Para estudiantes de Educación Básica y Media
UNIVERSIDAD DE CHILE

Clase 5: Manejo de datos con Python

FM849 - Programación Científica para Proyectos de Inteligencia Artificial (IA)

7 de enero de 2026

Antes de pandas: diccionarios

El diccionario es una estructura de datos que almacena datos en forma `llave:valor`.

```
>>> country_capitals = {
...     "Germany": "Berlin",
...     "Canada": "Ottawa",
...     "England": "London",
... }
>>> country_capitals["Canada"] # Buscamos la capital de Canadá
```

¿Cómo trabajaremos los datos?

- La información puede almacenarse en diferentes formatos: **grafos**, **árboles**, **tablas**, **diccionarios**, entre otros.
- En este curso, nos enfocaremos en datos tabulares (tablas). Este es el formato más simple y utilizado.
- El paquete `pandas` ofrece muchas funciones y estructuras de datos para procesar este formato de datos.

Paquete pandas en Python

¿Por qué pandas?

pandas es una librería muy flexible que permite manejar y visualizar datos.

- Contiene estructuras y herramientas para manejar datos de manera conveniente.
- Es mucho mas flexible que otros programas (p. ej., Excel), por lo que nos permite hacer cosas más complicadas y útiles para aplicaciones de inteligencia artificial.
- Es capaz de manejar grandes conjuntos de datos, pero existen librerías más sofisticadas como dask o polars.

Empezando con pandas

Primero importamos el paquete para hacer uso de todas sus funciones.

```
>>> import pandas # Importamos el paquete
>>> data = pandas.read_csv("datos.csv") # Cargamos un CSV
```

Existe una manera mas conveniente de hacer esto.

```
>>> import pandas as pd
>>> data = pd.read_csv("datos.csv")
```

Así, evitamos escribir `pandas` en cada llamada a una función. También, comúnmente usaremos la librería `numpy` en conjunto con pandas.

```
>>> import numpy as np
```

Estructuras de datos en pandas

Usando diccionarios podemos crear nuevas estructuras de datos. Nos enfocaremos en las *series* (`pd.Series`) y *data frames* (`pd.DataFrame`).

Dim.	Nombre	Descripción
1D	`pd.Series`	Arreglo de elementos 1D de un sólo tipo.
2D	`pd.DataFrame`	Arreglo 2D con columnas de tipo `pd.Series`.

Tabla 1: Tipos de estructuras de datos en `pandas`.

Objetos de tipo `pd.Series`

Un objeto de tipo `pd.Series` es un arreglo 1-dimensional que contiene una secuencia de valores del mismo tipo. Este arreglo tiene asociado un arreglo de etiquetas, llamado *index*.

INDEX	DATA
0	A
1	B
2	C
3	D
4	E
5	F

Figura 1: Ejemplo de un objeto de tipo `pd.Series`.

Objetos de tipo `pd.Series`

- Podemos crear un objeto de tipo `pd.Series` usando un diccionario.

```
>>> dicc = {"a": 3, "b": 2, "c" :1}
>>> obj = pd.Series(dicc)
>>> obj
```

Imprime

```
a 3
b 2
c 1
dtype: int64
```

Objetos de tipo `pd.Series`

- Podemos crear un objeto de tipo `pd.Series` usando una lista.

```
>>> obj = pd.Series([4, 7, -5, 3])
>>> obj
```

Imprime

```
0  4
1  7
2 -5
3  3
dtype: int64
```

Objetos de tipo `pd.DataFrame`

Un objeto de tipo `pd.DataFrame` representa una tabla rectangular de datos. Esta contiene una colección de columnas, donde cada una puede ser de un tipo diferente.

