ARCHIVOS EN PYTHON

UNPAZ

Introducción a la Programación

Archivos

Definición

Un archivo es un contenedor en dispositivos de almacenamiento informático que se utiliza para almacenar datos. También es un conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas registros que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajos denominadas campos.

En general, dividimos los archivos en dos categorías cuando se trabaja con Python: texto y binarios. Mientras que los archivos de texto son texto simple, los archivos binarios contienen datos que solo pueden ser interpretados por un ordenador. Python ofrece formas simples de manipular ambos tipos de archivos.

Archivos

Tipos

- Un ***archivo de texto*** es una secuencia de caracteres organizadas en líneas terminadas por un carácter de nueva línea. En estos archivos se pueden almacenar canciones, fuentes de programas, base de datos simples, etc. Los archivos de texto se caracterizan por ser planos, es decir, todas las letras tienen el mismo formato y no hay palabras subrayadas, en negrita, o letras de distinto tamaño o ancho.
- Un ***archivo binario*** es una secuencia de bytes que tienen una correspondencia uno a uno con un dispositivo externo. Así que no tendrá lugar ninguna traducción de caracteres. Además, el número de bytes escritos (leídos) será el mismo que los encontrados en el dispositivo externo. Ejemplos de estos archivos son tipo struct, fotografías, imágenes,texto con formatos, archivos ejecutables (aplicaciones), etc.

Archivos

Según su acceso

Tanto para archivos de texto como para archivo binarios, existen dos formas de acceder a los datos de los mismos:

Secuenciales: Los archivos secuenciales son aquellos en los que los datos se almacenan en una secuencia lineal de registros. Cada registro tiene una posición en el archivo, que se mide a partir del principio del archivo. Para acceder a un registro específico, se debe leer cada registro anterior en secuencia hasta llegar al registro deseado. Los archivos secuenciales son útiles para almacenar grandes cantidades de datos, pero no son eficientes para la búsqueda o modificación de datos específicos.

Archivos

Según su acceso

Directos: Los archivos directos son aquellos en los que se puede acceder directamente a cualquier registro en el archivo sin necesidad de leer los registros anteriores. Cada registro en el archivo tiene una dirección única asociada, que se utiliza para acceder directamente al registro deseado. Los archivos directos son más eficientes para la búsqueda y modificación de registros específicos, pero pueden ser menos eficientes para el almacenamiento de grandes cantidades de datos.

En este tipo de acceso, se utiliza un desplazamiento (offset) desde el inicio del archivo para indicar la posición deseada. Algunos ejemplos de acceso directo son el uso de la función seek() en Python o los métodos seek() y tell() en el manejo de archivos binarios.

En Python, los archivos se pueden abrir en modo secuencial o directo utilizando los métodos adecuados según el tipo de archivo. Por ejemplo, para abrir un archivo secuencial, se utiliza el método open() con el modo de apertura 'r' o 'w'. Para abrir un archivo directo, se utiliza el módulo os de Python junto con el método open() con el modo 'rb' o 'wb'.

Archivos

Operaciones

Cuando queremos leer o escribir en un archivo, primero debemos abrirlo. Cuando hayamos terminado, debe cerrarse para que se liberen los recursos que están vinculados con el archivo.

Por lo tanto, en Python, una operación de archivo se lleva a cabo en el siguiente orden:

1. abrir un archivo
2. Leer o escribir (realizar operación)
3. Cierra el archivo

Archivos

Apertura de Archivos Modalidades

Las modalidades para la apertura de un archivo pueden varían dependiendo la necesidad y el contexto en el que se trabaje. Python provee las siguientes.

