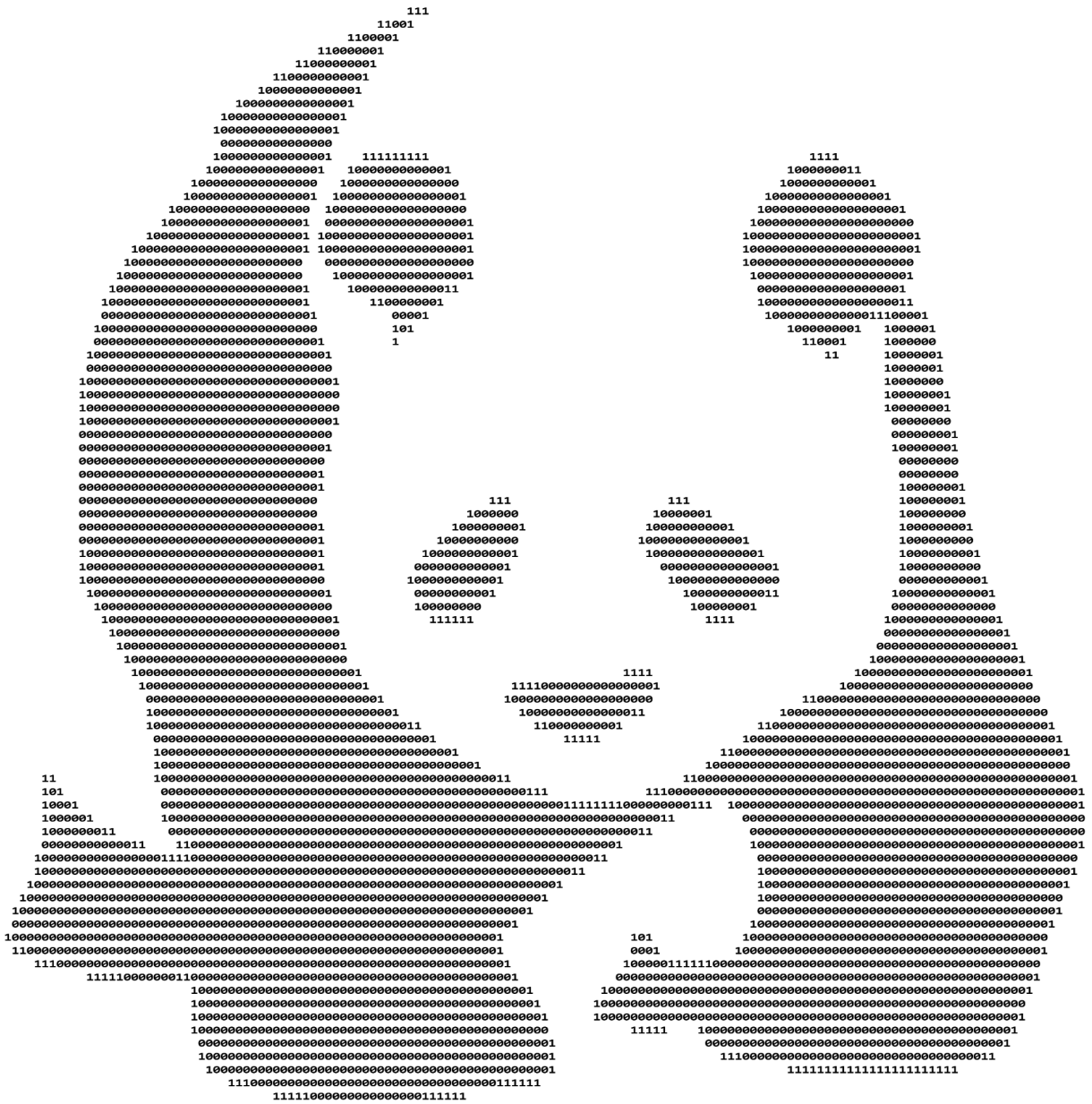
Proyectos de pandas



Objetivo del Documento

El propósito del presente documento es examinar el uso del módulo *pandas* del lenguaje de programación Python, a través de la implementación de proyectos prácticos. Esta metodología tiene como objetivo facilitar una comprensión profunda tanto de la documentación técnica como del uso adecuado de dicha herramienta, promoviendo simultáneamente el desarrollo de aplicaciones con valor funcional.

Los proyectos propuestos estarán principalmente enfocados en temáticas vinculadas a la economía y al trading algorítmico, con la finalidad de aportar soluciones prácticas y relevantes dentro de estos campos de estudio.

Proyecto I: Análisis de precios históricos de las acciones en el mercado.

Introducción

El presente proyecto se centra en el análisis de precios históricos de acciones cotizadas en el mercado financiero, utilizando para ello herramientas del ecosistema Python. En particular, se hará uso de la biblioteca yfinance, que permite acceder a datos bursátiles históricos, y de pandas, una librería fundamental para el procesamiento y análisis de datos estructurados.

El enfoque del proyecto estará orientado tanto a la aplicación técnica de estas herramientas como a la comprensión de los fundamentos del trading algorítmico, campo en el cual dichas bibliotecas cumplen un rol esencial. Se pondrá especial énfasis en la utilización de pandas para manipular y analizar la información obtenida, con el fin de obtener resultados precisos y útiles que deriven en la creación de un programa capaz de proporcionar información relevante sobre el comportamiento de los activos analizados.

Asimismo, se buscará integrar conocimientos teóricos propios de la ciencia de datos, con el objetivo de desarrollar salidas analíticas que respondan de manera eficaz a las necesidades planteadas por el entorno del trading algorítmico.

Parte I: Obtención de datos

La biblioteca pandas en Python constituye una herramienta fundamental para el manejo eficiente de grandes volúmenes de datos estructurados. No obstante, surge una pregunta esencial: ¿qué tipo de datos analizaremos y de dónde los obtendremos?

Con el fin de mantener una coherencia temática con el enfoque del trading algorítmico, se utilizará la biblioteca yfinance, la cual permite acceder y descargar información histórica de precios de activos financieros cotizados en el mercado de valores. Además de datos de precios (como apertura, cierre, máximos, mínimos y volumen), yfinance también proporciona información financiera relevante sobre las empresas, incluyendo balances generales, estados de resultados y flujos de efectivo (cash flow).

Si bien estos datos tienen un valor intrínseco, su análisis directo puede resultar complejo sin las herramientas adecuadas. En este contexto, pandas facilita la tarea al ofrecer estructuras de datos potentes y funcionalidades específicas para la manipulación, transformación y análisis eficiente de la información, convirtiéndola en un recurso accesible y útil para la toma de decisiones en entornos financieros.

1.1 Módulos a importar

Importamos tres librerías fundamentales para el análisis de datos financieros. pandas (abreviado como pd) permite manipular datos en forma de tablas, lo cual facilita enormemente el análisis y la transformación de la información. yfinance (abreviado como yf) se utiliza para descargar datos bursátiles históricos desde Yahoo Finance de manera sencilla y automatizada. Por último, datetime (abreviado como dt) sirve para gestionar fechas y horas dentro de Python, permitiendo calcular períodos de tiempo de forma precisa.

```
import pandas as pd
import yfinance as yf
import datetime as dt
```

1.2 Descarga de información

Posteriormente, se define una lista llamada tickers que contiene los símbolos de los activos financieros de interés: "YPFD.BA" (acción de YPF en la bolsa argentina), "AAPL" (Apple en la bolsa estadounidense) y "ALUA.BA" (Aluar en la bolsa argentina). A su vez, se establecen dos variables de fecha: start representa la fecha actual menos siete días, es decir, calcula una x cantidad de tiempo atrás desde hoy; mientras que end almacena la fecha y hora actuales. Estas variables serán utilizadas para establecer el rango temporal de descarga de los datos.

```
tickers = ["YPFD.BA","AAPL","ALUA.BA"]

start = dt.datetime.today() - dt.timedelta(7)
end = dt.datetime.today()
```

A continuación, se implementa un bucle for que recorre cada uno de los tickers de la lista. Dentro de este bucle, se utiliza la función **yf.download()** para descargar la información histórica de precios para cada activo. Esta descarga se realiza con un intervalo de 15 minutos **(interval="15m")** y cubriendo el rango de fechas comprendido entre hace siete días y hoy. La opción **auto_adjust=False** indica que los precios no deben ser ajustados automáticamente por eventos como dividendos o splits de acciones, preservando los valores originales.

