Modelos de Inteligencia Artificial

CIPFP Mislata
Centre Integrat Públic Formació Professional Superior

:: U2 ::

Introducción al tratamiento tabular de datos

Curso 2025-26

Índice de la presentación

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

1. **Entornos de trabajo**
2. **Claves de programación en Python**
3. **Trabajo con datos tabulares**
   - **Fundamentos** (dataframes, series, lectura/escritura, …)
   - **Preparación** (operaciones, tipos, índices, …)
   - **Análisis** (agrupaciones, tablas pivote, …)

1. Entornos de trabajo

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

2. Claves de Python

Importar librerías

- Python tiene un vasto ecosistema de librerías.
- La palabra clave `**import**` se usa para cargar una librería.
- A menudo se les da un alias (un nombre más corto) usando `as` para facilitar su uso.

Ejemplo

```
# Importar la librería math
import math
print("El valor de Pi es:", math.pi)

# Importar una librería con un alias, como se hará con pandas
import numpy as np
mi_array = np.array([1, 2, 3])
print("Array de NumPy:", mi_array)
```

Salida esperada:

```
El valor de Pi es: 3.141592653589793
Array de NumPy: [1 2 3]
```

2. Claves de Python

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Variables y tipos de datos

- En Python, no necesitas declarar el tipo de una variable; se infiere automáticamente.
- Python maneja varios tipos de datos esenciales para el análisis.

Ejemplo

```
entero = 10
flotante = 3.14
cadena = "Hola, Python"
booleano = True

print(f"Entero: {entero}, Tipo: {type(entero)}")
```

Salida esperada:

```
Entero: 10, Tipo: <class 'int'>
```

2. Claves de Python

Listas

- Las **listas** permiten almacenar una colección ordenada de elementos, que pueden ser de diferentes tipos.
- Son mutables, lo que significa que puedes cambiar sus elementos después de crearlas.

Ejemplo

```
frutas = ['manzana', 'banana', 'cereza']
# Añadir un elemento al final
frutas.append('naranja')
print("Lista con naranja:", frutas)
```

Salida esperada:

```
Lista con naranja: ['manzana', 'banana', 'cereza', 'naranja']
```

2. Claves de Python

Diccionarios

- Los **diccionarios** son colecciones desordenadasde elementos, ideales para representar datos estructurados.
- Cada elemento es un par `clave: valor`.

Ejemplo

```
persona = {
    'nombre': 'Elena',
    'edad': 28,
    'ciudad': 'Valencia'
}
# Acceder a valores por clave
print("Nombre:", persona['nombre'])

# Añadir o actualizar un par clave-valor
persona['profesion'] = 'Desarrolladora'
print("Diccionario actualizado:", persona)
```

Salida esperada:

```
Nombre: Elena
Diccionario actualizado: {'nombre': 'Elena', 'edad': 28, 'ciudad': 'Valencia', 'profesion': 'Desarrolladora'}
```

2. Claves de Python

Bucles

- Los bucles `**for**` se utilizan para iterar sobre una secuencia (como una lista).
- Permiten ejecutar un bloque de código repetidamente para cada elemento.

Ejemplo

```
numeros = [1, 2, 3, 4, 5]
suma = 0

# Bucle for para sumar todos los números de la lista
for numero in numeros:
    suma = suma + numero

print("La suma es:", suma)
```

Salida esperada:

```
La suma es: 15
```

2. Claves de Python

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

BLOQUE 1

Primeros pasos

3. Trabajo con Pandas

Series

- Se puede crear una **Serie** a partir de una lista, un array de NumPy o un diccionario.
- Una **serie** tiene una disposición visual vertical.

Ejemplo

```
import pandas as pd

# Desde una lista
mi_serie = pd.Series([10, 20, 30, 40])
print(mi_serie)
```

Salida esperada:

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

... cada elemento de una serie es accesible a partir del índice (lista) / clave par (diccionario)

```
print("Elemento en el índice 1:", mi_serie[1])
```

Salida esperada:

```
Elemento en el índice 1: 20
```

3. Trabajo con Pandas

Dataframes

- Un **DataFrame** es una tabla bidimensional, similar a una hoja de cálculo o una tabla de SQL, donde los datos están organizados en filas y columnas.
- Cada columna en un DataFrame es una **Serie**.

