Pandas

Introducción

Pandas se destaca como la biblioteca preeminente dentro del ecosistema de Python para la manipulación y análisis de datos. Su diseño es ágil y eficiente, lo que permite a los usuarios realizar operaciones complejas de manera rápida y sencilla. Con su potente capacidad para manejar datos estructurados, Pandas es ideal para trabajar con datos de diversas fuentes, como archivos CSV, bases de datos SQL y hojas de cálculo de Excel. Esto lo convierte en una herramienta esencial en el arsenal de cualquier analista de datos o científico de datos.

Pandas fue concebido y desarrollado por **Wes McKinney**, un analista financiero convertido en desarrollador de software. Wes reconoció la necesidad de una herramienta que pudiera abordar efectivamente los desafíos del análisis de datos en la industria financiera. En particular, observó que las herramientas de análisis de datos disponibles en ese momento eran limitadas y no cumplían con las exigencias de la industria, lo que lo llevó a crear una solución más robusta y versátil. Su visión era crear una biblioteca de Python que pudiera proporcionar las mismas capacidades de manipulación de datos encontradas en el software de hojas de cálculo populares y bases de datos relacionales. Este enfoque pragmático permitió que Pandas se convirtiera rápidamente en un estándar de facto para el análisis de datos en Python.

En 2008, Wes McKinney comenzó a trabajar en Pandas mientras estaba en AQR Capital Management. Durante este tiempo, se dio cuenta de que las herramientas existentes no cumplían con las exigencias de análisis de datos complejos, lo que lo llevó a desarrollar una solución que pudiera facilitar esta tarea. Su pasión y dedicación llevaron al lanzamiento de la primera versión de Pandas en 2009. Desde entonces, Pandas ha evolucionado continuamente, con contribuciones de numerosos desarrolladores y una comunidad activa que ha ampliado su funcionalidad y mejorado su rendimiento.

Características clave de Pandas: Pandas ofrece una serie de características convincentes, incluyendo:

- **DataFrame (núcleo)**: Un objeto DataFrame rápido y eficiente para la manipulación de datos sin problemas, completo con indexación integrada. Permite realizar operaciones como selección, filtrado y agrupamiento de datos de manera intuitiva. El DataFrame puede considerarse como una tabla de datos en la que las filas y columnas tienen etiquetas, lo que facilita la identificación y acceso a la información.
- **Entrada/Salida de Datos**: Lectura y escritura de datos simplificada en varios formatos, como Microsoft Excel, CSV, bases de datos SQL, y más. Esto facilita la integración de Pandas en flujos de trabajo de análisis de datos existentes, permitiendo a los usuarios cargar datos desde múltiples fuentes y exportar resultados de manera eficiente.
- **Manipulación de Datos Robusta**: Métodos integrados y eficientes para una amplia gama de manipulaciones de datos, incluyendo el manejo de datos faltantes, subconjuntos, fusión, y más. Estas funcionalidades son cruciales para limpiar y

preparar los datos para el análisis. Por ejemplo, Pandas permite rellenar o eliminar valores faltantes, lo que es fundamental para asegurar la calidad de los datos antes de realizar cualquier análisis.

- **Manejo de Datos Temporales**: Pandas sobresale en la gestión de datos temporales, convirtiéndose en la opción preferida para trabajar con datos de panel (de ahí su nombre). Ofrece potentes herramientas para la resampling, la conversión de tipos de fechas y la manipulación de series temporales. Esto incluye la capacidad de manejar operaciones como la agregación mensual de datos diarios o la creación de índices temporales que facilitan la visualización de tendencias a lo largo del tiempo.

- **Integración**: Integración suave con otras bibliotecas de análisis de datos y aprendizaje automático como scikit-learn, scipy, seaborn y plotly. Esto permite a los usuarios construir fácilmente pipelines de datos complejos que combinan diferentes técnicas y herramientas. Por ejemplo, se puede utilizar Pandas para preparar y limpiar datos antes de aplicar algoritmos de aprendizaje automático con scikit-learn.

- **Uso Generalizado**: Pandas goza de una adopción generalizada en sectores privados y académicos, convirtiéndose en una herramienta imprescindible para los entusiastas de los datos. Su popularidad ha llevado a una rica comunidad de usuarios que contribuyen con tutoriales, ejemplos y mejoras a la biblioteca. Además, hay una gran cantidad de recursos en línea, desde documentación oficial hasta cursos gratuitos y de pago, que ayudan a los nuevos usuarios a aprender a utilizar Pandas de manera efectiva.

- **Visualización de Datos**: Aunque Pandas no es una biblioteca de visualización por sí misma, se integra fácilmente con bibliotecas como Matplotlib y Seaborn, lo que permite crear gráficos atractivos y comprensibles a partir de DataFrames. Esto es esencial para el análisis exploratorio de datos, donde la visualización puede revelar patrones y relaciones que no son evidentes en los datos brutos.

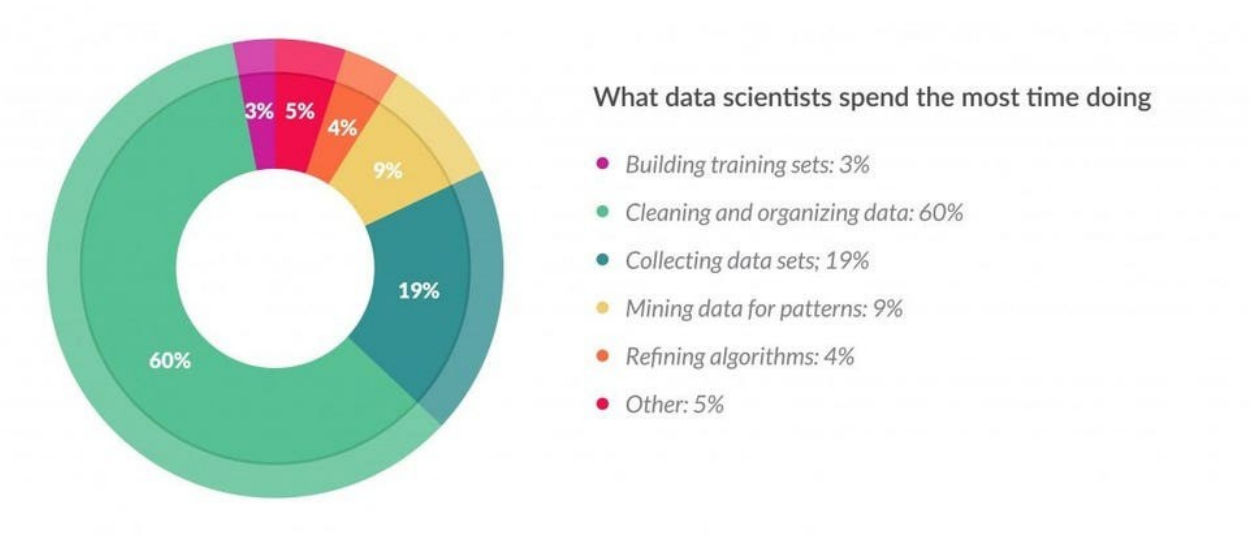
Potenciando el Análisis de Datos: Pandas proporciona estructuras de datos de alto nivel y funciones diseñadas para acelerar tu trabajo con datos estructurados o tabulares. Desde su debut en 2010, Pandas ha jugado un papel pivotal en elevar a Python como un entorno robusto y eficiente para el análisis de datos. Su flexibilidad permite a los usuarios adaptarse a diversas necesidades de análisis, desde tareas simples hasta proyectos de gran escala. Además, Pandas permite la creación de funciones personalizadas para extender su funcionalidad, lo que lo convierte en una herramienta extremadamente versátil.

Los objetos principales de Pandas que encontrarás en esta guía son el DataFrame —una estructura de datos tabular orientada a columnas con etiquetas de filas y columnas intuitivas— y la Serie —un arreglo unidimensional etiquetado. Ambos objetos están diseñados para facilitar la manipulación de datos y el análisis exploratorio, permitiendo a los usuarios descubrir patrones y tendencias de manera efectiva. Las operaciones entre estos objetos son rápidas y eficientes, lo que los hace ideales para el análisis de grandes conjuntos de datos.

Pandas une los principios de alto rendimiento de NumPy con las capacidades de manipulación de datos versátiles de las hojas de cálculo y bases de datos relacionales, como SQL. Introduce

características de indexación sofisticadas que simplifican la remodelación, el corte, la agregación y la selección de subconjuntos de datos. Estas características permiten a los usuarios realizar operaciones complejas con facilidad y rapidez, lo que es esencial en el análisis de datos en tiempo real y en la toma de decisiones basada en datos.

En resumen, Pandas empodera a analistas de datos, científicos y ingenieros para utilizar Python como una herramienta potente para una amplia gama de tareas relacionadas con datos, revolucionando la manera en que se gestiona y analiza los datos estructurados. Su enfoque en la eficiencia y la facilidad de uso lo ha convertido en una de las bibliotecas más queridas dentro de la comunidad de Python. Además, su desarrollo continuo y el apoyo de una comunidad activa garantizan que Pandas seguirá siendo relevante y útil en el futuro del análisis de datos.



