Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

VERSIÓN PRELIMINAR

Índice general

VERSIÓN PRELIMINAR

Parte I
Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

1 | Manipulación de datos

En este capítulo hablaremos en cómo manipular datos, independientemente del procesamiento de los mismos. Nuestro foco estará en cómo cargar o leer datos y herramientas para procesarlos, no en las operaciones que se realizarán sobre esos datos. En otras palabras y a modo de ejemplo, veremos cómo cargar una lista de datos en memoria como para calcular su promedio, pero no hablaremos del cálculo que se necesita para obtener el promedio.

Módulo	Versión
h5py	3.11.0
netCDF4	1.7.1
numpy	1.26.4
pandas[all]	2.2.2
requests	2.32.3

Código disponible

La fuente natural de datos son archivos si los datos ya están generados en una etapa previa, o algún dispositivo de entrada si los datos se están generando en el momento de forma continua. Notablemente, cuando queremos descargar datos de internet nos podemos encontrar con la misma situación: si los datos son pocos los podemos leer y procesar todos en la misma acción, o también podemos irlos bajando y procesando de a poco.

En cualquier caso los datos pueden estar en forma de "texto" o ser "binarios". En general (aunque no siempre) si los datos están en formato de texto estarán pensados para ser consumidos línea a línea, mientras que los datos binarios se procesarán byte a byte. Por razones pedagógicas nos enfocaremos en ambos casos por separado.

1.1. Leyendo archivos

Usamos el concepto "archivo" de forma muy genérica y al tiempo de forma muy poderosa, porque los sistemas informáticos utilizan ese concepto para proveer información desde varias fuentes. Entonces, trabajaremos como si fuesen archivos no sólo a archivos reales que tenemos en el dispositivo de almacenamiento de la computadora sino también a otras representaciones como dispositivos de entrada/salida de la computadora o respuestas de pedidos por la red.

En particular está el concepto de *file-like object*, "objetos que se parecen a archivos", que son objetos que podemos tener como resultados de operaciones en nuestro programa y que por simplicidad pueden tratarse como archivos aunque realmente no lo sean.

La forma más simple de abrir un archivo es utilizando la función integrada `open`. Sus parámetros más comunes son: el nombre del archivo, el modo, y la codificación.

El nombre del archivo (o el *path* completo) apunta al archivo en disco, que puede existir o no

según cómo vayamos a abrirlo.

El modo es justamente la forma en que vamos a abrir el archivo; es una cadena de varias letras cada una con un significado específico que se pueden combinar de algunas maneras puntuales.

- `'r'`: abre para solamente leer
- `'w'`: abre para solamente escribir, truncando el archivo a cero si es que ya existía
- `'x'`: abre para solamente escribir, pero exclusivamente creando el archivo (falla si el archivo ya existía)
- `'a'`: abre para solamente escribir, agregando el nuevo contenido al final
- `'b'`: modo binario
- `'t'`: modo texto
- `'+'`: abre para actualizar (leer y escribir)

En general construimos el modo indicando lectura/escritura y binario/texto, por ejemplo `'wb'` para escribir un archivo en binario, `'rt'` para leer un archivo en texto, etc. El `'+'` se usa relativamente poco, es para aquellos casos en que queremos escribir y leer al mismo tiempo, y se combina con las otras letras de lectura/escritura más para indicar qué sucedería con un archivo que ya existe o no.

Cell 01

```
# abrimos para escritura, escribimos, cerramos
fh = open("/tmp/archivoprueba", "wb")
fh.write(b"hola")
fh.close()

# abrimos para lectura, leemos, cerramos
fh = open("/tmp/archivoprueba", "rb")
print(fh.read())
fh.close()
```

```
b'hola'
```

Cell 02

```
# abrimos para *agregar* al archivo anterior, escribimos, cerramos
fh = open("/tmp/archivoprueba", "ab")
fh.write(b" mundo")
fh.close()

# vemos que en el archivo tenemos lo anterior y lo nuevo
fh = open("/tmp/archivoprueba", "rb")
print(repr(fh.read()))
fh.close()
```

```
b'hola mundo'
```

En realidad los archivos siempre son binarios; sólo pueden tener bytes dentro porque los medios de almacenamiento guardan bytes. Abrir el archivo en modo "texto" es solamente una

forma cómoda de trabajar con texto Unicode en nuestros programas sin pasar por una etapa donde el contenido está en binario: entonces en el programa siempre manejaremos texto y la conversión a o desde bytes la hará Python automáticamente, y es por eso que en estos casos hay que indicarle la codificación que usará para codificar o decodificar (en realidad es opcional, pero el default depende del sistema, entonces siempre es recomendable que esté indicado de forma explícita).

CELL 03

```
# notar como para texto especificamos la codificación
fh = open("/tmp/archivoprueba", "wt", encoding="utf8")
fh.write("moño")
fh.close()

# y aunque el archivo realmente tiene bytes, accedemos como texto
fh = open("/tmp/archivoprueba", "rt", encoding="utf8")
print(repr(fh.read()))
fh.close()
```

```
'moño'
```

CELL 04

```
# si accedemos al archivo en modo binario vemos lo que realmente tiene
fh = open("/tmp/archivoprueba", "rb")
data = fh.read()
fh.close()
print(repr(data))
```

```
b'mo\xc3\xb1o'
```

CELL 05

```
# obviamente podemos decodificarlo a mano, pero la idea es
# que si trabajamos con texto nunca manejarlo como bytes
print(repr(data.decode("utf8")))
```

```
'moño'
```

Hay algunas opciones más para lidiar con distintas especificidades (que surgen especialmente en distintas plataformas) pero ya con estas podemos comenzar a trabajar. Para profundizar lo mejor es su documentación.

La metodología de abrir y cerrar el archivo como mostramos antes es muy útil cuando el archivo se utiliza por tiempo indefinido o estructuramos algún objeto alrededor de su uso y los dos ciclos de vida están atados. Pero en general se usa otra técnica, ya que es muy común cerrar el archivo lo antes posible en el mismo bloque de código que se abrió; recordemos que mientras el archivo está abierto se está utilizando un recurso del sistema operativo, que hay límites con respecto a la cantidad de archivos abiertos, etc., por lo que siempre conviene prestar atención en cerrar el archivo lo antes posible cuando ya no lo utilizamos más.

Para simplificar eso Python nos permite usar un administrador de contexto ?? para interactuar con archivos:

CELL 06

```
with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:
    print(f"Estado dentro del contexto: closed={fh.closed}")
    data = fh.read()
print(f"Estado luego del contexto: closed={fh.closed}")
print("Leido:", data)
```

```
Estado dentro del contexto: closed=False
Estado luego del contexto: closed=True
Leido: moño
```

Vemos como usando el **with** tenemos al *file handler* abierto dentro del contexto y automáticamente cerrado luego de salir del mismo: de esta manera es clara la zona donde trabajamos con el archivo, con la seguridad que siempre se cerrará al archivo al salir de esa zona (incluso en el caso de una excepción, lo cual no sucedía en los ejemplos anteriores).

1.2. Trabajando con datos en formato textual

Cuando mencionamos archivos de forma genérica en 1.1 vimos la operatoria básica: abrir, cerrar, escribir, leer; entraremos en esta sección en detalles particulares al trabajar con texto.

Habiendo dicho eso, volveremos a mostrar aquí cómo escribir texto en un archivo aunque sea similar a lo que ya vimos. En el siguiente ejemplo bajamos un texto que tenemos como lista de cadenas a un archivo (respetando las líneas, para lo cual tenemos que agregar manualmente los saltos de línea).

CELL 01

```
lines = [
    "En este film velado en blanca noche",
    "El hijo tenaz de tu enemigo",
    "El muy verdugo cena distinguido",
    "Una noche de cristal que se hace añicos",
]

with open("/tmp/archivoprueba", "wt", encoding="utf8") as fh:
    for line in lines:
        fh.write(line + "\n")  # notar el agregado del salto de linea

with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:
    print(fh.read())
```

```
En este film velado en blanca noche
El hijo tenaz de tu enemigo
El muy verdugo cena distinguido
Una noche de cristal que se hace añicos
```

Para leer, por otro lado, aprovechamos la característica de que el *file handler* que devuelve `open` es iterable y nos devolverá línea por línea del archivo, lo cual es al mismo tiempo la forma más práctica y más eficiente de recorrer las líneas de un archivo:

CELL 02

```
with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:
    for line in fh:
        print(len(line), repr(line))
```

```
36 'En este film velado en blanca noche\n'
28 'El hijo tenaz de tu enemigo\n'
32 'El muy verdugo cena distinguido\n'
40 'Una noche de cristal que se hace añicos\n'
```

Hay otros métodos para trabajar con el *file handler*, por ejemplo `.readlines` que leerá todas las lineas del archivo almacenándolas en una lista, pero en general siempre será mejor leer linea por linea y en todo caso armar una lista con el resultado de un primer procesamiento de eso archivo.

Por supuesto, cuando trabajamos con texto también podemos leer carácter a carácter (similar a como vamos a ver cuando trabajemos con binarios, más adelante), buscando el salto de línea y armando las líneas a mano, pero en la casi absoluta cantidad de veces es suficiente la manera ya vista.