Modo	Descripción
r	Abre un archivo para leer. (por defecto)
w	Abre un archivo para escribir. Crea un nuevo archivo si no existe o trunca el archivo si existe.
x	Abra un archivo para la creación exclusiva. Si el archivo ya existe, la operación falla.
a	Abra un archivo para agregarlo al final del archivo sin truncarlo. Crea un nuevo archivo si no existe.
t	Abrir en modo texto. (por defecto)
b	Abrir en modo binario.
+	Abrir un archivo para actualizar (lectura y escritura)

Archivos

Apertura de Archivos

Para efectuar la apertura de un archivo supongamos la preexistencia de un txt llamado test.txt

Según las modalidades anteriormente descrita aquí algunos ejemplos de apertura

```
file1 = open("test.txt")      # equivalent to 'r' or 'rt'
file1 = open("test.txt",'w')  # write in text mode
file1 = open("img.bmp",'r+b') # read and write in binary mode
```

Archivos

Manejo de Excepciones de Archivos

Si ocurre una excepción cuando estamos realizando alguna operación con el archivo, el código sale sin cerrar el archivo. Una forma más segura es usar un bloque `try...finally` .

```
try:
    file1 = open("test.txt", "r")
    read_content = file1.read()
    print(read_content)

finally:
    # close the file
    file1.close()
```

Archivos

Atributos de un objeto FILE

Una vez que un archivo está abierto y tienes un objeto file puedes obtener mucha información relacionada con ese archivo.

Veamos una lista de los atributos relacionados con el objeto file:

1. f.closed : retorna True si el archivo está cerrado, si no, False.
2. f.mode : retorna el modo de acceso con el que el archivo ha sido abierto.
3. f.name : retorna el nombre del archivo.

```
1.  # Abre un archivo
2.  fo = open('foo.txt', 'wb') # Abrimos el archivo llamado foo.txt en modo escritura binaria
3.  print('Nombre del archivo: ', foo.name)
4.  print('Cerrado o no: ', foo.closed)
5.  print('Modo de acceso: ', foo.mode)
```

Archivos

Lectura de Archivos

Para la lectura de archivos los archivos deben estar abiertos en la modalidad adecuada.

```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)
```

Aquí, file1.read()lee el test.txt archivo y se almacena en la variable read_content. Otra manera de leer un archivo es llamar a cierto número de carácteres. Por ejemplo, con el siguiente código el intérprete leerá los primeros cinco carácteres y los retornará como una string:

```
1.  f = open('archivoPrueba.txt', 'r')
2.  print(f.read(5))
```

Archivos

Lectura de Archivos

Para la lectura de archivos los archivos deben estar abiertos en la modalidad adecuada.

1. miArchivo.read(): Esta función devuelve una cadena el contenido completo del archivo.
2. miArchivo.readline(): Esta función devuelve linea a linea. (de a una)
3. miArchivo.readlines(): Esta función todas las líneas en una lista

```
['Juan\n', 'Susana\n', 'Andrea\n', 'Melanie\n', 'Andrés\n', 'Aca puse un texo largo\n']
```

Archivos

Iterando la Lectura de Archivos

Cuando queremos leer o retornar todas las líneas de un archivo en una forma más eficiente en cuestiones de memoria y rapidez podemos utilizar el método de iterar sobre un archivo. La ventaja de usar este método es que el código que se utiliza es simple y fácil de leer.

```
##Metodo Apertura y lectura Simple
miArchivo = open('nombres.txt','r')
for linea in miArchivo:
    print(linea)
miArchivo.close()
```

Archivos

Dividiendo las líneas de los archivos en Python

Como un ejemplo final, exploremos una de las funciones únicas que nos permiten dividir las líneas tomadas de un archivo de texto. El siguiente programa está diseñado para dividir la string contenida en la variable data cuando sea que el intérprete encuentre un carácter espacio.

```
with open('nombres.txt', 'r') as archivo:
    contenido = archivo.readlines() # Separamos el contenido del archivo en lineas
    for linea in contenido:  # Iteramos a traves de las lineas
        palabras = linea.split()  # Dividimos la linea en palabras
        print(palabras)
```

Archivos

Acceso de archivos en Python (seek())

Se utiliza para mover el indicador de posición del archivo a una posición específica. La referencia del punto del acceso al archivo está determinado por el argumento **from_what** argument y acepta los siguientes valores:

- **0: sets the reference point at the beginning of the file (text and binary files)**
  - f.seek(0) # lo setea al inicio
- **1: sets the reference point at the current file position (direct/binary files)**
  - f.seek(2, 1) # Mueve dos caracteres adelante desde la posición actual
- **2: sets the reference point at the end of the file (binary files)**
  - f.seek(-10, 2) Mueve 10 caracteres desde el fin del archivo.