Source: Forbes

Instalación

Lo primero que siempre debes hacer es `pip3 install pandas`, `conda install pandas`

```
import pandas as pd
import numpy as np
```

Introducción a las estructuras de datos de Pandas

Para comenzar con **Pandas**, es crucial familiarizarse con sus dos estructuras de datos fundamentales: **Series** y **DataFrame**. Estas estructuras ofrecen una forma eficiente y flexible de manipular grandes volúmenes de datos, y son herramientas poderosas en cualquier flujo de trabajo de análisis de datos.

Series

Una **Serie** es una estructura de datos unidimensional en Pandas, que puede verse como un arreglo unidimensional (similar a una columna de una tabla), con la particularidad de que cada

elemento tiene una etiqueta asociada, conocida como **índice**. Esto la convierte en una combinación flexible de un arreglo de NumPy y un diccionario de Python.

Creación de Series: Las Series pueden crearse a partir de varios tipos de datos, como listas, diccionarios, o incluso a partir de un arreglo de NumPy. Aquí un ejemplo sencillo utilizando una lista:

```
data = [10, 20, 30, 40, 50]
series = pd.Series(data)
print(series)

0    10
1    20
2    30
3    40
4    50
dtype: int64
```

En este caso, se genera una Serie donde Pandas asigna automáticamente un índice numérico (0, 1, 2, ...) que corresponde a la posición de cada valor en la lista original. Si prefieres, puedes definir manualmente un índice:

```
series = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])
print(series)

a    10
b    20
c    30
d    40
e    50
dtype: int64
```

Esto asignará las etiquetas 'a', 'b', 'c', 'd', y 'e' como índices de la Serie.

Tipos de datos compatibles

Las Series en Pandas pueden contener varios tipos de datos:

- **Numéricos**: Enteros o flotantes, como en el ejemplo anterior.
- **Texto**: Pandas puede manejar cadenas de texto en una Serie:

```
series = pd.Series(['manzana', 'naranja', 'plátano', 'pera'])
print(series)

0    manzana
1    naranja
2    plátano
3       pera
dtype: object
```

- **Booleanos**: Puedes crear Series con valores booleanos:

```
series = pd.Series([True, False, True, False])
print(series)
```

```
0     True
1    False
2     True
3    False
dtype: bool
```

- **Fechas y Tiempos**: Pandas es particularmente útil para manejar datos de series temporales. Puedes utilizar el método `pd.date_range()` para generar Series con fechas:

```
dates = pd.date_range('20230101', periods=6)
series = pd.Series([1, 2, 3, 4, 5, 6], index=dates)
print(series)
```

```
2023-01-01    1
2023-01-02    2
2023-01-03    3
2023-01-04    4
2023-01-05    5
2023-01-06    6
Freq: D, dtype: int64
```

Operaciones básicas en Series

Las Series en Pandas permiten realizar una amplia gama de operaciones, muchas de ellas aprovechando el poder de las operaciones vectorizadas. A continuación, se describen algunas operaciones comunes:

Acceso a los elementos de una Serie

Puedes acceder a los valores de una Serie utilizando su índice:

```
series = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd',
'e'])
print(series['b'])  # Accede al valor correspondiente a la etiqueta
'b'
print(series[2])    # Accede al valor en la posición 2
```

```
20
30
```

También es posible utilizar el método `.loc[]` para acceder a los valores basados en el índice:

```
print(series.loc['b'])  # Usa el índice explícito
```

```
20
```

O usar `.iloc[]` para acceder a los valores por la posición numérica:

```
print(series.iloc[1])  # Usa la posición numérica
```

```
20
```

Modificación de valores

Puedes modificar el valor de un elemento en una Serie de la misma manera que accederías a él:

```
series['b'] = 100  # Cambia el valor de la etiqueta 'b'
print(series)
```

```
a     10
b    100
c     30
d     40
e     50
dtype: int64
```

Filtrado de datos

Pandas permite filtrar las Series utilizando condiciones booleanas:

```
filtered_series = series[series > 25]  # Filtra los valores mayores a 25
print(filtered_series)
```

```
b    100
c     30
d     40
e     50
dtype: int64
```

También puedes utilizar múltiples condiciones:

```
filtered_series = series[(series > 25) & (series < 100)]
print(filtered_series)
```

```
c    30
d    40
e    50
dtype: int64
```

Operaciones matemáticas

Las Series en Pandas soportan una amplia gama de operaciones aritméticas que se aplican de forma vectorizada. Esto significa que las operaciones se realizan en todos los elementos de la Serie simultáneamente, lo que mejora la eficiencia.

- **Suma**:

```
series = pd.Series([10, 20, 30, 40])
print(series + 10)  # Añade 10 a todos los valores de la Serie

0    20
1    30
2    40
3    50
dtype: int64
```

- **Multiplicación**:

```
print(series * 2)  # Multiplica todos los valores por 2

0    20
1    40
2    60
3    80
dtype: int64
```

- **Potencias y funciones**:

```
print(series ** 2)  # Eleva todos los valores al cuadrado

0     100
1     400
2     900
3    1600
dtype: int64

print(series.apply(np.sqrt))  # Aplica la función raíz cuadrada a cada
valor

0    3.162278
1    4.472136
2    5.477226
3    6.324555
dtype: float64
```

Manejo de Datos Faltantes

Es común trabajar con conjuntos de datos incompletos que contienen valores faltantes. Pandas representa estos valores faltantes con **NaN** (Not a Number). Al crear una Serie con un índice personalizado, si faltan datos para alguna etiqueta, Pandas asigna automáticamente NaN:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000}
states = ['Ohio', 'Texas', 'Oregon', 'Utah', 'California']

my_series = pd.Series(sdata, index=states)
print(my_series)
```

```
Ohio          35000.0
Texas         71000.0
Oregon        16000.0
Utah              NaN
California        NaN
dtype: float64
```

En este ejemplo, los valores correspondientes a 'Utah' y 'California' están marcados como NaN porque no había datos para esas etiquetas en el diccionario original.

Detección de NaN

Puedes detectar valores NaN utilizando el método `isna()`:

```
print(my_series.isna())  # Devuelve una Serie booleana que indica dónde están los NaN
```

```
Ohio          False
Texas         False
Oregon        False
Utah           True
California     True
dtype: bool
```

Relleno de valores faltantes

Para manejar los valores faltantes, puedes rellenarlos utilizando el método `fillna()`:

```
my_series.fillna(0, inplace=True)  # Rellena los NaN con 0
print(my_series)
```

```
Ohio          35000.0
Texas         71000.0
Oregon        16000.0
Utah              0.0
California        0.0
dtype: float64
```

También puedes usar otras técnicas, como reemplazar los NaN con el valor promedio de la Serie:

```
my_series.fillna(my_series.mean(), inplace=True)  # Rellena NaN con el valor promedio
print(my_series)
```

```
Ohio          35000.0
Texas         71000.0
Oregon        16000.0
Utah              0.0
```

```
California        0.0
dtype: float64
```

Aplicación de Funciones

Las Series en Pandas permiten aplicar funciones personalizadas a cada valor utilizando el método `apply()`. Esto es extremadamente útil cuando necesitas realizar una operación compleja en cada valor de la Serie:

```
def multiply_by_two(x):
    return x * 2

result_series = my_series.apply(multiply_by_two)
print(result_series)
```

```
Ohio           70000.0
Texas         142000.0
Oregon         32000.0
Utah               0.0
California         0.0
dtype: float64
```

Además de funciones personalizadas, puedes utilizar funciones de NumPy o incluso funciones lambda:

```
print(my_series.apply(np.log))  # Aplica la función logaritmo natural
a cada valor
```

```
Ohio          10.463103
Texas         11.170435
Oregon         9.680344
Utah                -inf
California          -inf
dtype: float64
```

```
# Uso de una función lambda para operaciones personalizadas
print(my_series.apply(lambda x: x + 100))
```

```
Ohio          35100.0
Texas         71100.0
Oregon        16100.0
Utah            100.0
California      100.0
dtype: float64
```

Reindexación

Reindexar una Serie significa cambiar el índice o reorganizarlo, lo que puede ser útil cuando deseas alinear datos con un nuevo índice:

```
new_index = ['California', 'Ohio', 'Oregon', 'Texas', 'Utah']
reindexed_series = my_series.reindex(new_index)
print(reindexed_series)

California        0.0
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Utah              0.0
dtype: float64
```

Si el nuevo índice contiene etiquetas que no estaban en la Serie original, se asignarán valores NaN a esos índices.

Ordenación y Clasificación

Pandas proporciona métodos para ordenar una Serie ya sea por su índice o por sus valores:

- **Ordenar por índice**:

