

# Función para generar un JWT
def create_jwt(payload, seconds):
    payload["exp"] = datetime.datetime.utcnow() + datetime.timedelta(seconds=seconds)
    payload["iat"] = datetime.datetime.utcnow()

    secret_key = "MI_EMPRESA_MASTER_PASSWORD_SEGURO$123"

    encoded_jwt = jwt.encode(payload, secret_key, algorithm="HS256")

    return encoded_jwt
```

```
token = create_jwt({ "usuario": "Alan", "correo": "alan@belatrix.com" }, 20)

print("Token:", token)
```

```
Token: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiQWxhbiIsImNvcnJlbyI6ImFsYW5AYmVsYXRyaXguY29tIiwiZXhwIjoxNzEzOTE4MzY4LCJpYXQiOjE3MTM5MTg
```

```
def verify_jwt(encoded_jwt):
    secret_key = "MI_EMPRESA_MASTER_PASSWORD_SEGURO$123"
    try:
        # Decodificar el JWT
        decoded_jwt = jwt.decode(encoded_jwt, secret_key, algorithms=["HS256"])
        return decoded_jwt
    except jwt.ExpiredSignatureError:
        return "Token expirado"
    except jwt.InvalidTokenError:
        return "Token inválido"
```

```
verify_jwt("eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiQWxhbiIsImNvcnJlbyI6ImFsYW5AYmVsYXRyaXguY29tIiwiZXhwIjoxNzEzOTE4MzY4LCJpYXQiOjE3MTM5MT
```

```
'Token expirado'
```

Interfaces gráficas con IPython Widgets

`ipywidgets`, también conocido como "IPython widgets", ofrece una variedad de controles que pueden ser utilizados para crear interfaces interactivas en Jupyter Notebooks. Estos widgets son útiles para la visualización de datos dinámica, la entrada de datos y la interactividad en general dentro de un notebook. Veamos algunos de los controles más comunes y útiles proporcionados por `ipywidgets`, así como ejemplos de cómo usarlos y cómo afectan sus propiedades más importantes.

1. **Widgets Básicos**

Botón (`Button`)

- **Descripción**: Se utiliza para ejecutar una acción cuando el usuario hace clic en él.
- **Propiedades importantes**:
  - `description`: Texto que aparece en el botón.
  - `button_style`: Estilo del botón, puede ser 'success', 'info', 'warning', 'danger' o '' (estilo por defecto).
  - `icon`: Agrega un ícono al botón (ejemplo: 'check', 'refresh').

```
button = widgets.Button(description="Click Me", button_style='success', icon='check')
display(button)

def on_button_clicked(b):
    print("Botón clickeado!")

button.on_click(on_button_clicked)
```

Slider (`IntSlider`, `FloatSlider`)

- **Descripción**: Permite seleccionar un valor numérico dentro de un rango mediante un deslizador.
- **Propiedades importantes**:
  - `value`: Valor inicial.
  - `min`: Mínimo valor del rango.
  - `max`: Máximo valor del rango.
  - `step`: Incremento entre valores del rango.

```
int_slider = widgets.IntSlider(value=10, min=0, max=100, step=1, description='Int Slider')
display(int_slider)
```

Caja de Texto (`Text`, `Textarea`)

- **Descripción**: Permite la entrada de texto por parte del usuario.
- **Propiedades importantes**:
  - `value`: Texto inicial en la caja de texto.
  - `placeholder`: Texto que se muestra en la caja de texto cuando está vacía.

```
text = widgets.Text(value='Hola', placeholder='Introduce texto aquí', description='String:')
display(text)
```

2. **Widgets para Selección**

Dropdown (`Dropdown`)

- **Descripción**: Permite seleccionar un valor de una lista desplegable.
- **Propiedades importantes**:
  - `options`: Lista de opciones o un diccionario de etiquetas y valores.
  - `value`: Valor seleccionado actualmente.
  - `description`: Descripción colocada al lado del widget.

```
dropdown = widgets.Dropdown(options=['Uno', 'Dos', 'Tres'], value='Dos', description='Número:')
display(dropdown)
```

RadioButton (`RadioButtons`)

- **Descripción**: Permite seleccionar una única opción de un conjunto visible.
- **Propiedades importantes**:
  - `options`: Lista de opciones disponibles.
  - `value`: Opción seleccionada actualmente.

```
radiobuttons = widgets.RadioButtons(options=['pepperoni', 'pineapple', 'anchovies'], description='Pizza topping:')
display(radiobuttons)
```

3. **Widgets para Visualización de Información**

ProgressBar (`IntProgress`, `FloatProgress`)

- **Descripción**: Muestra el progreso de una tarea.
- **Propiedades importantes**:
  - `value`: Valor actual del progreso.
  - `min`: Valor mínimo del progreso.
  - `max`: Valor máximo del progreso.
  - `description`: Descripción colocada al lado del progreso.

