

Archivos

Unidad 5

Apunte de cátedra

Pensamiento Computacional (90)
Cátedra: Balbiano

.UBA XXI

1. Archivos

Las Estructuras de Datos **persistentes** se conocen genéricamente como **archivos**. Que una estructura de datos sea persistente significa que perdura más allá de la vida de quien la genera. Claramente, un archivo debe guardarse en algún dispositivo que administre datos **no volátiles** (es decir, que no desaparezcan cuando se apaga el equipo). Por ello los archivos (en todos sus formatos y organizaciones) son las Estructuras de Datos que se emplean para compartir información entre usuarios, procesos, equipos.

Si generamos un grupo de datos corriendo un programa, ya sea que se ingresen por teclado o se generen como resultado de transformaciones, y deseamos acceder a ellos en otro momento, o desde otro programa, necesitamos guardarlos en una Estructura capaz de sobrevivir a la ejecución corriente (proceso activo). De esta manera, se pueden rescatar posteriormente y seguir trabajando con ellos. La manera de conservarlos para más tarde es grabándolos como archivos, con un nombre, una extensión, y claramente una organización interna conveniente.

Todos los lenguajes de Programación admiten la interacción con archivos. Cada lenguaje provee una colección de sentencias a tales efectos.

La diversidad de tipos de archivos es enorme. Cada uno de ellos tiene sus particularidades y requiere ciertas formas específicas en su manejo. Además, dependiendo de su morfología, el tipo de datos que maneja y sus dimensiones puede requerir algoritmos especiales para su uso y manipulación.

En este apunte simplemente daremos algunas directivas básicas para que puedas trabajar con tipos simples (y disponibles en cualquier lado) de archivos, y abarcaremos las operaciones esenciales que te requerirá su uso.

Lo primero que debemos recordar es que estamos haciendo programas que correrán en una **Arquitectura Von Neumann**. Por lo tanto, para que los datos puedan ser procesados, primero deben ser alojados en **Memoria Interna (MI)**. Y para que se puedan sacar datos y resultados (de la computadora) hacia dispositivos externos (discos, redes, etc.), donde estará alojado nuestro archivo, también deben pasar por la **MI**.

Así que no hay posibilidad de trabajar con los datos que están en un archivo si no los copiamos a MI, si los modificamos y deseamos guardar las modificaciones, deberemos bajar la versión que tengamos en MI al archivo.

Normalmente, los archivos contienen grandes volúmenes de datos, por ello, copiar un archivo completamente en MI puede ser una tarea muy costosa, sino imposible. Así que una solución es copiar los datos por partes. A medida que necesitamos información, la vamos subiendo a la MI. Y a medida que modificamos información, resguardamos una copia en el archivo.

El Sistema Operativo reserva una zona de memoria y establece un canal de comunicación entre el archivo y el programa. Genéricamente, estos miniespacios de intercambio entre un programa y un archivo se denominan **buffers**.

El SO lo que hace es implementar un sistema de tráfico de intercambio, lectura y escritura en bloque. Es decir, si el programa necesita un dato en particular, no se trae sólo ese dato, sino que se trae todo un bloque de información dentro del cual se incluye el dato. Esto es porque hacerlo por un sólo dato, resulta con un costo operativo muy alto; y porque se sobreentiende que si necesité el dato X del archivo, es probable que vaya a necesitar los datos cercanos al mismo pronto.

Abrir un archivo se traduce en el SO como pedirle que defina un buffer. Este va a tener el nombre del archivo, dónde se ubica en la estructura de directorios, qué haremos con él, y otras cosas y datos relevantes.

Un mismo programa puede abrir simultáneamente todos los archivos que necesite, por lo que necesita una forma de poder diferenciarlos. Así, se le asigna un **nombre interno (alias)** a cada archivo para referenciarlo con ese nombre dentro del programa.

1.1. Primitivas Generales

Para abrir un archivo en Python disponemos de la función **open()**. Esta función le solicita al SO que establezca el vínculo de trabajo y nos devuelva la dirección de inicio del buffer del archivo. Es importante que guardemos esa dirección en una variable; que será el nombre interno del archivo desde ese momento. Es decir que, el nombre interno de nuestro archivo es una etiqueta que apunta al inicio del buffer.

