




Seminario de Lenguajes - Python

Cursada 2024

Clase 4: Archivos (cont.)

-  Manejo de datos en archivos
-  Módulo datetime
-  Funciones mágicas en Jupyter

Lo visto en clase

Abrir, modificar, crear archivos con el módulo **csv** y **json**

Un programa puede estar formado por muchos módulos

Manejo de datos en archivos

Vamos a trabajar con archivos **csv** y probar algunas operaciones con los datos

```
In [ ]: from pathlib import Path
import csv
```

```
In [ ]: data_temp = Path('files') / "tas_climatology_annual-seasonal_cru_1961-1990_ARG.csv"
```

Vemos el contenido usando el container **with** si no usamos el container recuerden que se tiene que:

- abrir con `open()`
- cerrar con `close()` para evitar problemas con los archivos

Hay atajos de Jupyter que nos permiten conocer el contenido de directorios

```
In [ ]: %ls 'files'
```

El caracter "%" permite ejecutar un comando conocido, también podemos encontrarlo con el caracter "!"

```
!ls 'files'
```

En el caso del "!" es interpretado por el bash de los sistemas Unix, por lo cual puede que solo funcione en estos sistemas operativos o a través de git-bash de Windows.

Estos caracteres le indican que es un comando y los interpreta para ejecutarlos en Jupyter.

```
In [ ]: with open(data_temp) as file_temp:
    csv_reader = csv.reader(file_temp, delimiter=',')
    for line in csv_reader:
        print(line)
    header, data = next(csv_reader), list(csv_reader)
```

- ¿Qué pasa con los datos en la variable `csv_reader` después de recorrer un vez?

- ¿Qué tipo de variable es `csv_reader`?

Un **iterador** es un objeto que representa un flujo de datos.

Al abrir este flujo de datos, nos posicionamos en el primer ítem y luego, utilizando el método `__next__()` del iterador o al pasar la función incorporada **`next()`** vamos obteniendo los siguientes ítems del flujo. Este proceso finaliza cuando no hay más datos disponibles.

Para poder trabajar con los datos tenemos que guardarlo en una variable que no sea iterador.

```
In [ ]: with open(data_temp) as file_temp:
        csv_reader = csv.reader(file_temp, delimiter=',')
        header, data = next(csv_reader), list(csv_reader)
```

👀 Conozcamos los datos

```
In [ ]: print(header)
        for line in data:
            print(line)
```

Nombres de las columnas

- (DJF): Diciembre, Enero, Febrero (Verano)
- (MAM): Marzo, Abril, Mayo (Otoño)
- (JJA): Junio, Julio, Agosto (Invierno)
- (SON): Setiembre, Octubre, Noviembre (Primavera)

Si queremos tener un diccionario para acceder a los datos de cada jurisdicción directamente, ¿qué estructura podemos usar?

```
In [ ]: data = {}
        with open(data_temp) as file_temp:
            csv_reader = csv.DictReader(file_temp, delimiter=',')
            for row in csv_reader:
                print(row)
```

Recordemos que:

DictReader permite crear un diccionario donde el nombre de las columnas son las **keys** y el valor de la celda son los **values**

```
In [ ]: data = {}
        with open(data_temp) as file_temp:
            csv_reader = csv.DictReader(file_temp, delimiter=',')

            for row in csv_reader:
                data[row['Región']] = dict(row)
data
```

🍌 Mejorando la estructura para facilitar el procesamiento

Recordemos la estructura del archivo:

```
['Región', 'Annual', 'DJF', 'MAM', 'JJA', 'SON']
['Argentina', '14.60', '20.55', '14.45', '8.42', '14.98']
```

```
In [ ]: data = {}
with open(data_temp) as file_temp:
    csv_reader = csv.reader(file_temp, delimiter=',')
    header = next(csv_reader)
    for row in csv_reader:
        data[row[0]] = row[1:]
```

```
In [ ]: data
```

La primera línea es la anual, teniendo en cuenta que vamos a procesar las temperaturas de las jurisdicciones, podemos tener en cuenta esto cada vez que realizamos los cálculos, es decir empezar desde la segunda línea, o guardar los datos por separado.

```
In [ ]: data = {}
anual = []
with open(data_temp) as file_temp:
    csv_reader = csv.reader(file_temp, delimiter=',')
    header = next(csv_reader)
    anual = next(csv_reader)
    for row in csv_reader:
        data[row[0]] = row[1:]
```

```
In [ ]: anual
```

Desafío 1

Queremos obtener la temperatura anual promedio mínima.