Una vez descargados los datos, estos se convierten en un **DataFrame** de pandas llamado data. Se renombran las columnas del DataFrame para que tengan nombres específicos: "Adj Close", "Close", "High", "Low", "Open" y "Vol". Luego, se agregan dos nuevas columnas: una llamada Ticker, que guarda el nombre del activo correspondiente para identificar a qué empresa pertenece cada fila de datos, y otra llamada date, que toma el índice del DataFrame (que representa la fecha y hora de cada registro).

```
tickers = ["YPFD.BA","AAPL","ALUA.BA"]

start = dt.datetime.today() - dt.timedelta(7)
end = dt.datetime.today()

for ticker in tickers:
    yfinance_information = yf.download(ticker, start=start, end=end, interval="15m", auto_adjust=False)

  data = pd.DataFrame(yfinance_information)

  data.columns = ["Adj Close",'Close', 'High', 'Low', 'Open', 'Vol']
  data['Ticker'] = ticker
  data["date"] = data.index

  data.info()
```

Finalmente, el código ejecuta data.info(), una función que muestra un resumen general del DataFrame. Esta salida permite visualizar cuántas filas y columnas tiene el DataFrame, qué

tipo de datos contiene cada columna, y cuántos valores nulos existen, facilitando así una rápida verificación de que la información se haya descargado y organizado correctamente.

1.2.1 Datos en pandas

Pandas maneja principalmente tipos de datos estructurados en dos objetos principales: Series y DataFrames. Una Series es una estructura unidimensional que puede almacenar cualquier tipo de dato (números, cadenas de texto, valores booleanos, fechas, etc.), acompañada de un índice que identifica cada elemento. Se puede pensar en una Series como una columna de Excel: un conjunto de datos alineados uno debajo del otro, cada uno con su propio nombre o posición.

Por otro lado, un DataFrame es una estructura bidimensional, parecida a una hoja de cálculo o una tabla de base de datos, donde cada columna es en realidad una Series. Esto significa que cada columna de un DataFrame puede contener un tipo de dato diferente. Por ejemplo, una columna podría contener números enteros, otra cadenas de texto, otra fechas y otra valores booleanos (True/False), todo en el mismo DataFrame. Esto da a pandas una gran flexibilidad para trabajar con diferentes tipos de información al mismo tiempo.

En cuanto a los tipos de datos más comunes que pandas maneja dentro de las Series y DataFrames, podemos mencionar varios:

- Enteros (int): Son números enteros como 1, 5, 100 o -23. Dentro de pandas, suelen aparecer como int64, indicando que cada número ocupa 64 bits de memoria.
- Números decimales o flotantes (float): Representan números con parte decimal, como 3.14 o -0.01. Normalmente, pandas los almacena como float64.
- Cadenas de texto (object): Aunque parece extraño, pandas guarda las columnas de texto bajo el tipo object, porque puede contener cualquier tipo de objeto de Python, no solo texto. Desde 2020 en adelante, pandas también tiene un tipo más especializado llamado string, pero object sigue siendo muy usado para datos de texto.
- Valores booleanos (bool): Son valores True o False, y pandas los maneja como tipo bool, ocupando muy poca memoria.
- Fechas y tiempos (datetime64): Cuando trabajamos con fechas o tiempos, pandas utiliza el tipo datetime64[ns], donde ns significa nanosegundos, indicando la precisión de las marcas de tiempo.
- Datos categóricos (category): Este tipo especial se usa para columnas que tienen un número limitado de valores posibles (por ejemplo, colores, países, estados civiles). Guardar datos como category es más eficiente en memoria y puede acelerar algunas operaciones.

Finalmente, hay que mencionar que pandas también es capaz de trabajar con valores nulos, que representan datos faltantes. Estos pueden aparecer como NaN (Not a Number) en datos numéricos o como None en datos de texto o mezclados. Manejar correctamente los nulos es esencial para hacer análisis de datos confiables.

Parte II: Manipulación de datos con pandas

En esta sección se abordará la manipulación de datos utilizando la biblioteca pandas. Aunque los datos de mercado presentan una estructura funcional —ya que se obtienen de manera simultánea y con registro histórico—, resulta fundamental comprender cómo su procesamiento impacta directamente en la calidad y profundidad del análisis posterior. La correcta manipulación de los datos no solo permite mejorar la organización y presentación de la información, sino que también puede ser determinante en la obtención de resultados más precisos y representativos en los análisis realizados.

1.1 Información de yfinance

Cuando se descargan datos directamente a través de la biblioteca yfinance, la información se estructura en un DataFrame donde cada columna representa una variable de precios relevantes del activo financiero seleccionado. En el ejemplo proporcionado, las columnas principales son: Adj Close (precio de cierre ajustado), Close (precio de cierre), High (precio máximo alcanzado en el intervalo), Low (precio mínimo alcanzado), Open (precio de apertura) y Volume (volumen de operaciones).

Price	Adj Close	Close	High	Low	Open	Volume
Ticker	YPFD.BA	YPFD.BA	YPFD.BA	YPFD.BA	YPFD.BA	YPFD.BA
Datetime						
2025-04-21 18:30:00+00:00	36500.0	36500.0	36500.0	36275.0	36275.0	17534
2025-04-21 18:45:00+00:00	36350.0	36350.0	36500.0	36350.0	36500.0	8801
2025-04-21 19:00:00+00:00	36350.0	36350.0	36400.0	36325.0	36375.0	5412
...	...	...	...	...	...	...

Es importante señalar que los datos aparecen organizados bajo un índice temporal (Datetime), donde cada fila corresponde a un registro en un momento específico, en este caso, cada 15 minutos. Además, el nombre del activo (Ticker) se repite en cada columna como una segunda capa de índice (lo que técnicamente se denomina MultiIndex en pandas). Esto significa que la información no solo está clasificada por la variable de precio, sino también asociada explícitamente al símbolo bursátil correspondiente.

Sin embargo, este formato crudo, aunque contiene toda la información necesaria, no resulta inmediatamente práctico para ciertos análisis o manipulaciones posteriores. Es por ello que resulta habitual reorganizar, renombrar o simplificar esta estructura antes de proceder con tareas de procesamiento o visualización de datos financieros.

Con el objetivo de adaptar la estructura de los datos descargados para facilitar su manipulación y análisis, se procede a realizar una serie de transformaciones iniciales sobre el

DataFrame obtenido. En primer lugar, se redefinen los nombres de las columnas utilizando la instrucción data.columns = ["Adj Close", "Close", "High", "Low", "Open", "Vol"], asignando denominaciones más directas y uniformes a cada una de las variables. Esta acción elimina la capa de MultiIndex que originalmente asociaba cada columna al nombre del ticker, simplificando así el esquema de datos.

Posteriormente, se agrega una nueva columna denominada Ticker, a la cual se le asigna el valor correspondiente al activo financiero que se está procesando en ese momento, esto se debe a que probablemente en un futuro almacenemos grandes cantidades de información de mercado sobre cada ticker en la misma base, por lo que esto ayuda a identificar qué activo estamos analizando sin errores, y por otro lado, facilita el acceso a todos los datos de un activo de manera sencilla. Esto resulta fundamental para preservar la referencia al origen de cada dato dentro del conjunto, especialmente cuando se trabaja con múltiples activos simultáneamente. Finalmente, se crea una columna adicional llamada date, que extrae el índice temporal del DataFrame y lo convierte en una columna explícita, a su vez se agrega esta información por más de que se encuentre en el índice debido a que resulta beneficiosa a la hora de realizar ciertos análisis. Esta transformación es crucial para facilitar operaciones posteriores, como filtrados, agrupamientos o combinaciones de datos basadas en fechas específicas.