Nota:

Una vez abierto, sólo referenciamos al archivo por su nombre interno o alias.

1.1.1. Sintaxis de open:

```
alias = open(nombre_completo_archivo,modo)
```

Debe indicarse el nombre y extensión con el que fue guardado; así como localización completa, o relativa en los directorios (el inefable path). El modo se puede aclarar, pero tiene un valor por default de lectura.

Ejemplo:

```
archivo = open("texto.txt")
```

Una vez que **terminamos de usar** el archivo es importante realizar la **operación de cierre**. Ya que esta operación libera el espacio de buffer y completa el proceso de grabado, si hay algo pendiente. Así como el SO trae más información de la que le pedimos (por el abloqueamiento) sin preguntarnos, tampoco se molesta en salir corriendo a grabar un dato cuando se lo solicitamos, en realidad junta varios encargos de grabado antes de realizar la tarea. El cierre del archivo lo obliga a grabar todo lo que haya quedado pendiente. Para cerrar un archivo disponemos del método **close()**.

1.2. Modos de Apertura

Los modos (para qué será empleado el archivo dentro del programa) pueden ser los siguientes (debemos emplear el símbolo para especificar la acción):

Modo	Lee	Escribe	Existencia de Archivo	Posición Inicial de Lectura	Posición Inicial de Escritura
r	✓	-	Si el archivo no existe, da error.	Inicio	-
w	-	✓	Si el archivo no existe, lo crea. Si existe, le borra todo el contenido.	-	Inicio
a	-	✓	Si el archivo no existe, lo crea.	-	Final
r+	✓	✓	Si el archivo no existe, da error.	Inicio	Depende
w+	✓	✓	Si el archivo no existe, lo crea. Si existe, le borra todo el contenido.	Inicio	Depende

1.2.1. Modo **r** (read = leer)

Apertura

Si no aclaramos nada, el modo de apertura es **r**:

```
archivo = open("archivo.txt")
```

Pero también lo podemos aclarar, y es lo mismo:

```
archivo = open("archivo.txt", "r")
```

Lectura

En la materia vemos 3 métodos de lectura:

- **read**, que lee todo el archivo junto y lo devuelve en un string
- **readlines**, que lee cada una de las líneas y las devuelve como elementos de una lista. Cada línea contiene el caracter especial `\n` de forma explícita, que indica el cambio de línea.
- **readline**, que lee de a una línea cada vez que se llama a la función. Cada línea contiene el caracter especial `\n` pero no está de forma explícita, sino implícita. Es decir, se imprime con un espacio de nueva línea pero no vemos el caracter.

1.2.2. Modo **w** (**write** = **escribir**)

Apertura

```
archivo = open("archivo.txt", "w")
```

Escritura

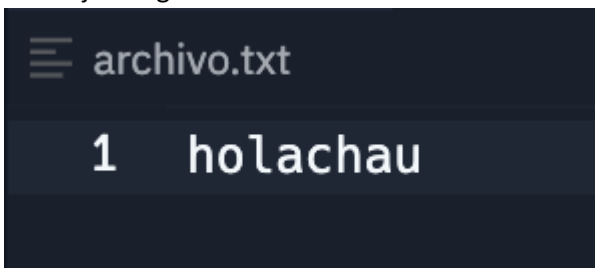
En la materia vemos 2 métodos:

- **write** recibe un string y lo escribe en el archivo. Si no le agregamos el caracter especial `\n`, va a imprimir una palabra detrás de otra.

Por ejemplo, el siguiente código:

```
archivo = open("archivo.txt", "w")
archivo.write("hola")
archivo.write("chau")
archivo.close()
```

Nos deja el siguiente contenido en el archivo:



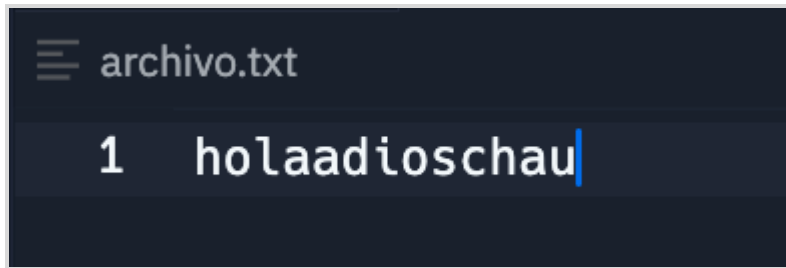
```
≡ archivo.txt
1 holachau
```

- **writelines** puede recibir un string o una lista de strings. De todas formas, si no le agregamos el caracter especial `\n`, va a imprimir una palabra detrás de otra.

```
archivo = open("archivo.txt", "w")
archivo.writelines(["hola", "adios"])
```

```
archivo.writelines("chau")  
archivo.close()
```