Es importante tener en cuenta que la función seek() sólo tiene efecto si se ha abierto el archivo en modo lectura ("r") o lectura y escritura ("r+")

Syntax: *f.seek(offset, from_what), where f is file pointer*
Parameters:
Offset: *Number of positions to move forward*
from_what: *It defines point of reference.*
Returns: *Return the new absolute position.*

Archivos

Escritura de Archivos

Hay dos cosas que debemos recordar al escribir en un archivo.

- Si intentamos abrir un archivo que no existe, se crea un nuevo archivo.
- Si un archivo ya existe, su contenido se borra y se agrega nuevo contenido al archivo.

1. miArchivo.write(): Escribe cadena dentro del archivo
2. miArchivo.writeline(): se utiliza para escribir listas de cadenas de caracteres en un archivo.)

Archivos

Escritura de Archivos

Para escribir en un archivo en Python, debemos abrirlo en modo de escritura pasándolo "w" open() como segundo argumento. Supongamos que no tenemos un archivo llamado test2.txt . Veamos qué sucede si escribimos contenido en el archivo. test2.txt

```
with open(test2.txt', 'w') as file2:

    # write contents to the test2.txt file
    file2.write('Programming is Fun.')
    fil2.write('Programiz for beginners')
```

Aquí, se crea un nuevo archivo test2.txt y este archivo tendrá contenidos especificados dentro del método write().

Archivos

Escritura de Archivos

```
#Forma clásica de crear un archivo:
f = open("archivotext.txt", "w") #Creamos el archivo
f.write("Creando archivo de texto en python de forma clásica") #<-Escribimos en el
f.close() #Cerramos el archivo

#Utilizando With As <------------------------Nuestra forma de hacerlo
with open ("archivotext.txt", "w") as f: #Creamos el archivo
    f.write("Creando archivo de texto en python usando whit as") #<-Escribimos en el
    f.close()
```

Anexar un archivo (append):

```
ruta = r"C:\Users\Ruta\PycharmProjects\pythonProject\Algoritmos y Programacion\Rawarchivo.txt"
with open(ruta, "w") as archivo:
    archivo.write("Viene desde el Raw!\n")

with open(ruta, "a") as archivo:
    archivo.write("al fondo que hay lugar!\n")
```

Archivos

Forzado de Escritura de Archivos fflush()

flush(): Este método se utiliza para asegurarse de que todos los datos escritos en el archivo se hayan guardado físicamente en el disco. En general, Python realiza un almacenamiento en búfer de los datos escritos en el archivo para mejorar la eficiencia. Sin embargo, puede haber casos en los que desees forzar la escritura inmediata de los

```python
with open("Mitexto.txt", "w") as archivo:
    archivo.write("Hola, mundo!\n")
    archivo.flush()
    archivo.write("Argentina es tricampeon mundial.\n")
    archivo.flush()
    archivo.write("Que locura!")
    archivo.flush()
    archivo.write("Campeones")
```

Archivos

Cerrar Archivos

Para cerrar un archivo debe estar abierto con algun tipo de modalidad

```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)

# close the file
file1.close()
```

Archivos

Apertura con cierre automático

Podemos usar la with...open sintaxis para cerrar automáticamente el archivo. with... open es una construcción en Python que se utiliza para abrir y manipular archivos de manera segura y eficiente. Proporciona un bloque de código estructurado que asegura que el archivo se cierre correctamente al finalizar, incluso si ocurren excepciones durante la ejecución.