La iteración de cada línea ni siquiera tiene que ser explícita, ya que muchas funciones de Python trabajan iterando naturalmente. En el siguiente ejemplo vemos una simple forma de obtener la línea más larga del archivo que recién creamos, usando `max` y aprovechando que se le puede indicar cual es indicador de orden (en nuestro caso el largo de cada línea, que es lo que devolverá la función `len` en cada caso):

CELL 03

```
with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:
    longest = max(fh, key=len)
print("Más larga:", repr(longest))
```

```
Más larga: 'Una noche de cristal que se hace añicos\n'
```

Veamos un ejemplo más complejo: sacar el top 5 de mantenedores de paquetes de Debian. La fuente de datos será un archivo público, la lista de *maintainers*, que en cada línea indica el paquete, uno o más espacios, y el nombre y mail de su mantenedor (para más detalles de la estructura que procesa el código ver la descripción en el Jupyter Notebook `textos.ipynb` de este capítulo).

CELL 04

```
from collections import Counter

names_counter = Counter()

with open("Maintainers", "rt", encoding="utf8") as fh:
    for line in fh:
        # separamos por blancos, pero como mucho una vez para tener intacto
        # el texto de todos los mantenedores
        package, maintainers = line.strip().split(maxsplit=1)

        # separamos por mayor-que y coma, para no cortar por la coma entre los nombres
        for maintainer in maintainers.split(">,"):
            if "<" in maintainer:
                # sacamos el resto del mail
                name, _ = maintainer.split("<")
            else:
                # el nombre *es* el mail
                name = maintainer.replace("<", "").replace(">", "")

            # contamos
            name = name.strip()
            names_counter[name] += 1

for name, quantity in names_counter.most_common(5):
    print(f"{quantity:3d}  {name}")
```

```
9002  Debian Kernel Team
7392  Debian GCC Maintainers
6544  Debian Rust Maintainers
5614  Debian Python Team
5010  Debian Haskell Group
```

Por supuesto, la estructura que acabamos de procesar es muy específica. Hay muchos formatos textuales que son más estándar, veamos algunos casos.

1.2.1. Tres formatos textuales muy comunes

Mencionaremos particularmente tres formatos que se utilizan mucho en los sistemas informáticos, CSV, JSON y XML.

1.2.1.1. CSV

Los archivos CSV, por *Comma-Separated Values*, o "valores separados por coma" es un tipo de archivo muy utilizado para guardar datos en forma de tabla.

Aunque no tiene prácticamente soporte para manejar distintos tipos de datos (la conversión a Unicode de los textos dependerá de cómo abramos el archivo, y sólo convertirá algunos campos numéricos en algún caso, como veremos luego), pero su simplicidad y facilidad de uso hizo que sea uno de los formatos más utilizados desde siempre.

Su estructura, a priori, es muy sencilla: cada línea es una fila, y cada columna se separa de la otra por comas. Pero, en la práctica, es todo mucho más complicado, porque no hay un estándar

fuerte sobre distintos aspectos como qué codificación se usa para guardar textos Unicode, cómo separar las columnas cuando el punto decimal en los números es la coma, o cuando los textos usan comillas o incluso si tienen saltos de línea dentro.

Entonces siempre recomendamos evitar caer en la tentación y procesar un archivo CSV "a mano" (leyendo línea por línea asumiendo que sólo hay saltos de línea al final de cada fila y directamente separar por comas para obtener las columnas). El procedimiento correcto es utilizar el módulo `csv` provisto por la Biblioteca Estándar de Python.

Veamos esto con un ejemplo. Construyamos un caso particularmente complicado donde tenemos cadenas Unicode, un texto vacío, y otros que tienen caracteres que a priori serían "de control" en el formato: comillas dobles y coma (y tilde y punto y coma, por las dudas), y un salto de línea.

Cell 05

```
import csv

original = [
    ["algún texto", 234, "", 'con comilla (")'],
    ["con tilde (')", 14.3, "coma y punto y coma (,;)", "con newline (\n)"],
]

with open("/tmp/prueba.csv", "wt") as fh:
    writer = csv.writer(fh)
    for row in original:
        writer.writerow(row)

with open("/tmp/prueba.csv", "rt") as fh:
    print(fh.read())
```

```
algún texto,234,,"con comilla ("")"
con tilde (',14.3,"coma y punto y coma (,;)","con newline (
)"
```

Grabamos el archivo en formato CSV usando el módulo ya mencionado para asegurarnos que el formato final sea el correcto, y luego lo abrimos y mostramos sin procesar. Allí vemos que aunque grabamos dos filas, tenemos efectivamente tres líneas y otros detalles que nos complicarían el procesamiento "a mano". Veamos entonces qué sucede si lo procesamos "mal":

Cell 06

```
with open("/tmp/prueba.csv", "rt") as fh:
    for line in fh:
        print(line.split(","))
```

```
['algún texto', '234', '', '"con comilla ("")"\n']
["con tilde (')", '14.3', '"coma y punto y coma (', ';)"', '"con newline (\n']
[')"\n']
```

Por supuesto, si utilizamos el módulo `csv` para procesar el archivo volvemos a tener cada uno de los textos de forma correcta:

CELL 07

```
with open("/tmp/prueba.csv", "rt") as fh:
    reader = csv.reader(fh)
    for row in reader:
        print(row)
```

```
['algún texto', '234', '', 'con comilla (")']
["con tilde (')", '14.3', 'coma y punto y coma (,;)', 'con newline (\n)']
```

Decíamos arriba que los archivos CSV pueden tener distintos formatos. El módulo `csv` trae soporte para interpretar los "dialectos" más comunes (`excel`, `unix`, etc.), y de esta manera podemos tener control sobre qué se usa para separar columnas, con qué se envuelven los campos que tienen caracteres especiales, cómo se escapan a su vez esos caracteres, el fin de línea, etc.

Podemos elegir cuál dialecto preferimos, armar uno nosotros, o incluso pasar modificadores al escribir o leer el CSV para afectar puntualmente algún comportamiento. En el siguiente ejemplo indicamos que al escribir se agreguen comillas alrededor de todos los campos no numéricos, entonces luego al leer el resto de los campos serán convertidos a `float` directamente:

CELL 08

```
with open("/tmp/prueba.csv", "wt") as fh:
    writer = csv.writer(fh, quoting=csv.QUOTE_NONNUMERIC)
    for row in original:
        writer.writerow(row)

with open("/tmp/prueba.csv", "rt") as fh:
    reader = csv.reader(fh, quoting=csv.QUOTE_NONNUMERIC)
    for row in reader:
        print(row)
```

```
['algún texto', 234.0, '', 'con comilla (")']
["con tilde (')", 14.3, 'coma y punto y coma (,;)', 'con newline (\n)']
```

Hasta este momento estamos mostrando fila por fila de datos sin acceder a una columna en particular. Habrán notado que cada fila termina siendo una lista, por lo que para acceder a cada dato debemos hacerlo posicionalmente:

CELL 09

```
with open("/tmp/prueba.csv", "rt") as fh:
    reader = csv.reader(fh, quoting=csv.QUOTE_NONNUMERIC)
    for row in reader:
        print(f"Cadena {row[3]!r} con valor {row[1]:.2f}")
```

```
Cadena 'con comilla (")' con valor 234.00
Cadena 'con newline (\n)' con valor 14.30
```

Podemos mejorar la legibilidad de nuestro código si en vez de usar el *reader* común usamos `DictReader`, al cual le pasamos el nombre de los campos y en vez de darnos una tupla nos dará un diccionario:

CELL 10

```
fieldnames = ["column1", "value", "column3", "text"]

with open("/tmp/prueba.csv", "rt") as fh:
    reader = csv.DictReader(fh, fieldnames=fieldnames, quoting=csv.QUOTE_NONNUMERIC)
    for row in reader:
        print("Cadena {text!r} con valor {value:.2f}".format_map(row))
```

```
Cadena 'con comilla (")' con valor 234.00
Cadena 'con newline (\n)' con valor 14.30
```

Incluso `DictReader` soporta tomar el nombre de las columnas de la primer fila del archivo; lo hace automáticamente si evitamos pasarle el parámetro `fieldnames`.

1.2.1.2. JSON

La ubicuidad del lenguaje JavaScript en la gran mayoría de navegadores web hizo que un formato pensado para ser evaluado de forma sencilla por ese lenguaje se hiciera moneda común a la hora de intercambiar información entre distintas partes.

Este formato es un subconjunto de la notación de JavaScript, llamado JSON por *JavaScript Object Notation*, "notación de objeto de JavaScript", y es a la vez flexible (porque soporta los tipos de datos más utilizados por los distintos lenguajes) y simple (porque es una cadena de caracteres mayormente legible y también soporta solamente unos pocos tipos de datos), lo cual impactó fuertemente en su interoperabilidad y por lo tanto en su adopción en el tiempo.