```
sorted_series = my_series.sort_index()
print(sorted_series)

California        0.0
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Utah              0.0
dtype: float64
```

- **Ordenar por valor**:

```
sorted_series = my_series.sort_values()
print(sorted_series)

Utah              0.0
California        0.0
Oregon        16000.0
Ohio          35000.0
Texas         71000.0
dtype: float64
```

Resumen de estadísticas

Pandas permite obtener rápidamente estadísticas resumen de las Series mediante métodos como `mean()`, `median()`, `std()`, etc.:

```
print(my_series.mean())    # Promedio de los valores

24400.0
```

```
print(my_series.median())  # Mediana de los valores
16000.0
print(my_series.max())     # Valor máximo
71000.0
print(my_series.min())     # Valor mínimo
0.0
print(my_series.std())     # Desviación estándar
29770.79105432034
```

Las Series de **Pandas** son una estructura de datos extremadamente útil y flexible para trabajar con datos unidimensionales. Proporcionan una amplia gama de operaciones y métodos que permiten el análisis eficiente de datos, desde operaciones básicas de acceso hasta operaciones más avanzadas, como el manejo de valores faltantes y la aplicación de funciones personalizadas. Con las Series, puedes realizar análisis complejos de datos de forma sencilla y rápida, lo que las convierte en una herramienta indispensable en el análisis de datos.

Dataframes

En Pandas, un DataFrame es una estructura de datos tabular bidimensional, de tamaño mutable y potencialmente heterogénea, con ejes etiquetados (filas y columnas). A menudo se compara con una hoja de cálculo o tabla SQL, ya que proporciona una forma conveniente de almacenar y manipular datos.

Entendiendo los DataFrames

- **Series**: Antes de sumergirnos en los DataFrames, es esencial entender el concepto de Series. Una Serie es un objeto similar a un arreglo unidimensional que puede contener varios tipos de datos. Los DataFrames son esencialmente colecciones de objetos Serie, cada uno representando una columna.
- **Columnas**: En un DataFrame, cada Serie representa una columna. Estas columnas pueden contener diferentes tipos de datos, como enteros, flotantes o cadenas.
- **Filas**: Las filas en un DataFrame están organizadas por etiquetas de índice. Cada fila corresponde a una entrada específica, y puedes acceder a las filas usando sus etiquetas de índice.

Los DataFrames son una herramienta poderosa para la manipulación, análisis y limpieza de datos. Ofrecen una forma estructurada de trabajar con datos, facilitando el filtrado, ordenamiento y cálculo de estadísticas en conjuntos de datos. Exploraremos varias operaciones y funcionalidades de DataFrame en esta guía.

Creación de DataFrames

Puedes crear DataFrames a partir de varias fuentes de datos, incluyendo diccionarios, listas, archivos CSV, bases de datos SQL y más. A continuación, algunos ejemplos sobre cómo hacerlo:

`Desde diccionarios con listas como valores`

```
# Crear un diccionario con columnas y datos
dict_states = {
    "estado": ["Oregón", "Utah", "Nuevo México", "Nebraska"],
    "año": [1900, 1898, 2000, 1900],
    "algo_más": [456, "ssdsd", 0.023, np.nan]
}

# Crear un DataFrame a partir del diccionario
df = pd.DataFrame(dict_states)

# Mostrar el DataFrame
df
```

```
         estado   año algo_más
0        Oregón  1900      456
1          Utah  1898    ssdsd
2  Nuevo México  2000    0.023
3      Nebraska  1900      NaN
```

En este ejemplo, primero importamos las bibliotecas Pandas y NumPy. Luego, creamos un diccionario `dict_states`, donde cada clave representa el nombre de una columna, y el valor asociado es una lista de datos para esa columna. Pasamos este diccionario a `pd.DataFrame()` para crear un DataFrame llamado `df`. Finalmente, mostramos el DataFrame.

Cuando usas Jupyter Notebook, los objetos DataFrame de Pandas se muestran como tablas HTML más amigables para el navegador, lo que facilita ver y explorar los datos de manera interactiva. Puedes encontrar más opciones y detalles de personalización para la visualización de DataFrame en la documentación de Pandas.

`Desde lista de diccionarios` Si creas un DataFrame a partir de una lista de diccionarios:

- Cada diccionario dentro de la lista representa una fila en el DataFrame.
- Las claves dentro de cada diccionario se convierten en los nombres de las columnas del DataFrame.
- Todos los diccionarios en la lista deben tener la misma estructura en términos de claves para asegurar la consistencia.

Aquí hay algunos ejemplos:

```
# Ejemplo 1: Crear un DataFrame con claves consistentes en cada diccionario
dict_states = {
    "estado": ["Oregón", "Utah", "Nuevo México", "Nebraska"],
```

```
    "año": [1900, 1898, 2000, 1900],
    "algo_más": [456, "ssdsd", 0.023, np.nan]
}

lista_de_diccionarios = [
    {"estado": "Oregón", "año": 1900, "algo_más": "oiuyghj"},  # Cada
diccionario representa una fila
    {"estado": "Utah", "año": 1989, "algo_más": 678},  # Estructuras
variables entre diccionarios
    {"estado": "Nuevo México", "año": 456, "algo_más": 87, "extra":
98765}  # Clave extra 'extra' en un diccionario
]

# Crear un DataFrame a partir de la lista de diccionarios
df = pd.DataFrame(lista_de_diccionarios)
df
```

```
         estado   año algo_más    extra
0        Oregón  1900  oiuyghj      NaN
1          Utah  1989      678      NaN
2  Nuevo México   456       87  98765.0
```

En el primer ejemplo, creamos un DataFrame `df` usando una lista de diccionarios `lista_de_diccionarios`, donde cada diccionario representa una fila con columnas que coinciden con las claves. En el segundo ejemplo, mostramos que puedes tener variaciones en la estructura de cada diccionario siempre y cuando tengan claves comunes.

Para más detalles, puedes referirte a la documentación de pandas sobre `from_dict`.

Carga de datos desde archivos

`.csv`

En este ejemplo, cargamos datos en un DataFrame `df` usando la función `pd.read_csv()`. Cargar datos desde archivos CSV es una operación común al trabajar con Pandas, ya que te permite traer datos externos a un DataFrame para su análisis y manipulación. El DataFrame resultante, `df`, contiene los datos estructurados del archivo CSV, haciéndolos accesibles para una exploración y análisis posteriores.

Conjunto de Datos de Precios de Aguacate: El conjunto de datos "Precios de Aguacate", obtenido de Kaggle, es un conjunto de datos ampliamente utilizado para proyectos de análisis de datos y aprendizaje automático. Proporciona datos históricos sobre precios y ventas de aguacates en varias regiones de los Estados Unidos. Este conjunto de datos es valioso para entender las tendencias en los precios de los aguacates, los volúmenes de ventas y su relación con diferentes factores.

Atributos Clave:

- **Columnas**: El conjunto de datos incluye varias columnas de información. Algunas de las columnas clave típicamente encontradas en este conjunto de datos incluyen:

  - **Fecha**: La fecha de observación.
  - **Precio Promedio**: El precio promedio de los aguacates.
  - **Volumen Total**: Volumen total de aguacates vendidos.
  - **4046**: Volumen de aguacates Hass pequeños vendidos.
  - **4225**: Volumen de aguacates Hass grandes vendidos.
  - **4770**: Volumen de aguacates Hass extra grandes vendidos.
  - **Bolsas Totales**: Total de bolsas de aguacates vendidas.
  - **Bolsas Pequeñas**: Bolsas de aguacates pequeños vendidas.
  - **Bolsas Grandes**: Bolsas de aguacates grandes vendidas.
  - **Bolsas Extra Grandes**: Bolsas de aguacates extra grandes vendidas.
  - **Tipo**: El tipo de aguacates, a menudo categorizados como convencionales u orgánicos.
  - **Región**: La región o ciudad dentro de los Estados Unidos donde se registraron los datos.

- **Rango de Fechas**: El conjunto de datos abarca un rango de fechas, lo que permite el análisis de series de tiempo. Puedes examinar cómo cambian los precios y ventas de aguacates a lo largo de diferentes estaciones y años.

- **Regiones**: Se proporciona información para varias regiones o ciudades a través de los Estados Unidos, lo que permite el análisis de variaciones de precios y ventas en diferentes mercados.

- **Tipos**: El conjunto de datos distingue entre diferentes tipos de aguacates, como convencionales y orgánicos, lo que puede ser útil para comparar tendencias de precios entre estas categorías.

- **Volumen**: Están disponibles datos sobre el volumen total de aguacates vendidos. Esta métrica de volumen se utiliza a menudo para analizar la demanda del mercado.

- **Precio Promedio**: El conjunto de datos contiene el precio promedio de los aguacates, una métrica fundamental para entender las tendencias de precios.

Casos de Uso:

- Este conjunto de datos se utiliza comúnmente para aprender y practicar el análisis de datos, visualización de datos y modelado de regresión en proyectos de ciencia de datos y aprendizaje automático.

- Sirve como un recurso valioso para entender cómo trabajar con datos del mundo real, extraer conocimientos y tomar decisiones basadas en datos.