El código de arriba nos deja el siguiente contenido en el archivo:

A screenshot of a text editor with a dark background. The title bar at the top says 'archivo.txt'. The editor shows a single line of text: '1 holaadioschau'. The cursor is positioned at the end of the line, after the 'u' in 'schau'.

1.2.3. Modo **a** (append = agregar)

Apertura

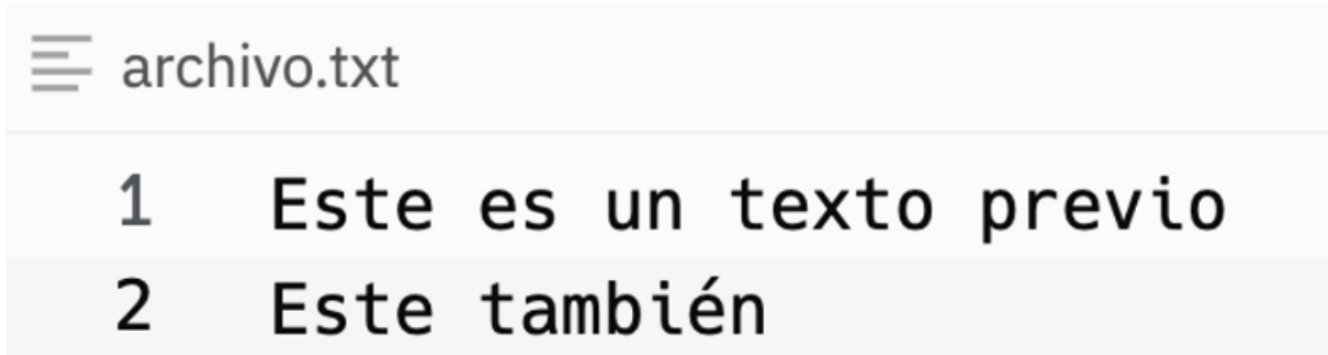
```
archivo = open("archivo.txt", "a")
```

Escritura

Se usan los mismos métodos de escritura que con **w**, la diferencia es que el modo **a** siempre agrega el contenido al final del archivo.

1.2.4. Problema de Escritura con Append

Si nuestro archivo no tiene una línea vacía extra, como acá:

A screenshot of a text editor with a light gray background. The title bar at the top says 'archivo.txt'. The editor shows two lines of text: '1 Este es un texto previo' and '2 Este también'. The lines are numbered on the left.

Lo que va a pasar es que nuestro “indicador de posición” en el archivo va a estar justo al final de la fila 2. Entonces, al agregar contenido con este código:

```
archivo = open("archivo.txt", "a")  
archivo.writelines(["hola", "adios"])
```

```
archivo.writelines("chau")  
archivo.close()
```

Vamos a obtener este resultado:

≡ archivo.txt

```
1 Este es un texto previo  
2 Este también hola adioschau
```

Posible Solución 1

Una posible solución es, previo a empezar a escribir o dentro mismo del primer string, agregar un `\n`:

```
archivo = open("archivo.txt", "a")  
archivo.writelines(["\nhola", "adios"])  
archivo.writelines("chau")  
archivo.close()
```

Entonces obtenemos:

≡ archivo.txt

```
1 Este es un texto previo  
2 Este también  
3 hola adioschau
```

Posible Solución 2

Otra es procurar que siempre todo lo que se agregue al archivo tenga un `\n` al final. De esta forma siempre vamos a terminar todas las líneas con un punto y aparte.

1.2.5. Modo **r+**

Apertura

```
archivo = open("archivo.txt", "r+")
```

Este modo y el ``w+`` son muy especiales y hay que tener cuidado.