Por ejemplo,

```
with open("test.txt", "r") as file1:
    read_content = file1.read()
    print(read_content)
```

Archivos

Codificación de caracteres

UTF-8 (Unicode Transformation Format, 8-bit) es un formato de codificación de caracteres de Unicode que utiliza uno o más bytes para representar los caracteres. UTF-8 es compatible con ASCII y puede representar todos los caracteres de Unicode (más de 130.000 caracteres) utilizando hasta 4 bytes por carácter. UTF-8 se ha convertido en el estándar de codificación de caracteres más utilizado en la actualidad debido a su compatibilidad con ASCII y su capacidad para representar todos los caracteres de Unicode.

Unicode es un estándar de codificación de caracteres que utiliza un conjunto de caracteres universal que puede representar prácticamente todos los idiomas del mundo, así como símbolos matemáticos, científicos y de otros tipos. Unicode utiliza varios esquemas de codificación de caracteres, incluyendo UTF-8, UTF-16 y UTF-32. El conjunto de caracteres de Unicode incluye más de 130.000 caracteres, y está diseñado para ser compatible con sistemas existentes de codificación de caracteres, como ASCII y ISO 8859-1.

En Python, los métodos encode() y decode() se utilizan para convertir entre cadenas de bytes y cadenas de caracteres Unicode.

Archivos

Texto como Binario

En Python, el prefijo b seguido de comillas simples o dobles (b'texto' o b"texto") indica que una cadena de caracteres está en formato de bytes en lugar de formato de texto. Cuando se utiliza b como prefijo, se crea un objeto de tipo bytes. Los objetos bytes en Python son secuencias ***inmutables*** de bytes, donde cada byte representa un valor entero en el rango de 0 a 255. Estos objetos se utilizan para manipular datos en su forma binaria original, en contraste con las cadenas de texto que representan caracteres legibles. Aquí hay un ejemplo de cómo se puede utilizar el prefijo b para crear una cadena de bytes:

```
cadena_bytes =b"Argentina es tricampeon mundial.\n" #Aca lo paso a bit con el b
```

En este caso, cadena_bytes es una instancia de bytes que contiene una secuencia de bytes que representan la cadena "Hola, mundo binario!". A diferencia de una cadena de texto, no se puede modificar directamente ningún carácter individual en una cadena de bytes debido a su **inmutabilidad**.

Archivos

Codificación

Es importante tener en cuenta que, al imprimir o mostrar una cadena de bytes, Python mostrará el contenido en formato de bytes en lugar de texto legible. Si deseas obtener una representación de texto legible, debes decodificar la cadena de bytes utilizando una codificación de caracteres adecuada, como UTF-8 o ASCII.

El método encode() se utiliza para convertir una cadena Unicode en una cadena de bytes en una codificación específica, como UTF-8 o ASCII. Este método toma un parámetro que especifica la codificación deseada. Por ejemplo:

```
1  # unicode string
2  string = 'pythön es Campeón mundial!'
3  # print string
4  print('The string is:', string)
5  # default encoding to utf-8
6  string_utf = string.encode("UTF-8")
7  # print result
8  print('The encoded version is:', string_utf)
```

```
The string is: pythön es Campeón mundial!
The encoded version is: b'pyth\xc3\xb6n es Campe\xc3\xb3n mundial!'
>
```

Archivos

Decodificación

El método decode() se utiliza para convertir una cadena de bytes en una cadena Unicode. Este método también toma un parámetro que especifica la codificación de los bytes. Por ejemplo:

main.py

Run

Shell

```
1 title = b'pyth\xc3\xb6n es tricampe\xc3\xb3n mundial'
2 # change encoding to utf-8
3 print(title.decode('utf-8'))
4
```

```
pythön es tricampeón mundial
>
```

Otro ejemplo:

Using decode()

```
print(b'Easy \xE2\x9C\x85'.decode("utf-8"))
```

Run Code »

Output

```
Easy ✅
```

Archivos

Binarios

Python proporciona funciones y métodos que nos permiten leer y escribir archivos binarios. A diferencia de los archivos de texto, los archivos binarios almacenan datos en un formato no legible por humanos, lo que los hace ideales para guardar información estructurada o compleja.