Recordemos el orden de los datos:

```
In [ ]: header
```

Guardamos en el diccionario desde el segundo elemento, ya que el dato en 'Región' es la key del diccionario:

```
In [ ]: header[1:]
```

```
In [ ]: min([data[temp][0] for temp in data])
```

¿Pero es esta la temperatura mínima?

Veamos los datos:

```
In [ ]: for prov in data:
        print(data[prov][0])
```

Mirando rápido encontramos varias temperaturas más bajas que 10.01:

- 4.86
- 7.76

Convertir los valores

El módulo **csv** considera como **str** los datos leídos, el valor que indicó es tomado con el orden alfabético. Por lo tanto, para hacer una comparación numérica tenemos que convertirlos a **float**, después de haberlos guardado en el diccionario o podemos leerlo nuevamente y hacer la conversión al guardarlos.

```
In [ ]: data = {}
anual = []
with open(data_temp) as file_temp:
    csv_reader = csv.reader(file_temp, delimiter=',')
    header = next(csv_reader)
    anual = next(csv_reader)
    anual[1:] = [float(elem) for elem in anual[1:]]
    for row in csv_reader:
        data[row[0]] = [float(elem) for elem in row[1:]]
```

```
In [ ]: data
```

Calculamos nuevamente la temperatura mínima anual:

```
In [ ]: min([data[temp][0] for temp in data])
```

Ahora sí es correcto el valor.

✨ Atajos mágicos en Jupyter

- Nos permite ejecutar operaciones que son interpretadas por el kernel de Jupyter
- Más arriba vimos %ls
- [Documentación oficial](#)
- Podemos consultar cuáles son todos los comandos disponibles

```
In [ ]: %lsmagic
```






Algunos que pueden ser interesantes son los que nos permiten medir el tiempo de ejecución de un bloque de código Python en *nuestra computadora*

```
In [ ]: %timeit L= min([data[temp][0] for temp in data])
```

```
In [ ]: %%timeit
minim = 999
for juri in data:
    if data[juri][0] < minim:
        minim = data[juri][0]
minim
```

✨ Atajos de celdas en Jupyter

Estando fuera del modo edición:

-  a: agrega una celda arriba de donde estamos
-  b: agrega una celda abajo de donde estamos
-  dd: borra la celda actual
-  m: convierte la celda en el formato markdown
-  y: convierte la celda en el formato código





Desafío 2

¿Qué jurisdicciones tuvieron una temperatura menor que la promedio anual?

```
In [ ]: anual
```

Recordemos que:

-  **filter**: permite filtrar elementos de una secuencia
-  **dict.items()**: me devuelve una lista con una tupla con las keys y los valores asociados



```
In [ ]: list(filter( lambda prov: prov[1][0] > anual[1], data.items()))
```

Ejercicio 1 para practicar

- Encontrar las jurisdicciones con las temperaturas mínimas y máximas de cada estación.