```
# Cargar conjunto de datos
df = pd.read_csv("datasets/avocado_kaggle.csv")

# Mostrar conjunto de datos
df
```

```
       Unnamed: 0        Date  AveragePrice  Total Volume     4046
4225  \
0               0  2015-12-27          1.33      64236.62  1036.74
54454.85
1               1  2015-12-20          1.35      54876.98   674.28
44638.81
2               2  2015-12-13          0.93     118220.22   794.70
109149.67
3               3  2015-12-06          1.08      78992.15  1132.00
71976.41
4               4  2015-11-29          1.28      51039.60   941.48
43838.39
...           ...         ...           ...           ...      ...
...
18244           7  2018-02-04          1.63      17074.83  2046.96
1529.20
18245           8  2018-01-28          1.71      13888.04  1191.70
3431.50
18246           9  2018-01-21          1.87      13766.76  1191.92
2452.79
18247          10  2018-01-14          1.93      16205.22  1527.63
2981.04
18248          11  2018-01-07          1.62      17489.58  2894.77
2356.13

         4770  Total Bags  Small Bags  Large Bags  XLarge Bags
type  \
0       48.16     8696.87     8603.62       93.25          0.0
conventional
1       58.33     9505.56     9408.07       97.49          0.0
conventional
2      130.50     8145.35     8042.21      103.14          0.0
conventional
3       72.58     5811.16     5677.40      133.76          0.0
conventional
4       75.78     6183.95     5986.26      197.69          0.0
conventional
...       ...         ...         ...         ...          ...
...
18244    0.00    13498.67    13066.82      431.85          0.0
organic
18245    0.00     9264.84     8940.04      324.80          0.0
organic
18246  727.94     9394.11     9351.80       42.31          0.0
organic
18247  727.01    10969.54    10919.54       50.00          0.0
organic
18248  224.53    12014.15    11988.14       26.01          0.0
organic
```

```
       year             region
0      2015             Albany
1      2015             Albany
2      2015             Albany
3      2015             Albany
4      2015             Albany
...     ...                ...
18244  2018  WestTexNewMexico
18245  2018  WestTexNewMexico
18246  2018  WestTexNewMexico
18247  2018  WestTexNewMexico
18248  2018  WestTexNewMexico

[18249 rows x 14 columns]
```

`.xlsx`, `.xls`, `.xlsm`, `.xlsb`, `.odf`, `.ods`, `.odt`

Pandas, una poderosa biblioteca de manipulación de datos en Python, proporciona una funcionalidad robusta para leer y escribir archivos Excel. Esto incluye el soporte para múltiples formatos de archivos Excel, permitiendo a los usuarios trabajar con datos de hojas de cálculo de una manera sencilla y eficiente.

Pandas simplifica el proceso de manejo de archivos Excel, permitiéndote integrar sin problemas datos de hojas de cálculo en tus flujos de trabajo de análisis de datos. Ya sea que estés tratando con archivos `.xls` clásicos, libros de trabajo `.xlsx` modernos o otros formatos compatibles con Excel como `.xlsm`, `.xlsb`, `.odf`, `.ods` y `.odt`, Pandas ofrece herramientas versátiles para importar y exportar datos.

Características Clave:

- **Leer Datos de Excel**: Pandas proporciona funciones para leer datos de Excel en DataFrames, preservando la estructura y el formato de las hojas de trabajo. Puedes leer datos de manera eficiente de múltiples hojas dentro de un libro de trabajo.
- **Escribir Datos en Excel**: Pandas te permite escribir DataFrames de vuelta a archivos Excel, permitiéndote guardar tus datos junto con cualquier modificación o análisis que hayas realizado.
- **Compatibilidad**: Pandas soporta varias extensiones de archivos Excel, incluyendo `.xls`, `.xlsx`, `.xlsm`, `.xlsb`, `.odf`, `.ods` y `.odt`, asegurando compatibilidad con diferentes versiones y formatos de Excel.
- **Preservación de Datos**: Al leer archivos Excel, Pandas retiene tipos de datos, fórmulas, estilos de celda y otros atributos, asegurando la integridad de los datos.
- **Manipulación de Datos**: Una vez que los datos están cargados en DataFrames de Pandas, puedes usar las extensas capacidades de manipulación y análisis de datos de Pandas para explorar, limpiar, transformar y visualizar tus datos.

Casos de Uso:

- **Extracción de Datos**: Extraer datos estructurados de archivos Excel para realizar análisis, informes o visualización.
- **Integración de Datos**: Combinar datos de múltiples hojas o libros de trabajo de Excel en un único conjunto de datos consolidado.
- **Exportación de Datos**: Guardar los resultados del análisis de datos realizado con Pandas de vuelta en archivos Excel para compartir o procesar más adelante.
- **Automatización**: Automatizar tareas de extracción y manipulación de datos incorporando Pandas en tu canal de datos o flujo de trabajo.

Pandas hace que trabajar con archivos Excel sea muy fácil, permitiéndote aprovechar el poder de Python para tus proyectos de análisis de datos mientras interactúas sin problemas con datos de Excel.

Para comenzar, explora la extensa documentación de Pandas sobre Entrada/Salida de Archivos Excel para aprender sobre los diversos métodos y opciones disponibles para leer y escribir archivos Excel.

```
# Otro conjunto de datos
df_from_excel = pd.read_excel("datasets/Online Retail.xlsx",
engine="openpyxl", nrows=5)
df_from_excel.head()
```

```
   InvoiceNo         InvoiceNo.1 InvoiceNo.2  \
0     536365 2010-12-01 08:26:00      85123A
1     536373 2010-12-01 09:02:00      85123A
2     536375 2010-12-01 09:32:00      85123A
3     536390 2010-12-01 10:19:00      85123A
4     536394 2010-12-01 10:39:00      85123A

                          InvoiceNo.3  InvoiceNo.4  InvoiceNo.5
InvoiceNo.6  \
0  CREAM HANGING HEART T-LIGHT HOLDER            6         2.55
15.3
1  CREAM HANGING HEART T-LIGHT HOLDER            6         2.55
15.3
2  CREAM HANGING HEART T-LIGHT HOLDER            6         2.55
15.3
3  CREAM HANGING HEART T-LIGHT HOLDER           64         2.55
163.2
4  CREAM HANGING HEART T-LIGHT HOLDER           32         2.55
81.6

   InvoiceNo.7     InvoiceNo.8
0        17850  United Kingdom
1        17850  United Kingdom
```

```
2        17850  United Kingdom
3        17511  United Kingdom
4        13408  United Kingdom
```

Leyendo diferentes hojas

```
# Definir la ruta al archivo Excel
excel_file_path = "datasets/Online Retail.xlsx"

# Leer la pestaña predeterminada (la primera)
# Usar la función read_excel para leer la primera hoja del archivo
Excel (comportamiento predeterminado)
df_default_tab = pd.read_excel(excel_file_path, engine="openpyxl",
nrows=5)

# Mostrar las primeras 5 filas del DataFrame
df_default_tab.head()
```

```
   InvoiceNo         InvoiceNo.1 InvoiceNo.2  \
0     536365 2010-12-01 08:26:00      85123A
1     536373 2010-12-01 09:02:00      85123A
2     536375 2010-12-01 09:32:00      85123A
3     536390 2010-12-01 10:19:00      85123A
4     536394 2010-12-01 10:39:00      85123A

                          InvoiceNo.3  InvoiceNo.4  InvoiceNo.5
InvoiceNo.6  \
0  CREAM HANGING HEART T-LIGHT HOLDER            6         2.55
15.3
1  CREAM HANGING HEART T-LIGHT HOLDER            6         2.55
15.3
2  CREAM HANGING HEART T-LIGHT HOLDER            6         2.55
15.3
3  CREAM HANGING HEART T-LIGHT HOLDER           64         2.55
163.2
4  CREAM HANGING HEART T-LIGHT HOLDER           32         2.55
81.6

   InvoiceNo.7     InvoiceNo.8
0        17850  United Kingdom
1        17850  United Kingdom
2        17850  United Kingdom
3        17511  United Kingdom
4        13408  United Kingdom
```

```
# Leer otra pestaña (por ejemplo, "nueva_pestaña")
# Especificar el nombre de la hoja como una cadena para leer una hoja
específica del archivo Excel
df_new_tab = pd.read_excel(excel_file_path, engine="openpyxl",
sheet_name="new_tab", nrows=5)
```

```
# Mostrar las primeras 5 filas del DataFrame de la hoja
"nueva_pestaña"
df_new_tab.head()
```

```
   column1 column2
0  example   hello
```

Desde bases de datos

`sql`: documentación

