Modulos importantes:

Json

json.dumps(objeto): Convierte un objeto de Python en una cadena JSON. El objeto puede ser una lista, un diccionario, una tupla, etc.

La función `json.dumps(objeto)` en Python es utilizada para convertir un objeto de Python en una cadena JSON. Esta función toma como entrada un objeto de Python, como una lista, un diccionario, una tupla u otro tipo de dato válido, y lo convierte en una cadena de caracteres en formato JSON.

El formato JSON es un estándar de intercambio de datos utilizado ampliamente en aplicaciones web y sistemas distribuidos. Permite representar datos de manera estructurada y legible tanto para los humanos como para las máquinas. Al convertir un objeto de Python en una cadena JSON, podemos compartirlo con otros sistemas que también entiendan JSON, como servicios web o bases de datos.

Aquí tienes un ejemplo práctico de cómo utilizar `json.dumps(objeto)`:

```
import json

datos = {
    "nombre": "Juan",
    "edad": 25,
    "hobbies": ["correr", "leer", "viajar"]
}

cadena_json = json.dumps(datos)
print(cadena_json)
```

En este ejemplo, creamos un diccionario llamado `datos` que contiene información sobre una persona, incluyendo su nombre, edad y una lista de hobbies. Luego, utilizamos `json.dumps(datos)` para convertir el diccionario en una cadena JSON. Finalmente, imprimimos la cadena JSON resultante.

El resultado impreso será:

```
{"nombre": "Juan", "edad": 25, "hobbies": ["correr", "leer", "viajar"]}
```

Observamos que el diccionario de Python se ha convertido en una cadena JSON válida, donde las claves y los valores están entre comillas dobles, y los elementos de la lista están entre corchetes.

json.loads(cadena_json): Convierte una cadena JSON en un objeto de Python. La cadena JSON debe estar en formato válido.

La función `json.loads(cadena_json)` en Python se utiliza para convertir una cadena JSON en un objeto de Python. Toma como entrada una cadena de caracteres en formato JSON y la convierte en un objeto de Python, como un diccionario, una lista, una tupla u otro tipo de dato válido.

Es importante destacar que la cadena JSON debe estar en formato válido para que la función `json.loads()` pueda realizar la conversión correctamente. De lo contrario, se generará una excepción `JSONDecodeError`.

A continuación, se muestra un ejemplo práctico de cómo utilizar `json.loads(cadena_json)`:

```
import json

cadena_json = '{"nombre": "Juan", "edad": 25, "hobbies": ["correr", "leer", "viajar"]}'
datos = json.loads(cadena_json)
print(datos)
```

En este ejemplo, tenemos una cadena JSON válida que representa la misma información que el ejemplo anterior. Utilizamos `json.loads(cadena_json)` para convertir la cadena JSON en un objeto de Python llamado `datos`. Finalmente, imprimimos el objeto `datos`.

El resultado impreso será:

```
{
    'nombre': 'Juan',
    'edad': 25,
```

```
    'hobbies': ['correr', 'leer', 'viajar']
}
```

Observamos que la cadena JSON se ha convertido en un objeto de Python, donde las claves y los valores se representan como en un diccionario y los elementos de la lista se representan como una lista de Python.

Es importante asegurarse de que la cadena JSON esté en formato válido para que la conversión sea exitosa. Si la cadena no cumple con la sintaxis JSON, se generará una excepción `JSONDecodeError`.

json.dump(objeto, archivo): Escribe un objeto de Python en un archivo en formato JSON. El objeto puede ser una lista, un diccionario, una tupla, etc.

La función `json.dump(objeto, archivo)` en Python se utiliza para escribir un objeto de Python en un archivo en formato JSON. Esta función toma como entrada un objeto de Python, como una lista, un diccionario, una tupla u otro tipo de dato válido, y lo guarda en un archivo en formato JSON.