En conjunto, estas modificaciones permiten transformar un formato de datos inicialmente complejo en una estructura más plana, ordenada y lista para su utilización en análisis financieros avanzados.

Datetime	Adj Close	Close	High	Low	Open	Vol	Ticker	date
2025-04-21 18:30:00+00:00	709.0	709.0	710.0	705.0	708.0	0	ALUA.BA	2025-04-21 18:30:00+00:00
2025-04-21 18:45:00+00:00	699.0	699.0	710.0	698.0	708.0	54225	ALUA.BA	2025-04-21 18:45:00+00:00
2025-04-21 19:00:00+00:00	688.0	688.0	699.0	685.0	699.0	63137	ALUA.BA	2025-04-21 19:00:00+00:00
2025-04-21 19:15:00+00:00	675.0	675.0	690.0	671.0	688.0	141137	ALUA.BA	2025-04-21 19:15:00+00:00
2025-04-21 19:30:00+00:00	666.0	666.0	681.0	663.0	675.0	160797	ALUA.BA	2025-04-21 19:30:00+00:00
...	...	...	...	...	...	...	...	...

La decisión de agregar una columna date en el DataFrame, además de mantener el índice temporal (Datetime), no es estrictamente necesaria desde el punto de vista técnico, ya que pandas permite acceder y manipular fácilmente datos utilizando directamente el índice. Cuando el índice contiene información temporal, se puede aplicar filtrado, ordenamiento, resampling y otras operaciones de series de tiempo de forma muy eficiente. Por ejemplo, es posible seleccionar datos de un día específico o de un rango de fechas simplemente utilizando métodos como .loc[] o funciones especializadas como .resample().

No obstante, incluir una columna explícita que contenga la fecha puede resultar conveniente en ciertos casos. Particularmente, cuando se requiere:

- Visualizar la fecha como una columna más al imprimir o exportar el DataFrame-
- Realizar operaciones de agrupamiento (por ejemplo, agrupar por día, mes o año) de manera más sencilla utilizando funciones como .groupby('date').
- Combinar o unir varios DataFrames basados en fechas específicas utilizando funciones como .merge() o .concat().
- Exportar los datos a formatos donde se prefiera la fecha como un campo explícito (por ejemplo, al guardar en CSV para trabajar luego en Excel).

1.2 Time Series

Dado que los datos financieros operan naturalmente como series de tiempo, es decir, como conjuntos de observaciones ordenadas cronológicamente, el manejo correcto del índice temporal en pandas adquiere una relevancia central. Una serie de tiempo permite analizar la evolución de una variable —como el precio de una acción o el volumen operado— a lo largo del tiempo. En este contexto, pandas ofrece una amplia gama de herramientas diseñadas específicamente para trabajar con datos temporales, tales como filtrado por fechas, cambios de frecuencia (resample()), cálculo de ventanas móviles (rolling()), y generación de estadísticas periódicas. Al mantener la información temporal como índice, se habilita el uso directo de estas funcionalidades, optimizando tanto el rendimiento como la expresividad del análisis. Sin embargo, como se mencionó anteriormente, disponer de la fecha también como columna puede complementar este enfoque al facilitar tareas de agrupamiento y fusión con otras estructuras de datos que no compartan el mismo índice.

```
daily_data = data.resample('D').agg({
   'Open': 'first',
   'High': 'max',
   'Low': 'min',
   'Close': 'last',
   'Volume': 'sum',
   'Adj Close': 'last'
})
```

Los datos de series temporales constituyen una forma fundamental de datos estructurados en numerosos campos del conocimiento, como las finanzas y la economía. En términos generales, una serie temporal se compone de observaciones o mediciones realizadas en distintos puntos del tiempo. Este tipo de datos puede presentarse con una frecuencia fija, en la cual las observaciones se realizan en intervalos regulares (por ejemplo, cada 15 segundos, cada 5 minutos o una vez por mes); o con una frecuencia irregular, donde las mediciones no siguen un patrón temporal constante.

La manera en que se representa o indexa una serie temporal depende en gran medida del contexto de aplicación. Entre las formas más comunes de representar series temporales se encuentran:

- **Timestamps:** instantes específicos en el tiempo, ampliamente utilizados en datos financieros y sensores digitales.
- **Períodos fijos:** unidades de tiempo regulares como meses, trimestres o años (por ejemplo, "enero de 2023").
- **Intervalos de tiempo:** definidos por un punto de inicio y uno de finalización, utilizados frecuentemente en estudios de duración o ventanas móviles.
- **Tiempo relativo o experimental:** en este caso, cada marca temporal representa un valor relativo a un evento inicial (por ejemplo, los segundos transcurridos desde que se encendió un horno).

1.2.1 Conversión entre representaciones temporales en pandas

En el análisis de series temporales, puede ser necesario transformar la forma en que se representa el tiempo según los requerimientos analíticos. La biblioteca pandas proporciona métodos robustos para convertir entre distintos tipos de índices temporales, tales como timestamps, períodos y intervalos. A continuación se describen los casos más comunes:

De Timestamp a Period: Un timestamp representa un instante específico en el tiempo, mientras que un period representa una unidad de tiempo como un mes, trimestre o año. Convertir un índice de tipo DatetimeIndex a PeriodIndex permite trabajar con agrupaciones temporales más generales.

```
# Supongamos una serie con índice datetime
rng = pd.date_range("2023-01-01", periods=5, freq="D")
ts = pd.Series(range(5), index=rng)

# Convertimos de timestamp a period mensual
ts_period = ts.to_period("M")
print(ts_period)
```

De Timestamp a Interval: Cuando se desea trabajar con bloques de tiempo definidos por un inicio y un fin (intervalos), es posible generar estructuras explícitas usando pd.IntervalIndex. Aunque pandas no tiene una función directa para convertir un índice temporal en intervalos, se puede hacer de forma manual agrupando por rangos:

```
# Crear intervalos de 2 días a partir de fechas
intervals = pd.interval_range(start="2023-01-01", end="2023-01-11", freq="2D")

# Crear un DataFrame con fechas
dates = pd.date_range("2023-01-01", periods=10, freq="D")
df = pd.DataFrame({"value": range(10)}, index=dates)

# Asignar intervalos según la fecha
df["interval"] = pd.cut(df.index, bins=intervals)
print(df.head())
```

Tiempo Relativo o Experimental: En experimentos, simulaciones o mediciones donde no se utiliza una fecha del calendario sino el tiempo transcurrido desde un punto de inicio, se puede usar un índice numérico que represente segundos, minutos o cualquier unidad relevante:

```
# Simulación de tiempo relativo: segundos desde el inicio
import numpy as np

elapsed_time = np.arange(0, 10, 0.5)  # cada 0.5 segundos
sensor_data = pd.Series(np.random.randn(len(elapsed_time)), index=elapsed_time)
sensor_data.index.name = "seconds_since_start"

print(sensor_data.head())
```

Parte III: Cálculo de indicadores

El cálculo de indicadores técnicos puede aplicarse de forma directa sobre la información obtenida a través de la biblioteca yfinance. No obstante, resulta fundamental comprender la lógica detrás de cada indicador, su método de cálculo, su utilidad en el análisis financiero y su interpretación dentro del contexto del comportamiento de las acciones.

Esta sección presenta los principales indicadores utilizados en el análisis técnico, destacando su aplicación práctica en el estudio de series temporales bursátiles y su relevancia en la toma de decisiones dentro del trading algorítmico.

1.1 Medias Móviles

Las medias móviles son indicadores fundamentales en el análisis técnico, utilizados para suavizar las fluctuaciones de precios y detectar tendencias en series temporales. Una de las variantes más empleadas es la Media Móvil Exponencial (EMA, por sus siglas en inglés), que asigna mayor peso a los datos más recientes, ofreciendo una respuesta más ágil ante cambios en el mercado.