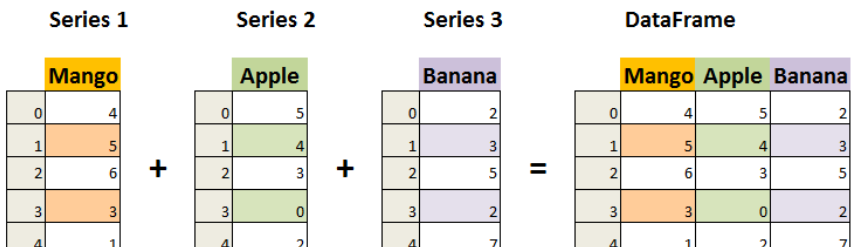


Figura 2: Ejemplo de un objeto de tipo `pd.DataFrame`.

Objetos de tipo `pd.DataFrame`

- Podemos crear un objeto de tipo `pd.DataFrame` usando un diccionario donde cada ítem representa una columna.

```
>>> data = {
...     "Name": ["Alice", "Bob", "Charlie"],
...     "Age": [25, 30, 35],
...     "City": ["Berlin", "Ottawa", "Londres"]
... }
>>> df = pd.DataFrame(data)
>>> df
```

	Name	Age	City
0	Alice	25	Berlin
1	Bob	30	Ottowa
2	Charlie	35	Londres

Básicos de pandas

Ahora veremos qué cosas básicas podemos hacer con un *DataFrame*. Volveremos a crear un pequeño conjunto de datos.

```
>>> data = {
...     "Name": ["Alice", "Bob", "Charlie", "Kim"],
...     "Age": [45, 50, 35, 20],
...     "City": ["Berlin", "Ottawa", "Londres", "Hong Kong"]
... }
>>> df = pd.DataFrame(data)
>>> df
```

	Name	Age	City
0	Alice	45	Berlin
1	Bob	50	Ottowa
2	Charlie	35	Londres
3	Kim	20	Hong Kong

Método `head`

- `df.head()` para mirar las primeras filas. Por defecto, se miran 5, pero podría usarse otro número como argumento (p. ej., `df.head(2)`).

	Name	Age	City
0	Alice	45	Berlin
1	Bob	50	Ottawa
2	Charlie	35	London
3	Kim	20	Hong Kong

`df.head(2)` →

	Name	Age	City
0	Alice	45	Berlin
1	Bob	50	Ottawa

Metodo `tail`

- `df.tail()` para mirar las últimas filas. Por defecto, se miran 5, pero podría usarse otro número como argumento (p. ej., `df.tail(2)`).

	Name	Age	City
0	Alice	45	Berlin
1	Bob	50	Ottawa
2	Charlie	35	London
3	Kim	20	Hong Kong

`df.tail(2)` →

	Name	Age	City
2	Charlie	35	London
3	Kim	20	Hong Kong

Atributo `shape`

- `df.shape` devuelve la forma del *DataFrame* en el formato (filas, columnas).

	Name	Age	City
0	Alice	45	Berlin
1	Bob	50	Ottawa
2	Charlie	35	London
3	Kim	20	Hong Kong

`df.shape` → **(4,3)**

Función `len`

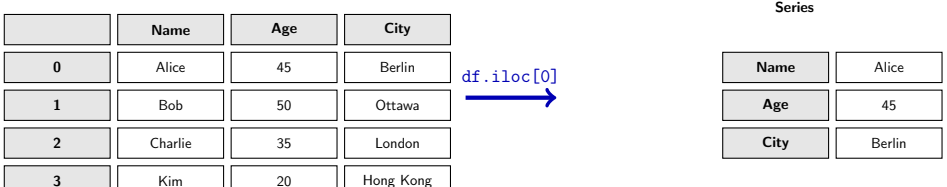
- `len(df)` devuelve el largo del *DataFrame* medido en número de filas.