Tenemos tres posibles comportamientos para ``r+``

Comportamiento 1: Sólo Lectura

Si sólo leemos, el modo `r+` se va a comportar como `r`



The screenshot shows a Python IDE with a file named `main.py` open. The code in the editor is:

```
1 archivo = open("archivo.txt", "r+")
2 lineas = archivo.readlines()
3 print(lineas)
4 archivo.close()
```

On the right, there is a console window showing the output of the `print` statement: `['Este es un texto previo\n', 'Este también\n']`. Below the console, there is a file explorer showing the contents of `archivo.txt`:

```
1 Este es un texto previo
2 Este también
3
```

Comportamiento 2: Sólo Escritura

El modo `r+` cuando sólo escribe, empieza a escribir desde el comienzo del archivo. Es decir, va a pisar, carácter a carácter, el contenido previo de cada línea.



The screenshot shows a Python IDE with a file named `main.py` open. The code in the editor is:

```
1 archivo = open("archivo.txt", "r+")
2 archivo.write("Texto nuevo que va a pisar todo")
3 archivo.close()
```

On the right, there is a console window showing the output of the `write` statement: `['Texto nuevo que va a pisar todo\n']`. Below the console, there is a file explorer showing the contents of `archivo.txt`:

```
1 Texto nuevo que va a pisar todombién
2
```

Comportamiento 3: Lectura y Escritura

Este es el punto complicado.

Cuando leemos y escribimos en modo ``r+``, lo que va a pasar es que, independientemente de qué hagamos antes, lo que escribamos en el archivo va a ir siempre al final.

Esto es porque, una vez que leemos del archivo, el “puntero de posición” para escribir pasa a estar inmediatamente al final.

Es como si le estuviésemos diciendo al modo de apertura ``r+``: “ojo que también quiero quedarme con lo que el archivo ya tiene”.

Al mismo tiempo, si le pedimos leer el contenido del archivo, siempre nos va a leer sólo el contenido que teníamos antes de abrirlo. Es decir, si escribo antes y después leo, igualmente nuestra nueva línea no va a estar. Esto es porque los ``write`` se guardan en el momento en el que se cierra el archivo.

Ejemplo de lectura y luego escritura:

Como vemos, le pedimos leer sólo 1 línea, e incluso así cuando escribí, escribió al final del archivo.

```

main.py
1  archivo = open("archivo.txt", "r+")
2  lineas = archivo.readline() # lee una sola línea
3  print(lineas)
4
5  archivo.write("Línea nueva")
6  archivo.close()
7
8
9  |

```

Este es texto previo

archivo.txt x notas.txt x arch1Par.txt x +

archivo.txt

```

1  Este es texto previo
2  Este también
3  Línea nueva

```

Ejemplo de escritura y luego lectura:

Lo mismo: escribimos una línea, que se agregó al final. Pero al momento de leer todas las del archivo, sólo nos devuelve aquellas que estaban en la copia de apertura original, no incluye la que agregamos.

```

main.py
1  archivo = open("archivo.txt", "r+")
2  archivo.write("Línea nueva")
3
4  lineas = archivo.readlines() # lee todas las líneas
5  print(lineas)
6
7  archivo.close()
8
9
10

```

['Este es texto previo\n', 'Este también\n']

archivo.txt x notas.txt x arch1Par.txt x +

archivo.txt

```

1  Este es texto previo
2  Este también
3  Línea nueva

```

1.2.6. Modo `w+`

Apertura

```
archivo = open("archivo.txt", "w+")
```

Como dijimos, este modo y el ``r+`` son muy especiales y hay que tener cuidado.

También tenemos tres posibles comportamientos para ``w+``

Comportamiento 1: Sólo Lectura

El modo sólo lectura para ``w+`` no funciona como uno pensaría, porque **lo primero que hace ``w+`` es borrar todo el archivo al abrirlo.**

Siempre empezamos con un archivo en blanco al abrir con ``w+`` (como pasa también con ``w``, porque si no existe, lo crea; y si existe, lo vacía).