Para abrir un archivo binario en Python, utilizamos la función open() con el modo de apertura correcto. En lugar de utilizar los modos de apertura de texto usuales (como 'r' para lectura o 'w' para escritura), utilizamos los siguientes modos para archivos binarios:

'rb': Para leer un archivo binario.

'wb': Para escribir en un archivo binario.

'ab': Para agregar datos a un archivo binario existente.

Archivos

Binarios

Una vez abierto el archivo en el modo adecuado, podemos leer o escribir datos binarios utilizando los métodos específicos proporcionados por Python. Algunos de los métodos más comunes son:

read(size): Lee y devuelve los siguientes size bytes del archivo. Si no se proporciona size, se leerá el archivo completo.

write(data): Escribe los bytes de data en el archivo.

seek(offset): Cambia la posición actual del archivo a offset bytes.

tell(): Devuelve la posición actual del archivo.

close(): Cierra el archivo.

Archivos

```
# Leer un archivo binario
with open('archivo.bin', 'rb') as file:
    data = file.read()
    # Procesar los datos leídos

# Escribir en un archivo binario
with open('archivo.bin', 'wb') as file:
    data = b'Datos binarios'
    file.write(data)
    # Archivo escrito correctamente
```

Archivos

Binarios - Pickle module

El módulo pickle en Python nos permite serializar objetos y almacenarlos en archivos binarios, y luego recuperarlos posteriormente. Esto es útil cuando necesitamos preservar la estructura y el estado de un objeto, incluso después de que el programa se haya cerrado.

Para utilizar el módulo pickle, necesitamos seguir los siguientes pasos:

1- Importar el módulo pickle: Para comenzar, importamos el módulo pickle en nuestro código Python.

```
import pickle
```

Archivos

Binarios - Serialización

2- Serialización de objetos: La serialización se realiza mediante la función pickle.dump(), que toma dos argumentos: el objeto que deseamos serializar y el archivo binario en el que deseamos almacenarlo. Aquí hay un ejemplo:

```
data = [1, 2, 3, 4, 5]  # Objeto a serializar

# Serializar el objeto y guardarlo en un archivo binario
with open('data.bin', 'wb') as file:
    pickle.dump(data, file)
```

Aclaración: NO es posible utilizar el método dump() del módulo pickle directamente con un file descriptor abierto en modo 'ab' (agregar datos a un archivo binario existente).

Archivos

Binarios - Deserialización

Deserialización de objetos: La deserialización se realiza mediante la función pickle.load(), que toma el archivo binario como argumento y devuelve el objeto original. Aquí hay un ejemplo:

```
# Deserializar el objeto desde el archivo binario
with open('data.bin', 'rb') as file:
    loaded_data = pickle.load(file)

print(loaded_data)  # Imprimir el objeto deserializado
```