Ejemplo

```
import pandas as pd

datos = {'Pais': ['España', 'Francia', 'Italia'],
         'Poblacion': [47, 65, 60]}
df = pd.DataFrame(datos)
print(df)
```

Salida esperada:

```
      Pais  Poblacion
0   España         47
1  Francia         65
2   Italia         60
```

3. Trabajo con Pandas

Dataframes

- Se puede inspeccionar rápidamente un DataFrame con atributos como **.shape** (dimensiones), **.columns** (nombres de columnas) y **.dtypes** (tipos de datos de cada columna).

Ejemplo

```
print("Dimensiones:", df.shape)
print("Columnas:", df.columns)
```

Salida esperada:

```
Dimensiones: (3, 2)
Columnas: Index(['Pais', 'Poblacion'], dtype='object')
```

3. Trabajo con Pandas

Leer un archivo CSV

- La función `**pd.read_csv()**` es la herramienta fundamental para cargar datos tabulares en un DataFrame.

Ejemplo

```
# Suponiendo que tienes un archivo llamado 'datos.csv'
# df = pd.read_csv('datos.csv')
# print(df.head()) # .head() muestra las primeras 5 filas
```

Salida esperada:

```
Esta celda requiere un archivo 'datos.csv' para funcionar. Mostraría las primeras filas del archivo.
```

3. Trabajo con Pandas

Escribir en un archivo CSV

- De manera similar, un DataFrame se puede guardar en un archivo CSV con el método `**.to_csv()**`. El argumento `**index=False**` evita que se guarde el índice del DataFrame en el archivo.

Ejemplo

```
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df.to_csv('nuevo_archivo.csv', index=False)
```

Salida esperada:

```
Se creará un archivo llamado 'nuevo_archivo.csv' en tu directorio de trabajo.
```

3. Trabajo con Pandas

Selección por etiqueta con .loc[]

- `.loc[]` se utiliza para seleccionar datos por los nombres de las filas y columnas.

Ejemplo

```
df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'b', 'c'], columns=['X', 'Y', 'Z']
print("Fila 'a':\n", df.loc['a'])
print("\nValor en ('b', 'Y'):", df.loc['b', 'Y'])
```

Salida esperada:

```
Fila 'a':
X   -0.573210
Y    0.147778
Z    1.523030
Name: a, dtype: float64

Valor en ('b', 'Y'): -0.234137
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Selección por posición con .iloc[]

- `.iloc[]` funciona con índices enteros, como si estuvieras trabajando con una lista o un array de NumPy.

Ejemplo

```
print("Primera fila (indice 0):\n", df.iloc[0])
print("\nÚltima fila (indice -1):\n", df.iloc[-1])
```

Salida esperada:

```
Primera fila (indice 0):
X   -0.573210
Y    0.147778
Z    1.523030
Name: a, dtype: float64

Última fila (indice -1):
X    0.469595
Y   -1.028919
Z   -0.485145
Name: c, dtype: float64
```

3. Trabajo con Pandas

Selección condicional (boolean indexing)

- Se pueden filtrar filas basándote en una condición, creando una máscara booleana.

Ejemplo

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
print(df[df['A'] > 1])
```

Salida esperada:

```
   A  B
1  2  5
2  3  6
```

3. Trabajo con Pandas

Selección condicional (boolean indexing)

- La condición pueda hacerse todo lo compleja que sea necesario, combinando diferente tipo de operadores.