```
# Importar SQLite
from sqlite3 import connect

conn = connect(':memory:')
df = pd.read_sql('SELECT columna_1, columna_2 FROM datos_muestra',
conn)

df.to_sql('datos_prueba', conn)
```

`mongodb`

```
import pymongo
from pymongo import MongoClient

client = MongoClient()
db = client.nombre_base_de_datos
coleccion = db.nombre_coleccion
datos = pd.DataFrame(list(coleccion.find()))
```

Análisis exploratorio de un dataframe

En esta sección, nos adentraremos en el mundo del análisis exploratorio de datos (EDA) utilizando Pandas. El EDA es un paso crucial en el proceso de análisis de datos, donde llegamos a conocer nuestros datos, entendemos sus características y descubrimos percepciones iniciales. El EDA no solo ayuda a entender los datos, sino que también es fundamental para formular preguntas de investigación y desarrollar hipótesis.

Historia y Origen del EDA

El término "Análisis Exploratorio de Datos" fue popularizado por el estadístico John Tukey en su libro de 1977 titulado "Exploratory Data Analysis". Tukey abogó por un enfoque más intuitivo y gráfico para el análisis de datos, en contraposición a los métodos puramente matemáticos y paramétricos que dominaban en ese momento. Su idea era que los analistas deberían sumergirse en los datos y utilizarlos para descubrir patrones, tendencias y relaciones, en lugar de depender únicamente de técnicas estadísticas predefinidas.

Tukey introdujo varios gráficos, como los diagramas de caja (boxplots) y los gráficos de dispersión (scatter plots), que se convirtieron en herramientas estándar en el EDA. Su enfoque enfatizaba la visualización como un medio poderoso para explorar los datos y generar hipótesis.

Desde entonces, el EDA se ha convertido en una práctica esencial en la ciencia de datos, análisis estadístico y aprendizaje automático, ya que permite a los analistas y científicos de datos descubrir insights importantes antes de aplicar técnicas más complejas.

Importancia del EDA

1. **Identificación de patrones y tendencias**: El EDA ayuda a visualizar los datos, lo que facilita la identificación de patrones y tendencias que pueden no ser evidentes a partir de estadísticas simples.

2. **Detección de valores atípicos (outliers)**: Los métodos gráficos y estadísticos en el EDA permiten identificar valores atípicos que podrían influir en el análisis y los modelos posteriores.

3. **Comprensión de la distribución de los datos**: A través del EDA, podemos comprender cómo se distribuyen los datos, lo que es esencial para seleccionar las técnicas estadísticas adecuadas y validar suposiciones.

4. **Preparación de datos**: El EDA revela la calidad de los datos y la necesidad de limpieza, transformación o imputación de datos faltantes, lo que mejora la calidad de los análisis posteriores.

5. **Generación de hipótesis**: Al explorar los datos, los analistas pueden formular preguntas y generar hipótesis que pueden ser probadas en análisis posteriores.

Utilizaremos un DataFrame, `df`, cargado desde el conjunto de datos "Advertising.csv" como ejemplo para realizar varios análisis exploratorios. Comencemos cargando el conjunto de datos y echando un vistazo a su contenido.

```
# cargar el conjunto de datos
df = pd.read_csv("datasets/Advertising.csv")

# Verificar las primeras filas
df.head()
```

```
      TV  Radio  Newspaper  Sales
0  230.1   37.8       69.2   22.1
1   44.5   39.3       45.1   10.4
2   17.2   45.9       69.3    9.3
3  151.5   41.3       58.5   18.5
4  180.8   10.8       58.4   12.9
```

Esto nos dará una idea de cómo se ven los datos y si hay columnas que necesitan limpieza o transformación.

Información meta

`shape, columns, dtypes, info, describe`

Cuando trabajas con datos en un DataFrame, es crucial entender las características básicas y la estructura del conjunto de datos. Para obtener percepciones sobre los datos, podemos recuperar información meta sobre el DataFrame. Esta información incluye detalles como la forma del DataFrame (número de filas y columnas), nombres de columnas, tipos de datos, información general y un resumen estadístico de los datos. Comprender estos aspectos es vital para tomar decisiones informadas durante el análisis.

En esta sección, exploraremos cómo usar varias funciones de Pandas para obtener información meta esencial sobre tu DataFrame. Este conocimiento te ayudará a comprender mejor y preparar tus datos para el análisis y la visualización.

```
# Forma del DataFrame (filas, columnas)
data_shape = df.shape
print(f"Forma del DataFrame: {data_shape}")
```

```
Forma del DataFrame: (200, 4)
```

Aquí, `shape` devuelve una tupla con el número de filas y columnas, lo cual es esencial para entender la escala de tus datos.

```
# Nombres de columnas
column_names = df.columns
print(f"Nombres de Columnas: {column_names}")
```

```
Nombres de Columnas: Index(['TV', 'Radio', 'Newspaper', 'Sales'],
dtype='object')
```

Conocer los nombres de las columnas es fundamental para acceder y manipular los datos de manera eficiente.

```
# Tipos de datos de cada columna
data_types = df.dtypes
print(f"Tipos de Datos:\n{data_types}")
```

```
Tipos de Datos:
TV           float64
Radio        float64
Newspaper    float64
Sales        float64
dtype: object
```

Entender los tipos de datos es crucial, ya que diferentes tipos requieren diferentes métodos de análisis. Por ejemplo, las columnas categóricas pueden necesitar codificación, mientras que las columnas numéricas pueden ser utilizadas directamente para análisis estadísticos.

```
# Información General sobre el DataFrame
data_info = df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   TV         200 non-null    float64
 1   Radio      200 non-null    float64
 2   Newspaper  200 non-null    float64
 3   Sales      200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
```

Este comando proporciona un resumen conciso que incluye el número de entradas, tipos de datos y memoria utilizada. Es útil para identificar rápidamente columnas con datos faltantes.

```
# Resumen Estadístico
data_summary = df.describe()
print("\nResumen Estadístico:")
print(data_summary)
```

```
Resumen Estadístico:
               TV       Radio   Newspaper       Sales
count  200.000000  200.000000  200.000000  200.000000
mean   147.042500   23.264000   30.554000   14.022500
std     85.854236   14.846809   21.778621    5.217457
min      0.700000    0.000000    0.300000    1.600000
25%     74.375000    9.975000   12.750000   10.375000
50%    149.750000   22.900000   25.750000   12.900000
75%    218.825000   36.525000   45.100000   17.400000
max    296.400000   49.600000  114.000000   27.000000
```

La función `describe()` calcula estadísticas descriptivas como la media, la desviación estándar, los valores mínimos y máximos, y los cuartiles para columnas numéricas, ayudando a identificar la distribución de los datos.

Previsualización

Antes de sumergirte en un análisis y manipulación profundos de tu conjunto de datos, a menudo es útil obtener un vistazo rápido de cómo se ven los datos. Pandas proporciona dos métodos útiles, `head()` y `tail()`, para ayudarte a previsualizar el principio y el final de tu DataFrame.

En esta sección, exploraremos cómo usar estos métodos para mostrar un subconjunto de tus datos, facilitando la comprensión de la estructura y el contenido de tu DataFrame. Estas herramientas simples pero poderosas son el primer paso para familiarizarte con tus datos, permitiéndote identificar cualquier patrón o problema inmediato.

`head`

```
df.head()
```

```
      TV  Radio  Newspaper  Sales
0  230.1   37.8       69.2   22.1
1   44.5   39.3       45.1   10.4
2   17.2   45.9       69.3    9.3
3  151.5   41.3       58.5   18.5
4  180.8   10.8       58.4   12.9
```

Por defecto, `head` muestra las primeras 5 filas, pero puedes especificar un número diferente como parámetro para ver más o menos filas. Esto es útil para tener una idea rápida de los datos y verificar si hay anomalías o valores inesperados.

`tail`

```
df.tail()
```

```
        TV  Radio  Newspaper  Sales
195   38.2    3.7       13.8    7.6
196   94.2    4.9        8.1    9.7
197  177.0    9.3        6.4   12.8
198  283.6   42.0       66.2   25.5
199  232.1    8.6        8.7   13.4
```

`tail` muestra las últimas filas del DataFrame, lo que te permite verificar cómo se comportan los datos al final del conjunto, especialmente útil si los datos están ordenados cronológicamente.

Ordenar un dataframe

Organizar y ordenar tus datos es una parte fundamental del análisis de datos. En esta sección, exploraremos cómo ordenar un DataFrame usando la biblioteca Pandas. Al ordenar datos, puedes obtener valiosas percepciones, identificar tendencias y hacer que tus datos sean más accesibles para el análisis.

Cubriremos varios escenarios, como ordenar por una o más columnas, en orden ascendente o descendente, y seleccionar columnas específicas para ver. Entender cómo organizar tus datos de manera efectiva puede mejorar significativamente tu capacidad para extraer información significativa de ellos.

Sumergámonos en las diferentes formas de ordenar y organizar tus datos usando Pandas.