A continuación, se presenta una función en Python que permite calcular la EMA a partir de una serie de precios y un período determinado:

```
def ema(series, length):
    return series.ewm(span=length, adjust=False).mean()
```

La función hace uso del método .ewm() de pandas, que genera un promedio ponderado exponencialmente. El parámetro span define la longitud de la media, mientras que adjust=False asegura que los pesos decrezcan de forma exponencial sin ajustes retrospectivos.

Para aplicar esta función sobre una columna de precios de cierre (Close), se puede proceder de la siguiente manera:

```
data["EMA"] = ema(data["Close"], 50)
```

1.2 Relative Strange Index (RSI)

El Índice de Fuerza Relativa (RSI, por sus siglas en inglés) es un indicador de momentum que mide la velocidad y el cambio de los movimientos de precios. Su valor oscila entre 0 y 100, y permite identificar posibles condiciones de sobrecompra o sobreventa en un activo

financiero. Generalmente, se interpreta que un RSI superior a 70 indica una situación de sobrecompra, mientras que un valor inferior a 30 sugiere una posible sobreventa.

A continuación, se presenta una función en Python que calcula el RSI a partir de una serie de precios de cierre (Close) y un período determinado:

```
def rsi(data, n):
    "Función para calcular el RSI"
    df = data.copy()

    # Calcular la diferencia de precios (cambio)
    change = df["Close"].diff()

    # Calcular ganancias y pérdidas
    df["gain"] = np.where(change > 0, change, 0)
    df["loss"] = np.where(change < 0, -change, 0)

    # Calcular las medias exponenciales de ganancias y pérdidas
    avgGain = df["gain"].ewm(span=n, min_periods=n).mean()
    avgLoss = df["loss"].ewm(span=n, min_periods=n).mean()

    # Evitar división por cero en RS
    rs = avgGain / avgLoss.replace(0, np.nan)

    # Calcular RSI
    df["RSI"] = 100 - (100 / (1 + rs))
    return df["RSI"]
```

La función utiliza pandas y numpy para calcular las ganancias y pérdidas promedio suavizadas exponencialmente, evitando errores como la división por cero al reemplazar pérdidas nulas con NaN.

Este indicador es ampliamente utilizado en estrategias de trading algorítmico, ya que permite establecer condiciones específicas para generar señales de compra o venta basadas en el comportamiento reciente del precio.

Estos representan solo algunos ejemplos de los múltiples indicadores técnicos que pueden implementarse en Python para el análisis de acciones. En términos generales, cualquier indicador utilizado en el análisis técnico puede ser desarrollado mediante herramientas del ecosistema Python, permitiendo no sólo su visualización, sino también su integración en sistemas automatizados de generación de señales de compra y venta, fundamentales en entornos de *trading* algorítmico.

Proyecto II: Análisis de balances.

Introducción

El presente proyecto tiene como objetivo la recopilación y análisis de los balances financieros de empresas seleccionadas, con el fin de construir una base de datos histórica que permita estudiar su evolución económica a lo largo del tiempo. A partir de la información obtenida mediante la biblioteca yfinance, se pretende evaluar el desempeño financiero de las compañías, determinando si presentan una situación de solidez y crecimiento sostenido, o por el contrario, signos de deterioro que puedan impactar negativamente en su valoración de mercado.

Este análisis se llevará a cabo mediante herramientas como pandas para la manipulación de datos, yfinance para la descarga automatizada de balances, y SQL para el almacenamiento estructurado y eficiente de la información recopilada.

El propósito final es disponer de datos cuantificables y trazables que puedan ser utilizados tanto en estudios económicos como en el desarrollo de futuros proyectos vinculados al trading algorítmico, en donde la evaluación financiera de las empresas constituya un pilar para la toma de decisiones automatizada.

Parte I: Descarga y manejo de datos.

Con el fin de brindar un contexto adecuado y facilitar el desarrollo posterior del análisis, en esta sección se describen de forma resumida los objetivos operativos fundamentales que debe cumplir el presente proyecto.

En primera instancia, el sistema debe verificar qué datos se encuentran disponibles para cada uno de los tickers seleccionados. A continuación, debe proceder a la descarga de la información correspondiente y evaluar si los datos presentan actualizaciones respecto a la versión previamente almacenada. Finalmente, en caso de detectarse cambios, los nuevos registros deben integrarse a la base de datos existente, reemplazando su versión anterior por la actualizada.

El objetivo principal de este proceso es construir una base de datos histórica, cuya evolución a lo largo del tiempo permita realizar análisis dinámicos y evaluar el desempeño financiero de las empresas bajo estudio.

1.1 Tratamiento de datos

En esta etapa se definen los procedimientos asociados al tratamiento de la información financiera obtenida. Para ello, se implementan rutinas automatizadas de verificación, actualización y consolidación de datos, asegurando la integridad del conjunto de información histórica.

Entre las tareas clave que conforman este bloque se encuentran:

a. Identificación del estado de los datos existentes: Se verifica la existencia de información previa para cada ticker y su nivel de actualización.
b. Descarga de datos financieros: A través de yfinance, se obtienen los balances actualizados de las empresas seleccionadas.
c. Comparación y decisión de actualización: Se evalúa si los datos nuevos difieren de los existentes. En caso afirmativo, se integran al histórico y se reemplaza la base almacenada.
d. Construcción incremental del histórico: Los datos actualizados se incorporan de forma acumulativa para mantener un registro temporal coherente y ordenado.

Este flujo de trabajo permite mantener una base de datos robusta y actualizada, esencial para estudios longitudinales orientados al análisis financiero y la toma de decisiones en contextos de inversión.

Lo primero que debe definirse es la ruta en la que se almacenarán los datos descargados. Para ello, se utiliza una conexión SQL mediante sqlite3.connect(ruta), donde la variable ruta representa el directorio en el cual se guardará la base de datos.

A continuación, se implementa un bucle for anidado, que recorre dos listas previamente definidas y se integra dentro de la función principal:

```
def get_balancesheets(types,tickers):
    ruta_balances = sqlite3.connect("xx/xx/xx")

    for type in types:
        for ticker in tickers:

types = ["yearly","quarterly"]
tickers = ["YPFD.BA","BYMA.BA","ALUA.BA","PAMP.BA","EDN.BA"]

get_balancesheets(types,tickers)
```

La primera lista (types) corresponde al tipo de balance que se desea obtener: puede ser anual (yearly) o trimestral (quarterly). Es conveniente almacenar ambos tipos para permitir un análisis más detallado y comparativo. La segunda lista (tickers) contiene los símbolos bursátiles de las empresas cuyas hojas de balance se desea descargar.

Una vez dentro del bucle, es necesario generar un nombre de tabla único para cada combinación de tipo de balance y empresa. Esto se logra eliminando el sufijo .BA del ticker y concatenando los valores de forma estructurada:

```
t = ticker.removesuffix(".BA")
tabla_nombre = f"balance_{type}_{t}"
```

Este identificador permite almacenar cada balance en una tabla distinta dentro de la base de datos, facilitando el acceso y análisis posterior de la información.

En segundo lugar, es necesario verificar si ya existen balances anuales o trimestrales previamente almacenados para cada ticker. Este paso permite evitar la descarga redundante de información. En caso de no encontrarse un balance previo, se procederá a descargar la información por primera vez. En caso contrario, los datos existentes serán cargados en una variable para su posterior comparación y análisis.