Ejemplo

```
df = pd.DataFrame({'Producto': ['A', 'B', 'A', 'C'], 'Precio': [10, 25, 12, 5], 'Stock': [100, 50, 0, 200]})

# Productos con Precio > 10 Y Stock > 0
print("Precio > 10 Y Stock > 0:")
print(df[(df['Precio'] > 10) & (df['Stock'] > 0)])

# Productos de tipo 'A' O 'C'
print("\nProductos A o C:")
print(df[(df['Producto'] == 'A') | (df['Producto'] == 'C')])
```

Salida esperada:

```
Precio > 10 Y Stock > 0:
  Producto  Precio  Stock
1        B      25     50

Productos A o C:
  Producto  Precio  Stock
0        A      10    100
2        A      12      0
3        C       5    200
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

¡Manos a la obra!

Realiza el ejercicio de repaso del **bloque 1**.

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

BLOQUE 2

Transformar datos

3. Trabajo con Pandas

Estructuración mediante multíndices

De Tabla Plana a MultiIndex

Grupo	Letra	Col1	Col2
Grupo1	A	0.5	-0.2
Grupo1	B	1.1	0.8
Grupo2	A	-0.4	1.5
Grupo2	B	0.1	-0.9

→

Grupo	Letra	Col1	Col2
Grupo1	A	0.5	-0.2
	B	1.1	0.8
Grupo2	A	-0.4	1.5
	B	0.1	-0.9

Un MultiIndex agrupa los datos bajo índices externos ('Grupo1', 'Grupo2'), haciendo las selecciones y los análisis agrupados más directos.

3. Trabajo con Pandas

Crear un "multiindex"

Puedes crear un **multiindex** a partir de una lista de arrays o tuplas.

Ejemplo

```
import pandas as pd
import numpy as np

arrays = [['Grupo1', 'Grupo1', 'Grupo2', 'Grupo2'], ['A', 'B', 'A', 'B']]
multi_index = pd.MultiIndex.from_arrays(arrays, names=('Grupo', 'Letra'))

df = pd.DataFrame(np.random.randn(4, 2), index=multi_index, columns=['Col1', 'Col2'])
print(df)
```

Salida esperada:

```
                Col1      Col2
Grupo  Letra
Grupo1 A     -0.895318  0.386902
       B     -0.510805 -1.180632
Grupo2 A      0.733618  0.223143
       B     -0.569338 -1.259468
```

3. Trabajo con Pandas

Seleccionar datos en un "multiindex"

La selección se puede realizar "por niveles" del índice creado.

Ejemplo

```
print("\nSeleccionar Grupo1:\n", df.loc['Grupo1'])
```

Salida esperada:

```
Seleccionar Grupo1:
         Col1      Col2
Letra
A     -0.895318  0.386902
B     -0.510805 -1.180632
```

3. Trabajo con Pandas

Crear un multiíndice con ".set_index()"

El uso del método "**.set_index()**" permite seleccionar columnas que funcionarán como índices.

Ejemplo

```
data = {'Grupo': ['Grupo1', 'Grupo1', 'Grupo2', 'Grupo2'],
        'Letra': ['A', 'B', 'A', 'B'],
        'Col1': [0.5, 1.1, -0.4, 0.1],
        'Col2': [-0.2, 0.8, 1.5, -0.9]}
df_plano = pd.DataFrame(data)

# Convertimos las columnas 'Grupo' y 'Letra' en el MultiIndex
df_multi = df_plano.set_index(['Grupo', 'Letra'])
print(df_multi)
```

Salida esperada:

```
                Col1  Col2
Grupo  Letra
Grupo1 A        0.5  -0.2
       B        1.1   0.8
Grupo2 A       -0.4   1.5
       B        0.1  -0.9
```

3. Trabajo con Pandas

Operaciones y transformación

- El método `**.apply()**` recorre cada elemento de una Serie y aplica la operación, devolviendo una nueva Serie con los resultados.
- La "operación" vendrá definida por una función.

Ejemplo

```
def al_cuadrado(numero):
    return numero ** 2

df['B_al_cuadrado'] = df['B'].apply(al_cuadrado)
print(df)
```

Salida esperada:

```
   A   B  A_por_100  B_al_cuadrado
0  1  10        100            100
1  2  20        200            400
2  3  30        300            900
```

3. Trabajo con Pandas

Operaciones y transformación

- Para transformaciones sencillas puede bastar con usar una función `**lambda**`.

Ejemplo

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [10, 20, 30]})
df['A_por_100'] = df['A'].apply(lambda x: x * 100)
print(df)
```