```
# Cargar el DataFrame desde un archivo CSV
df = pd.read_csv("datasets/avocado_kaggle.csv")

# Ordenar por una sola columna en orden descendente (año más reciente primero)
df.sort_values(by="year", ascending=False)
```

```
       Unnamed: 0        Date  AveragePrice  Total Volume        4046
\
9124           10  2018-01-14          0.90     950954.60   463945.73

8883            9  2018-01-21          0.78    1315329.83   613600.56

8905            7  2018-02-04          0.91    3316494.76  1609104.94

8906            8  2018-01-28          0.99    2533937.56  1387712.40

8907            9  2018-01-21          1.18    2208596.12  1051836.46

...           ...         ...           ...           ...         ...

10278           8  2015-11-01          1.36      52815.97    13987.25

10279           9  2015-10-25          1.41      48799.38    13800.24

10280          10  2015-10-18          1.62      33737.71    10370.68

10281          11  2015-10-11          1.72      28062.63     8093.67

0               0  2015-12-27          1.33      64236.62     1036.74


             4225      4770  Total Bags  Small Bags  Large Bags  XLarge
Bags  \
9124   188126.02  11227.47   287655.38   125408.69   162040.02
206.67
8883   246703.53  13332.26   441693.48   210780.39   226079.75
4833.34
8905   827998.81   9830.10   869560.91   678696.23   190573.14
291.54
8906   515698.93   6371.30   624154.93   474701.92   149437.52
15.49
8907   479599.90   7659.28   669500.48   562989.08   106461.22
50.18
...          ...       ...         ...         ...         ...
...
10278   20741.07      0.00    18087.65    17485.60      602.05
0.00
10279   21060.95      0.00    13938.19    12992.81      945.38
0.00
10280   16004.06      3.18     7359.79     6393.48      966.31
0.00
10281   14759.53      3.18     5206.25     5146.25       60.00
0.00
0       54454.85     48.16     8696.87     8603.62       93.25
0.00

               type  year            region
```

```
9124   conventional  2018  WestTexNewMexico
8883   conventional  2018     PhoenixTucson
8905   conventional  2018            Plains
8906   conventional  2018            Plains
8907   conventional  2018            Plains
...             ...   ...               ...
10278       organic  2015        LosAngeles
10279       organic  2015        LosAngeles
10280       organic  2015        LosAngeles
10281       organic  2015        LosAngeles
0      conventional  2015            Albany

[18249 rows x 14 columns]
```

```
# Ordenar por múltiples columnas en orden descendente (año y región)
df.sort_values(by=["year", "region"], ascending=False)
```

```
      Unnamed: 0        Date  AveragePrice  Total Volume       4046  \
9114           0  2018-03-25          0.84     965185.06  438526.12
9115           1  2018-03-18          0.88     855251.17  457635.79
9116           2  2018-03-11          0.94     897607.12  467501.55
9117           3  2018-03-04          0.88     935934.10  454269.43
9118           4  2018-02-25          0.88     895671.55  431217.01
...          ...         ...           ...           ...        ...
9173          47  2015-02-01          1.83       1228.51      33.12
9174          48  2015-01-25          1.89       1115.89      14.87
9175          49  2015-01-18          1.93       1118.47       8.02
9176          50  2015-01-11          1.77       1182.56      39.00
9177          51  2015-01-04          1.79       1373.95      57.42

           4225      4770  Total Bags  Small Bags  Large Bags  XLarge
Bags  \
9114  199585.90  11017.42   316055.62   153009.89   160999.10
2046.63
9115  137597.04   8422.08   251596.26   151191.85    98535.60
1868.81
9116  154130.63  11380.26   264594.68   152380.60   110322.16
1891.92
9117  164856.57   9907.85   306900.25   164965.35   138399.68
3535.22
9118  171532.79  10590.75   282331.00   125973.31   147040.26
9317.43
...         ...       ...         ...         ...         ...
...
9173      99.36      0.00     1096.03     1096.03        0.00
0.00
9174     148.72      0.00      952.30      952.30        0.00
0.00
9175     178.78      0.00      931.67      931.67        0.00
0.00
```

```
9176     305.12      0.00      838.44      838.44         0.00
0.00
9177     153.88      0.00     1162.65     1162.65         0.00
0.00

              type  year            region
9114  conventional  2018  WestTexNewMexico
9115  conventional  2018  WestTexNewMexico
9116  conventional  2018  WestTexNewMexico
9117  conventional  2018  WestTexNewMexico
9118  conventional  2018  WestTexNewMexico
...            ...   ...               ...
9173       organic  2015            Albany
9174       organic  2015            Albany
9175       organic  2015            Albany
9176       organic  2015            Albany
9177       organic  2015            Albany

[18249 rows x 14 columns]
```

```
# Ordenar por una sola columna en orden descendente (región)
df.sort_values(by="region", ascending=False)
```

```
      Unnamed: 0        Date  AveragePrice  Total Volume       4046  \
9124          10  2018-01-14          0.90     950954.60  463945.73
8426           1  2017-12-24          0.93     769970.84  402476.91
8428           3  2017-12-10          0.86     751137.79  384349.01
8429           4  2017-12-03          0.85     799879.00  359419.00
8430           5  2017-11-26          0.91     623094.00  354268.00
...          ...         ...           ...           ...        ...
9174          48  2015-01-25          1.89       1115.89      14.87
9175          49  2015-01-18          1.93       1118.47       8.02
9176          50  2015-01-11          1.77       1182.56      39.00
9177          51  2015-01-04          1.79       1373.95      57.42
0              0  2015-12-27          1.33      64236.62    1036.74

           4225      4770  Total Bags  Small Bags  Large Bags  XLarge
Bags  \
9124  188126.02  11227.47   287655.38   125408.69   162040.02
206.67
8426  150206.86   7966.08   209320.99    96489.71   112764.62
66.66
8428  133201.98   5774.97   227811.83   104486.24   123222.25
103.34
8429  224516.00   6690.00   209254.00    70520.00   138586.00
148.00
8430  104956.00   5465.00   158405.00    74684.00    83606.00
114.00
...         ...       ...         ...         ...         ...
...
```

```
9174     148.72      0.00      952.30      952.30         0.00
0.00
9175     178.78      0.00      931.67      931.67         0.00
0.00
9176     305.12      0.00      838.44      838.44         0.00
0.00
9177     153.88      0.00     1162.65     1162.65         0.00
0.00
0      54454.85     48.16     8696.87     8603.62        93.25
0.00

              type  year            region
9124  conventional  2018  WestTexNewMexico
8426  conventional  2017  WestTexNewMexico
8428  conventional  2017  WestTexNewMexico
8429  conventional  2017  WestTexNewMexico
8430  conventional  2017  WestTexNewMexico
...            ...   ...               ...
9174       organic  2015            Albany
9175       organic  2015            Albany
9176       organic  2015            Albany
9177       organic  2015            Albany
0     conventional  2015            Albany

[18249 rows x 14 columns]
```

```
# Ordenar por una sola columna en orden descendente (año)
df.sort_values(by="year", ascending=False)
```

```
       Unnamed: 0        Date  AveragePrice  Total Volume        4046
\
9124           10  2018-01-14          0.90     950954.60   463945.73

8883            9  2018-01-21          0.78    1315329.83   613600.56

8905            7  2018-02-04          0.91    3316494.76  1609104.94

8906            8  2018-01-28          0.99    2533937.56  1387712.40

8907            9  2018-01-21          1.18    2208596.12  1051836.46

...           ...         ...           ...           ...         ...

10278           8  2015-11-01          1.36      52815.97    13987.25

10279           9  2015-10-25          1.41      48799.38    13800.24

10280          10  2015-10-18          1.62      33737.71    10370.68

10281          11  2015-10-11          1.72      28062.63     8093.67
```

```
0               0  2015-12-27          1.33      64236.62     1036.74


            4225      4770  Total Bags  Small Bags  Large Bags  XLarge
Bags  \
9124   188126.02  11227.47   287655.38   125408.69   162040.02
206.67
8883   246703.53  13332.26   441693.48   210780.39   226079.75
4833.34
8905   827998.81   9830.10   869560.91   678696.23   190573.14
291.54
8906   515698.93   6371.30   624154.93   474701.92   149437.52
15.49
8907   479599.90   7659.28   669500.48   562989.08   106461.22
50.18
...          ...       ...         ...         ...         ...
...
10278   20741.07      0.00    18087.65    17485.60      602.05
0.00
10279   21060.95      0.00    13938.19    12992.81      945.38
0.00
10280   16004.06      3.18     7359.79     6393.48      966.31
0.00
10281   14759.53      3.18     5206.25     5146.25       60.00
0.00
0       54454.85     48.16     8696.87     8603.62       93.25
0.00

               type  year            region
9124   conventional  2018  WestTexNewMexico
8883   conventional  2018     PhoenixTucson
8905   conventional  2018            Plains
8906   conventional  2018            Plains
8907   conventional  2018            Plains
...             ...   ...               ...
10278       organic  2015        LosAngeles
10279       organic  2015        LosAngeles
10280       organic  2015        LosAngeles
10281       organic  2015        LosAngeles
0      conventional  2015            Albany

