Tarea	Biblioteca/Función	Descripción
Automatizar acciones en la web	Selenium	Permite automatizar la navegación web, interactuar con formularios, hacer scraping, etc.
Automatización de tareas de sistema	os / shutil	os permite interactuar con el sistema operativo (crear, eliminar archivos, etc.), shutil para operaciones con archivos y directorios.
Automatización de tareas programadas	schedule	Permite programar la ejecución de funciones en intervalos regulares (horas, minutos, etc.).
Automatización de correos electrónicos	smtplib / email	smtplib se usa para enviar correos electrónicos, y email para construir mensajes más complejos.
Automatización de entrada y salida de datos	csv / json	Leer y escribir archivos CSV o JSON para intercambiar datos de manera automatizada.
Automatización de tareas de red	requests / http.client	requests simplifica las peticiones HTTP, mientras que http.client permite más control sobre las solicitudes.
Automatización de Excel	openpyxl / pandas	openpyxl permite trabajar con archivos Excel (.xlsx), y pandas se puede usar para manipular y procesar datos de Excel.
Automatización de interfaces gráficas	pyautogui	Permite controlar el mouse y el teclado, simulando acciones del usuario en una interfaz gráfica.
Automatización de bases de datos	sqlite3 / SQLAlchemy / pymysql	sqlite3 para bases de datos SQLite, SQLAlchemy para bases de datos relacionales, y pymysql para MySQL.
Automatización de tareas en el navegador	pyppeteer	Es una alternativa a Selenium que permite automatizar tareas en un navegador (basado en Chromium).
Automatización de tareas en la terminal	subprocess	Permite ejecutar comandos del sistema desde un script Python, interactuar con la terminal.

Schedule:

Podemos crear un script en Python que realice una función específica y luego usar **schedule** para programar esa función para que se ejecute a intervalos determinados, como diariamente a una hora específica.

Por ejemplo, si tienes una función que envía un correo electrónico o hace alguna tarea automática, puedes usar schedule para que se ejecute todos los días a una hora específica.

Ejemplo:

1. Primero, instala schedule:

```
pip install schedule
```

```
import schedule
import time

# Define la función que quieres ejecutar
def mi_tarea():
    print("¡La tarea se ejecutó!")

# Programa la tarea para que se ejecute todos los días a las 14:00
schedule.every().day.at("14:00").do(mi_tarea)

# Bucle para mantener el script en ejecución
while True:
    schedule.run_pending()  # Ejecuta las tareas pendientes
    time.sleep(60)  # Espera 1 minuto antes de volver a verificar
```

¿Cómo funciona?

1. **mi_tarea()**: Esta es la función que quieres ejecutar (en este caso solo imprime un mensaje, pero puede ser cualquier tarea que desees automatizar).
2. **schedule.every().day.at("14:00").do(mi_tarea)**: Esta línea dice que se ejecute mi_tarea todos los días a las 14:00.
3. El bucle while True mantiene el script en ejecución y permite que **schedule** verifique si hay tareas pendientes para ejecutar.

¿Qué pasa cuando se ejecuta el script?

- El script se mantiene ejecutándose, revisando cada minuto si hay alguna tarea que debe ejecutarse.
- Cuando llega la hora especificada (14:00 en este ejemplo), ejecuta la tarea (en este caso, imprime un mensaje en la consola).

Puedes cambiar el contenido de mi_tarea() para hacer lo que necesites (enviar correos, mover archivos, hacer consultas a la base de datos, etc.).

Otras bibliotecas de "tiempo".

1. APScheduler (Advanced Python Scheduler)

- **Descripción**: Es una biblioteca más avanzada que **schedule**, con más opciones y flexibilidad para ejecutar tareas programadas. Permite programar tareas de manera muy específica, como ejecutar una tarea en intervalos de tiempo personalizados, a una hora concreta del día, en días específicos de la semana, y más. También soporta la ejecución en segundo plano, persistencia de tareas, y la integración con bases de datos para almacenar trabajos programados.

2. Celery

- **Descripción**: Celery es una herramienta muy potente para la gestión de tareas en segundo plano, especialmente cuando necesitas escalar y distribuir tareas de forma asíncrona. Se usa comúnmente en aplicaciones web para procesar trabajos en cola de forma concurrente. Aunque Celery no está diseñado exclusivamente para programar tareas como **schedule**, es muy eficaz para ejecutar tareas de larga duración o cuando se necesita procesar trabajos en paralelo.

3. Timeloop

- **Descripción**: **Timeloop** es una opción simple y ligera para programar tareas recurrentes en Python. Es menos compleja que **APScheduler** y **Celery**, pero aún permite programar tareas para que se ejecuten en intervalos regulares o en momentos específicos.

4. Croniter (para cron jobs)

- **Descripción**: Si estás familiarizado con la sintaxis de cron en sistemas Unix/Linux, **croniter** es una biblioteca que permite trabajar con la misma sintaxis para la programación de tareas en Python. Permite calcular la próxima ejecución de un trabajo programado utilizando una cadena cron.

Bibliotecas de eventos:

1. watchdog

- **Descripción**: **watchdog** es una de las bibliotecas más populares para monitorear archivos y directorios en tiempo real. Puedes usarla para ejecutar un script cuando un archivo o directorio específico sea creado, modificado, renombrado o eliminado.

2. pynotifier (para notificaciones de eventos del sistema)

- **Descripción**: **pynotifier** permite crear notificaciones del sistema en respuesta a ciertos eventos o condiciones. Si bien esta biblioteca no ejecuta tareas directamente por otros eventos del sistema (como la creación de un archivo), es útil si necesitas que tu script notifique al usuario cuando ocurre algo importante.

3. pyudev (para detectar dispositivos conectados/desconectados)

- **Descripción**: **pyudev** permite monitorear eventos del sistema relacionados con dispositivos conectados, como USBs, discos externos, etc. Puedes usar esta biblioteca para ejecutar un script cuando un dispositivo sea conectado o desconectado.

4. psutil (para detectar eventos relacionados con el sistema)

- **Descripción**: **psutil** permite monitorear y gestionar recursos del sistema, como CPU, memoria, procesos, discos y redes. Puedes usar esta biblioteca para ejecutar un script cuando el sistema alcance un umbral de recursos, como cuando la CPU llega a un cierto nivel de uso o un proceso específico comienza o termina.

Resumen

- **watchdog**: Ejecuta un script cuando ocurre un cambio en un archivo o directorio (creación, modificación, eliminación, etc.).
- **pyudev**: Ejecuta un script cuando se conecta o desconecta un dispositivo (como un USB o disco duro).
- **psutil**: Monitorea el uso de recursos del sistema y ejecuta acciones cuando los recursos alcanzan ciertos umbrales.
- **pynotifier**: Permite enviar notificaciones del sistema, aunque no está orientado a eventos del sistema en sí.