Salida esperada:

```
   A   B  A_por_100
0  1  10        100
1  2  20        200
2  3  30        300
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Aplicar Funciones sobre Filas con `axis=1`

- Por defecto, `.apply()` opera sobre columnas. Sin embargo, al especificar `**axis=1**`, le indicamos a Pandas que aplique la función a cada fila.

- En este caso, la función lambda recibe la fila completa como un objeto Serie, lo que permite acceder a los valores de múltiples columnas a la vez para realizar cálculos complejos.

Ejemplo

```
df_ventas = pd.DataFrame({'Precio': [10, 20, 5], 'Cantidad': [2, 3, 4]})

# Calcular el total para cada fila
df_ventas['Total'] = df_ventas.apply(lambda row: row['Precio'] * row['Cantidad'], axis=1)

print(df_ventas)
```

Salida esperada:

```
   Precio  Cantidad  Total
0      10         2     20
1      20         3     60
2       5         4     20
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Operaciones y transformación

- El método `**.map()**` es útil para sustituir cada valor en una Serie por otro valor, basándose en un diccionario o una función.

Ejemplo

```
df_map = pd.DataFrame({'Letra': ['a', 'b', 'c']})
df_map['Vocal'] = df_map['Letra'].map({'a': 'Si', 'b': 'No', 'c': 'No'})
print(df_map)
```

Salida esperada:

```
  Letra Vocal
0     a    Si
1     b    No
2     c    No
```

3. Trabajo con Pandas

Manejo de tipos de datos

- Python infiere el tipo de una variable automáticamente.
- Python maneja varios tipos de datos esenciales para el análisis.
- El atributo `**.dtypes**` te da un resumen rápido de los tipos de datos de cada columna en tu DataFrame.

Ejemplo

```
import pandas as pd
data = {'Edad': ['25', '30', '22'], 'Salario': [50000.5, 60000.0, 45000.8]}
df = pd.DataFrame(data)
print(df.dtypes)
```

Salida esperada:

```
Edad       object
Salario    float64
dtype: object
```

3. Trabajo con Pandas

Manejo de tipos de datos

- El método `**.astype()**` es la forma más directa de convertir una columna a un tipo de dato específico.
- En el siguiente ejemplo la columna 'Edad' es de tipo 'object' (texto) y la convertimos a 'int'.

Ejemplo

```
df['Edad'] = df['Edad'].astype(int)
print(df.dtypes)
```

Salida esperada:

```
Edad         int64
Salario    float64
dtype: object
```

3. Trabajo con Pandas

Manejo de tipos de datos

- A veces, los datos numéricos contienen caracteres no válidos. `**pd.to_numeric()**` es más robusto que `**.astype()**`, ya que permite manejar errores, por ejemplo, convirtiendo valores no válidos en NaN.

Ejemplo

```
s = pd.Series(['1', '2.5', '3a', '4'])
numerico = pd.to_numeric(s, errors='coerce') # 'coerce' convierte errores en NaNs
print(numerico)
```

Salida esperada:

```
0    1.0
1    2.5
2    NaN
3    4.0
dtype: float64
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Manejo de tipos de datos

- La función `**pd.to_datetime()**` es esencial, ya que convierte cadenas de texto a un tipo de dato `datetime` especializado, permitiendo a Pandas realizar operaciones de series temporales.

Ejemplo

```
fechas_texto = pd.Series(['2023-01-15', '2023/02/20', 'Mar 25, 2023'])
fechas_datetime = pd.to_datetime(fechas_texto)
print(fechas_datetime)

# Ahora puedes acceder a propiedades como el día de la semana
print("\nDia de la semana (0=Lunes):\n", fechas_datetime.dt.dayofweek)
```

Salida esperada:

```
0   2023-01-15
1   2023-02-20
2   2023-03-25
dtype: datetime64[ns]

Día de la semana (0=Lunes):
0    6
1    0
2    5
dtype: int32
```

3. Trabajo con Pandas

Optimización y memoria

- El primer paso para optimizar es saber cuánta memoria está usando tu DataFrame. Puedes obtener un informe detallado con `**df.info()**`.

Ejemplo