Ordenar los datos

Queremos ordenar los datos de forma descendente según los valores de las temperaturas de la estación Verano. Recordemos que:

-  **sort**: nos permite ordenar una secuencia sin poder especificar el criterio.
-  **sorted**: nos permite ordenar indicando varias opciones para la acción.

```
In [ ]: header
```

Recordemos que **DJF** es el **verano**

```
In [ ]: data
```

data.items contiene una tupla con los elementos:

(key, values)

Los values es una lista donde tiene en la posición 1 la temperatura del verano.

```
In [ ]: dict(sorted(data.items(), key=lambda item: item[1][1], reverse=True))
```

Y en invierno de forma ascendente:

```
In [ ]: dict(sorted(data.items(), key=lambda item: item[1][2]))
```

Ejemplo de paquetes en el IDE

```
In [ ]: !ls files
```

Procesamiento de logs

- Trabajamos con el logs de los registros de Moodle de la Autoevaluación 1

```
In [ ]: logs_file = 'logs_act1.csv'
dir_files = Path('files').resolve()
```

```
In [ ]: with open(dir_files / logs_file) as file_temp:
    csv_reader = csv.reader(file_temp, delimiter=',')
    header, data_logs = next(csv_reader), list(csv_reader)
```

```
In [ ]: data_logs[:5]
```

Desafío 3

Encontrar los registros que tienen las IP que no son de la facultad.

```
In [ ]: header
```

```
In [ ]: for row in data_logs:
        if not row[8].startswith('163.10.'):
            print(row)
```

En la salida vemos que los nombres se **repiten** según la cantidad de veces que accedió cada persona.

Podemos guardar los datos en una estructura **sin repetición**.

```
In [ ]: header
```

```
In [ ]: data_names = {}
        for row in data_logs:
            if not row[1] in data_names:
                data_names[row[1]] = row
```

```
In [ ]: data_names
```

Vemos los 5 primeros.

```
In [ ]: list(data_names.items())[:5]
```

Módulo datetime

Datetime es un módulo para trabajar con las fechas

- ▶ Nos permite trabajar y realizar cálculos con el tiempo.
- ▶ Documentación módulo **datetime**

Contiene varias **clases** para manejar fechas y tiempo:

- ✓ **date**: se utiliza para fechas pero no guarda hora, minutos ni segundos
- ✓ **time**: se utiliza para operar valores de tiempo al nivel de minutos a microsegundos
- ✓ **datetime**: contiene información de las fechas **date** y el tiempo **time**

Algunas de la características que comparten estos objetos son:

- ▶ Son **inmutables**.
- ▶ Son **"hashable"**, es decir que se pueden usar como keys en un diccionario.

```
In [ ]: from datetime import date
        d1 = date(2024, 3, 26)
        d1
```

```
In [ ]: from datetime import time
        t1 = time(11, 10, 9)
        t1
```

⚠ **OJO** que existe el módulo **time** que tiene funcionalidades similares, en comparación con el submódulo de **datetime**, el módulo **time** tiene un poco más de precisión el nivel de microsegundos y es recomendado de usar solo en sistemas operativos UNIX.

```
In [ ]: from datetime import datetime
my_datetime = datetime.combine(d1, t1)
my_datetime
```

Los objetos **date**, **time** y **datetime** pueden operar entre sí, se usan según lo que se necesite.

🕒 ¿Qué podemos realizar con la clase **datetime**?

¿Cómo podemos ver los métodos de una clase?

- ▶▶ `dir()`
- ▶▶ en Jupyter, el tab nos ayuda para autocompletar, donde nos marca cuáles son funciones y cuáles son variables de la clase



```
In [ ]: datetime.today()
```

Al crear podemos opcionalmente especificar la información de `tzinfo`.





```
In [ ]: from datetime import timezone
print(datetime.now(timezone.utc))
print(datetime.now(timezone.utc).tzinfo)
#en python 3.11
#datetime.now(datetime.timezone.utc)
```

Algunos de los métodos que pueden ser interesantes de **datetime.datetime**, para simplificar asumamos que los importamos:

```
from datetime import datetime
```

- ✓ **`datetime.now()`**: nos da la fecha actual
- ✓ **`datetime.timestamp(datetime.now())`**: convierte la hora actual en el formato timestamp, que

es utilizado para guardar fecha de forma única

-  **datetime.strptime()**: convierte un string con el formato que le indicamos en un objeto *datetime*
-  **date.strftime("%m/%d/%Y, %H:%M:%S")**: convierte un objeto *datetime* a un string
-  **date.weekday()**: podemos saber el día de la semana de la fecha
-  **date.hour**: podemos saber la hora