[18249 rows x 14 columns]
```

```
# Seleccionar columnas específicas (año y región) después de ordenar
df.sort_values(by="year", ascending=False)[["year", "region"]]
```

```
       year            region
9124   2018  WestTexNewMexico
8883   2018     PhoenixTucson
8905   2018            Plains
```

```
8906   2018            Plains
8907   2018            Plains
...     ...               ...
10278  2015        LosAngeles
10279  2015        LosAngeles
10280  2015        LosAngeles
10281  2015        LosAngeles
0      2015            Albany

[18249 rows x 2 columns]
```

```
# Crear una lista de nombres de columnas para crear un subconjunto de columnas
subset_columns = list(df.columns)[1:4]

# Seleccionar un subconjunto de columnas usando la lista de nombres de columnas
df[subset_columns]
```

```
             Date  AveragePrice  Total Volume
0      2015-12-27          1.33      64236.62
1      2015-12-20          1.35      54876.98
2      2015-12-13          0.93     118220.22
3      2015-12-06          1.08      78992.15
4      2015-11-29          1.28      51039.60
...           ...           ...           ...
18244  2018-02-04          1.63      17074.83
18245  2018-01-28          1.71      13888.04
18246  2018-01-21          1.87      13766.76
18247  2018-01-14          1.93      16205.22
18248  2018-01-07          1.62      17489.58

[18249 rows x 3 columns]
```

`sample`

En el análisis de datos, a menudo es esencial trabajar con un subconjunto representativo de tu conjunto de datos para varios propósitos, como exploración de datos, pruebas o entrenamiento de modelos. Pandas proporciona el método `sample` para facilitar el muestreo aleatorio de filas de un DataFrame. Este método te permite obtener filas aleatorias o una fracción específica de tus datos, convirtiéndolo en una herramienta valiosa para el análisis estadístico y tareas de aprendizaje automático. En esta sección, exploraremos cómo usar el método `sample` para extraer muestras aleatorias de un DataFrame y entender sus diversas opciones y aplicaciones.

```
# Muestreo de una fila aleatoria del DataFrame
df.sample()
```

```
       Unnamed: 0        Date  AveragePrice  Total Volume  4046
4225  4770  \
11959          26  2016-06-26          1.53       1922.75   0.0
```

```
162.87   0.0

       Total Bags  Small Bags  Large Bags  XLarge Bags     type  year
region
11959     1759.88     1759.88         0.0          0.0  organic  2016
Albany

# Muestreo de una fracción aleatoria (20%) de filas del DataFrame
df.sample(frac=0.2)

       Unnamed: 0        Date  AveragePrice  Total Volume        4046
\
6524            7  2017-11-12          1.56     226866.00     6490.00

17323          38  2017-04-09          1.42      10153.14     1802.26

13237           4  2016-11-27          1.70      86492.45     2136.81

13056          31  2016-05-22          1.38      17710.51     4850.71

11505          39  2015-03-29          1.42      60007.19    28872.83

...           ...         ...           ...           ...         ...

7288           29  2017-06-11          1.41     401365.48   246114.25

11804          26  2015-06-28          1.64     659947.03   192182.13

8740           10  2018-01-14          1.16     322932.06    79986.22

8377            5  2017-11-26          1.18    4182730.00  1677367.00

7263            4  2017-12-03          1.31     318626.00   202974.00


              4225      4770  Total Bags  Small Bags  Large Bags
XLarge Bags  \
6524    162389.00    206.00    57781.00    55528.00     2253.00
0.00
17323     1023.40      3.43     7324.05     1400.00     5924.05
0.00
13237    31954.82   1694.84    50705.98    30998.32    19707.66
0.00
13056     8099.11      0.00     4760.69      496.56     4264.13
0.00
11505     5897.44      0.00    25236.92    25143.34       93.58
0.00
...           ...       ...         ...         ...         ...
...
7288     41215.61     61.99   113973.63    61632.80    46130.83
6210.00
```

```
11804   332739.27   5956.87   129068.76    80006.40    49062.36
0.00
8740     96281.04   7841.07   138823.73    71330.43    67126.63
366.67
8377   1002610.00  79478.00  1423276.00   817296.00   595877.00
10103.00
7263     23478.00    987.00    91187.00    60278.00    30905.00
3.00

               type  year               region
6524   conventional  2017  HartfordSpringfield
17323       organic  2017              StLouis
13237       organic  2016             Midsouth
13056       organic  2016             LasVegas
11505       organic  2015         SouthCentral
...             ...   ...                  ...
7288   conventional  2017              Orlando
11804       organic  2015              TotalUS
8740   conventional  2018             LasVegas
8377   conventional  2017                 West
7263   conventional  2017              Orlando

[3650 rows x 14 columns]
```

`display`

Cuando trabajas con Jupyter Notebook o JupyterLab, puedes usar la función `display` para renderizar DataFrames de Pandas en un formato más visualmente atractivo e interactivo. Aunque la visualización tabular predeterminada de Pandas es informativa, la función `display` proporciona opciones adicionales de flexibilidad y personalización.

Al usar `display`, puedes aprovechar las capacidades mejoradas de formato de tablas de los entornos Jupyter, incluyendo columnas ordenables, diseño adaptable y una representación visual mejorada de tus datos. Es especialmente útil cuando se trata con conjuntos de datos más grandes o cuando quieres presentar tus datos de una manera más limpia e interactiva para la exploración de datos o informes.

```
# Mostrar el DataFrame usando la función display (equivalente a print)
display(df)
```

```
      Unnamed: 0        Date  AveragePrice  Total Volume     4046
4225  \
0              0  2015-12-27          1.33      64236.62  1036.74
54454.85
1              1  2015-12-20          1.35      54876.98   674.28
44638.81
2              2  2015-12-13          0.93     118220.22   794.70
109149.67
3              3  2015-12-06          1.08      78992.15  1132.00
71976.41
```

```
4               4  2015-11-29          1.28      51039.60   941.48
43838.39
...           ...         ...           ...           ...      ...
...
18244           7  2018-02-04          1.63      17074.83  2046.96
1529.20
18245           8  2018-01-28          1.71      13888.04  1191.70
3431.50
18246           9  2018-01-21          1.87      13766.76  1191.92
2452.79
18247          10  2018-01-14          1.93      16205.22  1527.63
2981.04
18248          11  2018-01-07          1.62      17489.58  2894.77
2356.13

         4770  Total Bags  Small Bags  Large Bags  XLarge Bags
type  \
0       48.16     8696.87     8603.62       93.25          0.0
conventional
1       58.33     9505.56     9408.07       97.49          0.0
conventional
2      130.50     8145.35     8042.21      103.14          0.0
conventional
3       72.58     5811.16     5677.40      133.76          0.0
conventional
4       75.78     6183.95     5986.26      197.69          0.0
conventional
...       ...         ...         ...         ...           ...
...
18244    0.00    13498.67    13066.82      431.85          0.0
organic
18245    0.00     9264.84     8940.04      324.80          0.0
organic
18246  727.94     9394.11     9351.80       42.31          0.0
organic
18247  727.01    10969.54    10919.54       50.00          0.0
organic
18248  224.53    12014.15    11988.14       26.01          0.0
organic

       year            region
0      2015            Albany
1      2015            Albany
2      2015            Albany
3      2015            Albany
4      2015            Albany
...     ...               ...
18244  2018  WestTexNewMexico
18245  2018  WestTexNewMexico
```

```
18246  2018  WestTexNewMexico
18247  2018  WestTexNewMexico
18248  2018  WestTexNewMexico

[18249 rows x 14 columns]
```

Valores NaN

En el análisis de datos, la falta de datos es una ocurrencia común y puede impactar significativamente la precisión y fiabilidad de tus análisis. Una forma de representar datos faltantes en Pandas y muchas otras bibliotecas de análisis de datos es mediante el uso del valor especial "NaN", que significa "Not A Number" (No Es Un Número).

NaN es esencialmente un marcador de posición para puntos de datos faltantes o indefinidos, y generalmente se trata como un valor de punto flotante. Esto permite que Pandas trabaje con datos faltantes mientras mantiene los tipos de datos dentro de un DataFrame.

Manejar los valores NaN es un aspecto crucial de la limpieza y preprocesamiento de datos, ya que puede afectar los cálculos estadísticos, visualizaciones y modelos de aprendizaje automático. En esta sección, exploraremos varias técnicas y funciones en Pandas para tratar con datos faltantes y asegurar que tu análisis de datos produzca resultados precisos y significativos.