```
import pandas as pd
import numpy as np
data = {'Pais': ['España', 'Francia', 'España', 'Italia', 'Francia'] * 10000,
        'Ventas': np.random.randint(100, 1000, 50000)}
df = pd.DataFrame(data)
df.info(memory_usage='deep')
```

Salida esperada:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Pais    50000 non-null  object
 1   Ventas  50000 non-null  int64
dtypes: int64(1), object(1)
memory usage: 3.2 MB
```

3. Trabajo con Pandas

Optimización y memoria

- Si una columna de texto (object) tiene pocos valores únicos (baja cardinalidad), convertirla al tipo **'category'** puede ahorrar una enorme cantidad de memoria. Pandas almacenará cada valor único una sola vez y usará punteros enteros.

Ejemplo

```
df['Pais'] = df['Pais'].astype('category')
df.info(memory_usage='deep')
```

Salida esperada:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Pais    50000 non-null  category
 1   Ventas  50000 non-null  int64
dtypes: category(1), int64(1)
memory usage: 440.0 KB
```

3. Trabajo con Pandas

Optimización y memoria

- Pandas a menudo usa tipos de datos más grandes de lo necesario (como int64). Si sabes que los valores de una columna caben en un tipo más pequeño (como int32 o int16), puedes hacer **'downcasting'** para ahorrar memoria.

Ejemplo

```
print("Tipo original:", df['Ventas'].dtype)
# Convertir a un entero más pequeño que aún puede contener los valores
df['Ventas'] = pd.to_numeric(df['Ventas'], downcast='integer')
print("Tipo nuevo:", df['Ventas'].dtype)
df.info(memory_usage='deep')
```

Salida esperada:

```
Tipo original: int64
Tipo nuevo: int16
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   País    50000 non-null  category
 1   Ventas  50000 non-null  int16
dtypes: category(1), int16(1)
memory usage: 147.0 KB
```

3. Trabajo con Pandas

Manipulación de texto

- Normalizar el texto es el primer paso. Los métodos `.str.**lower**()`, `.str.**upper**()` y `.str.**strip**()` (elimina espacios al inicio y final) son esenciales.

Ejemplo

```
import pandas as pd
s = pd.Series(['  Producto A  ', ' producto b', 'PRODUCTO C '])
print("Original:\n", s)
print("\nEn minúsculas y sin espacios:\n", s.str.lower().str.strip())
```

Salida esperada:

```
Original:
0      Producto A
1         producto b
2      PRODUCTO C
dtype: object

En minúsculas y sin espacios:
0    producto a
1    producto b
2    producto c
dtype: object
```

3. Trabajo con Pandas

Manipulación de texto

- Puedes buscar subcadenas con `.str.**contains**()` (útil para filtrar) y reemplazar texto con `.str.**replace**()`.

Ejemplo

```
df = pd.DataFrame({'SKU': ['SKU-001-A', 'SKU-002-B', 'INV-003-A']})
# Filtrar filas que contienen 'SKU'
print("Filas con 'SKU':\n", df[df['SKU'].str.contains('SKU')])

# Reemplazar 'SKU-' con 'ID_'
df['SKU'] = df['SKU'].str.replace('SKU-', 'ID_')
print("\nDataFrame con reemplazo:\n", df)
```

Salida esperada:

```
Filas con 'SKU':
          SKU
0  SKU-001-A
1  SKU-002-B

DataFrame con reemplazo:
          SKU
0   ID_001-A
1   ID_002-B
2  INV-003-A
```

3. Trabajo con Pandas

Manipulación de texto

- El método `.str.**split**()` es increíblemente útil para dividir una columna en varias.
- Usando `**expand=True**`, el resultado se lleva directamente en nuevas filas/columnas de un DataFrame. Si el valor es **False,** devuelve una lista por cada fila con los valores.

Ejemplo

```
df_nombres = pd.DataFrame({'NombreCompleto': ['García, Juan', 'Pérez, Ana', 'Sánchez, Luis']})
df_nombres[['Apellido', 'Nombre']] = df_nombres['NombreCompleto'].str.split(', ', expand=True)
print(df_nombres)
```

Salida esperada:

```
  NombreCompleto Apellido Nombre
0   García, Juan   García   Juan
1     Pérez, Ana    Pérez    Ana
2  Sánchez, Luis  Sánchez   Luis
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

¡Manos a la obra!

