Argenitna Programa 4.0

Universidad Nacional de San Luis

Desarrollador Python

Lenguaje de Programación Python: Archivos y Módulos

Autor:

Dr. Mario Marcelo Berón

UNIDAD 5

LENGUAJE DE PROGRAMACIÓN PYTHON: ARCHIVOS Y MÓDULOS

5.1. Archivos

En muchas situaciones de la vida cotidiana es necesario que la información que se ingresa y/o produce un sistema no se pierda cuando el programa deja de ejecutar. Para realizar esta actividad los sistemas deben almacenar la información en almacenamiento secundario (discos rígidos, pendrive, etc). Para éste fin, los lenguajes de programación proporcionan un conjunto de sentencias que permiten que el programador almacene la información en un *archivo*.

Archivo

Un sector de un dispositivo de memoria secundaria destinado a almacenar información relacionada. Se utiliza para almacenar permanentemente datos en una memoria no volátil.

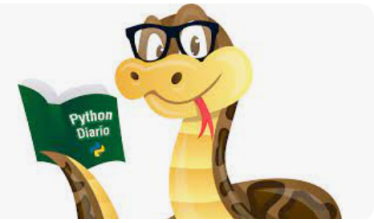
En las secciones siguientes se describen las operaciones principales

que se pueden realizar con archivos.

5.1.1. Operaciones con Archivos

Para poder trabajar con archivos en general se deben aplicar los siguientes pasos:

1. Abrir el archivo.
2. Procesar el archivo.
3. Cerrar el archivo.



5.1.2. Abrir un Archivo

Para abrir un archivo se utiliza la sentencia *open(archivo, modo)*[1] esta sentencia recibe como entrada básicamente dos parámetros:

- El primero de ellos es el nombre del archivo.
- El segundo el modo.

Respecto del primer parámetro se coloca el nombre del archivo directamente o bien el paso con el nombre incluido[2]. En lo que respecta al modo se puede decir que existen diferentes opciones:

- " r": permite abrir el archivo en modo lectura se produce un error si el archivo no existe.
- " a": permite abrir el archivo para incorporar nuevos elementos al final del archivo. En caso de que el archivo no existe se crea.

[1]Esta es la verisión más sencilla de *open*. El método tiene más parámetros pero su descripción está fuera del alcance de este curso.

[2]Cabe destacar que existen librerías que permiten especificar pasos de los archivos de forma tal que sea independiente del sistema operativo. Este tipo de librerías están fuera del alcance de este curso.

- "w": permite abrir el archivo para escribir. Cuando se escribe borra el archivo en su totalidad. En caso de que el archivo no exista se crea.
- "x": crea el archivo y produce un error si el archivo existe.

Ejemplo de Apertura de Archivo

```
f=open("archivoDeDemostración")
f=open("archivoDeDemostración","a")
f=open("archivoDeDemostración","w")
```

5.1.3. Cerrar un Archivo

Para cerrar un archivo se utiliza la sentencia *objetoArchivo.close()*. Es una buena práctica de programación cerrar los archivos abiertos de forma tal de evitar errores cuya detección es compleja.

Ejemplo de Procesamiento de Archivo

```
#Abrir el archivo
f=open("Datos.txt")
#Procesar el archivo
print(f.read())
#Cerrar el archivo
f.close()
```

5.1.4. Procesamiento de un Archivo

El procesamiento del archivo difere de acuerdo a la aplicación. Sin embargo, básicamente el archivo se tiene que *abrir*, *leer o escribir o leer y escribir (junto con otras tareas)* y *cerrar*. En las subsecciones siguientes se explica como se realiza la lectura y escritura de un archivo.

Lectura de Archivo

Para leer un archivo se utiliza el método *read()* por defecto este método lee el contenido del archivo en su totalidad aunque también se pueden especificar, en el caso de los archivos de texto, cuántos caracteres se desean leer o la cantidad de bytes en el caso de un archivo binario. Para leer un archivo se deben seguir los siguientes pasos:

1. Abrir el archivo en modo lectura.
2. Leer el contenido.
3. Procesar el archivo.
4. Cerrar el archivo

El cuadro que se muestra a continuación muestra un ejemplo de apertura y procesamiento de un archivo.