Los siguientes son los tipos de datos soportados por JSON, con la aclaración de a qué tipo corresponden en Python:

- Números enteros (`int`) y con parte fraccional separada por punto (`float`), en ambos casos positivos y negativos
- Cadenas de texto Unicode (`str`)
- Booleanos (**`True`** y **`False`**)
- Null (**`None`**)
- Arreglos (o *arrays*) con cualquier otro tipo de dato soportado dentro (`list`)
- Objetos o colecciones no ordenadas de pares clave-valor, donde la clave tiene que ser una cadena sí o sí y el valor puede ser cualquier tipo (`dict`, con el detalle de restricción del tipo de la clave)

Veamos algunos ejemplos codificando estructuras de Python al formato JSON:

CELL 11

```
import json

structures = [
    "una cadena unicode áÑ",  # eso
    35,  # un entero
    2.3,  # punto flotante
    [1, "hola", [3, 2], 'comilla "'],  # lista con cosas (inclusive otra lista)
    (1, 2, 3, True, False, None),  # tupla
    {"a": 3, "b": [], 'c': ''}  # diccionario, ojo a las claves siempre string
]

for structure in structures:
    encoded = json.dumps(structure)
    print(f"Python: {structure!r} - JSON: {encoded!r}")
```

```
Python: 'una cadena unicode áÑ' - JSON: '"una cadena unicode \\u00e1\\u00d1"'
Python: 35 - JSON: '35'
Python: 2.3 - JSON: '2.3'
Python: [1, 'hola', [3, 2], 'comilla "'] - JSON: '[1, "hola", [3, 2], "comilla \\""]'
Python: (1, 2, 3, True, False, None) - JSON: '[1, 2, 3, true, false, null]'
Python: {'a': 3, 'b': [], 'c': ''} - JSON: '{"a": 3, "b": [], "c": ""}'
```

Podemos ver en el ejemplo algunos detalles que vale la pena mencionar. Las cadenas siempre se rodean con comilla doble (cuando la cadena en sí tiene comilla doble se escapa con barra invertida) y cuando tienen caracteres no-ASCII los mismos se codifican a UTF-16 y se incluyen los valores escapados de los bytes. `True`, `False` y `None` se escriben casi igual en JSON, lo que sumado a muchas otras similitudes hacen que sea un formato muy legible si estamos acostumbrados a Python.

Por último, tanto la tupla como la lista se transforman en el mismo tipo de *array* (que usa como delimitador los mismos corchetes que Python para las listas). Un efecto secundario de esto es que podemos inadvertidamente "perder" las tuplas al pasar por JSON:

CELL 12

```
the_tuple = (1, 2, 3, True, False, None)
print("Original:", repr(the_tuple))
encoded = json.dumps(the_tuple)
print("El JSON:", repr(encoded))
decoded = json.loads(encoded)
print("Volvimos:", repr(decoded))
```

```
Original: (1, 2, 3, True, False, None)
El JSON: '[1, 2, 3, true, false, null]'
Volvimos: [1, 2, 3, True, False, None]
```

Hasta ahora convertimos de Python a JSON como cadena de texto (y volvimos de esa cadena de texto a Python) usando las funciones `dumps` y `loads` del módulo `json`. La mayoría de las veces no nos interesa obtener esa cadena *per se* sino que su destino final es un archivo o enviarla a través de la red; en esos casos podemos grabar o leer directamente de un archivo (o de un objeto *file like*) con las funciones `dump` y `load`.

CELL 13

```
somestuff = dict(answer=42, message="hola campeón", sequence=[1, 2, 3.14])
with open("/tmp/prueba.json", "wt", encoding="utf8") as fh:
    json.dump(somestuff, fh)

with open("/tmp/prueba.json", "rt", encoding="utf8") as fh:
    loaded = json.load(fh)
print(loaded)
```

```
{'answer': 42, 'message': 'hola campeón', 'sequence': [1, 2, 3.14]}
```

Tambien podemos trabajar con otros objetos que no sean realmente archivos pero que expongan la misma interfaz (los llamamos *file-like objects*). Un caso típico es la respuesta de un request HTTP, que expone el método `read` sobre el cual podemos operar.

En el siguiente ejemplo usamos la biblioteca `requests` para obtener la temperatura actual del Museo de Ciencias Naturales de La Plata, Buenos Aires, Argentina:

CELL 14

```
import requests

forecast_url = "https://api.open-meteo.com/v1/forecast"

# pedimos el clima actual en las coordenadas del museo
lat, long = -34.9153435, -57.9331994
params = dict(latitude=lat, longitude=long, current_weather=True)

# para que no venga comprimido
headers = headers={"Accept-Encoding": "identity"}

resp = requests.get(forecast_url, stream=True, headers=headers, params=params)
print(resp.headers["Content-Type"])
print(resp.raw)
```

```
application/json; charset=utf-8
<urllib3.response.HTTPResponse object at 0x7352b41ff580>
```

Cuando hacemos el `get` con `stream=True` la biblioteca no va a leer toda la respuesta inmediatamente, solamente leerá las cabeceras, y vemos también que tenemos el atributo `raw` que es el objeto-respuesta a más bajo nivel, la cual usamos luego para leer directamente decodificando el JSON de la respuesta:

CELL 15

```
data = json.load(resp.raw)
weather = data["current_weather"]
weather
```

```
{'time': '2024-09-24T12:00',
 'interval': 900,
 'temperature': 16.0,
 'windspeed': 23.3,
 'winddirection': 107,
 'is_day': 1,
 'weathercode': 0}
```

JSON soporta todos los tipos de datos que mencionamos antes pero si lo usamos con continuidad notaremos que siempre en algún momento necesitamos enviar en este formato algún tipo de dato no soportado. Incluso en el ejemplo que acabamos de ver tenemos un caso de esto: la "fecha hora" es una cadena y no un objeto `datetime`.

Para este ejemplo alcanza con convertir esa cadena al objeto correspondiente, pero si va a ser recurrente podemos armar un *decoder* específico para nuestras necesidades:

CELL 16

```
import datetime

def decoder(json_dict):
    for key, value in json_dict.items():
        try:
            json_dict[key] = datetime.datetime.fromisoformat(value)
        except (ValueError, TypeError):
            pass
    return json_dict

resp = requests.get(forecast_url, stream=True, headers=headers, params=params)
data = json.load(resp.raw, object_hook=decoder)
weather = data["current_weather"]
weather
```

```
{'time': datetime.datetime(2024, 9, 24, 12, 0),
 'interval': 900,
 'temperature': 16.0,
 'windspeed': 23.3,
 'winddirection': 107,
 'is_day': 1,
 'weathercode': 0}
```

Vemos como podemos indicar una función al momento de decodificar que recibirá todos los "objetos" a nivel JSON (que en Python conceptualizamos como diccionarios). No hay manera de recibir las cadenas directamente, pero con recibir todo el objeto nos alcanza, podemos iterar las claves y valores del mismo y convertir a fecha-hora lo que se pueda.

Para intercambios de datos más complejos y estructurados es normal utilizar claves con nombres particulares que indican el tipo de objeto que está representado en ese JSON e instanciar un objeto Python de ese tipo con esos atributos.

1.2.1.3. XML

El XML, por sus siglas en inglés *eXtensible Markup Language* ("lenguaje de marcado extensible") nos permite codificar datos de una manera extremadamente flexible y compartible, ya que se pueden tener reglas para cualquier tipo de dato y al mismo tiempo no está atado a ningún lenguaje de programación o plataforma.

Veremos que su estructura es parecida a la del mucho más conocido HTML. Ambos provienen de un lenguaje adoptado por la ISO llamado SGML (*Standard Generalized Markup Language*, una evolución del GML creado por IBM en la década de 1970).

En comparación con los formatos que vimos anteriormente, XML es por lejos el más flexible, ya que permite codificar no sólo la información que necesitamos almacenar o transmitir sino tambien todo tipo de metadata necesaria para interpretar esa información. Esta flexibilidad tiene el costo asociado de la complejidad del formato y su muy baja legibilidad.

Su estructura se basa en una estructura de árbol donde cada nodo puede tener un dato u otros nodos, además de "marcas" que proveen información extra sobre esos datos o nodos.

Para visualizar más fácil esto armemos en formato XML lo que podría ser la respuesta del servicio que usamos antes para saber el clima de una ciudad (escribiendo sólo la parte que usamos de clima actual, por brevedad):

```
<?xml version="1.0"?>
<data>
    <current_weather>
        <temperature unit="celsius">20.3</temperature>
        <windspeed unit="knots">10.7</windspeed>
        <winddirection>132.0</winddirection>
        <weathercode>3</weathercode>
        <is_day>1</is_day>
        <time format="ISO">2023-04-22T15:00</time>
    </current_weather>
    <etc>
        ...
    </etc>
</data>
```

Allí vemos la estructura de árbol que mencionamos, así como también información extra asociada a cada dato, útil tanto para parsear el texto del dato (como en el caso de la fecha-hora) como para también potencialmente mostrarle esa información al usuario (como la unidad de temperatura).

Veamos un caso más real: el XML resultante al exportar la página de Argentina de Wikipedia en castellano:

Cell 17

```
resp = requests.get("https://es.wikipedia.org/wiki/Especial:Exportar/Argentina")
fullxml = resp.text
len(fullxml)
```

```
118595
```

No la mostramos toda (¡117 mil bytes!) pero usemos el módulo `textwrap` para ver de forma sencilla y prolija la primera parte del contenido:

CELL 18

```
import textwrap

for chunk in textwrap.wrap(fullxml, width=80, max_lines=10):
    print(chunk)
```

```
<mediawiki xmlns="http://www.mediawiki.org/xml/export-0.11/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.mediawiki.org/xml/export-0.11/
http://www.mediawiki.org/xml/export-0.11.xsd" version="0.11" xml:lang="es">
<siteinfo>     <sitename>Wikipedia</sitename>     <dbname>eswiki</dbname>
<base>https://es.wikipedia.org/wiki/Wikipedia:Portada</base>
<generator>MediaWiki 1.43.0-wmf.23</generator>     <case>first-letter</case>
<namespaces>       <namespace key="-2" case="first-letter">Medio</namespace>
<namespace key="-1" case="first-letter">Especial</namespace>       <namespace
key="0" case="first-letter" />       <namespace key="1" case="first- [...]
```

Una buena manera de ver un XML completo bien estructurado y con la facilidad extra de ocultar/mostrar distintas partes del árbol es abriendo ese XML en un navegador web; prueben esto abriendo la URL del XML que acabamos de descargar.