Realiza el ejercicio de repaso del **bloque 2**.

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

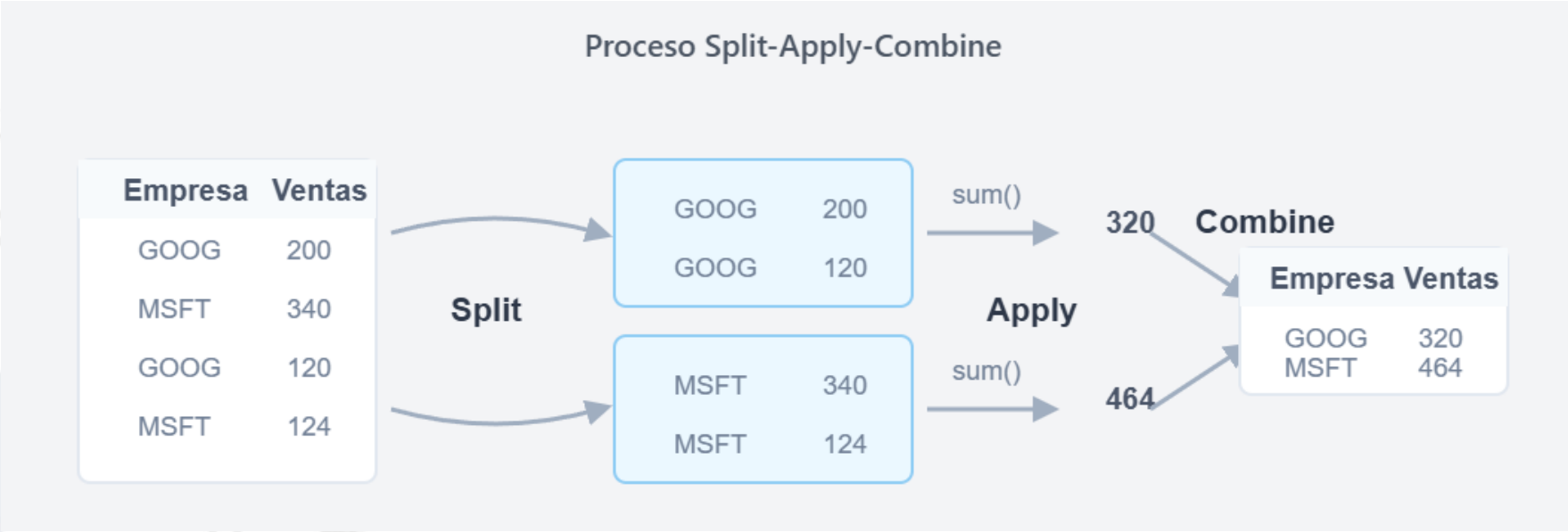
BLOQUE 3

Resumir datos

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Agrupaciones



3. Trabajo con Pandas

Agrupaciones

- Podemos **agrupar** por una columna y luego **aplicar** funciones de agregación.
- Las más comunes son: `**sum()**` (suma total de los valores), `**mean()**` (calcula el promedio), `**count()**` (cuenta el número de elementos no nulos), `**max()**` (valor máximo) y `**min()**` (valor mínimo).

Ejemplo

```
import pandas as pd
data = {'Empresa': ['GOOG', 'GOOG', 'MSFT', 'MSFT', 'FB', 'FB'],
        'Persona': ['Sam', 'Charlie', 'Amy', 'Vanessa', 'Carl', 'Sarah'],
        'Ventas': [200, 120, 340, 124, 243, 350]}
df = pd.DataFrame(data)

ventas_por_empresa = df.groupby('Empresa').sum()
print("Suma de ventas por empresa:\n", ventas_por_empresa)
```

Salida esperada:

```
Suma de ventas por empresa:
         Ventas
Empresa
FB          593
GOOG        320
MSFT        464
```

3. Trabajo con Pandas

Agrupaciones

- El método `**.agg()**` permite aplicar múltiples funciones de agregación a la vez, dándote un resumen muy completo.