Este procedimiento puede implementarse mediante un bloque try-except que intente leer la tabla correspondiente desde la base de datos. Si la tabla existe, se carga en un DataFrame; si no, se inicializa un DataFrame vacío.

```
try:
        balance_guardado = pd.read_sql_query(f"SELECT * FROM {tabla_nombre}", ruta_balances)
        print(f"\nBalance previamente guardado para {ticker}:\n", balance_guardado)
except Exception as e:
        balance_guardado = pd.DataFrame()
        print(f"\nNo se encontró balance previo para {ticker}: {e}")
```

2.1 Descarga de balances

Una vez verificados los datos iniciales, se procede a la obtención de la información financiera del activo correspondiente a través de la API proporcionada por yfinance. En particular, se utiliza el método get_balance_sheet() especificando la frecuencia deseada (anual o trimestral) mediante el parámetro freq. Esta metodología permite mantener una base de datos actualizada con la información más reciente disponible, evitando a su vez la duplicación de registros previamente incorporados.

```
activo = yf.Ticker(ticker)
balance = activo.get_balance_sheet(freq=type)

if balance is not None and not balance.empty:
        balance = balance.T  # Transponer: fechas como filas
        balance.reset_index(inplace=True)
        balance.rename(columns={"index": "Date"}, inplace=True)
        balance["Ticker"] = ticker
```

Se verifica que el balance descargado no sea nulo ni se encuentre vacío. En caso de cumplir estas condiciones, se aplica una transposición al DataFrame (.T). Esto se debe a que, por defecto, las fechas se presentan como columnas y los distintos conceptos del balance como índices. Al transponer la tabla, se invierte esta estructura, de modo que las fechas pasen a ser filas y los conceptos contables columnas, lo cual facilita el acceso, procesamiento y análisis de los datos. Finalmente, se renombra el índice como "Date" y se incorpora una nueva columna denominada "Ticker" que identifica el activo correspondiente.

3.1 Actualización de datos

Como mencionamos anteriormente, a medida que se obtengan nuevos datos estos se irán agregando a las bases con el fin de tener los valores históricos para saber la evolución de la empresa.

```
# Une balances si hay datos nuevos
        if not balance_guardado.empty:
            balance["Date"] = pd.to_datetime(balance["Date"])
            balance_guardado["Date"] = pd.to_datetime(balance_guardado["Date"])

            combinado = pd.concat([balance_guardado, balance], ignore_index=True)
            combinado.drop_duplicates(subset=["Date"], keep="last", inplace=True)
        else:
            combinado = balance

        combinado.to_sql(tabla_nombre, ruta_balances, if_exists="replace", index=False)
```

En esta etapa, se comprueba si ya existen registros previos almacenados en la base de datos (balance_guardado). En caso afirmativo, se convierten las columnas de fecha al formato datetime para asegurar una correcta comparación y manipulación. Luego, se concatenan los nuevos datos descargados con los ya existentes mediante pd.concat(), y se eliminan posibles duplicados utilizando la función drop_duplicates(), conservando el registro más reciente para cada fecha.

Si no existen datos anteriores, se asigna directamente el nuevo balance como conjunto de datos consolidado. Finalmente, el DataFrame resultante se guarda en la base de datos SQL correspondiente utilizando el método to_sql(), reemplazando la tabla existente con la versión actualizada.

Parte II: Análisis de los balances

En esta sección se presentará, de manera simplificada, la aplicación de un análisis de liquidez sobre los balances financieros obtenidos previamente. Se mostrará cómo incorporar funciones específicas al flujo de procesamiento ya implementado, con el objetivo de enriquecer el análisis y facilitar la interpretación de los datos contables.

Inicialmente, el enfoque estará puesto en una única función destinada a calcular indicadores clásicos de liquidez, como la liquidez corriente y la liquidez ácida, aplicados directamente sobre los balances descargados. Esta función será integrada dentro del proceso de actualización de balances, permitiendo que cada registro quede automáticamente acompañado por estos indicadores.

A medida que se identifiquen nuevas necesidades analíticas, se irán incorporando funciones adicionales o mejorando las existentes, con el fin de profundizar el análisis financiero y aportar valor agregado a la herramienta desarrollada.

1.1 Indicadores de Liquidez

El análisis de liquidez permite evaluar la capacidad de una empresa para hacer frente a sus obligaciones de corto plazo. A continuación, se describen los dos indicadores seleccionados:

Liquidez corriente: mide la proporción entre los activos corrientes y los pasivos corrientes. Refleja la capacidad de la empresa para cubrir sus deudas a corto plazo utilizando sus activos líquidos. Se calcula mediante la siguiente fórmula:

$$\text{Liquidez corriente} = \text{Activos Corrientes Totales} / \text{Pasivos Corrientes Totales}$$

Liquidez ácida: es una medida más conservadora, ya que excluye las existencias (inventario) de los activos corrientes, considerando que estos pueden no ser fácilmente realizables en el corto plazo. Su fórmula es:

$$\text{Liquidez ácida} = \frac{\text{Activos Corrientes Totales - Inventario}}{\text{Pasivos Corrientes Totales}}$$

Estos indicadores se calcularán automáticamente dentro del proceso de actualización de balances, y se agregarán como nuevas columnas a la base de datos. A medida que se identifiquen nuevas necesidades analíticas, se irán incorporando funciones adicionales o mejorando las existentes, con el fin de profundizar el análisis financiero y aportar valor agregado a la herramienta desarrollada.

2.1 Explicación de la función liquidity(data)

La función liquidity(data) tiene como propósito calcular automáticamente dos indicadores financieros esenciales: la liquidez corriente y la liquidez ácida. Estos indicadores permiten evaluar la capacidad de una empresa para afrontar sus obligaciones de corto plazo, utilizando como insumo la información contenida en un DataFrame que representa un balance contable previamente descargado y procesado.

En primer lugar, la función realiza una copia del DataFrame original recibido como parámetro. Esta práctica es recomendable para evitar modificaciones directas sobre el conjunto de datos inicial, preservando así la integridad de los datos fuente durante el proceso de análisis. A continuación, se establece una lista con las columnas mínimas requeridas para el cálculo de los indicadores: "TotalAssets", "Inventory" y "TotalDebt". Si alguna de estas columnas no se encuentra presente en el DataFrame, la función emite un mensaje de advertencia indicando las ausencias y retorna el DataFrame sin cambios, previniendo errores posteriores.

Superada esta verificación, se procede al cálculo de los indicadores. La liquidez corriente se obtiene dividiendo los activos totales por el total de deudas (TotalAssets / TotalDebt). Aunque la definición tradicional de este indicador utiliza únicamente activos y pasivos corrientes, en esta implementación inicial se utiliza una versión simplificada basada en los totales, asumiendo que representan suficientemente la situación de corto plazo.

Por su parte, la liquidez ácida ofrece una perspectiva más conservadora respecto de la capacidad de pago de la empresa, ya que excluye del numerador los inventarios. Esta exclusión se justifica en el hecho de que los inventarios pueden no ser fácilmente realizables en el corto plazo, y por ende no representar activos líquidos disponibles para afrontar obligaciones inmediatas. La fórmula empleada para este cálculo es la siguiente:

(TotalAssets - Inventory) / TotalDebt.

Cabe realizar una aclaración relevante en el contexto del presente proyecto: dado que se trabaja con acciones de empresas argentinas, es frecuente que la información provista por la API de yfinance no incluya el dato correspondiente a “Inventory”. Esta ausencia puede deberse tanto a limitaciones en la disponibilidad de datos como a la naturaleza del sector en el que opera la empresa (por ejemplo, entidades financieras o de servicios que no manejan inventarios tradicionales). Por este motivo, y tal como se ha señalado anteriormente, este tipo de situaciones serán abordadas progresivamente a medida que se avance con el desarrollo del proyecto, incorporando validaciones o ajustes específicos para lograr un análisis más preciso y representativo.

Una vez realizados ambos cálculos, el DataFrame es devuelto con dos nuevas columnas: "Liquidity" y "Acid Liquidity", lo que permite integrar estos indicadores de forma automática dentro del proceso general de análisis financiero.