Pasemos a procesar este XML.

En Python tenemos distintas herramientas para trabajar con este formato. Veamos los distintos casos todos sobre el mismo contenido que acabamos de descargar, siempre con el objetivo de encontrar el título de la página, el identificador y timestamp de esta revisión en particular, y el largo del texto en sí de la página.

La primer biblioteca que usaremos es ElementTree, que nos da una forma simple y eficiente para parsear XMLs, ya que mapea directamente el concepto de "árbol de nodos" de este formato con dos tipos de objetos: el `ElementTree` que representa el documento XML completo, y `Element` que nos permite trabajar con cada nodo.

CELL 19

```
from xml.etree import ElementTree

node = ElementTree.fromstring(fullxml)
print("El primer nodo:", node)
print("Su tag:", node.tag)
print("Atributos:")
for key, value in node.attrib.items():
    print(f"    {key!r}: {value!r}")
print("Texto útil no tiene:", repr(node.text))
print("Pero sí nodos hijos:")
for child in node:
    print("    ", child)
```

```
El primer nodo: <Element '{http://www.mediawiki.org/xml/export-0.11/}mediawiki' at 0x7352af5dd210>
Su tag: {http://www.mediawiki.org/xml/export-0.11/}mediawiki
Atributos:
    '{http://www.w3.org/2001/XMLSchema-instance}schemaLocation':
    ↳ 'http://www.mediawiki.org/xml/export-0.11/ http://www.mediawiki.org/xml/export-0.11.xsd'
    'version': '0.11'
    '{http://www.w3.org/XML/1998/namespace}lang': 'es'
Texto útil no tiene: '\n  '
Pero sí nodos hijos:
     <Element '{http://www.mediawiki.org/xml/export-0.11/}siteinfo' at 0x7352af5df380>
     <Element '{http://www.mediawiki.org/xml/export-0.11/}page' at 0x7352af5b9850>
```

Como ya tenemos el XML completo en una cadena lo parseamos de allí (`ElementTree`, como el resto de las bibliotecas que veremos aquí nos permite tanto leer el contenido desde una cadena como de un archivo). Nos devuelve directamente el primer nodo del documento, que cómo es una estructura de árbol tendemos a llamar "raiz", pero atención que en este caso tenemos más de un nodo en el primer nivel.

Vemos que su tag es `{http://www.mediawiki.org/xml/export-0.10/}mediawiki`, no simplemente `mediawiki`. Esto es porque ElementTree nos muestra siempre (y obliga a trabajar) cada nombre en su espacio de nombre. Los espacios de nombre son para identificar unívocamente a qué documento pertenece un nombre; sería muy util si mezclamos este XML con otro documento que también tiene los tags `mediawiki` o `page`, por ejemplo, pero para manejar un sólo documento puede ser un poco engorroso.

Más allá de eso, ElementTree nos deja acceder de forma bastante directa a toda la información del nodo: tenemos sus atributos en `.attrib`, el texto en `.text`, y lo iteramos directamente para ir a por los hijos. Y también tenemos el método `.find` para buscar un nodo hijo en particular, usémoslo para recorrer el árbol y obtener la información que queríamos:

CELL 20

```
page = node.find("{http://www.mediawiki.org/xml/export-0.11/}page")
title = page.find("{http://www.mediawiki.org/xml/export-0.11/}title")
revision = page.find("{http://www.mediawiki.org/xml/export-0.11/}revision")
rev_id = revision.find("{http://www.mediawiki.org/xml/export-0.11/}id")
rev_tstamp = revision.find("{http://www.mediawiki.org/xml/export-0.11/}timestamp")
rev_text = revision.find("{http://www.mediawiki.org/xml/export-0.11/}text")
textlen = rev_text.attrib["bytes"]
print(f"Página {title.text!r} r.{rev_id.text} ({rev_tstamp.text}) largo={textlen}")
```

```
Página 'Argentina' r.162365205 (2024-09-09T23:59:56Z) largo=115597
```

La segunda herramienta es muy parecida a la anterior pero su API es estándar, definida por la *World Wide Web Consortium (W3C)*. De allí viene el nombre del módulo, DOM, por *Document Object Model*.

Cuando parseamos el XML nos devuelve un "documento", al cual le pedimos el primer nodo hijo:

CELL 21

```
from xml.dom.minidom import parseString

dom = parseString(fullxml)
node = dom.firstChild

print("El primer nodo:", node)
print("Su tag:", node.nodeName)
print("Atributos:")
for key, value in node.attributes.items():
    print(f"    {key!r}: {value!r}")
print('Los nodos "hijos":')
for child in node.childNodes:
    print("    ", child)
```

```
El primer nodo: <DOM Element: mediawiki at 0x7352b425fad0>
Su tag: mediawiki
Atributos:
    'xmlns': 'http://www.mediawiki.org/xml/export-0.11/'
    'xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance'
    'xsi:schemaLocation': 'http://www.mediawiki.org/xml/export-0.11/
    ↳ http://www.mediawiki.org/xml/export-0.11.xsd'
    'version': '0.11'
    'xml:lang': 'es'
Los nodos "hijos":
     <DOM Text node "'\n  '">
     <DOM Element: siteinfo at 0x7352b425f770>
     <DOM Text node "'\n  '">
     <DOM Element: page at 0x7352af5d7020>
     <DOM Text node "'\n'">
```

Si comparamos los atributos de este nodo con lo que teníamos en ElementTree vemos que esta otra biblioteca no nos está manejando los espacios de nombres: los tenemos sin expandir en los atributos del primer nodo y no está incluido en el tag.

Al iterar un nodo del DOM obtenemos tanto los nodos hijos como otros "pseudo nodos" de

tipo `Text`, lo cual es un poco confuso de trabajar. Este detalle es aún más notorio cuando queremos obtener la información necesaria del XML, donde aparece incluso cuando queremos el largo del texto: en el atributo `bytes` del nodo `text` para sacar el valor debemos obtener una especie de nodo hijo y acceder a sus datos:

CELL 22

```
page = dom.getElementsByTagName("page")[0]
title = page.getElementsByTagName("title")[0]
revision = page.getElementsByTagName("revision")[0]
rev_id = revision.getElementsByTagName("id")[0]
rev_tstamp = revision.getElementsByTagName("timestamp")[0]
rev_text = revision.getElementsByTagName("text")[0]
textlen = rev_text.attributes["bytes"].firstChild.data

title_text = title.firstChild.data
rev_id_text = rev_id.firstChild.data
rev_tstamp_text = rev_tstamp.firstChild.data

print(f"Página {title_text!r} r.{rev_id_text} ({rev_tstamp_text}) largo={textlen}")
```

```
Página 'Argentina' r.162365205 (2024-09-09T23:59:56Z) largo=115597
```

En este caso de uso es explícito que cuando buscamos cada nodo hijo (por ejemplo `page`) realmente nos estamos quedando con el primero, potencialmente podría haber más. Es lo mismo que pasaba con ElementTree, donde usábamos el método `.find` que también nos trae el primero que encuentre.

Finalmente, la tercer biblioteca nos provee un conjunto de herramientas que implementan una forma de acceder a la información de cada nodo al parsear el XML: la *Simple API for XML (SAX)*. En este caso no tendremos toda la estructura del XML como un árbol sino que tendremos que implementar nuestros propios mecanismos para acceder a la información correspondiente.

La forma más fácil de lograr esto es heredar del `ContentHandler` que trae el módulo por default e implementar tres métodos que se irán llamando automáticamente al parsear el XML: `startElement` para cuando arranca un elemento, `endElement` para cuando termina, y `characters` para el texto de cada nodo.

En la siguiente celda vemos como armamos nuestra clase siguiendo ese esquema, y luego usamos esa clase para parsear el XML, mostrando directamente la información necesaria al final:

CELL 23

```
import xml.sax

class Handler(xml.sax.handler.ContentHandler):

    def __init__(self):
        self.for_text = {
            ("mediawiki", "page", "title"): "title",
            ("mediawiki", "page", "revision", "id"): "rev_id",
            ("mediawiki", "page", "revision", "timestamp"): "rev_tstamp",
        }
        self.for_attribs = {
            ("mediawiki", "page", "revision", "text"): "text_attr",
        }
        self.data = {}
        self.branch = []

    def startElement(self, name, attrs):
        self.branch.append(name)
        branch = tuple(self.branch)
        if branch in self.for_attribs:
            key = self.for_attribs[branch]
            self.data[key] = dict(attrs)

    def endElement(self, name):
        assert self.branch[-1] == name
        self.branch.pop()

    def characters(self, content):
        branch = tuple(self.branch)
        if branch in self.for_text:
            key = self.for_text[branch]
            self.data[key] = self.data.get(key, "") + content

handler = Handler()
xml.sax.parseString(fullxml, handler)
print("Página {title!r} r.{rev_id} ({rev_tstamp}) largo={text_attr[bytes]}".format_map(handler.data))
```

```
Página 'Argentina' r.162365205 (2024-09-09T23:59:56Z) largo=115597
```

En el método de inicialización tenemos algunas estructuras necesarias para el procesamiento, basadas en el concepto de "rama" del árbol del XML: una secuencia de nodos desde una "raíz" hasta la "hoja" que necesitamos:

- `data`: un diccionario donde guardaremos los resultados del procesamiento
- `for_text` y `for_attribs`: dos diccionarios donde la clave es la rama para indicar la hoja que nos interesa (en el primer caso guardar su texto y en el segundo sus atributos) y el valor es a su vez la clave con que guardaremos esa información en el atributo `data`.
- `branch`: una lista que nos representa "la rama actual"

En `startElement` y `endElement` agregamos y sacamos el nodo en cuestión de `self.branch` para construir/mantener en qué rama estamos; además, en el primero de los dos (donde entramos al

nodo y recibimos sus atributos) verificamos si la rama actual es interesante justamente para guardar esos atributos.