Ejemplo de Lectura de un Archivo

```
#Abrir un archivo en modo lectura
f=open("Datos.txt","r")
#Leer el contenido
dato=f.read()
#Realizar algún procesamiento con el contenido
print(len(dato))
#Cerrar el archivo
close(f)
```

Se debe tener presente que el procesamiento al ser variable puede requerir el uso de sentencias especializadas de lectura. Por ejemplo, en lugar de leer todo el archivo con una sola sentencia, se puede necesitar leer algunas líneas o datos del mismo. A continuación, se mencionan algunas sentencias que generalmente se utilizan en la lectura de archivos:

- *archivo.read(n)*: lee todo el contenido de archivo. Si se especifica n retorna n bytes de archivo en formato de string.
- *archivo.readline(n)*: lee una línea en forma de string desde archivo. Si se especifica n este indica el número de bytes que se leerán. Se debe tener en cuenta que si n excede la longitud de la línea, la función no considera la próxima línea.

Ejemplo de Procesamiento de Archivo

Si por ejemplo se desea imprimir una línea del Datos.txt el cual se supone que existe, las sentencias son las siguientes:

```
#Abrir el archivo
f=open("Datos.txt")
#Procesar el archivo
print(f.readline())
```

- *archivo.readlines()*: lee todas las líneas de archivo y las retorna en una lista.

Ejemplo de Procesamiento de Archivo

```
f = open("Datos.txt", "r")
print(f.readlines())
```

Escritura de un Archivo

Para escribir en un archivo se utiliza el método *write()*. Para escribir en un archivo se deben seguir los siguientes pasos[3]:

1. Abrir el archivo en modo escritura (o append).
2. Capturar el dato que se desea escribir.

[3]Estos pasos son ilustrativos y están pensados para que el lector pueda comprender en términos generales el proceso. No obstante, los pasos que se deben seguir dependerán de la aplicación.

3. Escribir el dato en el archivo.

4. Cerrar el archivo

Ejemplo de Escritura de un Archivo

```
#Abrir un archivo en modo escritura
f=open("Datos.txt","w")
#Escribir el contenido en el archivo
contendio="Hola Mundo"
dato=f.write(contenido)
#Cerrar el archivo
close(f)
```

Si el programador desea escribir más de una cadena en el archivo puede utilizar el método *archivo.writelines(l)* el cual recibe como parámetro una lista de strings.

Ejemplo de Escritura de un Archivo

```
#Abrir un archivo en modo escritura
f=open("Datos.txt","w")
#Escribir el contenido en el archivo
l=["Dato-1","Dato-2","Dato-3"]
dato=f.writelines(l)
#Cerrar el archivo
close(f)
```

5.1.5. Archivos Binarios

Los archivos binarios siguen la misma regla que lo explicado con anterioridad, lo cual se intentó hacer lo más general posible. No obstante, los ejemplos utilizados fueron realizados con archivos de texto. La operaciones

de lectura y escritura se realizan de manera muy similar utilizando las sentencias read y write y abriendo los archivos con el modo binario (rb o wb etc). Actualmente se trabaja poco con archivos binarios. Esto se debe a que se tiene que tener presente muchos parámetros como por ejemplo el formato que van a tener los datos, las conversiones a y desde formato binario lo cual es requerido para que los datos de usuario puedan ser almacenados en disco y luego recuperados en un formato interpretable por el usuario. Es decir trabajar con archivos binarios con las primitivas brindadas en este curso es hacerlo a bajo nivel, lo cual, en la actualidad, no se utiliza, al menos en los sistemas tradicionales, porque lo que se requiere es productividad e interoperabilidad. Por esta razón para usar este tipo de archivo, cuando la situación así lo requiera, Python provee el módulo *Pickle*, el cual tiene funciones de alto nivel que le simplifican la tarea al programador. *Pickle* hace sencillo el almacenamiento y recuperación de estructuras de datos. El estudio de este módulo está fuera del alcance de este curso pero se lo menciona para que el interesado en trabajar con archivos binarios pueda estudiar el módulo y usarlos con facilidad.