```
# Verificar la información del conjunto de datos
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18249 entries, 0 to 18248
Data columns (total 14 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Unnamed: 0    18249 non-null  int64
 1   Date          18249 non-null  object
 2   AveragePrice  18249 non-null  float64
 3   Total Volume  18249 non-null  float64
 4   4046          18249 non-null  float64
 5   4225          18249 non-null  float64
 6   4770          18249 non-null  float64
 7   Total Bags    18249 non-null  float64
 8   Small Bags    18249 non-null  float64
 9   Large Bags    18249 non-null  float64
 10  XLarge Bags   18249 non-null  float64
 11  type          18249 non-null  object
 12  year          18249 non-null  int64
 13  region        18249 non-null  object
dtypes: float64(9), int64(2), object(3)
memory usage: 1.9+ MB
```

Este comando muestra la cantidad de valores no nulos en cada columna, lo que es útil para identificar rápidamente las columnas que contienen datos faltantes.

```
# Verificar si hay valores faltantes en el DataFrame
missing_values = pd.isnull(df)
```

Este comando genera un DataFrame booleano que indica la ubicación de los valores NaN.

```
# Contar los valores faltantes en cada columna
missing_counts = missing_values.sum()
print("Valores Faltantes en Cada Columna:\n", missing_counts)
```

```
Valores Faltantes en Cada Columna:
 Unnamed: 0      0
Date             0
AveragePrice     0
Total Volume     0
4046             0
4225             0
4770             0
Total Bags       0
Small Bags       0
Large Bags       0
XLarge Bags      0
type             0
year             0
region           0
dtype: int64
```

Esto te dará una serie que indica el número total de valores faltantes por columna, lo cual es esencial para la limpieza de datos.

```
# Contar columnas con valores faltantes
columns_with_missing = missing_counts[missing_counts > 0].count()
print("\nNúmero de Columnas con Valores Faltantes:",
columns_with_missing)
```

```
Número de Columnas con Valores Faltantes: 0
```

Esto te permite conocer cuántas columnas contienen al menos un valor faltante, ayudándote a determinar si necesitas aplicar técnicas de imputación o eliminación de datos.

```
# Verificar si todas las columnas tienen valores faltantes
all_columns_missing = missing_counts.all()
print("¿Todas las Columnas Tienen Valores Faltantes?:",
all_columns_missing)
```

```
¿Todas las Columnas Tienen Valores Faltantes?: False
```

Este comando verifica si hay columnas que están completamente vacías, lo que puede ser un indicativo de que se deben eliminar.

```
# Calcular el número total de valores faltantes
total_missing_values = missing_counts.sum()
print("\nTotal de Valores Faltantes en el DataFrame:",
total_missing_values)


Total de Valores Faltantes en el DataFrame: 0
```

Esto te da una idea de la magnitud del problema de datos faltantes en el conjunto de datos.

Al finalizar esta sección, habrás obtenido un entendimiento más profundo de tus datos, habrás identificado valores faltantes y estarás mejor preparado para realizar un análisis más profundo.

Resumen: Puntos Clave Sobre Pandas

- **Pandas es una Biblioteca Poderosa:** Pandas es una biblioteca de Python versátil utilizada para trabajar con datos estructurados de manera eficiente, lo que facilita la manipulación y análisis de grandes volúmenes de datos.
- **Construido sobre NumPy:** Está construido sobre NumPy, lo que lo hace increíblemente rápido y eficiente para la manipulación de datos. Gracias a su estructura, permite realizar operaciones vectorizadas que optimizan el rendimiento.
- **Ampliamente Utilizado:** Pandas es ampliamente adoptado por otras bibliotecas de datos y herramientas, convirtiéndolo en una parte fundamental del ecosistema de datos en Python. Se usa frecuentemente en combinación con bibliotecas como Matplotlib y Seaborn para visualización.
- **Datos Tabulares:** Pandas sobresale en el manejo de datos tabulares, que consisten en filas y columnas. Los DataFrames de Pandas permiten representar datos en un formato fácil de usar y manipular.
- **Python y Pandas Juntos:** Al trabajar con datos, a menudo usas tanto Python como Pandas en conjunto para alcanzar tus objetivos, facilitando tareas complejas de análisis y transformación de datos.

Análisis Exploratorio de Datos (EDA)

- **Vista Previa de Datos:** Puedes previsualizar rápidamente tus datos usando métodos como `head()`, `tail()` y `sample()`. Estos métodos te permiten inspeccionar el principio, final o partes aleatorias de tu conjunto de datos, lo que ayuda a identificar patrones y problemas.
- **Visión General de Datos:** Los métodos `info()` y `describe()` proporcionan una visión general de tus datos, incluyendo tipos de datos, conteo de valores no nulos y estadísticas descriptivas, como media, mediana, mínimo y máximo. Esto es esencial para entender la distribución de tus datos.

- **Valores Únicos:** Usa `unique()` o `value_counts()` para encontrar valores únicos o contar ocurrencias en una columna. Esto es útil para comprender la diversidad de los datos en columnas categóricas.
- **Ordenamiento:** Puedes ordenar tus datos usando el método `sort_values()`, lo cual es útil para organizar datos por columnas específicas. La capacidad de ordenar puede ayudar a destacar tendencias o anomalías en los datos.
- **Subconjunto de Datos:** La selección de subconjuntos te permite extraer filas o columnas específicas. Usa corchetes (`[]`) para seleccionar columnas, y la indexación booleana para filas basadas en condiciones. Esto permite un análisis más focalizado.

Creando DataFrames

- **Desde Listas de Diccionarios:** Puedes crear DataFrames a partir de listas de diccionarios, donde cada diccionario representa una fila. Esta es una forma intuitiva de construir DataFrames, especialmente en análisis ad-hoc.
- **Desde Diccionarios con Listas:** Alternativamente, los DataFrames pueden ser creados a partir de diccionarios con listas como valores. Asegúrate de que las listas tengan la misma longitud para evitar errores.
- **Lectura de Datos:** Pandas proporciona funciones como `read_csv()` para leer datos de varias fuentes, como archivos CSV, URLs, bases de datos SQL, archivos Excel, y más. Puedes usar tanto rutas absolutas como relativas, e incluso cargar datos desde fuentes remotas como repositorios de GitHub. Esto permite una integración fluida de datos desde múltiples orígenes.

Estos son algunos de los conceptos y operaciones esenciales en Pandas, permitiéndote manipular y explorar tus datos de manera efectiva. Dominar estos elementos es clave para cualquier científico de datos o analista que trabaje con Python.

Hoja de Referencia de Pandas

```
# Muestra las primeras filas del DataFrame (por defecto, 5 filas)
df.head()

# Muestra las últimas filas del DataFrame (por defecto, 5 filas)
df.tail()

# Descripción estadística de las columnas numéricas
df.describe()

# Información general del DataFrame, incluyendo tipos de datos y
conteo de valores no nulos
df.info()

# Muestra los nombres de las columnas
df.columns
```

```python
# Muestra el índice del DataFrame
df.index

# Muestra los tipos de datos de cada columna
df.dtypes

# Crea un gráfico del DataFrame
df.plot()

# Crea un histograma de las columnas numéricas
df.hist()

# Cuenta los valores únicos de una columna específica
df.col.value_counts()

# Devuelve los valores únicos de una columna específica
df.col.unique()

# Copia el DataFrame
df.copy()

# Elimina columnas o filas; usa axis=0 para filas y axis=1 para
columnas
df.drop()

# Elimina las filas que contienen valores nulos
df.dropna()

# Rellena los valores nulos con un valor específico
df.fillna()

# Muestra las dimensiones del DataFrame (filas, columnas)
df.shape

# Selecciona solo las columnas numéricas
df._get_numeric_data()

# Renombra columnas del DataFrame
df.rename()

# Reemplaza valores en columnas de tipo cadena
df.str.replace()

# Cambia el tipo de datos de una o más columnas
df.astype(dtype='float32')

# Localiza filas y columnas usando índices
df.iloc[]

# Localiza filas y columnas usando etiquetas
df.loc[]
```

```
# Transpone el DataFrame (filas se convierten en columnas y viceversa)
df.transpose()
df.T

# Muestra una muestra aleatoria de filas del DataFrame
df.sample(n, frac)

# Calcula la suma de los valores en una columna
df.col.sum()

# Encuentra el valor máximo en una columna
df.col.max()

# Encuentra el valor mínimo en una columna
df.col.min()

# Selecciona una columna específica del DataFrame
df[col]

# Selecciona una columna usando la notación de punto
df.col

# Devuelve un DataFrame con los valores nulos
df.isnull()

# Devuelve un DataFrame con los valores nulos (alternativa)
df.isna()

# Devuelve un DataFrame con los valores que no son nulos
df.notna()

# Elimina filas duplicadas del DataFrame
df.drop_duplicates()

# Restablece el índice del DataFrame y lo sobreescribe
df.reset_index(inplace=True)
```

Más material

- Read the docs!
- Cheatsheet
- Exercises to practice
- More on merge, concat, and join. And even more!