En `characters` recibimos el texto de los distintos nodos, el cual guardaremos si la rama está indicada para ello. En este método hay que tener en cuenta que podemos recibir el contenido de un nodo en varias llamadas, así que no alcanza con guardarlo directamente en el diccionario de resultados: tenemos que concatenar las posibles partes.

Vemos que para casos simples es mucho más fácil utilizar alguna de las dos bibliotecas anteriores. Pero el fuerte de SAX es que no tiene que cargar y procesar el XML completo de una sola vez, ya que irá llamando a nuestros métodos y será responsabilidad de nuestro código de extraer sólo la información relevante. De esta manera podemos procesar eficientemente XMLs muy grandes, lo cual sería muy costoso si hay muchos y grandes, o directamente imposible si alguno es más grande que la memoria de nuestra computadora.

1.3. Trabajando con datos binarios

Cuando mencionamos archivos de forma genérica en 1.1 vimos la operatoria básica: abrir, cerrar, escribir, leer; entraremos en esta sección en detalles particulares al trabajar con formatos binarios.

Habiendo dicho eso, volveremos a mostrar aquí cómo escribir bytes en un archivo aunque sea similar a lo que ya vimos.

CELL 01

```
some_bytes = b"abcd\x01\x02\x03\x04"

with open("/tmp/archivoprueba", "wb") as fh:
    fh.write(some_bytes)

with open("/tmp/archivoprueba", "rb") as fh:
    print(fh.read())
```

```
b'abcd\x01\x02\x03\x04'
```

Para leer, a diferencia de cuando trabajamos con texto, en general no iteraremos el *file handler*; el mismo sí es iterable, y nos devolverá el valor de cada byte uno a uno, pero es raro necesitar los bytes individualmente.

Además de leer *todo* el archivo (como recién mostramos para visualizar lo que acabamos de escribir) también podemos leer por partes. Si al `read` le pedimos la cantidad de bytes devolverá solamente lo indicado (a menos que se le termine un archivo, en cual caso devolverá todo lo que pueda), dejando un puntero en donde dejó de leer. Si volvemos a leer, leerá desde allí (y si no indicamos la cantidad, leerá desde esa posición hasta el final).

CELL 02

```
fh = open("/tmp/archivoprueba", "rb")
# leemos dos bytes
fh.read(2)
```

```
b'ab'
```

CELL 03

```
# vemos donde está el puntero
fh.tell()
```

```
2
```

CELL 04

```
# leemos tres bytes más
fh.read(3)
```

```
b'cd\x01'
```

CELL 05

```
# leemos hasta el final
fh.read()
```

```
b'\x02\x03\x04'
```

A menos que el archivo sea trivial siempre estaremos leyendo sólo lo que necesitamos y no todo el binario en una sola acción. En función de la estructura de los datos a leer podemos incluso evitar leer secuencialmente el archivo para llegar a la parte que necesitamos: si sabemos en qué posición del archivo está la información que necesitamos podemos mover el puntero hasta allí y leer.

CELL 06

```
with open("/tmp/archivoprueba", "rb") as fh:
    # vamos hasta la cuarta posición
    fh.seek(4)
    # leemos
    print(fh.read(2))
```

```
b'\x01\x02'
```

En el ejemplo mostrado el `seek` posiciona al puntero en la cuarta posición desde el *principio* del archivo. Pero también podemos movernos relativamente a la posición actual o al final del archivo:

CELL 07

```
import os

with open("/tmp/archivoprueba", "rb") as fh:
    # leemos los últimos dos bytes
    fh.seek(-2, os.SEEK_END)
    print(fh.read())
```

```
b'\x03\x04'
```

No es raro al trabajar con archivos binarios ir "saltando" por distintas partes del archivo para leer sólo lo necesario (en contraposición a una lectura puramente secuencial, como es normal en los archivos de texto).

Más allá de la forma de lectura, el siguiente paso es "interpretar" la información que estamos leyendo. Normalmente no nos interesan los bytes "crudos" al leer un archivo binario, sino convertirlos a números, texto, o las estructuras pertinentes para los tipos de datos con los que estamos trabajando.

Para ilustrar esto tomemos una secuencia de bytes simple (que grabamos primero en el archivo para simular el ejemplo) y luego leemos en función de la siguiente estructura predefinida:

- el primer byte nos indica la longitud de un texto guardado (codificado en UTF-8)
- el segundo byte nos indica la cantidad de números de 32 bits a encontrar
- luego viene el texto (longitud variable, indicada al principio)
- y por último los números que mencionamos

CELL 08

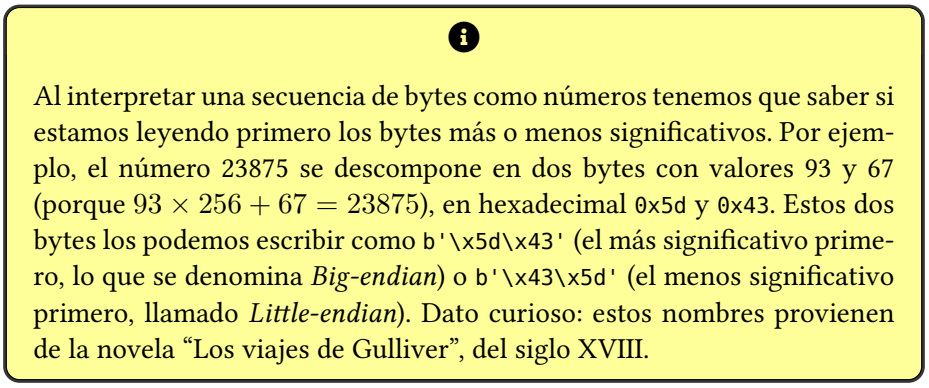
```
# escribimos la secuencia que vamos a leer
sequence = (
    b"\x05\x07A\xc3\xb1os"
    b"\x00\x00\x07\xe3\x00\x00\x07\xe7"
    b"\x00\x00\x07\xe2\x00\x00\x07\xd5"
    b"\x00\x00\x07\xc9\x00\x00\x07\xcb"
    b"\x00\x00\x07\xd6"
)
with open("/tmp/archivoprueba", "wb") as fh:
    fh.write(sequence)

# leemos e interpretamos los bytes del archivo
with open("/tmp/archivoprueba", "rb") as fh:
    len_text = int.from_bytes(fh.read(1), byteorder="big")
    quant_numbers = int.from_bytes(fh.read(1), byteorder="big")
    text_bytes = fh.read(len_text)
    print(f"(debug: {text_bytes=})")
    numbers = []
    for _ in range(quant_numbers):
        number = int.from_bytes(fh.read(4), byteorder="big")
        numbers.append(number)
    print(f"(debug: {numbers=})")

text = text_bytes.decode("utf8")
num_sequence = ', '.join(map(str, numbers))
print(f"{text}: {num_sequence}")
```

```
(debug: text_bytes=b'A\xc3\xb1os')
(debug: numbers=[2019, 2023, 2018, 2005, 1993, 1995, 2006])
Años: 2019, 2023, 2018, 2005, 1993, 1995, 2006
```

Vemos que luego de abrir el archivo leemos un sólo byte y lo interpretamos como número para el largo del texto, y luego hacemos lo mismo para la cantidad de números. Para obtener el texto leemos la cantidad necesaria de bytes y decodificamos. Para los números vamos leyendo de a 4 bytes y convirtiendo a entero (indicando que están ordenados según *Big-endian*). Más allá de dos líneas de *debug* que dejamos para que se entienda mejor el proceso, finalmente mostramos los datos de forma "elegante".



Al interpretar una secuencia de bytes como números tenemos que saber si estamos leyendo primero los bytes más o menos significativos. Por ejemplo, el número 23875 se descompone en dos bytes con valores 93 y 67 (porque $93 \times 256 + 67 = 23875$), en hexadecimal `0x5d` y `0x43`. Estos dos bytes los podemos escribir como `b'\x5d\x43'` (el más significativo primero, lo que se denomina *Big-endian*) o `b'\x43\x5d'` (el menos significativo primero, llamado *Little-endian*). Dato curioso: estos nombres provienen de la novela "Los viajes de Gulliver", del siglo XVIII.

Esta técnica es sencilla y nos puede resultar efectiva para estructuras pequeñas. Pero para casos más complejos tiende a ser muy propensa a errores, porque las distintas lecturas e interpretaciones están distribuidas a lo largo de varias líneas de código.

