Apuntes EXTENDIDOS: subprocess, os, psutil, time, threading (con ejemplos y notas)

Introducción

Este documento ofrece explicaciones detalladas, ejemplos prácticos y buenas prácticas para los módulos: **subprocess**, **os**, **psutil**, **time** y **threading**. Incluye notas sobre seguridad, errores comunes y comportamiento en distintos sistemas (Windows/Linux/Mac).

1. Módulo: subprocess

Qué hace: Permite lanzar procesos externos (comandos, programas) desde Python, capturar su salida, comunicarte con ellos (stdin/stdout/stderr) y gestionar tiempo de ejecución. Es la forma recomendada en Python 3 para interactuar con la shell o ejecutar otros ejecutables.

Cuándo usarlo: - Ejecutar utilidades del sistema (ls, dir, ping). - Llamar a scripts externos. - Automatizar tareas que no tienen API Python. Evita usar shell=True si no es necesario por seguridad.

Riesgos y buenas prácticas: - Nunca pase strings sin sanitizar a shell=True con datos del usuario (riesgo de inyección). - Use listas de argumentos cuando sea posible (ej: ["ls", "-l"]). - Controle timeouts y errores (capture returncode). - Use encoding/text=True para trabajar con strings en vez de bytes.

Ejemplo básico (subprocess.run):

```
import subprocess

# Ejecuta un comando y captura su salida como texto
result = subprocess.run(['echo', 'Hola desde subprocess'], capture_output=True, text=True)
print('returncode:', result.returncode)
print('stdout:', result.stdout.strip())
```

Ejemplo con timeout y manejo de errores:

```
import subprocess

try:
    # timeout en segundos
    r = subprocess.run(['sleep', '5'], timeout=2)
except subprocess.TimeoutExpired:
    print('El proceso tardó demasiado y fue interrumpido')
```

Ejemplo avanzado: comunicar con un proceso (Popen) y leer salida en streaming

```
import subprocess

p = subprocess.Popen(['ping', '-c', '4', '8.8.8.8'], stdout=subprocess.PIPE, stderr=subprocess.

# Leer línea a línea en tiempo real
for line in p.stdout:
    print('SALIDA:', line.strip())

p.wait()
print('Código de salida:', p.returncode)
```

Notas: - En Windows, 'ping' usa argumentos distintos a Linux ('-n' en vez de '-c'). - Si necesitas pasar input: subprocess.run(..., input='texto', text=True). - Para ejecuciones simples, subprocess.run es suficiente; Popen es para control fino (pipes, lectura asíncrona, comunicación bidireccional).

2. Módulo: os

Qué hace: Abstracción para interactuar con el sistema operativo: rutas, archivos, variables de entorno, procesos, permisos.

Usos comunes: Navegar directorios, leer/crear/eliminar archivos y directorios, obtener info del sistema, ejecutar comandos simples (os.system), y manipular variables de entorno.

Advertencia: Evita usar os.system para capturar salida (usa subprocess). Ten en cuenta permisos y diferencias entre sistemas operativos (separador de rutas, permisos Unix, etc.).

Ejemplo: operaciones con archivos y rutas (portátil)

```
import os

# Obtener el directorio actual
cwd = os.getcwd()
print('Directorio actual:', cwd)

# Construir rutas de forma portable
ruta = os.path.join(cwd, 'ejemplo', 'subcarpeta', 'archivo.txt')
print('Ruta construida:', ruta)

# Crear estructura si no existe
os.makedirs(os.path.dirname(ruta), exist_ok=True)

# Escribir un archivo
with open(ruta, 'w', encoding='utf-8') as f:
    f.write('Contenido de ejemplo\n')

# Listar contenidos del directorio
print('Listado:', os.listdir(os.path.join(cwd, 'ejemplo')))

# Limpiar
os.remove(ruta)
os.rmdir(os.path.join(cwd, 'ejemplo', 'subcarpeta'))
os.rmdir(os.path.join(cwd, 'ejemplo'))
```

Variables de entorno:

```
import os

# Leer
path = os.environ.get('PATH')
print('PATH tiene longitud:', len(path) if path else 'vacío')

# Escribir temporalmente (solo para el proceso actual)
os.environ['MI_VAR'] = 'valor'
print('MI_VAR:', os.environ['MI_VAR'])
```

Información del proceso y PID:

```
import os, sys

print('PID actual:', os.getpid())
print('UID (si aplica):', getattr(os, 'getuid', lambda: 'N/A')())
```

Permisos y modos (Unix): usa os.stat() para metadatos; cambie permisos con os.chmod(path, mode).

3. Módulo: psutil (instalar con pip)

Qué hace: psutil es una librería multiplataforma para obtener métricas del sistema y gestionar procesos. No es parte de la biblioteca estándar; instálala con: pip install psutil.

Por qué usarla: Proporciona una API consistente para CPU, memoria, discos, red y procesos. Ideal para monitorización, herramientas de administración y diagnósticos.