5.1.6. Ejercicios

Ejercicio 1: Escriba un programa que permita que el usuario ingrese el nombre de un archivo el programa lee el contenido y lo imprime por pantalla.

Ejercicio 2: Escriba un programa que permita que el usuario ingrese el nombre de un archivo lo abra y escriba una cadena de carácteres ingresada por el usuario. Luego el programa debe cerrar el archivo.

Ejercicio 3: Escriba un programa que permita que el usuario agrege la cadena "Fin de Archivo" al final de un archivo cuyo nombre es ingresado por teclado.

Ejercicio 5: Escriba un programa que permita que el usuario cree un archivo y le grabe n cadena de caracteres.

Ejercicio 6: Escriba un programa que permita que el usuario ingrese el nombre de un archivo. Luego el programa debe leer el archivo y contabilizar para cada palabra que aparece en el archivo la cantidad de veces que la misma se repite. Luego el programa debe imprimir por pantalla las cinco palabras que más se repitieron en el archivo.

Ejercicio 7: Escriba un programa que pase a mayúsculas todas las letras que contiene un archivo a minúsculas. El nombre del archivo es ingresado por el usuario.

Ejercicio 8: Escriba un programa que cuente la cantidad de: mayúsculas, minúsculas y números que contiene un archivo ingresado por el usuario.

Ejercicio 9: Escriba un programa que sume los números que se encuentran en un archivo de texto cuyo nombre es ingresado por el usuario.

Ejercicio 10: Escriba un programa que invierta el orden de las líneas de un archivo de texto.

Ejercicio 11: Escriba un programa que invierta las palabras de un archivo de texto.

Ejercicio 12: Escriba un programa que invierta el orden y a su vez las palabras de un archivo de texto.

5.2. Módulos y Paquetes

En esta sección se describen dos componentes fundamentales para el desarrollo de aplicaciones escritas en el lenguaje de programación Python como lo son los *módulos* y *paquetes*.

5.2.1. Módulos

Un módulo es un archivo .py. Todos los programas que se escriben en Python son módulos y programas. La principal diferencia es que los programas están pensados para que se ejecuten, mientras que los módulos

tienen como finalidad ser importados por otros programas. El cuadro que se muestra a continuación visualiza la sintaxis que utiliza Python para importar módulos.

Sintaxis para Importar un Módulo

```
import importable
import importable1, importable2, ..., importablen
import importable as nombre_preferido
```

En este contexto *importable* puede ser un módulo o un paquete. En este último caso cada parte está separada por ".".

La tercera opción permite que el programador le asigne un nombre personalizado al módulo o paquete. Esta posibilidad puede conducir a conflictos de nombres y, por lo tanto, no es muy utilizada. El renombre puede ser útil cuando se desea trabajar con dos implementaciones distintas de un mismo módulo o paquete. Es común colocar los *import* al comienzo del programa después de la línea que indica el intérprete que ejecutará el programa (*shebang line*) y después de la documentación. También se recomienda colocar primero los *import* de los módulos la librería estándar, luego los módulos de las librerías de terceros y finalmente los módulos de quién escribe el programa. El cuadro que se muestra a continuación ejemplifica el uso de *import*.

Ejemplo de Importación de Módulos

```
import collections
import os.path
```


Otras formas de importar módulos

```
from importable import objeto as nombre-preferido
from importable import objeto-1,objeto-2,...,objeto-n
from importable import (objeto-1,objeto-2,objeto-3, \
    objeto-4,objeto-5,objeto-6,...,objeto-n)
from importable import *
```

Estas formas de importar módulos pueden causar conflictos de nombres porque permite el acceso directo a las funciones, variables y métodos.

Comentario Importante

La sintaxis:

from importable **import** *

Indica que se importe todo aquello que no sea privado es decir todo aquello cuyos nombres no comiencen con guiones bajos.

Comentario Importante

La sintaxis:

from .... **import** ....

permiten referenciar a los objetos sin calificación.