Hay una forma más simple de realizar esto, usando la función `unpack` del módulo `struct`, que nos permite especificar el formato de la estructura que queremos interpretar desde la cadena de bytes y luego hace todo el trabajo:

CELL 09

```
import struct

with open("/tmp/archivoprueba", "rb") as fh:
    len_text, quant_numbers = struct.unpack("BB", fh.read(2))
    complex_format = f">{len_text}s{quant_numbers}i"
    text_bytes, *numbers = struct.unpack(complex_format, fh.read())
    print(f"(debug: {text_bytes=})")
    print(f"(debug: {numbers=})")

text = text_bytes.decode("utf8")
num_sequence = ', '.join(map(str, numbers))
print(f"{text}: {num_sequence}")
```

```
(debug: text_bytes=b'A\xc3\xb1os')
(debug: numbers=[2019, 2023, 2018, 2005, 1993, 1995, 2006])
Años: 2019, 2023, 2018, 2005, 1993, 1995, 2006
```

La primera vez que lo usamos es para sacar las dos cantidades: el largo del texto y cuantos números hay. El formato usado es `'BB'`, que indica que hay dos enteros sin signo de 1 byte cada uno.

Luego armamos un formato más complejo, donde con `'>'` indicamos que es *Big-endian* (tiene que estar al principio del formato), luego el largo del texto y `'s'` para indicar una cadena de cuantos bytes, y finalmente la cantidad de números y `'i'` para la cantidad de enteros con signo de 4 bytes de largo cada uno. Termina armando `'>5s7i'`, que nos devolverá una cadena de cinco bytes y los siete números. Cabe aclarar que la repetición del formato `'s'` funciona distinto que la de la mayoría de los otros formatos (como el `'i'`); en el primer caso devuelve *una* cadena de N de largo, mientras que en el caso más común la repetición devuelve *N* elementos.

Entonces cuando desempacamos usando ese formato complejo obtendremos una cadena de bytes y algunos números, los cuales asignamos respectivamente a `text` y `numbers` (que como tiene un * al principio del nombre consume todos los elementos remanentes, armando efectivamente

una lista). Podemos validar en las líneas de *debug* que obtenemos la misma información que en el ejemplo anterior, y obviamente la línea final mostrada será también igual.

Un efecto colateral fantástico de usar `struct` para interpretar estructuras de bytes (o para armarlas) es que una sola llamada a nivel Python alcanza para todo el procesamiento, que se realiza a nivel de C compilado, con lo cual es extremadamente eficiente y permite procesar una gran cantidad de información en poco tiempo.

Veamos otro ejemplo apenas más complejo pero más real: interpretar el contenido de una imagen con formato PNG.

CELL 10

```
from collections import namedtuple

IHDR_struct = '>IIbbbbb'
IHDR = namedtuple('IHDR', 'width height bit_depth color_type compression filter interlace')

with open("logo.png", "rb") as fh:
    header = fh.read(8)
    assert header == b"\x89PNG\r\n\x1A\n"

    while True:
        octet = fh.read(8)
        if not octet:
            break
        length, chunk_type = struct.unpack(">I4s", octet)
        print(f"Chunk {chunk_type!r} len={length}")
        chunk_data = fh.read(length)
        if chunk_type == b"IHDR":
            ihdr = IHDR._make(struct.unpack(IHDR_struct, chunk_data))
            print(f"    width={ihdr.width} height={ihdr.height}")

        fh.read(4)  # CRC, not for this example
```

```
Chunk b'IHDR' len=13
    width=1280 height=640
Chunk b'pHYs' len=9
Chunk b'tEXt' len=25
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=1393
Chunk b'IEND' len=0
```

Al principio del código definimos dos estructuras que nos servirán para parsear el campo `IHDR` de un PNG (que nos da información sobre la imagen como su ancho y alto, que es lo que mostraremos); primero el formato para `struct` y luego una tupla con nombres para guardar los distintos items resultantes al parsear ese header con el formato indicado. El PNG tiene otros campos pero en este ejemplo sólo parsearemos el ya nombrado.

Luego abrimos el archivo y antes que nada validamos que los primeros bytes del mismo correspondan a la firma que todos los PNG tienen, para confirmar que el formato es ese. Luego entramos en un bucle "infinito" realizando los siguientes pasos:

- tratamos de leer ocho bytes; si no obtuvimos nada es que se terminó el archivo y salimos
- interpretamos esos ocho bytes como un entero de 32 bits (el largo de la cantidad de bytes a continuación) y una cadena de cuatro bytes-caracteres (su tipo); mostramos ambos datos
- leemos la porción de bytes que viene a continuación, usando el largo recién obtenido
- si el tipo es `IHDR`, lo interpretamos como dos enteros de 32 bits y cuatro enteros de 8 bits, según la estructura definida arriba, metiendo todos esos datos directamente en una `namedtuple` para poder acceder a la info por nombre
- de la estructura `IHDR` mostramos el ancho y alto del PNG
- consumimos los 4 bytes de la verificación del bloque (que no usamos en este simple ejemplo)

Veamos otro caso donde leemos directamente desde un dispositivo de hardware: el mouse. Al mismo podemos acceder, en Linux, abriendo un *path* especial (con permisos de *root*) como si fuese un archivo:

```
1  import struct
2
3  with open("/dev/input/mice", "rb") as fh:  # needs root permissions
4      while True:
5          try:
6              data = fh.read(3)
7              button, x, y = struct.unpack(">bbb", data)
8
9              left = button & 0x1
10             right = button & 0x2
11             middle = button & 0x4
12
13             print(f"{x=}, {y=}, {left=}, {middle=}, {right=}")
14         except KeyboardInterrupt:
15             break
```

Para leer del dispositivo tenemos una estructura de "bucle infinito hasta que se interrumpe con CTRL-C", donde en cada ciclo leemos 3 bytes; cada uno es un número entero: el primero tiene información de los botones, y el segundo y tercero el delta X e Y de la posición. Interpretamos y mostramos.

Cabe la pena mencionar que aunque `struct` nos facilita enormemente el interpretar distintas secuencias de bytes no nos ayuda para interpretar distintos bits dentro de un mismo byte. Es por eso que en las líneas 9 a 11 tenemos que aplicar distintas máscaras para detectar si algunos de los tres bits menos significativos (que corresponden a los tres botones más normales de un mouse) están activados.

Una salida posible de este programa, al mover un poco el mouse y luego apretar el botón derecho, sería algo como lo siguiente:

```
\$ sudo python3 raton.py
x=0, y=-1, left=0, middle=0, right=0
```

```
x=-1, y=-1, left=0, middle=0, right=0
x=-1, y=0, left=0, middle=0, right=0
x=-3, y=-2, left=0, middle=0, right=0
x=-1, y=0, left=0, middle=0, right=0
x=-2, y=0, left=0, middle=0, right=0
x=-1, y=-1, left=0, middle=0, right=0
x=-2, y=0, left=0, middle=0, right=0
x=-1, y=0, left=0, middle=0, right=0
x=-2, y=-1, left=0, middle=0, right=0
x=-2, y=0, left=0, middle=0, right=0
x=-2, y=-1, left=0, middle=0, right=0
x=0, y=0, left=0, middle=0, right=1
```

1.3.1. Un formato binario muy usado: HDF5

El formato `HDF5` es la versión 5 del HDF (*Hierarchical Data Format*, "formato jerárquico de datos"), un formato de archivo diseñado para almacenar y organizar grandes cantidades de datos.

En esta versión la estructura del archivo está simplificada para incluir dos tipos de objetos: los conjuntos de datos (*datasets*) dispuestos como arreglos multidimensionales con el tipo definido, y grupos, que son básicamente contenedores con un nombre que dentro pueden tener a los conjuntos de datos u otros grupos.

La biblioteca `h5py` representa toda esa estructura como un diccionario, donde cada clave será el nombre de un grupo y el valor su contenido, que será otro diccionario si es un grupo o directamente el conjunto de datos.

Para ejemplificar su uso abriremos y usaremos un archivo HDF5 con información atmosférica de Estados Unidos (descargamos el `all.h5` de allí). Abramos entonces el archivo y exploremos cuales son sus grupos en el primer nivel:

CELL 11

```
import h5py

fh = h5py.File("all.h5", "r")
fh.keys()
```

```
<KeysViewHDF5 ['datetime', 'latitude', 'longitude', 'mean', 'n', 'std', 'wk']>
```

Cabe aclarar que al archivo lo podemos abrir usando la sintaxis de administrador de contexto (como ya vimos **??** para otros archivos), pero en este caso lo abrimos de forma clásica para poder mostrarlo mejor en estos ejemplos.

Este archivo es bastante simple, tiene directamente los conjuntos de datos sin ningún subgrupo:

CELL 12

```
for dataset in fh.values():
    print(dataset)
```

```
<HDF5 dataset "datetime": shape (1, 61368), type "|S14">
<HDF5 dataset "latitude": shape (581777,), type "<f4">
<HDF5 dataset "longitude": shape (581777,), type "<f4">
<HDF5 dataset "mean": shape (581777, 1), type "<f4">
<HDF5 dataset "n": shape (581777, 1), type "<i4">
<HDF5 dataset "std": shape (581777, 1), type "<f4">
<HDF5 dataset "wk": shape (581777, 1), type "<f4">
```

Podría ser incluso más simple, ya que casi todos los arreglos tienen una dimensión extra (de valor uno) que no tiene sentido. Veamos cómo acceder en los distintos casos y exploremos apenas los valores del grupo `latitude`:

CELL 13

```
print("datetime:", fh["datetime"][0])  # el primer elemento tiene todo el array
print("latitude:", fh["latitude"][42])  # aquí lo tenemos directamente
print("mean:", fh["mean"][123])  # y acá cada valor está dentro de una lista

fh["latitude"][:20]
```

```
datetime: [b'20110801230000' b'20110801220000' b'20110801210000' ...
 b'20110402020000' b'20110402010000' b'20110402000000']
latitude: 24.026184
mean: [7.5135026]

array([24.01368 , 24.014069, 24.01445 , 24.014832, 24.01519 , 24.015564,
       24.015923, 24.016289, 24.016632, 24.01699 , 24.017326, 24.017662,
       24.018005, 24.018349, 24.01867 , 24.018997, 24.019302, 24.019623,
       24.019936, 24.020256], dtype=float32)
```