```
progress = widgets.IntProgress(value=70, min=0, max=100, description='Loading:')
display(progress)
```

4. **Organización con Layouts**

`ipywidgets` también proporciona `Box`, `HBox` y `VBox` para organizar widgets en la página, permitiendo la alineación horizontal o vertical.

```
text1 = widgets.Text(description='Nombre:')
text2 = widgets.Text(description='Apellido:')
button = widgets.Button(description='Guardar')

vbox = widgets.VBox([text1, text2, button])
display(vbox)
```

Estos widgets y controles son solo una muestra de lo que puedes hacer con `ipywidgets`. A medida que experimentes y combines estos widgets, descubrirás formas aún más ricas y dinámicas de interactuar con tus datos y usuarios

en Jupyter Notebooks.

```
import ipywidgets as widgets
from IPython.display import display

# Función que se llama cuando el botón es clickeado
def calcular(b):
    resultado.value = str(int(numero1.value) + int(numero2.value))

# Crear widgets
numero1 = widgets.Text(value='0', description='Número 1:', continuous_update=False)
numero2 = widgets.Text(value='0', description='Número 2:', continuous_update=False)
sumar_btn = widgets.Button(description='Sumar')
resultado = widgets.Label(value='Resultado aparecerá aquí')

# Mostrar widgets
display(numero1, numero2, sumar_btn, resultado)

# Definir evento del botón
sumar_btn.on_click(calcular)
```

Número 1: 345

Número 2: 654

Sumar

999

```
import ipywidgets as widgets
from IPython.display import display

button = widgets.Button(description="Pulsame", button_style='info', icon='info')

display(button)

def on_pulsado(b):
    print("Botón pulsado")

button.on_click(on_pulsado)
```

Pulsame

Botón pulsado

```
import ipywidgets as widgets
from IPython.display import display

contador = 0

label = widgets.Label(value=f"Contador: {contador}")
display(label)

buttonInc = widgets.Button(description="Incrementar", button_style='success', icon='check')
buttonDec = widgets.Button(description="Decrementar", button_style='success', icon='check')

hbox = widgets.HBox([buttonInc, buttonDec])
display(hbox)

def incrementar(b):
  global contador
  contador += 1
  label.value = f"Contador: {contador}"

def decrementar(b):
  global contador
  contador -= 1
  label.value = f"Contador: {contador}"

buttonInc.on_click(incrementar)
buttonDec.on_click(decrementar)
```

Contador: 0

Incrementar Decrementar

```
import ipywidgets as widgets
from IPython.display import display

mySlider = widgets.IntSlider(value=10, min=0, max=100, step=5)

out1 = widgets.Output()

display(mySlider, out1)

def on_slider_change(observer):
  with out1:
    print("Salida:", observer["old"], observer["new"])

mySlider.observe(on_slider_change, names="value")
```

15

```
Salida: 10 15
Salida: 15 30
Salida: 30 45
Salida: 45 50
Salida: 50 55
Salida: 55 60
Salida: 60 65
Salida: 65 70
Salida: 70 65
Salida: 65 50
Salida: 50 45
Salida: 45 40
Salida: 40 5
Salida: 5 10
Salida: 10 100
Salida: 100 95
Salida: 95 0
Salida: 0 5
Salida: 5 35
Salida: 35 20
Salida: 20 15
```

```
import ipywidgets as widgets
from IPython.display import display

mySlider = widgets.IntSlider(value=10, min=0, max=100, step=5)
display(mySlider)

label = widgets.Label(value="Valor=10")
display(label)

def on_slider_change(observer):
  label.value = "Salida: {} >>> {}".format(observer["old"], observer["new"])
```

Subir un archivo desde Google Colab

```
from google.colab import files

uploaded = files.upload()

for filename in uploaded.keys():
  print('Archivo "{name}" con longitud {length} bytes subido exitosamente.'.format(
      name=filename, length=len(uploaded[filename])))
```

Elegir archivos adult.csv

- **adult.csv**(text/csv) - 3974305 bytes, last modified: 15/4/2024 - 100% done

```
Saving adult.csv to adult.csv
Archivo "adult.csv" con longitud 3974305 bytes subido exitosamente.
```

Descargar un archivo desde Google Colab

```
from google.colab import files

# Supongamos que tienes un archivo llamado 'example.csv' en tu entorno de Colab
files.download('candado.jpg')
```

```
import ipywidgets as widgets
from IPython.display import display
import random
import json
from google.colab import files

button = widgets.Button(description="Pulsame", button_style='info', icon='info')

display(button)

def on_pulsado(b):
  frutas = []
  for i in range(100):
    fruta = {
        "id": random.randint(1, 100_000),
        "nombre": random.choice(["Manzana", "Pera", "Mango"]),
        "precio": random.uniform(1, 100)
    }
    frutas.append(fruta)
  data = json.dumps(frutas)
  with open("data.json", "w") as outputFile:
    outputFile.write(data)
  files.download('data.json')

button.on_click(on_pulsado)
```

Pulsame