Aquí tienes un ejemplo práctico de cómo utilizar `json.dump(objeto, archivo)`:

```
import json

datos = {
    "nombre": "Juan",
    "edad": 25,
    "hobbies": ["correr", "leer", "viajar"]
}

with open("datos.json", "w") as archivo:
    json.dump(datos, archivo)
```

En este ejemplo, tenemos un diccionario llamado `datos` que contiene información sobre una persona. Utilizamos `json.dump(datos, archivo)` para escribir el diccionario en un archivo llamado "datos.json". El modo `"w"` indica que se abrirá el archivo en modo escritura.

Después de ejecutar el código, se creará un archivo llamado "datos.json" en el directorio actual y se guardará el diccionario en formato JSON dentro de ese archivo.

El contenido del archivo "datos.json" será el siguiente:

```
{"nombre": "Juan", "edad": 25, "hobbies": ["correr", "leer", "viajar"]}
```

Observamos que el diccionario de Python se ha guardado correctamente en formato JSON en el archivo.

json.load(archivo): Lee un archivo JSON y lo convierte en un objeto de Python.

La función `json.load(archivo)` en Python se utiliza para leer un archivo JSON y convertirlo en un objeto de Python. Toma como entrada un archivo en formato JSON y devuelve un objeto de Python, como un diccionario, una lista, una tupla u otro tipo de dato válido.

A continuación se muestra un ejemplo práctico de cómo utilizar `json.load(archivo)`:

```
import json

with open("datos.json", "r") as archivo:
    datos = json.load(archivo)
    print(datos)
```

En este ejemplo, abrimos el archivo "datos.json" en modo lectura (`"r"`) utilizando la instrucción `with open`. A continuación, utilizamos `json.load(archivo)` para leer el contenido del archivo y convertirlo en un objeto de Python llamado `datos`. Finalmente, imprimimos el objeto `datos`.

El resultado impreso será el mismo diccionario de Python que utilizamos en ejemplos anteriores:

```
{
    'nombre': 'Juan',
    'edad': 25,
    'hobbies': ['correr', 'leer', 'viajar']
}
```

Observamos que el contenido del archivo JSON se ha leído correctamente y se ha convertido en un objeto de Python.

Es importante destacar que el archivo JSON debe estar en formato válido para que la función `json.load()` pueda realizar la conversión correctamente. Si el archivo no cumple con la sintaxis JSON, se generará una excepción `JSONDecodeError`. Además, asegúrate de que el archivo exista y se encuentre en la ruta correcta.

time

Una de las funciones más básicas de la biblioteca `time` es `time()`. Esta función devuelve el tiempo actual en segundos desde el 1 de enero de 1970, también conocido como la "época Unix". Podemos utilizar esta función para medir el tiempo de ejecución de nuestro código o para realizar cálculos temporales. Por ejemplo, podemos usar `time()` para calcular la duración de un proceso:

```
import time

inicio = time.time()

# Realizar alguna tarea

fin = time.time()
duracion = fin - inicio
print("La tarea tardó:", duracion, "segundos")
```

Otra función útil de la biblioteca `time` es `sleep()`. Esta función nos permite pausar la ejecución de nuestro programa durante un número determinado de segundos. Es especialmente útil cuando necesitamos introducir una pausa entre operaciones o cuando queremos controlar el ritmo de ejecución de nuestro código. Por ejemplo, podemos usar `sleep()` para agregar una pausa entre iteraciones de un bucle:

```
import time

for i in range(5):
    print("Realizando operación", i)
    time.sleep(1)  # Pausa durante 1 segundo entre iteraciones
```

La función `ctime()` nos permite convertir un número de segundos en una cadena de caracteres que representa una fecha y hora legible para los humanos. Esto puede ser útil cuando queremos mostrar el tiempo de una manera más comprensible. Por ejemplo, podemos utilizar `ctime()` para imprimir la fecha y hora actual:

```
import time

segundos = time.time()
fecha_hora = time.ctime(segundos)
print("La fecha y hora actual es:", fecha_hora)
```