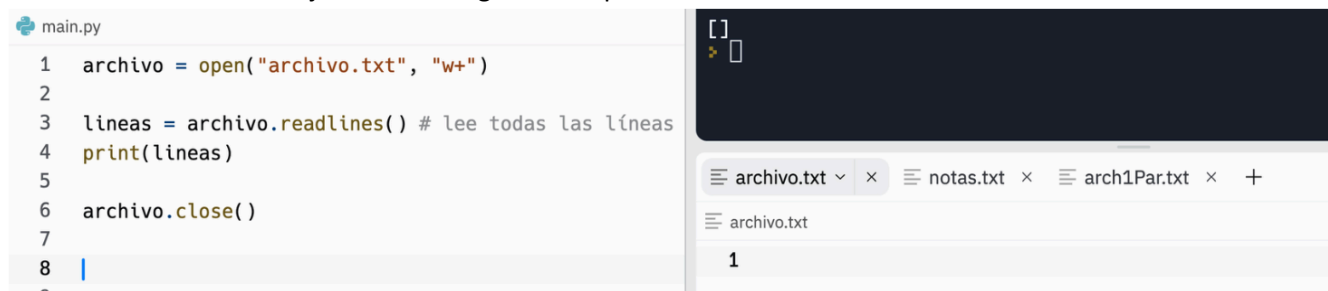
Este es el estado previo de nuestro archivo:



```
≡ archivo.txt

1  Línea 1
2  Línea 2
3
```

Y este es el resultado de ejecutar el código de la izquierda:



```
main.py
1  archivo = open("archivo.txt", "w+")
2
3  lineas = archivo.readlines() # lee todas las líneas
4  print(lineas)
5
6  archivo.close()
7
8  |
```

Output: []

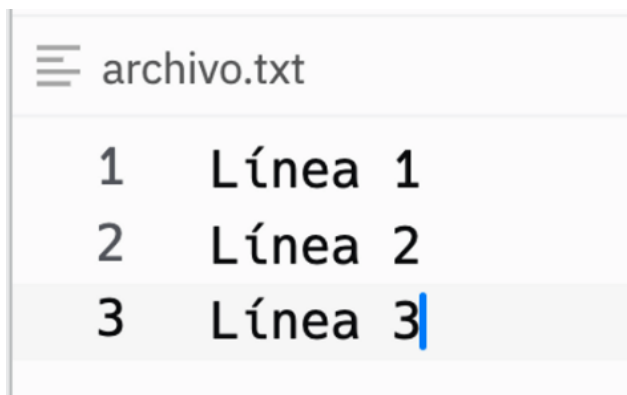
File Explorer: archivo.txt, notas.txt, arch1Par.txt, +

Preview: archivo.txt

Content: 1

Comportamiento 2: Sólo Escritura

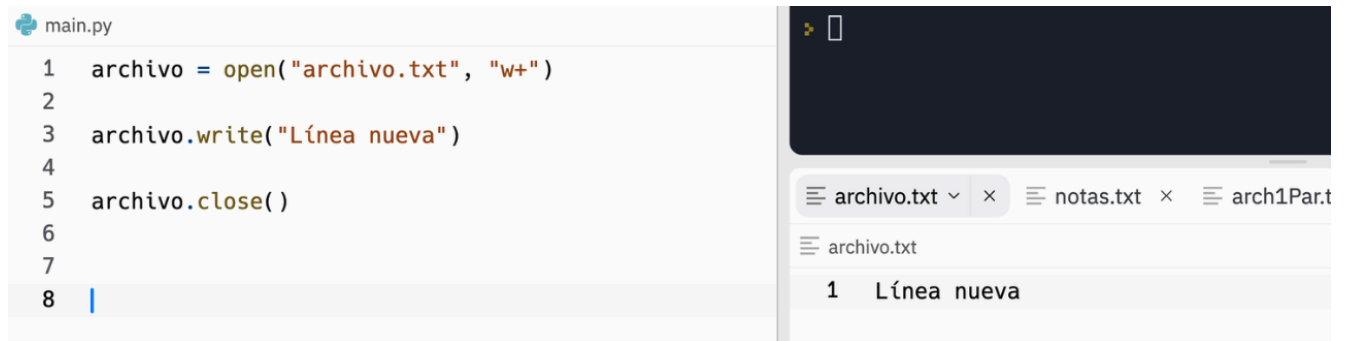
El estado previo del archivo es el siguiente:



```
≡ archivo.txt

1  Línea 1
2  Línea 2
3  Línea 3
```

Al abrirlo, se borra el archivo por completo. Luego, escribimos dentro del archivo vacío:



The screenshot shows a code editor with a file named `main.py`. The code is as follows:

```
1 archivo = open("archivo.txt", "w+")
2
3 archivo.write("Línea nueva")
4
5 archivo.close()
6
7
8
```

On the right, there is a terminal window showing the output of the script:

```
1 Línea nueva
```

Comportamiento 3: Lectura y Escritura

Este de nuevo es el punto complicado.