Ejemplo de Importación de Módulos

```
from decimal import *
/*Acceso sin calificación*/
d=Decimal("10101")
```

5.2.2. Paquetes

Un paquete es simplemente un directorio que contiene un conjunto de módulos y un arhcivo llamado __init__.py. En consecuencia para crear un módulo se necesita:

1. Crear un directorio.
2. Colocar los archivos .py que formarán parte del paquete en el directorio creado con anterioridad.
3. Crear el archivo __init__.py y almacenarlo en el directorio creado.

Una vez realizadas las tareas descritas se puede importar el módulo que se desee.

Ejemplo de Creación de un Paquete

Suponga que se tiene los archivos: Bmp.py, Jpeg.py, Png.py, Tiff.py, Xpm.py. Se desea crear el paquete Imágenes el cual contendrá todos los archivos .py mencionados con anterioridad, entonces:

- Se crea el directorio Imágenes.
- Se almacenan en Imágenes los archivos: Bmp.py, Jpeg.py, Png.py, Tiff.py, Xpm.py.
- Se crea en el directorio Imágenes el archivo __init__.py vacío.
- Una vez realizadas las tareas antes mencionadas, un programa que utilice el paquete puede importarlos de la siguiente manera:

```
import Imágenes.Bmp
#Se asume que Bmp.py define load
imagen= Imágenes.Bmp.load("MiImagen.bmp")
import Imágenes.Jpeg as Jpeg
#Se asume que Jpeg.py define load
imagen2= Imágenes.Jpeg.load("MiImagen2.jpeg")
```

También se pueden realizar importaciones utilizando *from... import* como se muestra a continuación:

```
from Imágenes import Png
image3 = Png.load("Imagen3.png")
```

Si el programador desea importar todos los paquetes con una sentencia from paquete import * entonces debe definir la lista __all__ en el módulo __init__.py de la siguiente manera:

```
__all__=["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

Una vez realizada esta tarea se puede hacer:

```
from Imágenes import *
```

dicha sentencia incluirá todos los formatos de imágenes.

Comentario Importante

Se pueden crear submódulos de la misma forma en la que se crean módulos. En este caso se deberá crear un subdirectorio del directorio donde se encuentra el módulo padre y luego se copian los archivos .py que forman parte del submódulo y se crea el archivo __init__.py vacío o con la variable __all__ definida.

5.2.3. Ejercicios

Ejercicio 1: Implemente un módulo que contenga las siguientes funciones:

1. cantidadDeLíneas la función recibe como parámetro el nombre de un archivo. La función retorna como resultado la cantidad de líneas que tiene el archivo.
2. números la función recibe como parámetro el nombre de un archivo y retorna como resultado una lista de números los cuales se encuentran en el archivo recibido como parámetro.
3. carácteres la función recibe como parámetro el nombre de un archivo y retorna como resultado la cantidad de caracteres que tiene el archivo.

Ejercicio 2 : Escriba un programa que:

1. Permita que el usuario ingrese el nombre de un archivo.
2. Calcule la cantidad de: líneas , números y carácteres que tiene el archivo.
3. Almacene en el archivo resultado los datos calculados en el ítem anterior.

Nota: Para resolver este ejercicio importe el módulo definido en el ejercicio anterior.

Ejercicio 3: Escriba funciones que permitan calcular el área y perímetro de las siguientes figuras geométricas: Cuadrado, Triángulo, Rectángulo, Circunferencia. Luego cree el paquete FigurasGeométricas el cual contiene todas las funciones que definición para : Cuadrado, Triángulo, Rectángulo, Circunferencia.

Ejercicio 4: Escriba un programa que pida la base y la altura de un triángulo y de un rectángulo y a partir de dichos datos calcule el área y perímetro de esas figuras geométricas.

Nota: Importe el paquete definido en el ejercicio anterior.

5.3. Funcionalidades útiles para Archivos

En esta sección se presentan algunas funcionalidades útiles para trabajar con archivos y directorios que Python posee. El lenguaje posee muchas más, no obstante las aquí mencionadas son muy utilizadas en el desarrollo de aplicaciones.

5.3.1. Construcción with...as

En Python se puede usar *with ... as* para trabajar con archivos. De esta forma el archivo se cierra automátiamente cuando sale del bloque *with...as*.

Ejemplo de with...as

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

5.3.2. Verificar si un Arhivo Existe

El módulo del sistema operativo proporciona múltiples funciones para interactuar con el sistema operativo.

En realidad existen varias formas de verificar si un archivo existe. Una forma sin usar librerías especializadas es la que se muestra a continuación.

Apertura de un Archivo con Excepciones