	Name	Age	City
0	Alice	45	Berlin
1	Bob	50	Ottawa
2	Charlie	35	London
3	Kim	20	Hong Kong

4

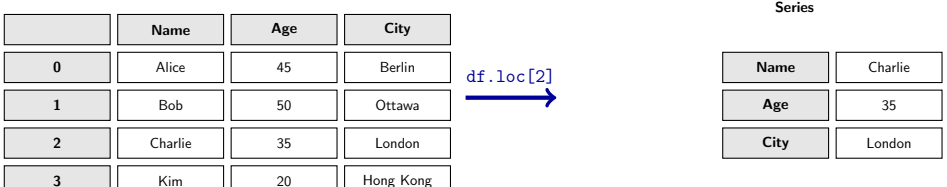
Método `iloc`

- `df.iloc(k)` devuelve un objeto del tipo `pd.Series` generado con la fila en la posición k. Si lo viéramos como una lista, acá lo que importa es la posición entera (que ya sabemos, parte de 0).



Método `loc`

- `df.loc(k)` devuelve un objeto del tipo `pd.Series` generado con la fila indexada por el valor k. Aquí lo que importa es el valor del índice del *DataFrame* (que puede ser cualquier cosa, no necesariamente números enteros consecutivos).



Leer una columna específica

En Python, podemos acceder a los atributos de objetos usando `[]` después del objeto.

- `df[col]` devuelve un objeto de tipo `pd.Series` generado con la columna `col`.
- También podemos usar `df[["col1", "col2"]]` para obtener varias columnas a la vez (en este caso, `col1` y `col2`).

	Name	Age	City
0	Alice	45	Berlin
1	Bob	50	Ottawa
2	Charlie	35	London
3	Kim	20	Hong Kong

`df['Age']` →

Series

0	45
1	50
2	35
3	20

Resumen de *DataFrames* en `pandas`

- `df.read_csv()` para cargar un conjunto de datos desde un archivo `.csv`.
- `df.head()` para mirar las primeras filas.
- `df.tail()` para mirar las últimas filas.
- `df.shape` para obtener las dimensiones de la tabla.
- `len(df)` para obtener el número de filas en la tabla.
- `df.columns` para obtener los nombres de las columnas en la tabla.
- `df.iloc[k]` para obtener la fila ubicada en la posición entera k.
- `df.loc[index]` para obtener la fila asignada al indice `index`.
- `df.to_csv()` para guardar el *DataFrame* en un archivo `.csv`.

Sin embargo, ¡`pandas` tiene más funciones! Veremos esto mañana....

DataFrames a partir de conjuntos de datos reales

En los ejemplos anteriores, generamos *series* y *DataFrames* usando datos creados de manera manual. Sin embargo, en proyectos reales debemos cargar datos desde archivos o páginas web. Esto se puede realizar con las siguientes funciones (Tabla 2).

Función	**Uso**	**Ejemplo**
`read_csv`	Cargar datos delimitados por comas desde un archivo local o una URL	`pd.read_csv("datos.csv")`
`read_excel`	Cargar datos tabulares desde un archivo Excel	`pd.read_excel("datos.xlsx")`
`read_pickle`	Cargar datos serializados en formato pickle	`pd.read_pickle("datos.pkl")`
`read_json`	Cargar datos desde archivos JSON	`pd.read_json("datos.json")`

Tabla 2: Funciones de `pandas` para carga de datos.

¿Cómo guardar datos con pandas?

Lo último que vamos a ver es cómo guardar un objeto de tipo `pd.DataFrame`. Podemos guardarlo usando las siguientes funciones (Tabla 3).

Función	Uso	Ejemplo
`to_csv`	Guardar un *DataFrame* en un archivo delimitado por comas (CSV)	`df.to_csv("datos.csv")`
`to_excel`	Guardar un *DataFrame* en un archivo Excel	`df.to_excel("datos.xlsx")`
`to_pickle`	Serializar y guardar un *DataFrame* en formato pickle	`df.to_pickle("datos.pkl")`
`to_json`	Guardar un *DataFrame* en un archivo JSON	`df.to_json("datos.json")`

Tabla 3: Funciones de pandas para guardar datos.

Ejemplo práctico

Exploraremos un conjunto de datos que tiene información sobre Pokémon en Google Colab, para aplicar lo aprendido en esta clase.