La función `gmtime()` nos permite convertir un número de segundos en una estructura de tiempo en el formato UTC (Tiempo Universal Coordinado). Esto es útil cuando necesitamos trabajar con fechas y horas en diferentes zonas horarias. Por ejemplo, podemos utilizar `gmtime()` para obtener la fecha y hora UTC actual:

```
import time

segundos = time.time()
tiempo_utc = time.gmtime(segundos)
print("La hora UTC actual es:", tiempo_utc)
```

Por otro lado, la función `localtime()` nos permite convertir un número de segundos en una estructura de tiempo local, basada en la configuración de la máquina en la que se está ejecutando el código. Esto es útil cuando necesitamos trabajar con fechas y horas en la zona horaria local. Por ejemplo, podemos utilizar `localtime()` para obtener la fecha y hora local actual:

```
import time

segundos = time.time()
tiempo_local = time.localtime(segundos)
print("La hora local actual es:", tiempo_local)
```

La función `strftime()` nos permite formatear una estructura de tiempo en una cadena de caracteres utilizando un formato especificado. Podemos utilizar esta función para personalizar la forma en que se muestra la fecha y hora en nuestros programas. Por ejemplo, podemos usar `strftime()` para mostrar solo la fecha en un formato específico:

```
import time

tiempo_actual = time.localtime()
fecha_formateada = time.strftime("%Y-%m-%d", tiempo_actual)
print("La fecha actual en formato AAAA-MM-DD es:", fecha_formateada)
```

Estas son solo algunas de las funciones más utilizadas de la biblioteca `time` en Python. La biblioteca también proporciona otras funciones y utilidades que permiten realizar operaciones más avanzadas con el tiempo, como mediciones de tiempo más precisas, manipulación de fechas y cálculos de tiempo. Al utilizar la biblioteca `time`, podemos tener un mayor control sobre la gestión del tiempo en nuestras aplicaciones y realizar tareas relacionadas con el tiempo de manera más eficiente.

Es importante tener en cuenta que la biblioteca `time` utiliza la hora del sistema de la máquina en la que se está ejecutando el código. Si necesitamos trabajar con fechas y horas en zonas horarias específicas o realizar cálculos más avanzados, podemos considerar el uso de otras bibliotecas en Python, como `datetime` o `pytz`, que ofrecen funcionalidades adicionales y una mayor flexibilidad en el manejo del tiempo.

datetime

La biblioteca `datetime` en Python es una herramienta poderosa para trabajar con fechas y horas de una manera más avanzada y precisa. Nos permite crear, manipular y formatear objetos de fecha y hora, realizar cálculos de tiempo y manejar zonas horarias. En este artículo, exploraremos las principales funcionalidades de la biblioteca `datetime` y cómo podemos utilizarlas en nuestras aplicaciones.

La biblioteca `datetime` incluye varias clases principales que nos permiten trabajar con fechas y horas. Estas clases son:

1. `datetime`: Representa una fecha y hora específica, incluyendo año, mes, día, hora, minuto, segundo y microsegundo.

2. `date`: Representa una fecha, incluyendo año, mes y día.
3. `time`: Representa una hora, incluyendo hora, minuto, segundo y microsegundo.

Además, la biblioteca `datetime` también proporciona otras clases y funciones útiles, como `timedelta` para representar diferencias de tiempo y `timezone` para trabajar con zonas horarias.

Para comenzar a utilizar la biblioteca `datetime`, primero debemos importarla en nuestro código:

```
import datetime
```

Una de las funcionalidades básicas de la biblioteca `datetime` es crear objetos de fecha y hora. Podemos utilizar las clases `datetime`, `date` y `time` para esto. Por ejemplo, podemos crear un objeto `datetime` para representar la fecha y hora actual:

```
import datetime

fecha_hora_actual = datetime.datetime.now()
print(fecha_hora_actual)
```