Ejemplo: uso de CPU y memoria:

```
import psutil

print('CPU percent (1s):', psutil.cpu_percent(interval=1))
vm = psutil.virtual_memory()
print('Memoria total (bytes):', vm.total)
print('Memoria usada (%):', vm.percent)
```

Ejemplo: listar procesos y obtener más info:

```
import psutil

for proc in psutil.process_iter(['pid', 'name', 'username', 'cpu_percent']):
    try:
        print(proc.info)
    except (psutil.NoSuchProcess, psutil.AccessDenied):
        pass
```

Ejemplo: cómo terminar un proceso (con cuidado)

```
import psutil

pid = 12345  # ejemplo
try:
    p = psutil.Process(pid)
    p.terminate()  # enviar SIGTERM en Unix
    p.wait(timeout=3)
except psutil.NoSuchProcess:
    print('El proceso no existe')
except psutil.TimeoutExpired:
    print('No respondió, se forzará kill')
    p.kill()
```

Notas de seguridad y permisos: - Algunas operaciones (inspeccionar/terminar procesos de otros usuarios) requieren privilegios de administrador/root. - En entornos compartidos, ten cuidado con la información sensible (nombres de usuario, rutas).

4. Módulo: time

Qué hace: Funciones para medir tiempo, pausar ejecución, y convertir entre formatos de tiempo.

Funciones clave y cuándo usarlas:

- time.time(): timestamp Unix (segundos desde 1970). Útil para guardar tiempos absolutos.

- time.perf_counter(): reloj de alta resolución, ideal para medir intervalos (benchmarking).

- time.process_time(): tiempo de CPU consumido por el proceso (no incluye tiempo inactivo).

- time.sleep(secs): pausa la ejecución; evita sleeps largos en GUIs o servidores (bloquea el hilo).

Ejemplo: medir duración de una función con perf_counter

```
import time

start = time.perf_counter()
# código a medir
sum(range(10_000_000))
end = time.perf_counter()
print('Duración:', end - start, 'segundos')
```

Ejemplo: formateo de fechas

```
import time

ts = time.time()
print('Timestamp:', ts)
print('Fecha legible:', time.ctime(ts))
print('Local time struct:', time.localtime(ts))
```

Notas: - Para manipulación y formateo de fechas a alto nivel, preferir el módulo datetime (más rico y seguro para zonas horarias). - Evita usar sleep en aplicaciones sensibles a latencia; para esperar sin bloquear, considera hilos o asyncio (event loop).

5. Módulo: threading

Qué hace: threading permite ejecutar tareas concurrentes mediante hilos (threads). En CPython existe el GIL (Global Interpreter Lock), que afecta la ejecución de hilos en operaciones CPU■bound (no paraleliza threads de Python que hagan cálculos intensos). Sin embargo, threading sigue siendo útil para I/O■bound (red, disco, espera).

Conceptos clave: - Thread: unidad de ejecución. - Lock: para sincronizar acceso a recursos compartidos. - Daemon threads: mueren cuando el hilo principal finaliza. - join(): esperar a que termine un hilo.

Ejemplo: hilos con lock para proteger recurso compartido

```
import threading
import time

contador = 0
lock = threading.Lock()

def sumar():
    global contador
    for _ in range(10000):
        with lock:
            contador += 1

threads = [threading.Thread(target=sumar) for _ in range(5)]
for t in threads:
    t.start()
for t in threads:
    t.join()

print('Valor final contador:', contador)
```

Ejemplo: hilos daemon y no daemon

```
import threading, time

def tarea_larga():
    for i in range(10):
        print('tarea', i)
        time.sleep(1)

t = threading.Thread(target=tarea_larga, daemon=True)
t.start()
print('El hilo demonio puede morir si el proceso principal termina')
```

Notas sobre GIL y alternativas: - Para paralelismo en CPU■bound usa multiprocessing (multiprocess) o librerías nativas en C. - Para I/O concurrido, threading o asyncio son buenas opciones. - Evita deadlocks: mantén consistencia en el orden de adquisición de locks, utiliza timeout en acquire cuando sea necesario.

6. Patrones avanzados y ejemplos combinados

Ejemplo combinado: Monitoriza un proceso externo con subprocess y psutil, midiendo uso de CPU y tiempo con time, en un hilo.

```
import subprocess, psutil, time, threading

def monitor(pid):
    try:
        p = psutil.Process(pid)
    except psutil.NoSuchProcess:
        print('Proceso no encontrado')
        return
    while p.is_running():
        print('CPU %', p.cpu_percent(interval=0.5), 'Mem %', p.memory_percent())
        time.sleep(0.5)

# Lanzar un proceso que consuma algo de tiempo (ejemplo 'sleep' en Unix)
proc = subprocess.Popen(['sleep', '5'])
t = threading.Thread(target=monitor, args=(proc.pid,))
t.start()
proc.wait()
t.join()
print('Proceso terminado')
```

Ejemplo: comunicarse con proceso que espera input

```
import subprocess, time

p = subprocess.Popen(['python', '-u', '-c', "print(input())"], stdin=subprocess.PIPE, stdout=su
stdout, stderr = p.communicate(input='Hola desde parent\n', timeout=3)
print('Salida child:', stdout)
```

Checklist de depuración y errores comunes:

- Si subprocess devuelve returncode != 0, revisa stderr para diagnosticar. - En Windows, usa argumentos adecuados (-n vs -c en ping). - Si psutil no lista procesos, comprueba permisos (admin/root). - Evita sleeps largos en el hilo principal de programas interactivos. - Usa locks para proteger variables compartidas en threading. - Considera asyncio para concurrencia basada en eventos (I/O).

Notas finales y recursos

- Para manipulación avanzada de fechas, usa 'datetime' y 'zoneinfo' en Python 3.9+.

- psutil: documentación en https://psutil.readthedocs.io (instalar con pip).

- Para ejecutar tareas paralelas CPU-bound, mira 'multiprocessing' o 'concurrent.futures.ProcessPoolExecutor'.

- Para patrones async I/O, mira 'asyncio' y 'aiohttp' (para HTTP asíncrono).