Bibliotecas Python para operar con el Sistema Operativo

os

Documentantación oficial: https://docs.python.org/es/3.10/library/os.html
Resúmen: El módulo **os** en Python es una parte esencial de la biblioteca estándar que proporciona una interfaz para interactuar con funciones específicas del sistema operativo subyacente. Este módulo es particularmente útil para realizar tareas relacionadas con el sistema operativo, como manipulación de archivos, operaciones de directorios, acceso a variables de entorno y más. Aquí hay una descripción más detallada de algunas de las funciones y características clave del módulo **os**:

1. Manipulación de archivos y directorios:

   os.listdir(path): Devuelve una lista de nombres de archivos y subdirectorios en la ruta especificada.
   os.mkdir(path): Crea un nuevo directorio en la ruta especificada.
   os.makedirs(path): Crea directorios anidados, creando todos los componentes necesarios.
   os.rename(old, new): Cambia el nombre de un archivo o directorio.
   os.remove(path): Elimina un archivo.
   os.rmdir(path): Elimina un directorio vacío.

2. Información del sistema:

   os.getcwd(): Devuelve el directorio de trabajo actual.
   os.path.abspath(path): Devuelve la ruta absoluta de una ruta relativa.
   os.path.exists(path): Comprueba si la ruta especificada existe.
   os.path.isfile(path): Comprueba si la ruta es un archivo.
   os.path.isdir(path): Comprueba si la ruta es un directorio.

3. Variables de entorno:

   os.environ: Un diccionario que contiene las variables de entorno del sistema.
   os.getenv(variable): Devuelve el valor de una variable de entorno específica.

4. Operaciones de sistema:

   os.system(command): Ejecuta un comando del sistema en una subshell.
   os.startfile(file): Abre un archivo con el programa predeterminado del sistema (solo en Windows).

5. Manipulación de rutas:

   os.path.join(path, *paths): Une componentes de ruta en una única ruta.
   os.path.basename(path): Devuelve el último componente de la ruta (nombre del archivo).
   os.path.dirname(path): Devuelve el directorio padre de la ruta.

6. Permisos y propietario:

   os.chmod(path, mode): Cambia los permisos de un archivo o directorio.
   os.chown(path, uid, gid): Cambia el propietario (usuario y grupo) de un archivo o directorio.

A continuación se muestra un ejemplo de cómo usar el módulo os para crear un directorio nuevo:

```
import os
os.mkdir("my_new_directory")
```

Este código creará un directorio nuevo llamado "**my_new_directory**" en el directorio de trabajo actual.

pathlib

pathlib es un módulo de la biblioteca estándar de **Python** que proporciona una interfaz orientada a objetos para trabajar con rutas de archivos y directorios. Fue introducido en **Python 3.4** para simplificar y mejorar la manipulación de rutas y archivos en comparación con los métodos más antiguos proporcionados por el módulo os y otros.

La principal idea detrás de **pathlib** es tratar las rutas y los archivos como objetos. Esto significa que puedes manipularlos utilizando métodos y propiedades que son más naturales y legibles para los programadores. Algunas características clave de **pathlib** incluyen:

1. Sintaxis Orientada a Objetos:

   En lugar de usar funciones sueltas para operaciones de manejo de archivos y directorios, **pathlib** utiliza objetos que representan rutas y archivos. La clase principal es Path, que se utiliza para instanciar objetos que representan rutas en el sistema de archivos.

2. Operaciones Intuitivas:

   Los objetos Path proporcionan métodos y propiedades para realizar operaciones comunes de manera más intuitiva. Por ejemplo, puedes usar el operador **/** para concatenar componentes de la ruta, y los métodos como **resolve()**, parent y name para acceder y manipular partes específicas de la ruta.

3. Independencia del Sistema Operativo:

   pathlib se encarga de las diferencias entre sistemas operativos en cuanto a la representación de rutas. Esto significa que puedes escribir código que funcione en diferentes plataformas sin preocuparte por las convenciones específicas del sistema.