Ejemplo

```
print("\nAgregación múltiple de ventas:\n", df.groupby('Empresa')['Ventas'].agg(['mean', 'std', 'count']))
```

Salida esperada:

```
Agregación múltiple de ventas:
            mean         std  count
Empresa
FB         296.5   75.660426      2
GOOG       160.0   56.568542      2
MSFT       232.0  152.735065      2
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Tablas "pivote"

De Formato "Largo" a Formato "Ancho"

Día	Ciudad	Temperatura
Lunes	Madrid	28
Martes	Madrid	30
Lunes	Barcelona	22
Martes	Barcelona	24

pivot →

Ciudad	Lunes	Martes
Barcelona	22	24
Madrid	28	30

3. Trabajo con Pandas

Crear una tabla "pivote"

- Usamos `pivot_table()` especificando los valores a agregar (`values`), el índice para las nuevas filas (`index`) y las columnas para las nuevas columnas (`columns`).

Ejemplo

```
import pandas as pd
data = {'Día': ['Lunes', 'Martes', 'Miércoles', 'Lunes', 'Martes', 'Miércoles'],
        'Ciudad': ['Madrid', 'Madrid', 'Madrid', 'Barcelona', 'Barcelona', 'Barcelona'],
        'Temperatura': [28, 30, 31, 22, 24, 25]}
df = pd.DataFrame(data)

pivot_df = df.pivot_table(values='Temperatura', index='Ciudad', columns='Día')
print(pivot_df)
```

Salida esperada:

```
Día        Lunes  Martes  Miércoles
Ciudad
Barcelona     22      24         25
Madrid        28      30         31
```

3. Trabajo con Pandas

Tabla "pivote" con diferente agregación

- Por defecto, `pivot_table` calcula la media. Podemos cambiar esto con el argumento `**aggfunc**`.

Ejemplo

```
df_ventas = pd.DataFrame({'Vendedor': ['Ana', 'Luis', 'Ana', 'Luis'], 'Producto': ['A', 'A', 'B', 'B'], 'Cantidad': [10, 5, 8, 12]})
# Calcular la suma total en lugar de la media
pivot_sum = df_ventas.pivot_table(values='Cantidad', index='Vendedor', columns='Producto', aggfunc='sum')
print(pivot_sum)
```

Salida esperada:

```
Producto    A   B
Vendedor
Ana        10   8
Luis        5  12
```

3. Trabajo con Pandas

Unir DataFrames

- `**concat()**` apila DataFrames uno encima del otro (eje 0) o uno al lado del otro (eje 1).

Ejemplo

```
import pandas as pd
df1 = pd.DataFrame({'A': ['A0', 'A1'], 'B': ['B0', 'B1']})
df2 = pd.DataFrame({'A': ['A2', 'A3'], 'B': ['B2', 'B3']})

concatenado = pd.concat([df1, df2], ignore_index=True) # ignore_index reinicia el índice
print("Concatenación con índice reseteado:\n", concatenado)
```

Salida esperada:

```
Concatenación con índice reseteado:
    A   B
0  A0  B0
1  A1  B1
2  A2  B2
3  A3  B3
```

3. Trabajo con Pandas

Fundir DataFrames

- `**merge()**` es la función de Pandas para realizar "fusiones" de dataframes al estilo de las bases de datos SQL.
- La opción "how" permite especificar el tipo de fusión que se aplicará ("inner", "outer", "left", "join", ...).

Ejemplo

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2'], 'A': ['A0', 'A1', 'A2']})
right = pd.DataFrame({'key': ['K0', 'K1', 'K3'], 'B': ['B0', 'B1', 'B3']})

outer_join = pd.merge(left, right, on='key', how='outer')
print("\nOuter Join:\n", outer_join)
```