Primero lectura y luego escritura

Como al abrir el archivo, se vacía, siempre tratar de leer con **w+** antes de escribir no va a devolver nada.



The screenshot shows a code editor with a file named `main.py`. The code is as follows:

```
1 archivo = open("archivo.txt", "w+")
2
3 lines = archivo.readlines()
4 print(lines)
5
6 archivo.write("Línea nueva")
7
8 archivo.close()
9
10
```

On the right, there is a terminal window showing the output of the script:

```
1 Línea nueva
```

Primer escritura y luego lectura

Lo mismo, el archivo se borra al abrirlo. Por lo que el resultado es idéntico al caso anterior.

Una forma de lograr leer sobre el archivo que acabamos de escribir sería volver a posicionarnos con el comando **seek(0)** en el inicio del archivo, pero no es algo que vayamos a ver en la materia.

1.2.7. Otra forma de abrir archivos

Otra forma de abrir archivo sin la necesidad de usar **close** es usando el **with open as**:



The screenshot shows a code editor with a file named `main.py`. The code is as follows:

```
1 with open("archivo.txt", "r") as archivo:
2     lines = archivo.readlines()
3     print(lines)
4
```

Donde todo el código dentro del bloque **with** va a mantener el archivo abierto. Saliendo del mismo, cualquier intento de acceder al archivo va a devolver un error.

1.3. Tipos de Archivo

1.3.1. Archivos Planos – Archivos de Texto (txt)

Un archivo plano, también conocido como archivo de texto plano o archivo de texto simple, es un tipo de archivo informático que almacena datos de manera estructurada pero sin utilizar ningún formato de codificación o estructura de datos compleja.

Los archivos planos llevan la extensión “.txt” al final. Por ejemplo: “archivo.txt”.

Veamos este ejemplo:

- **Agrega oraciones en arch1.txt**

```
vaca_txt=open('arch1.txt','r+')
t = vaca_txt.readline()
print(t)
```

```
t = input('Ingresa un texto con vaca: ')
while (t.lower()).count('vaca')==0:
```

```
    t = input('Ingresa un texto con vaca: ')
vaca_txt.write(t+'\n')
vaca_txt.close()
```

```
vaca_txt=open('arch1.txt')
todas = vaca_txt.readlines()
for linea in todas:
    print(linea.strip('\n'))
vaca_txt.close()
```

Nota:

En este ejemplo tenemos en la misma carpeta del programa un archivo: **arch1.txt** que contiene al momento de correr el programa lo siguiente:

```
La vaca LOCA
VACAYENDO gente al baileee
siga la vaca
vacaciones en faMilia...
es lila la vaca de MILKA
```

Por lo que, si ejecutamos el programa, tendremos una salida como esta:

```

>_ Console × Output × +

La vaca LOCA

Ingresá un texto con vaca: señora vaca, señora vaca
La vaca LOCA
VACAYENDO gente al baileee
siga la vaca
vacaciones en faMIlia...
es lila la vaca de MILKA
  
```

Y el archivo **arch1.txt** contendrá ahora lo siguiente:

```

La vaca LOCA
VACAYENDO gente al baileee
siga la vaca
vacaciones en faMIlia...
es lila la vaca de MILKA
señora vaca, señora vaca
  
```

¿Qué hicimos en el programa?