La función que permite calcular los indicadores de liquidez queda determinada de la siguiente manera:

```
def liquidity(data):
   df = data.copy()

   required_cols = ["TotalAssets", "Inventory", "TotalDebt"]
   if not all(col in df.columns for col in required_cols):
         print(f"Faltan columnas requeridas: {', '.join([col for col in required_cols if col not in df.columns])}")
       return df

   # Calcular liquidez corriente
   df["Liquidity"] = df["TotalAssets"] / df["TotalDebt"]

   # Calcular liquidez ácida
   df["Acid Liquidity"] = (df["TotalAssets"] - df["Inventory"]) / df["TotalDebt"]

   return df
```

Como se observa, esta función toma como parámetro un conjunto de datos contables y verifica que contenga las columnas necesarias para efectuar los cálculos: "TotalAssets", "Inventory" y "TotalDebt". En caso de que falte alguna de ellas, se informa al usuario y se devuelve el DataFrame sin alteraciones. Si las columnas están presentes, se calculan e incorporan dos nuevos indicadores: la liquidez corriente (Liquidity) y la liquidez ácida (Acid Liquidity), retornando finalmente el DataFrame enriquecido con esta información.

3.1 Integración al flujo de trabajo principal

Su aplicación dentro del proceso general de descarga y actualización de balances implica una modificación puntual dentro de la función get_balancesheets(). La integración se da en la parte del código donde se combinan los balances nuevos con los ya existentes. El bloque afectado queda de la siguiente manera:

```
if not balance_guardado.empty:
                balance["Date"] = pd.to_datetime(balance["Date"])
                balance_guardado["Date"] = pd.to_datetime(balance_guardado["Date"])

                combinado = pd.concat([balance_guardado, balance], ignore_index=True)
                combinado.drop_duplicates(subset=["Date"], keep="last", inplace=True)
                combinado = liquidity(combinado, ticker)
```

```
else:
        combinado = balance
        combinado = liquidity(combinado, ticker)
```

De este modo, se asegura que los datos combinados —ya sea por actualización o carga inicial— pasen por la función liquidity, incorporando así automáticamente las métricas de análisis financiero correspondientes. Esta integración no solo automatiza el cálculo, sino que también permite estandarizar el análisis de liquidez en todos los balances almacenados, facilitando su interpretación y comparación a lo largo del tiempo.

Proyecto III: Análisis de datos económicos del gobierno.

El presente proyecto se fundamenta en los datos recopilados durante el desarrollo del Proyecto II de web scraping, con el propósito de examinar la manipulación y el procesamiento de grandes volúmenes de metadatos mediante el empleo de las bibliotecas pandas y SQL. Estas herramientas permiten abordar de forma eficiente la estructuración, consulta y transformación de la información extraída.

El objetivo final consiste en generar un análisis que posibilite la aplicación automatizada de los valores obtenidos, promoviendo su integración en procesos analíticos o de toma de decisiones posteriores. De esta manera, se busca establecer una base sólida para el desarrollo de sistemas de procesamiento de datos dinámicos y escalables.

Parte I: Acceso a los datos descargados.

Los datos utilizados en este trabajo han sido obtenidos mediante técnicas de web scraping desarrolladas en el Proyecto II y se encuentran almacenados en un archivo en formato SQL. En consecuencia, la utilización de esta tecnología resulta esencial para la extracción y gestión estructurada de los metadatos recolectados.

Una vez realizada la conexión y extracción de los datos, se recurre al uso de la biblioteca pandas, debido a su eficiencia y flexibilidad en la manipulación de grandes volúmenes de información tabular. Este enfoque permite transformar, limpiar y preparar los datos para su posterior análisis.

Finalmente, con el objetivo de facilitar la interpretación de los resultados, se implementarán herramientas de visualización como Matplotlib o Plotly, que permiten representar gráficamente los patrones y relaciones presentes en los datos. Estas representaciones visuales aportan una perspectiva más profunda y enriquecedora del conjunto de datos analizado, favoreciendo la toma de decisiones basada en evidencia empírica.

1.1 Acceder a los datos descargados.

Para llevar a cabo la manipulación y análisis de los datos, se procede a importar las bibliotecas necesarias. En primer lugar, se utiliza la biblioteca pandas para el tratamiento de estructuras tabulares, y sqlite3 para establecer la conexión con la base de datos en formato SQL. Adicionalmente, se importa el módulo goverment_info, desarrollado en el Proyecto II, el cual contiene la funcionalidad para automatizar la descarga del archivo requerido.

```
import pandas as pd
import sqlite3
import goverment_info as gi
```

La función principal contenida en el módulo goverment_info permite ejecutar el proceso de web scraping previamente implementado, asegurando así la disponibilidad del archivo actualizado que será posteriormente cargado y procesado en el entorno de análisis.

1.1.1 Inicialización del proceso de análisis.

Para iniciar el proceso de análisis, se ejecuta la función download_goverment_info() proveniente del módulo goverment_info, desarrollado en el Proyecto II. Esta función automatiza el procedimiento de descarga de la base de datos en formato SQLite a través de técnicas de web scraping previamente implementadas.

```
gi.download_goverment_info()
```

Al ejecutarse, dicha función crea de forma automática una carpeta denominada datos_sqlite, dentro de la cual se almacena el archivo .sql que contiene la información estructurada requerida para el análisis posterior. Este enfoque garantiza la organización de los datos y su disponibilidad en un formato estandarizado para su manipulación mediante bibliotecas como pandas y sqlite3.

1.2 Carga de datos

Una vez completado el proceso de descarga y descompresión, el archivo de base de datos en formato SQLite se encuentra disponible dentro del directorio datos_sqlite. Para acceder a su contenido, se define la función cargar_datos(), la cual establece la conexión con la base de datos, ejecuta una consulta SQL sobre una tabla específica y devuelve los resultados en forma de un DataFrame de pandas.

Esta función encapsula el proceso de carga, permitiendo una mayor reutilización del código y facilitando la adaptación a distintos conjuntos de datos o estructuras de bases.

```
def cargar_datos(nombre_archivo, nombre_tabla):
    ruta = os.path.join("datos_sqlite", nombre_archivo)
    conexion = sqlite3.connect(ruta)
    df = pd.read_sql_query(f"SELECT * FROM {nombre_tabla}", conexion)
    conexion.close()
    return df
```

La función construye dinámicamente la ruta al archivo utilizando os.path.join(), lo cual asegura la compatibilidad con distintos sistemas operativos. A continuación, se abre una conexión, se consulta el contenido completo de la tabla indicada (SELECT *) y se retorna un DataFrame que puede ser manipulado fácilmente en las etapas posteriores de análisis.

1.2.1 ¿Qué son los metadatos?

En el ámbito de la gestión de la información, los metadatos se definen como datos estructurados que describen, explican, localizan o facilitan la recuperación, uso y gestión de otros datos. En otras palabras, los metadatos son datos sobre los datos: no contienen el contenido principal en sí mismo, sino que proporcionan un conjunto de atributos que permiten caracterizarlo, contextualizar o clasificarlo.

El uso de metadatos es fundamental en sistemas de información modernos, ya que permiten organizar grandes volúmenes de información de manera eficiente, mejorando los procesos de búsqueda, análisis, interoperabilidad y reutilización de los datos. En bases de datos relacionales, bibliotecas digitales, sistemas de archivos y portales gubernamentales, los metadatos constituyen la columna vertebral de la estructura informativa.

Los metadatos pueden clasificarse en diversas categorías, dependiendo de su propósito:

- Metadatos descriptivos: proporcionan información sobre el contenido del recurso, como título, autor, fecha de creación, palabras clave o resumen.
- Metadatos estructurales: describen cómo se organizan los componentes internos de un recurso, por ejemplo, la secuencia de páginas en un documento o la relación entre diferentes archivos.
- Metadatos administrativos: ofrecen detalles técnicos como el formato, tamaño, derechos de acceso, condiciones de uso o procedencia del archivo.