```
try:
  file = open("Archivo.txt")
  print(file) # Procesamiento
  file.close()
except FileNotFoundError:
  print("El archivo no existe")
  exit()
```

En este caso la sentencia que intenta abrir un archivo se encuentra encerrada en un bloque *try-except*. En el caso que *Archivo.txt* no exista se dispara una excepción *FileNotFoundError* el cual es manejada en el bloque *except*.

5.3.3. El Módulo os

El módulo *os* proporciona un conjunto muy útil de funciones que permiten interactuar con el sistema operativo. Este módulo proporciona funciones útiles para chequear la existencia y la no existencia de rutas, archivos y directorios. A continuación se muestran ejemplos de esas operaciones.

El ejemplo que se muestra a continuación ejemplifica la existencia de rutas de acceso.

Ejemplo de path

```
import os

os.path.exists('archivo.txt')
True

os.path.exists('directorio')
True

os.path.exists('paso')
False
```

Si se desea probar la existencia de un archivo (no un directorio), se utiliza el método *isfile ()*.

Ejemplo de isfile

```
import os

os.path.isfile('archivoPrueba.txt')
True

os.path.isfile('directorioPrueba/')
False

os.path.isfile('archivoNoExiste.txt')
False

os.path.isfile('directorioPrueba/otroArchivo.txt')
True
```

Para el chequeo de existencia de directorios se puede usar el método

isdir().

Ejemplo de isdir

```
import os

os.path.isdir('archivoPrueba.txt')
False

os.path.isdir('directorioPrueba')
True

os.path.isdir('otroArchivo.txt')
False
```

El método *chdir()* permite cambiar de directorio y recibe como argumento un cadena de caracteres que indica la ruta al directorio donde se desea cambiar.

Ejemplo de chdir

```
import os
os.chdir("./Vídeos")
```

El método *mkdirs()* permite cambiar de directorio y recibe como argumento un cadena de caracteres que indica la ruta del directorio que desea crear. Este método crea todos los directorios intermedios si los mismos no se encuentra en el dispositivo de almacenamiento secundiario.

Ejemplo de mkdirs

```
import os
os.mkdirs("/Vídeos/Acción")
```

El método *mkdir()* permite cambiar de directorio y recibe como argumento un cadena de caracteres que indica el direcotrio que se desea crear. Este método a diferencia del anterior no crea las carpetas intermedias.

Ejemplo de mkdir

```
import os
os.mkdir("/Acción")
```

Comentario Importante

Existen muchas más funcionalidades provistas por el módulo os. En este documento solo se han mencionado algunas de las más utilizadas.

5.3.4. El Módulo Pathlib

El módulo pathlib proporciona diferentes funciones para trabajar con rutas. El trabajo con rutas es fundamental cuando se hacen programas que requieren persistencia de datos. Python provee funcionalidades que ayudan al programador en este aspecto.

El método *cwd()* retorna como resultado el directorio actual.

Obtención del Directorio Actual

```
from pathlib import Path
Path.cwd()
```

Los siguientes cuadros muestran ejemplos de como se puede comprobar la existencia de rutas, archivos y directorios.

Comprobación de si una Ruta Existe

```
from pathlib import Path

Path('archivoPrueba.txt').exists()
True

Path('archivoNoExiste.txt').exists()
False

Path('directorioPrueba').exists()
True
```

Comprobación de si una Ruta apunta a un Archivo

```
Path('archivoPrueba.txt').is_file()
True

Path('directorioPrueba').is_file()
False
```

Comprobación de si una Ruta apunta a un Diretorio

```
Path('archivoPrueba.txt').is_dir()
False

Path('directrioPrueba').is_dir()
True
```


Comentario Importante

Existen muchas más funcionalidades provistas por el módulo *pathlib*. En este documento solo se han mencionado algunas de las más utilizadas.