Código	Efecto
# pruebaarch1.py	
<code>vaca_txt=open('arch1.txt','r+')</code>	Abre arch1.txt para lectura y escritura, con la codificación por defecto del SO y define nombre interno o alias vaca_txt
<code>t=vaca_txt.readline()</code>	Lee desde el archivo la primer línea (todos los caracteres hasta encontrar un <code>\n</code> y se guarda en t
<code>print(t)</code>	Muestra la línea
<code>t=input('Ingresá un texto con vaca: ')</code>	Pide al usuario un nuevo texto con la palabra vaca
<code>while (t.lower()).count('vaca') == 0:</code>	Valida que el texto incluya al menos vez
<code>t=input('Ingresá un texto con vaca: ')</code>	vaca
<code>vaca_txt.write(t+'\n')</code>	Graba en el archivo la línea leída
<code>vaca_txt.close()</code>	Cierra el archivo
<code>vaca_txt=open('arch1.txt')</code>	Reabre el archivo, pero ahora sólo lectura
<code>todas=vaca_txt.readlines()</code>	Carga la lista todas con las líneas del archivo. Cada línea es un elemento
<code>for linea in todas:</code>	Muestra el contenido de la lista quitando
<code>print(linea.strip('\n'))</code>	<code>\n</code> para evitar doble interlineado, uno sale naturalmente por el <code>print()</code>
<code>vaca_txt.close()</code>	Cierra el archivo

1.3.2. Archivos Planos – Comma Separated Values (csv)

Mirá un ejemplo de cómo podés integrar datos de tres **Hojas de Cálculo** en un único archivo. Desde Excel podés guardar una hoja de cálculo con formato **csv** (valores separados por coma). Esta clase de archivo tiene un formato interno estándar, se puede tratar como un archivo plano, cada dato en la línea está separado por, o ;

Nota:

*El caracter de separación de campos en un archivo **csv** puede variar (',' o ';') dependiendo de la Planilla que lo abre. Por ejemplo, en Excel[™] se usa ';', mientras que en algunas Planillas Open se emplea ','. Averiguar cuál usar es simple. Generas un archivo **csv** en tu Planilla y luego lo abres que un editor simple de texto y miras con cuál símbolo se separan los campos. Así sabrás qué separador indicar en tu programa.*

datos1.csv

		santiago del	
Luciana	juárez	estero	95
		santiago del	
Julián	manzur	estero	98
Mariano	álvarez	san luis	68
rogelio	linares	córdoba	88
Anabel	llanes	mendoza	83

datos2.csv

Jazmín	gerez	formosa	95
ana laura	jiménez	chaco	98
Roberto	luna	corrientes	68
Joaquín	ortiz alba	entre ríos	88
Antonio	peñate	río negro	83
Marcelo	villate	santa cruz	99
Mariano	villafañez	santa cruz	98
Nicolás	luzuriaga	santa cruz	97
Agustín	méndez	formosa	77

datos3.csv

Andrés	gil gómez	caba	85
Andrea	paz	caba	99
		buenos	
Pablo	paz iraola	aires	77
Paulina	estarda	la pampa	88
Paula	garcía	río negro	83
Lucía	gonzález	neuquén	99
Luciano	pinto	la rioja	81
Rossana	del olmo	chubut	84
Victoria	antón	salta	62

Nota: Supondremos que cada una de ellas tiene datos relacionados con puntajes obtenidos por personas en un concurso. Nuestro programa va a guardar la información de todos en un archivo de tipo **.csv** (se podrá abrir desde block de notas, Excel, etc.); y la presentará ordenada de mayor a menor puntaje obtenido.

Nota: Emplearemos separación de campos con ';'.

Código	Efecto
#Genera Planilla Integrada de Datos	
completo=open('datosCompleto.csv','w')	Crea el archivo datosCompleto
b=[]	Crea la lista b vacía
dat=open('datos1.csv')	Abre datos1 sólo para lectura como dat
a=dat.readlines()	Carga en la lista a todas las líneas
dat.close()	Cierra archivo
for i in range(len(a)):	
a[i]=a[i].strip('\n')	Quita bajada de línea a cada línea
a[i]=a[i].split(';')	Arma tabla quitando ;
a[i][3]=int(a[i][3])	A la columna 3 la convierte a entero
b=b+a	Agrega la lista a a b
dat=open('datos2.csv')	Lo mismo que antes pero para datos2
a=dat.readlines()	
dat.close()	
for i in range(len(a)):	
a[i]=a[i].strip('\n')	
a[i]=a[i].split(';')	
a[i][3]=int(a[i][3])	
b=b+a	
dat=open('datos3.csv')	Lo mismo que antes pero para datos3
a=dat.readlines()	
dat.close()	
for i in range(len(a)):	
a[i]=a[i].strip('\n')	
a[i]=a[i].split(';')	
a[i][3]=int(a[i][3])	
b=b+a	
b.sort(reverse=True,key=lambda b:(b[3],b[2]))	Ordena la tabla b que tiene todas las filas de datos1, datos2 y datos3
for elemento in b:	
print(elemento)	Muestra las filas de b
for elemento in b:	
elemento[3]=str(elemento[3])	Para cada fila convierte columna 3 en texto
ele=';'.join(elemento)+'\n'	Arma string con separaciones de ; y agrega bajada de línea al final
completo.write(ele)	Graba la línea en el archivo completo
completo.close()	Cierra el archivo