En el contexto de este trabajo, los metadatos provienen de publicaciones oficiales del gobierno argentino y fueron obtenidos a través de técnicas de web scraping. Estos metadatos no contienen el texto completo de las resoluciones, disposiciones o normativas, sino atributos que permiten identificarlas y clasificarlas, tales como:

Número de expediente.

- Fecha de emisión.
- Organismo emisor.
- Tipo de norma jurídica.
- Estado del documento.
- Título descriptivo.

El análisis de estos metadatos permite generar una visión estructurada de la actividad administrativa, identificar patrones de comportamiento institucional, y establecer bases para la automatización en procesos de clasificación documental.

1.3 Estructura de los metadatos.

Con el objetivo de comprender la organización interna de los metadatos descargados, en esta sección se construye un DataFrame a partir de la tabla correspondiente de la base de datos SQLite. Esta tabla contiene la estructura que describe los distintos conjuntos de datos disponibles, funcionando como una capa de información contextual esencial para cualquier análisis posterior.

Dado que los metadatos constituyen una colección de atributos sobre los datos principales —tales como su procedencia, fecha de actualización, descripción, categoría temática, entre

otros— es necesario examinar la totalidad de las columnas presentes para identificar cuáles serán de utilidad en la etapa de exploración y modelado.

A continuación, se muestra el código utilizado para la carga de dicha tabla y la visualización de las columnas contenidas en ella:

```
data = cargar_datos("series-tiempo.sqlite","metadatos")
data = pd.DataFrame(data)
print(data.columns)
```

Este fragmento de código realiza tres operaciones fundamentales:

- Carga de la tabla "metadatos" desde el archivo series-tiempo.sqlite, ubicado en el directorio datos_sqlite.
- Conversión a un DataFrame de pandas, lo que facilita su posterior manipulación y análisis.
- Visualización de las columnas que conforman la estructura interna del conjunto de metadatos.

La exploración de estas columnas permitirá determinar qué variables serán utilizadas en el análisis, cuáles representan información descriptiva relevante, y qué campos podrían considerarse redundantes o prescindibles en función del objetivo analítico del proyecto.

1.3.1 Descripción de las columnas de metadatos.

Una vez cargada la tabla metadatos en un DataFrame, podemos inspeccionar las columnas que la componen. Cada columna representa un atributo o descriptor que contextualiza los datos principales en el archivo. A continuación, se detallan algunas de las columnas más relevantes que se encuentran en los metadatos:

Columna	Descripción
id_documento	Identificador único de cada documento en la base de datos. Se utiliza para referenciar unívocamente cada registro.
titulo	Título descriptivo del documento o resolución. Facilita la identificación rápida del contenido de la norma o documento.
organismo_emisor	Nombre del organismo o institución que emite el documento, permitiendo clasificar los datos por entidad responsable.
fecha_emision	Fecha en la cual el documento fue emitido. Es fundamental para ordenar y filtrar los documentos según su antigüedad.
tipo_documento	Categoría del documento (por ejemplo, "Ley", "Resolución", "Disposición", etc.). Permite una clasificación temática.
categoria	Categoría o tema principal tratado en el documento (por ejemplo, "Economía", "Salud", "Educación", etc.). Es útil para agrupaciones temáticas.
fecha_actualizacion	Fecha en que el documento fue actualizado por última vez. Este metadato es clave para el seguimiento de cambios y versiones.
estado_documento	Indica el estado actual del documento (por ejemplo, "Aprobado", "En revisión"). Es esencial para determinar la validez y aplicabilidad del documento.
ubicacion_archivo	Ruta o dirección donde se encuentra el archivo físico (si aplica). Ayuda a localizar el archivo original para su consulta.
descripcion	Resumen breve del contenido del documento. Este metadato proporciona un contexto inicial para entender el propósito del documento.
unidad_medida	En ciertos casos, si el documento contiene datos numéricos o financieros, esta columna especifica las unidades de medida correspondientes (por ejemplo, "USD", "kg").

Parte II: Los datos.

Una vez descargados los datos, es fundamental comprender su estructura y composición para asegurar un análisis adecuado. Aunque anteriormente se ha inspeccionado la composición de las columnas, en esta sección se procederá a validar la calidad de los datos antes de su utilización. Esto es importante, ya que algunas series pueden estar desactualizadas o incluso marcadas como discontinuadas.

En particular, cabe destacar que el dataset incluye indicadores booleanos representados como 0 (falso) y 1 (verdadero). Sin embargo, se observa una inconsistencia lógica: todas las series aparecen marcadas como discontinuadas (serie_discontinuada = 1), incluso aquellas que también están etiquetadas como actualizadas (serie_actualizada = 1). Este comportamiento contradice la expectativa de que una serie actualizada no debería estar discontinuada. A pesar de ello, el conjunto de datos sigue siendo útil para los fines de este proyecto, ya que ofrece una oportunidad adecuada para aplicar técnicas de manipulación y consulta con pandas y SQL.

2.1 Filtrado de datos.

Con el fin de simplificar el análisis y enfocarnos únicamente en los elementos relevantes, se procederá a filtrar el dataset original y construir un DataFrame reducido que contenga únicamente las columnas de interés. Dado que la base de datos contiene información complementaria (enriquecida) que no resulta esencial para esta etapa del proyecto, se trabajará inicialmente con un subconjunto mínimo de variables.

El siguiente código realiza dicha selección:

```
df_filtrado = data[[
    'serie_titulo',
    'serie_unidades',
    'serie_indice_inicio',
    'serie_indice_final',
    'serie_valor_ultimo',
    'serie_valor_anterior',
    'serie_var_pct_anterior',
    'serie_actualizada',
    'serie_discontinuada'
]]
```

Este filtrado permitirá trabajar de manera más eficiente, concentrándose en el comportamiento de las series temporales y su vigencia.

Una vez filtrado el conjunto de datos, es posible aplicar distintos tipos de análisis exploratorios que nos permitan comprender mejor la estructura y calidad de la información contenida. Uno de los primeros aspectos a verificar es la actualización de las series y la cantidad de registros disponibles para cada título, lo cual resulta útil para determinar la densidad de información asociada a cada temática.

Además, se observa que aunque algunos títulos puedan repetirse, estos pueden estar asociados a distintas unidades de medida, lo que implica la necesidad de prestar especial atención a las unidades al momento de interpretar los datos. Esta diferenciación es clave, ya que la misma variable puede representarse en términos porcentuales, absolutos o en distintas monedas, dependiendo del contexto.

Para ejemplificar este análisis, se aplica un filtro que selecciona únicamente las series que aparecen tanto actualizadas como discontinuadas, y se procede a contar la cantidad de registros por título:

```
# Mostrar resultado ordenado por título
df_filtrado = df_filtrado[(df_filtrado['serie_actualizada'] == 1) &
(df_filtrado['serie_discontinuada'] == 1)]

# Ver cantidad de registros por título
print(df_filtrado.value_counts('serie_titulo'))
```

2.2 Descarga y gestión de datos.

Uno de los componentes fundamentales del presente proyecto corresponde al módulo de descarga y gestión de datos. Tras la etapa de filtrado, se procede a evaluar la necesidad de conservar un nuevo conjunto de datos o descartar por resultar redundante. Este proceso se automatiza mediante la descarga diaria del archivo correspondiente, permitiendo determinar si se han producido actualizaciones respecto de versiones almacenadas previamente.

Para detectar duplicaciones, se comparan los valores y fechas contenidos en el archivo nuevo con los registros previamente guardados en el sistema. En caso de no haberse registrado cambios significativos, el archivo descargado es eliminado a fin de evitar redundancias innecesarias en el almacenamiento local.

