Módulo 6.2: Intro a Pandas

Prof. Carlos Cedeño

¿Qué es Pandas y por qué es tan importante?

Pandas es una librería de código abierto para Python diseñada para trabajar con datos **estructurados** o **tabulares**. Su nombre proviene de "Panel Data" (Datos de Panel), un término de econometría para conjuntos de datos que incluyen observaciones a lo largo del tiempo para los mismos individuos.

¿Por qué la usamos?

- **Fácil de usar:** Tiene una sintaxis intuitiva y flexible.
- **Potente:** Permite manejar grandes volúmenes de datos de manera eficiente.
- **Versátil:** Puede leer y escribir datos en una gran variedad de formatos.

Las Estructuras de Datos Clave: Series y DataFrames

Pandas tiene dos estructuras de datos principales que deben conocer: `Series` y `DataFrame`.

1. Series

Una **Serie** es como una columna en una tabla. Es un arreglo unidimensional que puede contener cualquier tipo de dato (números, texto, fechas, etc.). Cada elemento tiene un **índice** asociado, que por defecto es numérico (0, 1, 2...).

Imagina una lista de frutas:

```
import pandas as pd

frutas = pd.Series(['Manzana', 'Banana', 'Cereza'])
print(frutas)
```

Salida:

```
0    Manzana
1     Banana
2     Cereza
dtype: object
```

2. DataFrame

El **DataFrame** es la estructura más importante y utilizada en Pandas. Es una tabla de dos dimensiones, similar a una hoja de cálculo, donde los datos están organizados en **filas** y **columnas**. ¡Cada columna de un DataFrame es una Serie!

Pensemos en un DataFrame como un diccionario de Series, donde las claves son los nombres de las columnas y los valores son las propias columnas (las Series).

```python
import pandas as pd

# Creamos un DataFrame desde un diccionario
datos = {
    'Nombre': ['Ana', 'Luis', 'Marta', 'Juan'],
    'Edad': [28, 34, 29, 42],
    'Ciudad': ['Quito', 'Guayaquil', 'Cuenca', 'Quito']
}

df_personas = pd.DataFrame(datos)
print(df_personas)
```

Salida:

```
  Nombre  Edad     Ciudad
0    Ana    28      Quito
1   Luis    34  Guayaquil
2  Marta    29     Cuenca
3   Juan    42      Quito
```

Leyendo y Escribiendo Archivos

Una de las tareas más comunes es cargar datos desde un archivo. Pandas hace esto increíblemente fácil.

Leer Datos desde Archivos

Pandas puede leer una multitud de formatos. Los más comunes son CSV, Excel y JSON.

Leer un Archivo CSV (`.csv`)

Los archivos CSV (Comma-Separated Values) son el pan de cada día en el mundo de los datos. Son simples archivos de texto donde los valores están separados por comas. Para leer un CSV, usamos la función `pd.read_csv()`.

```
# Suponiendo que tenemos un archivo llamado 'datos_ventas.csv'
# pd.read_csv('ruta/al/archivo/datos_ventas.csv')

# Ejemplo:
df_ventas = pd.read_csv('datos_ventas.csv')
print(df_ventas.head()) # .head() muestra las primeras 5 filas
```

Leer un Archivo de Excel (`.xlsx`)

Para leer archivos de Excel, usamos `pd.read_excel()`. Es importante saber que podrías necesitar instalar una librería adicional como `openpyxl`. Si no la tienes, puedes instalarla con: `pip install openpyxl`.

```
# Para leer una hoja específica de un archivo Excel
df_reporte = pd.read_excel('reporte_anual.xlsx', sheet_name='Ventas2024')
print(df_reporte.head())
```

Leer un Archivo JSON (`.json`)

JSON (JavaScript Object Notation) es otro formato muy popular, especialmente para datos provenientes de APIs web.

```
# Para leer un archivo JSON
df_usuarios = pd.read_json('usuarios.json')
print(df_usuarios.head())
```

Guardar Datos en Nuevos Archivos

Después de manipular tus datos (limpiar, transformar, analizar), querrás guardar los resultados. El proceso es igual de sencillo.

Guardar en CSV

Para guardar un DataFrame en un archivo CSV, usamos el método `.to_csv()`.

```
# Guardamos el DataFrame de personas en un nuevo archivo CSV
# El argumento index=False evita que Pandas guarde el índice del DataFrame como una columna
df_personas.to_csv('personas_guardado.csv', index=False)
```

Guardar en Excel

De manera similar, para guardar en Excel, usamos `.to_excel()`.

```
# Guardamos el DataFrame en un archivo Excel
df_personas.to_excel('personas_guardado.xlsx', index=False, sheet_name='DatosPersonas')
```

Explorando un DataFrame

Una vez que has cargado tus datos, los primeros pasos siempre son para conocerlos. Aquí tienes los comandos más útiles:

- `df.head(n)`: Muestra las primeras `n` filas (por defecto 5).
- `df.tail(n)`: Muestra las últimas `n` filas (por defecto 5).
- `df.shape`: Devuelve una tupla con el número de filas y columnas `(filas, columnas)`.