4. Código Más Limpio y Legible:

   La sintaxis y los métodos de **pathlib** permiten escribir código más limpio y legible, lo que facilita la comprensión de las operaciones que se están realizando en las rutas y los archivos.

Aquí hay un ejemplo simple de cómo se vería el uso de **pathlib** para crear una nueva carpeta y un archivo en ella:

```
from pathlib import Path
# Crear una nueva carpeta
nueva_carpeta = Path("mi_carpeta")
nueva_carpeta.mkdir()
# Crear un archivo en la nueva carpeta
archivo = nueva_carpeta / "mi_archivo.txt"
archivo.touch()
```

En resumen, **pathlib** es una herramienta útil en Python para trabajar con rutas y archivos de manera más elegante y eficiente, especialmente en comparación con las funciones tradicionales de manejo de archivos del módulo os.

shutil

Documentación oficial: https://docs.python.org/es/3.10/library/shutil.html
Resúmen: El módulo **shutil** en Python es otro componente importante de la biblioteca estándar que proporciona funciones de alto nivel para realizar operaciones relacionadas con archivos y directorios. La principal ventaja de **shutil** es que simplifica tareas comunes y complejas de manipulación de archivos y directorios, y ofrece métodos más amigables y seguros que los proporcionados por el módulo os. Aquí tienes una descripción de algunas de las funciones clave del módulo **shutil**:

1. Copiado, Movimiento y Eliminación:

   shutil.copy(src, dst): Copia un archivo de la ubicación src a la ubicación dst.
   shutil.copy2(src, dst): Similar a **shutil.copy()**, pero también intenta mantener metadatos (timestamps, permisos, etc.).
   shutil.copytree(src, dst): Copia un directorio completo de la ubicación **src** a la ubicación **dst**.
   shutil.move(src, dst): Mueve un archivo o directorio de **src** a **dst**.
   shutil.rmtree(path): Elimina un directorio y todo su contenido.

2. Archivos de Compresión y Extracción:

   shutil.make_archive(base_name, format, root_dir): Crea un archivo de compresión (zip, tar, etc.) desde un directorio.
   shutil.unpack_archive(archive_name, extract_dir): Descomprime un archivo de compresión en un directorio.

3. Operaciones de Archivo:

   shutil.which(cmd): Busca la ubicación de un comando ejecutable en el sistema.

shutil.disk_usage(path): Devuelve información sobre el uso de disco en una ruta.

4. Operaciones de Directorio:

   shutil.disk_usage(path): Devuelve información sobre el uso de disco en una ruta.
   shutil.get_archive_formats(): Devuelve una lista de formatos de archivo de compresión compatibles.

5. Cambio de Permisos y Propietario:

   shutil.copystat(src, dst): Copia los metadatos de un archivo, como permisos y timestamps, de **src** a **dst**.

El módulo **shutil** es especialmente útil cuando necesitas realizar tareas que involucran la manipulación de archivos y directorios en un nivel más alto y abstracto que lo que ofrece el módulo **os**. Además, la mayoría de las funciones en **shutil** son conscientes del sistema operativo subyacente y adaptan su comportamiento en consecuencia, lo que hace que tu código sea más portable.

glob

Documentación oficial: https://docs.python.org/es/3.10/library/glob.html
Resúmen: El módulo **glob** en **Python** es una parte de la biblioteca estándar que te permite buscar archivos y directorios en un sistema de archivos utilizando patrones de coincidencia similares a los que se utilizan en la línea de comandos. Es particularmente útil para encontrar archivos que coinciden con ciertos patrones en un directorio dado. Aquí hay una descripción más detallada del módulo glob:

1. Funciones Principales:

   glob.glob(pattern): Devuelve una lista de rutas que coinciden con el patrón especificado. El patrón puede contener caracteres comodín, como * (cualquier cantidad de caracteres) y ? (un solo carácter).
   glob.iglob(pattern): Similar a **glob.glob()**, pero devuelve un iterador en lugar de una lista. Esto es útil para manejar grandes conjuntos de archivos.