HDF5 permite agregarle atributos a cada conjunto de datos (para "anotarlos" con cualquier información relevante), así como también nombrar sus distintas dimensiones, pero en este archivo esas facilidades no fueron utilizadas:

CELL 14

```
print("Dimensiones sin nombre:")
for dim in fh["mean"].dims:
    print("    ", dim)
print("Tampoco hay atributos:", len(fh["latitude"].attrs))
```

```
Dimensiones sin nombre:
     <"" dimension 0 of HDF5 dataset at 129675587088320>
     <"" dimension 1 of HDF5 dataset at 129675587088320>
Tampoco hay atributos: 0
```

Los *datasets* de `HDF5` son conceptualmente similares a los arreglos de NumPy ?? y `hp5y` nos da un objeto que se les parece mucho, pero no lo son exactamente. De cualquier manera es trivial crear un arreglo de NumPy a partir del *dataset* para hacer análisis más complejos, como el siguiente ejemplo donde calculamos la media de temperatura para los puntos con latitud entre

27 y 28 y en cualquier longitud:

Cell 15

```
import numpy as np

latitudes = np.array(fh["latitude"])
latitudes
```

```
array([24.01368 , 24.014069, 24.01445 , ..., 49.291092, 49.283897,
       49.27669 ], dtype=float32)
```

Cell 16

```
lat_27 = (latitudes >= 27) & (latitudes < 28)
np.count_nonzero(lat_27)
```

```
52402
```

Cell 17

```
means = np.squeeze(fh["mean"])
means
```

```
array([7.4238734, 7.4596395, 7.5191693, ..., 9.926428 , 9.930207 ,
       9.933213 ], dtype=float32)
```

Cell 18

```
means[lat_27].mean()
```

```
6.9832783
```

Tomamos las latitudes como arreglo de NumPy para poder crear una máscara con `True` cuando el valor está entre 27 y 28, mostrando que hay más de cincuenta mil puntos con esa característica. Luego tomamos los valores medios de cada punto, pero como ese *dataset* tiene una dimensión extra que no queremos tenemos que usar `squeeze` de NumPy, viendo cómo nos queda un arreglo listo para usar. Luego aplicamos la máscara quedándonos con los valores que corresponden a la latitud que nos interesa, y calculamos el valor medio del total de esos datos.

1.4. Pandas

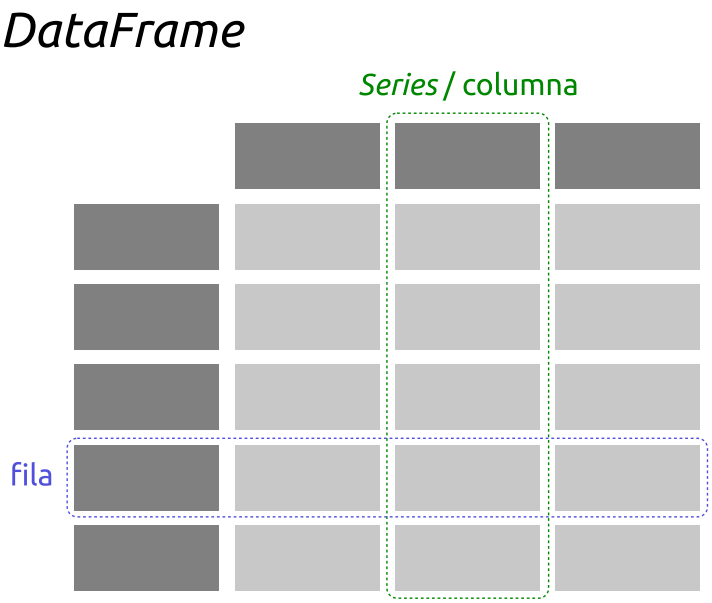
La biblioteca Pandas se ha establecido como la gran herramienta para trabajar con tablas de datos, ofreciendo utilidades que permite realizar diversos procesamientos de forma simple pero poderosa.

Se enfoca en permitir manejar tablas de dos dimensiones. A priori parece una restricción pero tengamos en cuenta que esto modela exactamente la mayoría de las fuentes de datos como pueden ser bases de datos, planillas de cálculo, archivos separados por coma, etc. Pandas denomina *DataFrame* a esta tabla de datos.

Cada *dataframe* puede ser considerado entonces como un conjunto de columnas donde cada una de estas últimas será una serie de escalares de algún tipo de datos específico. A estas se las

llama *Series*.

Más allá de esas dos denominaciones puntuales y en función de la naturaleza tabular de la estructura de datos, al trabajar con Pandas también es costumbre hablar de "columnas" (una serie de datos puntual) y "filas" (los datos para la misma posición de las distintas columnas).



Veamos estos conceptos en un ejemplo. Descargamos un CSV de la tasa de Natalidad por 1000 habitantes registrada en la República Argentina y lo abrimos con Pandas:

Cell 01

```
import pandas as pd

nats = pd.read_csv("tasa-natalidad-deis-2000-2022.csv")
type(nats)
```

```
pandas.core.frame.DataFrame
```

Uno de los primeros pasos al trabajar con estructuras de datos es entender exactamente esas estructuras: qué columnas hay y con qué tipo de dato cada una, si contienen valores nulos, cuantas filas hay en total, etc.

Podemos pedirle a Pandas que nos de un resumen:

CELL 02

```
nats.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23 entries, 0 to 22
Data columns (total 26 columns):
 #   Column                                    Non-Null Count  Dtype
---  ------                                    --------------  -----
 0   indice_tiempo                             23 non-null     object
 1   natalidad_argentina                       23 non-null     float64
 2   natalidad_ciudad_autonoma_de_buenos_aires 23 non-null     float64
 3   natalidad_buenos_aires                    23 non-null     float64
 4   natalidad_catamarca                       23 non-null     float64
 5   natalidad_cordoba                         23 non-null     float64
 6   natalidad_corrientes                      23 non-null     float64
 7   natalidad_chaco                           23 non-null     float64
 8   natalidad_chubut                          23 non-null     float64
 9   natalidad_entre_rios                      23 non-null     float64
 10  natalidad_formosa                         23 non-null     float64
 11  natalidad_jujuy                           23 non-null     float64
 12  natalidad_la_pampa                        23 non-null     float64
 13  natalidad_la_rioja                        23 non-null     float64
 14  natalidad_mendoza                         23 non-null     float64
 15  natalidad_misiones                        23 non-null     float64
 16  natalidad_neuquen                         23 non-null     float64
 17  natalidad_rio_negro                       23 non-null     float64
 18  natalidad_salta                           23 non-null     float64
 19  natalidad_san_juan                        23 non-null     float64
 20  natalidad_san_luis                        23 non-null     float64
 21  natalidad_santa_cruz                      23 non-null     float64
 22  natalidad_santa_fe                        23 non-null     float64
 23  natalidad_santiago_del_estero             23 non-null     float64
 24  natalidad_tucuman                         23 non-null     float64
 25  natalidad_tierra_del_fuego                23 non-null     float64
dtypes: float64(25), object(1)
memory usage: 4.8+ KB
```

Vemos que tenemos un dataframe con 23 filas (numeradas del 0 al 22, también podemos verlas con `df.index`) y 26 columnas (también podemos ver sus títulos con `df.columns`) donde todas menos la primera (las natalidades) son punto flotante y esa primera (el índice de tiempo) es un objeto genérico; el conteo de "no nulas" es igual al total de filas, o sea que tenemos una tabla completa.

A cada una de esas columnas (*series*) podemos referenciarlas por su nombre o título; aquí vemos los valores de la natalidad de toda Argentina (los primeros cinco, por brevedad):

CELL 03

```
natalidad_argentina = nats["natalidad_argentina"]
natalidad_argentina.head()
```

```
0    19.0
1    18.2
2    18.3
3    18.4
4    19.3
Name: natalidad_argentina, dtype: float64
```

También podemos mencionar varias columnas, lo que efectivamente nos da otro dataframe (¿copiando los valores o es una vista que sólo los referencia? No lo podemos saber de antemano, Pandas hace lo más eficiente según el momento en cada caso):

CELL 04

```
natarg_con_tiempos = nats[["indice_tiempo", "natalidad_argentina"]]
natarg_con_tiempos.head()
```

```
  indice_tiempo  natalidad_argentina
0    2000-01-01                 19.0
1    2001-01-01                 18.2
2    2002-01-01                 18.3
3    2003-01-01                 18.4
4    2004-01-01                 19.3
```

Para entender las columnas de punto flotante (que son los valores medidos) podemos calcular un análisis de cada serie de manera muy sencilla:

CELL 05

```
natarg_con_tiempos.describe()
```

```
       natalidad_argentina
count            23.000000
mean             16.956522
std               2.540549
min              10.700000
25%              16.350000
50%              17.900000
75%              18.500000
max              19.300000
```

Para filtrar parte de la tabla podemos trabajar con máscaras como ya vimos con NumPy:

CELL 06

```
mask = natarg_con_tiempos["indice_tiempo"] >= "2015-01-01"
natarg_con_tiempos[mask]
```

```
   indice_tiempo  natalidad_argentina
15    2015-01-01                 17.9
16    2016-01-01                 16.7
17    2017-01-01                 16.0
18    2018-01-01                 15.4
19    2019-01-01                 13.9
20    2020-01-01                 11.8
21    2021-01-01                 11.6
22    2022-01-01                 10.7
```

Volviendo al *dataframe* original, podemos hacer referencia a secciones de la tabla también por índice a través del método `iloc` (por *integer-location*), donde usamos la sintaxis de *slicing* también de NumPy (que es la normal de Python llevada a varias dimensiones) que para dos dimensiones podemos señalar como `rows_slice,cols_slice`.