- `df.info()`: Proporciona un resumen técnico del DataFrame: el índice, las columnas, los tipos de datos de cada columna y el uso de memoria. ¡Esencial para detectar valores nulos!
- `df.describe()`: Calcula estadísticas descriptivas para las columnas numéricas (conteo, media, desviación estándar, mínimo, máximo, etc.).
- `df.columns`: Muestra los nombres de todas las columnas.
- `df['NombreColumna'].value_counts()`: Cuenta la frecuencia de cada valor único en una columna. Muy útil para columnas categóricas.

Ejemplo práctico:

```
print("Forma del DataFrame (filas, columnas):", df_personas.shape)
print("\nInformación del DataFrame:")
df_personas.info()
print("\nEstadísticas descriptivas:")
print(df_personas.describe())
print("\nConteo de personas por ciudad:")
print(df_personas['Ciudad'].value_counts())
```

Seleccionando y Filtrando Datos

Ahora viene lo divertido: ¡hacer preguntas a nuestros datos!

Selección de Columnas

Puedes seleccionar una o más columnas de un DataFrame.

```
# Seleccionar una sola columna (devuelve una Serie)
nombres = df_personas['Nombre']
print(nombres)

# Seleccionar múltiples columnas (devuelve un nuevo DataFrame)
# Nota el uso de dobles corchetes [[...]]
info_basica = df_personas[['Nombre', 'Ciudad']]
print(info_basica)
```

Selección de Filas: `loc` vs. `iloc`

Para seleccionar filas, Pandas nos da dos métodos principales que a menudo causan confusión al principio: `.loc` e `.iloc`. Entender su diferencia es clave.

La Diferencia Fundamental

- `.loc` (Location) selecciona datos por su **etiqueta** o **nombre de índice**. Es selección basada en *nombres*.
- `.iloc` (Integer Location) selecciona datos por su **posición numérica** (entera). Es selección basada en el *número de fila*, comenzando desde 0.

Pensemos en nuestro `df_personas`:

```
  Nombre  Edad     Ciudad
0    Ana    28      Quito     <- Etiqueta de fila: 0, Posición de fila: 0
1   Luis    34  Guayaquil     <- Etiqueta de fila: 1, Posición de fila: 1
2  Marta    29     Cuenca     <- Etiqueta de fila: 2, Posición de fila: 2
3   Juan    42      Quito     <- Etiqueta de fila: 3, Posición de fila: 3
```

En este caso, las etiquetas del índice coinciden con las posiciones (0, 1, 2, 3), pero esto no siempre es así.

Ejemplos Prácticos

Usemos la sintaxis `[fila, columna]` para ver la diferencia.

Usando `.loc` (basado en etiquetas):

```
# Seleccionar la fila con la etiqueta de índice 2
print("Fila con etiqueta 2 usando .loc:\n", df_personas.loc[2])

# Seleccionar la celda en la fila con etiqueta 1 y la columna 'Nombre'
print("\nCelda en fila 1, columna 'Nombre' usando .loc:", df_personas.loc[1, 'Nombre'])

# Slicing (rebanado) con .loc INCLUYE el último elemento
# Selecciona filas desde la etiqueta 1 HASTA la etiqueta 3
print("\nFilas de 1 a 3 (inclusive) con .loc:\n", df_personas.loc[1:3])
```

Usando `.iloc` (basado en posición numérica):

```
# Seleccionar la tercera fila (que está en la posición 2)
print("Tercera fila (posición 2) usando .iloc:\n", df_personas.iloc[2])

# Seleccionar la celda en la segunda fila (posición 1) y la tercera columna (posición 2)
print("\nCelda en posición [1, 2] usando .iloc:", df_personas.iloc[1, 2]) # Esto corresponde a la ciudad de Luis

# Slicing (rebanado) con .iloc EXCLUYE el último elemento (como en Python regular)
# Selecciona filas desde la posición 1 HASTA la posición 3 (sin incluir la 3)
print("\nFilas en posiciones 1 a 2 con .iloc:\n", df_personas.iloc[1:3])
```

Resumen Clave

Característica	`.loc`	`.iloc`
Tipo de Selección	Basada en **etiquetas** (nombres)	Basada en **posición** (números enteros)
Argumentos	`df.loc[etiqueta_fila, etiqueta_columna]`	`df.iloc[posicion_fila, posicion_columna]`
Slicing (Rebanado)	**Inclusivo** (incluye el final)	**Exclusivo** (excluye el final)
Cuándo usar	Cuando te importan los nombres/índices	Cuando te importa la posición numérica

Filtrado Condicional

Esta es una de las funcionalidades más poderosas. Permite seleccionar filas que cumplen una cierta condición. **Importante:** El filtrado condicional se realiza principalmente con `[]` o `.loc`, no con `.iloc`.

```
# Filtrar personas que tienen más de 30 años
mayores_de_30 = df_personas[df_personas['Edad'] > 30]
print(mayores_de_30)

# Filtrar personas que viven en Quito
# Usar .loc es considerado una buena práctica para el filtrado
personas_quito = df_personas.loc[df_personas['Ciudad'] == 'Quito']
print(personas_quito)
```

Puedes combinar múltiples condiciones usando `&` (y) y `|` (o). ¡No olvides los paréntesis!

```
# Personas de Quito Y que tienen más de 30 años
filtro_combinado = df_personas.loc[(df_personas['Ciudad'] == 'Quito') & (df_personas['Edad'] > 30)]
print(filtro_combinado)
```