2. Patrones de Coincidencia:

   Los patrones que se utilizan en **glob** son similares a los patrones utilizados en la línea de comandos para buscar archivos.

   *: Coincide con cualquier número de caracteres.
   ?: Coincide con un solo carácter.
   [...]: Coincide con cualquier carácter dentro del rango especificado.
   [!...]: Coincide con cualquier carácter que no esté en el rango especificado.

Ejemplos:

```
import glob
# Buscar todos los archivos .txt en el directorio actual
txt_files = glob.glob("*.txt")
```

```
# Buscar todos los archivos que comienzan con "file" y tienen extensión .csv
csv_files = glob.glob("file*.csv")
# Buscar todos los archivos en subdirectorios que tengan la extensión .jpg
all_jpg_files = glob.glob("**/*.jpg", recursive=True)
```

El módulo **glob** es especialmente útil cuando necesitas trabajar con múltiples archivos que siguen un patrón específico, como cargar todos los archivos de datos de un cierto tipo en un directorio o procesar archivos en lotes. Además, la capacidad de utilizar patrones de coincidencia familiares hace que **glob** sea una herramienta intuitiva y fácil de usar.

subprocess

Documentación oficial: https://docs.python.org/es/3.10/library/subprocess.html
Resúmen: El módulo **subprocess** en **Python** es una herramienta poderosa que permite ejecutar comandos del sistema operativo desde un programa de **Python**. Proporciona una interfaz más segura y flexible para interactuar con procesos externos en comparación con el módulo más antiguo **os.system()**.
El módulo **subprocess** ofrece varias funciones y clases para crear, administrar y comunicarse con subprocesos. Aquí hay una descripción de algunos de los conceptos clave y funciones en el módulo **subprocess**:

1. Funciones Principales:

   subprocess.run(args, ...): Ejecuta un comando y espera a que termine. Reemplaza a **os.system()** y es más flexible. Devuelve un objeto **CompletedProcess** con información sobre la ejecución del comando.
   subprocess.call(args, ...): Similar a subprocess.run(), pero devuelve el código de retorno del proceso en lugar de un objeto CompletedProcess.
   subprocess.Popen(args, ...): Inicia un nuevo proceso y devuelve un objeto Popen que puede usarse para controlar la ejecución, comunicación y finalización del proceso.

2. Argumentos:

   Los argumentos para estas funciones generalmente se proporcionan como una lista o cadena. Si se pasa como una cadena, se divide en una lista automáticamente. Por ejemplo:

   ```
   import subprocess
   result = subprocess.run(["ls", "-l"], capture_output=True, text=True)
   ```

3. Control de Procesos:

   Popen.poll(): Comprueba si el proceso ha terminado.
   Popen.wait(): Espera hasta que el proceso finalice.
   Popen.communicate(input=None, timeout=None): Comunica con el proceso, enviando datos y recibiendo resultados.
   Popen.terminate(): Envía una señal **SIGTERM** al proceso para solicitar su terminación.
   Popen.kill(): Envía una señal **SIGKILL** al proceso para forzar su terminación.

4. Redireccionamiento de Entrada/Salida:

   Puedes redirigir la entrada/salida estándar de un proceso a través de los parámetros **stdin**, **stdout** y **stderr** en las funciones de **subprocess**.

5. Manejo de Excepciones:

   El módulo **subprocess** genera excepciones, como **CalledProcessError**, en caso de que ocurra un error durante la ejecución del proceso.

El módulo **subprocess** es útil cuando necesitas ejecutar comandos del sistema operativo desde tu programa de **Python**, interactuar con la salida de esos comandos y controlar la ejecución de los procesos de manera más avanzada. Sin embargo, dado que involucra la ejecución de procesos externos, es importante tener en cuenta la seguridad y evitar la inyección de comandos maliciosos. Usar argumentos bien formateados y evitar la interpolación directa de entradas del usuario es esencial para escribir código seguro.