Que producirá la siguiente salida en pantalla:

```
[ 'marcelo', 'villate', 'santa cruz', 99]
[ 'lucía', 'gonzález', 'neuquén', 99]
[ 'andrea', 'paz', 'caba', 99]
[ 'julián', 'manzur', 'santiago del estero', 98]
[ 'mariano', 'villafañez', 'santa cruz', 98]
[ 'ana laura', 'jiménez', 'chaco', 98]
[ 'nicolás', 'luzuriaga', 'santa cruz', 97]
[ 'luciana', 'juárez', 'santiago del estero', 95]
[ 'jazmín', 'gerez', 'formosa', 95]
[ 'paulina', 'estarda', 'la pampa', 88] [ 'joaquín',
'ortiz alba', 'entre ríos', 88] [ 'rogelio ',
'linares', 'córdoba', 88]
[ 'andrés', 'gil gómez', 'caba', 85]
[ 'rossana', 'del olmo', 'chubut', 84]
[ 'antonio', 'peñate', 'río negro', 83]
[ 'paula', 'garcía', 'río negro', 83] [ 'anabel',
'llanes', 'mendoza', 83]
[ 'luciano', 'pinto', 'la rioja', 81]
[ 'agustín', 'méndez', 'formosa', 77]
[ 'pablo', 'paz iraola', 'buenos aires', 77]
[ 'mariano', 'álvarez', 'san luis', 68] [ 'roberto',
'luna', 'corrientes', 68] [ 'victoria', 'antón',
'salta', 62]
```

Y generará el siguiente archivo. Para verlo lo abriremos desde Excel:

datosCompleto.csv

Marcelo	villate	santa cruz	99
Lucía	gonzález	Neuquén	99
Andrea	paz	Caba	99
		santiago del	
Julián	manzur	estero	98
Mariano	villafañez	santa cruz	98
ana laura	jiménez	Chaco	98
Nicolás	luzuriaga	santa cruz	97
		santiago del	
Luciana	juárez	estero	95
Jazmín	gerez	Formosa	95
Paulina	estarda	la pampa	88
Joaquín	ortiz alba	entre ríos	88
rogelio	linares	Córdoba	88
Andrés	gil gómez	Caba	85
Rossana	del olmo	Chubut	84
Antonio	peñate	río negro	83
Paula	garcía	río negro	83

Anabel	llanes	Mendoza	83
Luciano	pinto	la rioja	81
Agustín	méndez	Formosa	77
Pablo	paz iraola	buenos aires	77
Mariano	álvarez	san luis	68
Roberto	Luna	Corrientes	68
Victoria	Antón	Salta	62

Una mejora:

Como lo que se emplea para abrir un **archivo** en la función **open()** es una string que contiene el nombre, podemos editar esa string y automatizar, si cabe, la construcción del nombre del **archivo** a abrir.

- **Integración con bucle de Carga**

```
completo=open('datosCompletos.csv','w')
b=[]

j in range(1,4):
    dat=open('datos'+str(j)+'.csv')
    a=dat.readlines()
    dat.close()
for i in range(len(a)):
    a[i]=a[i].strip('\n')
    a[i]=a[i].split(';')
    a[i][3]=int(a[i][3])

    b=b+a
b.sort(reverse=True,key=lambda b:(b[3],b[2]))
for elemento in b:
    print(elemento)
for elemento in b:
    elemento[3]=str(elemento[3])
    ele=';'.join(elemento)+'\n'
    completo.write(ele)
completo.close()
```