También podemos crear objetos `date` y `time` por separado si solo necesitamos trabajar con fechas o horas específicas:

```
import datetime

fecha_actual = datetime.date.today()
print(fecha_actual)

hora_actual = datetime.time(hour=12, minute=30, second=45)
print(hora_actual)
```

La biblioteca `datetime` nos permite realizar operaciones matemáticas con fechas y horas utilizando el objeto `timedelta`. Por ejemplo, podemos calcular la diferencia de

tiempo entre dos fechas:

```
import datetime

fecha1 = datetime.date(2022, 1, 1)
fecha2 = datetime.date(2023, 1, 1)
diferencia = fecha2 - fecha1
print("Diferencia de días:", diferencia.days)
```

También podemos realizar operaciones aritméticas con fechas y horas utilizando el objeto `timedelta`. Por ejemplo, podemos agregar o restar un período de tiempo a una fecha:

```
import datetime

fecha_actual = datetime.date.today()
un_dia = datetime.timedelta(days=1)
fecha_mañana = fecha_actual + un_dia
print("Mañana es:", fecha_mañana)
```

La biblioteca `datetime` también nos permite formatear fechas y horas en diferentes formatos utilizando el método `strftime()`. Podemos especificar el formato deseado utilizando códigos de formato específicos. Por ejemplo, podemos formatear una fecha en formato "AAAA-MM-DD":

```
import datetime

fecha_actual = datetime.date.today()
fecha_formateada = fecha_actual.strftime("%Y-%m-%d")
print(fecha_formateada)
```

Además, podemos analizar una cadena de caracteres en un objeto de fecha y hora utilizando el método `strptime()`. Debemos especificar el formato de la cadena de entrada para que la biblioteca `datetime` pueda interpretarla correctamente. Por ejemplo, podemos analizar una cadena en formato "AAAA-MM-DD" en un objeto de fecha:

```
import datetime

cadena_fecha = "2022-12-31"
fecha_objeto = datetime.datetime.strptime(cadena_fecha, "%Y-%m-%d")
print(fecha_objeto)
```

La biblioteca `datetime` también nos permite trabajar con zonas horarias utilizando el objeto `timezone`. Podemos crear objetos `timezone` para representar zonas horarias específicas y convertir fechas y horas entre diferentes zonas horarias. Sin embargo, el manejo de zonas horarias puede ser complejo y requiere una comprensión adecuada de los conceptos relacionados con las zonas horarias y la sincronización del tiempo.

Práctica de conceptos

Supongamos que estamos desarrollando una aplicación de clima que necesita obtener datos meteorológicos de una API externa. Utilizaremos la API OpenWeatherMap, que proporciona información sobre el clima actual en diferentes ubicaciones.

En primer lugar, necesitamos comprender el concepto de API. Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permite a diferentes aplicaciones comunicarse entre sí. En nuestro caso, la API de OpenWeatherMap nos permite solicitar datos meteorológicos y recibir una respuesta en formato JSON.

JSON (JavaScript Object Notation) es un formato ligero y legible por humanos para el intercambio de datos. Utiliza una estructura basada en pares clave-valor, similar a un diccionario en Python. La biblioteca `json` en Python nos permite convertir objetos de Python en formato JSON y viceversa.

Ahora, vamos a utilizar la biblioteca `datetime` para trabajar con fechas y horas. Supongamos que queremos mostrar la fecha y hora actual junto con la información del clima.

```
import requests
import json
import datetime

# Obtener la fecha y hora actual
```

```
fecha_actual = datetime.datetime.now()
fecha_formateada = fecha_actual.strftime("%Y-%m-%d %H:%M:%S")

# Realizar la solicitud a la API de OpenWeatherMap
url = "https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=YOUR_API_KEY"
response = requests.get(url)

# Convertir la respuesta JSON en un objeto de Python
datos_clima = json.loads(response.text)

# Extraer la información del clima
temperatura = datos_clima['main']['temp']
descripcion = datos_clima['weather'][0]['description']

# Imprimir la información del clima junto con la fecha y hora actual
print("Fecha y hora actual:", fecha_formateada)
print("Temperatura actual:", temperatura, "grados Celsius")
print("Descripción del clima:", descripcion)
```

En este ejemplo, primero importamos las bibliotecas necesarias: `requests`, `json` y `datetime`. Luego, utilizamos `datetime` para obtener la fecha y hora actual y formatearla como una cadena legible.