Salida esperada:

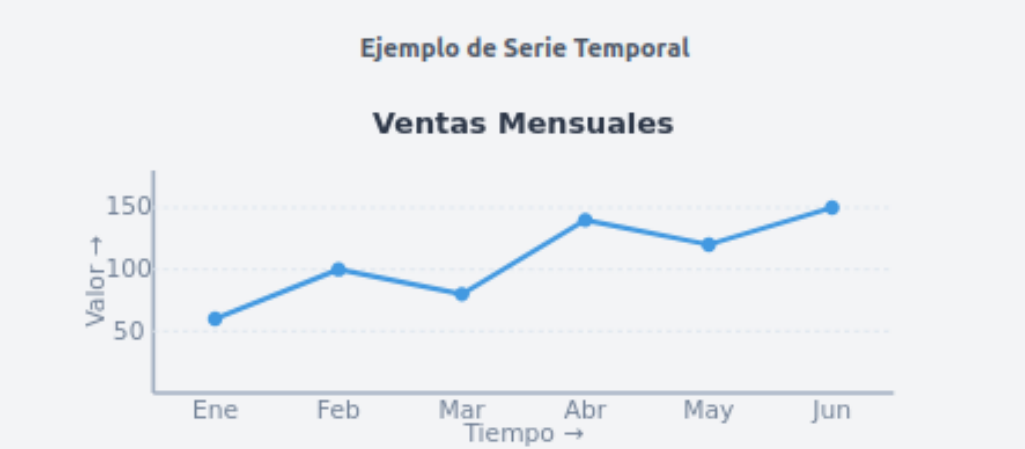
```
Outer Join:
  key    A    B
0  K0   A0   B0
1  K1   A1   B1
2  K2   A2  NaN
3  K3  NaN   B3
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Crear y muestrear Series Temporales

- Una serie temporal es un tipo de columna cuyo datos son fechas ordenadas cronológicamente.
- Su uso es muy común en el análisis de problemas financieros, meterológicos, ...



Un gráfico de serie temporal muestra cómo una variable (Valor) cambia a lo largo del tiempo.

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

Crear y muestrear Series Temporales

- Podemos crear un rango de fechas con `**date_range()**` y usarlo como índice. `**resample()**` permite cambiar la frecuencia de la serie temporal (por ejemplo, de diaria a mensual).

Ejemplo

```
import pandas as pd
import numpy as np
dates = pd.date_range('20230101', periods=100)
ts = pd.Series(np.random.randint(0, 500, len(dates)), index=dates)

# Remuestrear a frecuencia mensual y calcular la media
print("Media mensual:\n", ts.resample('M').mean())
```

	Fechas
2023-01-01	168
2023-01-02	285
2023-01-03	28
2023-01-04	348
2023-01-05	351
...	...
2023-04-06	141
2023-04-07	301
2023-04-08	265
2023-04-09	343
2023-04-10	212

100 rows × 1 columns

Salida esperada:

```
Media mensual:
2023-01-31    253.903226
2023-02-28    251.535714
2023-03-31    239.548387
2023-04-30    242.000000
Freq: M, dtype: float64
```

3. Trabajo con Pandas

Crear y muestrear Series Temporales

- El método `**shift()**` es útil para desplazar datos hacia adelante o hacia atrás en el tiempo, lo que es común para calcular diferencias o rendimientos.

Ejemplo

```
print("Serie original:\n", ts.head(3))
print("\nSerie desplazada 1 día:\n", ts.shift(1).head(3))
```

Salida esperada:

```
Serie original:
2023-01-01    [valor aleatorio]
2023-01-02    [valor aleatorio]
2023-01-03    [valor aleatorio]
Freq: D, dtype: int64

Serie desplazada 1 día:
2023-01-01              NaN
2023-01-02    [valor de 01-01]
2023-01-03    [valor de 01-02]
Freq: D, dtype: float64
```

3. Trabajo con Pandas

CIPFP Mislata
Centre Integrat Públic
Formació Professional Superior

¡Manos a la obra!

Realiza el ejercicio de repaso del **bloque 3**.