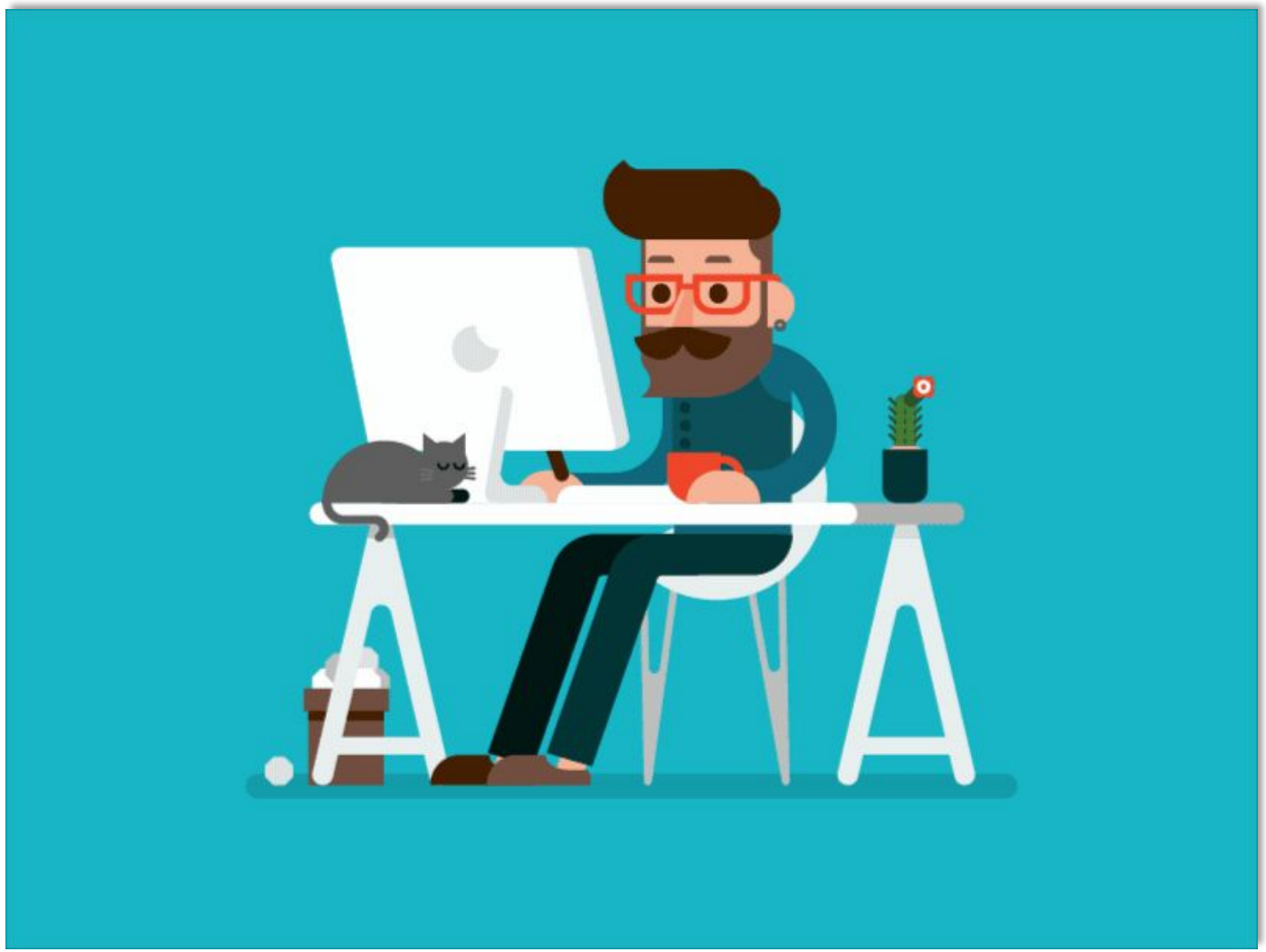
Archivos

Binarios - Pickle module - Conclusiones

Es importante destacar que al utilizar pickle, debes tener en cuenta las siguientes consideraciones:

- La serialización y deserialización con pickle solo funcionarán correctamente si tanto el proceso de escritura como el de lectura se realizan con la misma versión de Python y las mismas clases o estructuras de datos definidas.
- Ten en cuenta que pickle no es seguro contra datos maliciosos o de fuentes no confiables. Solo debes utilizarlo para objetos en los que confíes.
- Con el módulo pickle, puedes persistir y recuperar objetos complejos, como listas, diccionarios, instancias de clases y más, lo que te brinda una forma conveniente de trabajar con datos estructurados en tus aplicaciones.

Gracias



Fuente: https://www.programiz.com/python-programming/