Después, utilizamos la biblioteca `requests` para realizar una solicitud a la API de OpenWeatherMap. Proporcionamos la URL de la API junto con nuestra clave de API para acceder a los datos meteorológicos de Londres.

Una vez que recibimos la respuesta de la API, utilizamos `json.loads()` para convertir la respuesta JSON en un objeto de Python. Luego, extraemos la información del clima que nos interesa, como la temperatura y la descripción.

Finalmente, imprimimos la información del clima junto con la fecha y hora actual en la consola.

Explicación sobre el manejo de elementos contenidos en variables:

```
temperatura = datos_clima['main']['temp']
descripcion = datos_clima['weather'][0]['description']
```

En estas líneas estamos asignando valores a las variables `temperatura` y `descripcion` a partir de los datos obtenidos de la respuesta de la API de OpenWeatherMap.

En la primera línea, `datos_clima['main']['temp']`, estamos accediendo al valor de la temperatura en la respuesta de la API. La variable `datos_clima` es un objeto de Python que contiene la información del clima convertida desde el formato JSON. Mediante la sintaxis `datos_clima['main']`, estamos accediendo a un diccionario dentro del objeto `datos_clima` que contiene información principal sobre el clima, y con `datos_clima['main']['temp']` accedemos al valor de la temperatura dentro de ese diccionario. Luego, ese valor se asigna a la variable `temperatura`.

En la segunda línea, `datos_clima['weather'][0]['description']`, estamos accediendo al valor de la descripción del clima en la respuesta de la API. Al igual que en la línea anterior, `datos_clima['weather']` nos permite acceder a una lista dentro del objeto `datos_clima` que contiene información sobre el clima. Al agregar `[0]` después de `datos_clima['weather']`, estamos accediendo al primer elemento de esa lista, y con `datos_clima['weather'][0]['description']` obtenemos el valor de la descripción dentro de ese elemento. Finalmente, ese valor se asigna a la variable `descripcion`.

Es importante destacar que estas líneas asumen que la respuesta de la API está estructurada de una manera específica, con claves y valores predefinidos. Es posible que la estructura de la respuesta varíe según la API que estés utilizando, por lo que es necesario revisar la documentación de la API para comprender la estructura de los datos y adaptar el acceso a las variables según corresponda.

Actividad práctica:

import pandas as pd
import json

Cargar el archivo JSON

with open('productos.json') as file:
data = json.load(file)

Normalizar el JSON en un DataFrame

df = pd.json_normalize(data, 'productos')

Imprimir el DataFrame resultante

print(df)

En este ejemplo, primero importamos las bibliotecas necesarias: Pandas (pd) y json. Luego, cargamos el archivo JSON utilizando json.load(). El resultado se almacena en la variable data.

A continuación, utilizamos pd.json_normalize() para normalizar el JSON en un DataFrame plano. Los argumentos que proporcionamos son:

data: El objeto JSON que queremos normalizar.
'productos': El camino (path) a los datos que queremos normalizar en el JSON. En este caso, los datos de interés están dentro de la clave 'productos'.

{
"productos": [
{
"id": 1,
"nombre": "Camisa",
"precio": 25.99,
"stock": 10,
"detalles": {
"color": "azul",
"talla": "M"
}
},
{
"id": 2,
"nombre": "Pantalón",
"precio": 39.99,
"stock": 5,
"detalles": {
"color": "negro",
"talla": "L"
}
},
{

"id": 3,
"nombre": "Zapatos",
"precio": 59.99,
"stock": 8,
"detalles": {
"color": "marrón",
"talla": "42"
}
}
]
}