Este procedimiento permite conservar exclusivamente los archivos que aportan información novedosa, facilitando la construcción de un repositorio histórico depurado, útil para el análisis de series temporales. Con el objetivo de garantizar trazabilidad, cada archivo es renombrado incluyendo la fecha de descarga mediante el siguiente fragmento de código:

```
fecha_hoy = datetime.today().strftime('%Y-%m-%d')
nuevo_nombre_sqlite = f"series-tiempo_{fecha_hoy}.sqlite"

 try:
      for archivo in os.listdir(carpeta_destino):
         if archivo.endswith(".sqlite"):
            ruta_antigua = os.path.join(carpeta_destino, archivo)
            ruta_nueva = os.path.join(carpeta_destino, nuevo_nombre_sqlite)
            os.rename(ruta_antigua, ruta_nueva)
            print(f"Archivo renombrado a: {nuevo_nombre_sqlite}")
            pass
 except:
      try:
         os.remove(os.path.join(carpeta_destino, "series-tiempo.sqlite"))
      except:
         pass
```

Posteriormente, se ejecuta una función que contrasta el contenido del nuevo archivo con la versión previa más reciente. Este paso resulta crucial para mantener la integridad y pertinencia del repositorio de datos a lo largo del tiempo:

```
def conservar_nuevo_y_eliminar_viejo(nombre_nuevo_archivo, carpeta="datos_sqlite"):

   data_nueva = cargar_datos(nombre_nuevo_archivo, "metadatos")
   df_nuevo = pd.DataFrame(data_nueva)[[
      'serie_titulo',
      'serie_unidades',
      'serie_indice_inicio',
      'serie_indice_final',
      'serie_valor_ultimo',
      'serie_valor_anterior',
      'serie_var_pct_anterior',
      'serie_actualizada',
      'serie_discontinuada'
   ]]

   # Buscar archivo anterior más reciente (que no sea el nuevo)
   archivos_sqlite = sorted([
      f for f in os.listdir(carpeta) if f.endswith(".sqlite") and f != nombre_nuevo_archivo
   ], reverse=True)
```

```
    if archivos_sqlite:
        archivo_anterior = archivos_sqlite[0]
        os.remove(os.path.join(carpeta, archivo_anterior))
        print(f"Archivo anterior eliminado: {archivo_anterior}")
    else:
        print("No hay archivos anteriores para eliminar.")

    print(f"Archivo nuevo conservado: {nombre_nuevo_archivo}")
    return df_nuevo
```

2.2.1 Teoría y fundamento de la función conservar_nuevo_y_eliminar_viejo.

La función conservar_nuevo_y_eliminar_viejo responde a una lógica de mantenimiento eficiente de datos en contextos donde se recolectan series temporales de forma diaria. Su implementación persigue el objetivo de conservar únicamente información actualizada y relevante, eliminando registros obsoletos que podrían distorsionar el análisis longitudinal de las variables observadas.

Desde un enfoque teórico, esta función opera sobre dos principios claves:

- Minimización de la redundancia informativa: En sistemas donde los datos pueden no variar día a día, es innecesario almacenar versiones repetidas. La eliminación del archivo anterior garantiza que el almacenamiento solo conserve la versión más reciente y, por ende, relevante.
- Gestión incremental del historial de datos: Conservando exclusivamente el último archivo, se asegura una base de datos depurada en tiempo real, evitando acumulaciones innecesarias de versiones que no aportan modificaciones sustanciales.

La estructura de la función contempla:

1. Carga del nuevo archivo: Se transforma en un DataFrame para facilitar su posterior procesamiento.
2. Búsqueda del archivo anterior: Se identifican y ordenan los archivos .sqlite en el directorio especificado, excluyendo el archivo recién descargado.
3. Eliminación del archivo obsoleto: Se elimina el archivo más reciente anterior al nuevo, bajo la suposición de que éste ya ha sido reemplazado.
4. Confirmación de conservación: Se confirma que el nuevo archivo queda almacenado para futuros análisis.

Esta estrategia resulta coherente con los principios de la ingeniería de datos, al promover la eficiencia, la trazabilidad y la consistencia temporal en la construcción de repositorios de datos históricos.

Parte III: Análisis.

En la sección anterior se introdujo una modificación sustancial al sistema de obtención de datos, permitiendo no solo su descarga automatizada, sino también la actualización efectiva del conjunto de información almacenado. Una vez implementado este mecanismo de gestión y preservación de datos únicos para cada serie aún vigente, es necesario establecer una metodología adecuada para la preparación y organización del material con fines analíticos.

El análisis de series temporales económicas requiere que los datos se encuentren debidamente estructurados y segmentados. Aunque el conjunto original ya presenta una organización preliminar, para poder abordar el estudio individualizado de cada serie resulta fundamental descomponer el archivo en subconjuntos específicos. Esta segmentación deberá realizarse en función de una columna que actúa como identificador único o repetitivo para cada serie económica.

3.1 División de datos.

El DataFrame obtenido tras la descarga y filtrado contiene las siguientes columnas:

- 'serie_titulo',
- 'serie_unidades',
- 'serie_indice_inicio',
- 'serie_indice_final',
- 'serie_valor_ultimo',
- 'serie_valor_anterior',
- 'serie_var_pct_anterior',
- 'serie_actualizada',
- 'serie_discontinuada'

Para proceder con el análisis individualizado, se selecciona la columna serie_titulo como criterio de agrupación. Esta variable actúa como identificador nominal de cada serie, permitiendo distinguirlas unas de otras y facilitando la creación de subconjuntos homogéneos. La elección de serie_titulo responde a su naturaleza descriptiva, la cual permanece constante para una misma serie a lo largo del tiempo, convirtiéndose en un criterio robusto para la segmentación de datos.

Esta división permitirá, en etapas posteriores, aplicar métricas estadísticas, transformaciones o visualizaciones a cada serie de forma independiente, manteniendo la trazabilidad y consistencia de los resultados obtenidos.

3.1.1 División por título

Una vez consolidado el conjunto actualizado de datos, el siguiente paso consiste en segmentarlos según su categoría temática. En este proyecto, cada serie económica está identificada por el campo serie_titulo, el cual representa el nombre único de la variable que se desea analizar a lo largo del tiempo.

Para facilitar el tratamiento individual de cada serie, se implementa una función que permite filtrar el DataFrame principal y conservar únicamente los registros asociados a un título específico. A continuación, se presenta la función filtrar_por_titulo, encargada de realizar esta tarea:

```
def filtrar_por_titulo(df, titulo_serie):
    df_filtrado = df[df['serie_titulo'] == titulo_serie].copy()
    return df_filtrado

# Ejemplo de uso
titulo_deseado = "gtos_primarios_despues_figurativos_2017"
df_serie = filtrar_por_titulo(ultima_infromacion, titulo_deseado)
```

Este procedimiento es esencial para aislar cada serie de interés y proceder con su análisis específico, como visualizaciones, estadísticas descriptivas o modelado temporal.

3.2 Estado actual y limitaciones

Durante el desarrollo de este proyecto se ha identificado una discrepancia en los datos proporcionados: múltiples series aparecen simultáneamente marcadas como actualizadas y descontinuadas. Esta inconsistencia en la base de datos impide la continuidad del análisis en esta etapa, ya que compromete la validez de las interpretaciones y resultados que puedan derivarse de dichas series.

En consecuencia, el avance del presente proyecto se encuentra temporalmente suspendido hasta que se verifiquen nuevas actualizaciones en la información disponible o se resuelva el conflicto detectado en los metadatos.

No obstante, este trabajo sienta las bases para una futura herramienta de análisis automatizado de indicadores económicos. El objetivo a mediano plazo es contar con una aplicación funcional que permita interpretar, segmentar y evaluar variables clave del entorno económico, con el fin de facilitar la toma de decisiones informadas en contextos de inversión.

Bibliografías

- Beaulieu, A. (2020). Learning SQL (3rd ed.). O'Reilly Media.
- McKinney, W. (2018). Python for data analysis (2nd ed.). O'Reilly Media.
- The pandas development team. (n.d.). Getting started — pandas documentation. pandas. https://pandas.pydata.org/docs/getting_started/intro_tutorials/index.html
- VanderPlas, J. (2017). Python data science handbook. O'Reilly Media.