En el siguiente paso entonces hacemos referencia a todas las filas (nada antes de la coma) y desde la segunda hasta la séptima columna, efectivamente obteniendo toda la info de las primeras cinco provincias (mostramos las primeras tres filas por brevedad, como en muchos de los próximos ejemplos):

CELL 07

```
provinces = nats.iloc[:, 2:7]
provinces[:3]
```

```
   natalidad_ciudad_autonoma_de_buenos_aires  natalidad_buenos_aires  \
0                                       14.3                    17.5
1                                       13.9                    16.9
2                                       13.6                    17.0

   natalidad_catamarca  natalidad_cordoba  natalidad_corrientes
0                 25.8               17.2                  22.7
1                 24.9               15.9                  21.9
2                 24.4               16.6                  23.3
```

Los *dataframes* tienen muchos métodos para realizar operaciones en ellos, y en general nos permiten indicar sobre qué eje se realiza la operación. Por ejemplo para sacar el mínimo y máximo de cada fila le indicamos que es en el eje 1, y obtenemos una nueva columna que incorporamos directamente a nuestra tabla general de natalidades:

CELL 08

```
nats["prov_min"] = provinces.min(axis=1)
nats["prov_max"] = provinces.max(axis=1)
reduced = nats[["indice_tiempo", "natalidad_argentina", "prov_min", "prov_max"]]
reduced[:3]
```

```
  indice_tiempo  natalidad_argentina  prov_min  prov_max
0    2000-01-01                 19.0      14.3      25.8
1    2001-01-01                 18.2      13.9      24.9
2    2002-01-01                 18.3      13.6      24.4
```

Allí también vemos como sacamos una nueva tabla de la general (`reduced`) (porque ya no nos interesa el dato de cada provincia). Esta nueva tabla "reducida" de la general tiene solamente las cuatro columnas que indicamos, y podemos seguir operando sobre ella.

Pandas nos permite realizar operaciones entre las columnas (elemento por elemento a través de las filas), por ejemplo para agregar una nueva columna `range` que es la diferencia entre el máximo y el mínimo de cada fila:

CELL 09

```
reduced.loc[:, "range"] = reduced["prov_max"] - reduced["prov_min"]
reduced.loc[:, "año"] = reduced["indice_tiempo"].str[:4]
reduced[:3]
```

```
  indice_tiempo  natalidad_argentina  prov_min  prov_max  range   año
0    2000-01-01                 19.0      14.3      25.8   11.5  2000
1    2001-01-01                 18.2      13.9      24.9   11.0  2001
2    2002-01-01                 18.3      13.6      24.4   10.8  2002
```

También tenemos acceso al atributo `str` para manejar cada valor como si fuese una cadena (con los métodos de las cadenas que estamos acostumbrados o directamente obteniendo parte de la misma como estamos haciendo en el ejemplo) e incluso un `apply` al que le podríamos pasar una función de Python creada por nosotros.

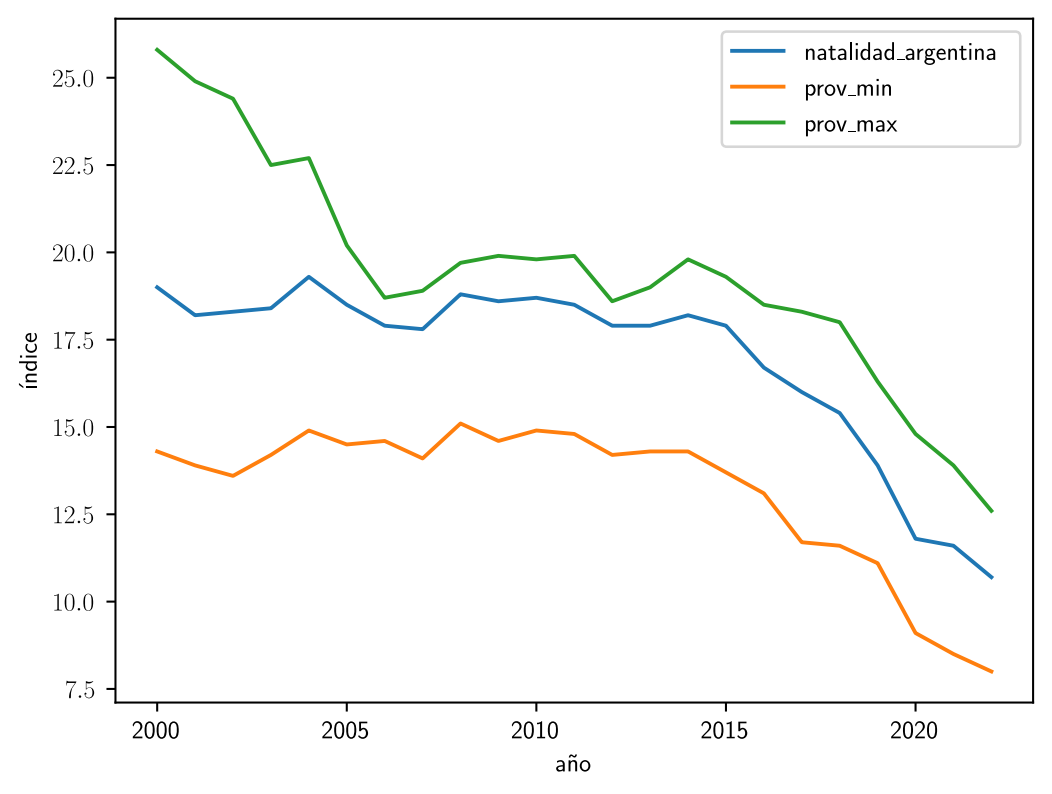
Noten como usamos otra sintaxis para crear una nueva columna en la tabla. Esto es un requerimiento de Pandas porque a priori no se sabe si `reduced` es efectivamente una vista de `nats` o una copia de una sección de la misma (Pandas resuelve qué hacer en cada caso en tiempo de ejecución).

Una vez que están los datos procesados en general vamos a querer sacar algún gráfico o guardarlos para un procesamiento posterior.

Pandas tiene una muy buena integración con Matplotlib, la biblioteca de gráficos que vimos en otras secciones del libro. Vemos como podemos plotear directamente desde un *dataframe*, incluso pasando opciones útiles para generar el gráfico según queramos.

Cell 10

```
cols = reduced[["año", "natalidad_argentina", "prov_min", "prov_max"]]
subplot = cols.plot(x="año", ylabel="índice")
```



Incluso obtenemos el *plot* de Matplotlib, así que en realidad tenemos todo su potencial al alcance de la mano.

Cell 11

```
type(subplot)
```

```
matplotlib.axes._axes.Axes
```

Si quisiéramos trabajar los datos a más bajo nivel podemos obtener un arreglo multidimensional de NumPy sin esfuerzo:

Cell 12

```
reduced[:3].to_numpy()
```

```
array([['2000-01-01', 19.0, 14.3, 25.8, 11.5, '2000'],
       ['2001-01-01', 18.2, 13.9, 24.9, 10.999999999999998, '2001'],
       ['2002-01-01', 18.3, 13.6, 24.4, 10.799999999999999, '2002']],
      dtype=object)
```

También podemos exportar los datos a diversos formatos, como por ejemplo los que ya vimos en este capítulo CSV y HDF5, pero hay muchas más opciones.

CELL 13

```
csv_dumped = reduced[:3].to_csv()
for linea in csv_dumped.split("\n"):
    print(linea)   # para mostrarlo más sencillamente aqui
```

```
,indice_tiempo,natalidad_argentina,prov_min,prov_max,range,año
0,2000-01-01,19.0,14.3,25.8,11.5,2000
1,2001-01-01,18.2,13.9,24.9,10.999999999999998,2001
2,2002-01-01,18.3,13.6,24.4,10.799999999999999,2002
```

CELL 14

```
key = "/data"  # el grupo base en la estructura de HDF5
reduced.to_hdf(path_or_buf="/tmp/testpandas.h5", key=key)
```

CELL 15

```
import h5py

fh = h5py.File("/tmp/testpandas.h5", "r")
[title.decode("utf8") for title in fh["data"]["axis0"]]
```

```
['indice_tiempo',
 'natalidad_argentina',
 'prov_min',
 'prov_max',
 'range',
 'año']
```

Para terminar, unas palabras sobre el proceso de instalación de Pandas. Aunque es trivial instalarlo (usando `pip install` en un entorno virtual como ya vimos en ??) tenemos que ser conscientes que hay distintas "opciones", donde cada opción instalará dependencias en función de la utilización que vamos a hacer de la biblioteca. Por ejemplo, si vamos a usar Pandas para realizar gráficos a partir de los datos necesitamos hacer `pip install pandas[plot]`, si vamos a exportar o importar datos de planillas de cálculo haremos `pip install pandas[excel]`, etc.

Pueden revisar las distintas dependencias opcionales, aunque siempre es más sencillo (pero menos eficiente) indicarle que traiga directamente todo con `pip install pandas[all]`.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [2].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.