Los datos que se guardan en los **logs**, en general van acompañados de un **timestamp**, que es un número que representa de forma única a un momento específico. Es un número que representa el tiempo pasado desde 1/1/1970.

```
In [ ]: my_timestamp = datetime.timestamp(datetime.now())
my_timestamp
```

Se puede guardar en ese formato o como un string, lo cual es más legible.

```
In [ ]: my_log = datetime.fromtimestamp(my_timestamp)
```

```
In [ ]: my_log.strftime("%m/%d/%Y, %H:%M:%S")
```

Desafío 4

Encontrar cuánto tiempo pasó desde el primer registro hasta el último.

Recordemos cómo está almacenado el dato de las fechas en el archivo:

```
['19/03/24, 15:57:50',
 'T_L15',
 'T_L15',
 'Cuestionario: Autoevaluación 1 - Turno MAÑANA',
 'Cuestionario',
 'Intento de cuestionario visualizado',
 "The user with id '22862' has viewed page '2' of the attempt with id
 '82865' belonging to the user with id '22862' for the quiz with course
 module id '39686'.",
 'web',
 '186.23.0.228']
```

La posición 0 contiene el string con la fecha, veamos cómo podemos convertir ese string en un objeto *datetime*.

Veamos los 5 primeros

```
In [ ]: format = "%d/%m/%y, %H:%M:%S"
for row in data_logs[:5]:
    print(datetime.strptime(row[0], format))
```

El método **strptime()** permite convertir un **str** en un objeto *datetime*, pero primero debemos indicar el **formato** para que interprete el string:

```
format = "%d/%m/%y, %H:%M:%S"
```

La variable **format** es un string que tenemos que armar en función de cómo está guardado la fecha, esto varía según sea el caso.

Veamos qué operaciones se pueden realizar con las fechas que están guardadas. Como vimos esta información está en la primera posición, en formato **string**:


```
In [ ]: date_str = data_logs[0][0]
date_str
```

Y para convertir a un objeto datetime usaremos el método **strptime**:

```
In [ ]: my_date = datetime.strptime(data_logs[0][0], format)
my_date
```

Veamos otro ejemplo con otro formato.

```
In [ ]: format = "%d/%m/%Y, %H:%M:%S"
date = datetime.strptime(data_logs[0][0], format)
date
```

¿Por qué creen que da error?

Da error porque el año especificado en la variable **format** (%Y) al ponerlo en mayúscula espera cuatro dígitos y en el archivo sólo tiene dos dígitos.

Veamos con un ejemplo con otro formato en el dato.

```
In [ ]: date_oth = '26/03/2024, 17:36:54'
format = "%d/%m/%Y, %H:%M:%S"
date = datetime.strptime(date_oth, format)
date
```

En este caso al estar con 4 dígitos el string no hubo problema.

Volvamos con las fechas del archivo

```
In [ ]: format = "%d/%m/%y, %H:%M:%S"
date = datetime.strptime(data_logs[0][0], format)
date
```

```
In [ ]: len(data_logs)
```

```
In [ ]: data_logs[len(data_logs)-1]
```

Calculamos las fechas de **inicio** y **final** del archivo de logs:

```
In [ ]: date_start = datetime.strptime(data_logs[len(data_logs)-1][0], format)
date_start
```

```
In [ ]: date_end = datetime.strptime(data_logs[0][0], format)
date_end
```

Y el tiempo que pasó entre ambas fechas:

```
In [ ]: date_end - date_start
```

🚧 Ejercicio 2 para practicar

- Calcular la cantidad de registros que hay por cada día de la semana.

Recordemos que:

`date_start.weekday()`

nos permite acceder al día de la semana y el módulo Counter nos ayuda a encontrar las cantidades

correspondientes.

¿Se animan a hacerlo con list comprehension?

Conociendo el entorno de Jupyter

Analice qué información nos brindan los siguientes comandos:

```
In [ ]: %history
```

```
In [ ]: who
```

```
In [ ]: whos
```