Introducción a Python para ML

Día 2: Estructuras de Datos e Introducción a pandas

EAE Business School Barcelona

3 de febrero de 2026

Plan del Día

Primera Parte (9:00-11:00)

1. Repaso rápido del Día 1
2. Estructuras de datos: listas y diccionarios
3. Introducción a pandas

Segunda Parte (11:30-13:30)

1. Inspección de datos con pandas
2. Selección e indexación de datos
3. Práctica con datos de viviendas Barcelona

Repaso Rápido: Día 1

¿Qué vimos ayer?

- **Variables y tipos básicos**: `int`, `float`, `str`, `bool`
- **Operadores**: aritméticos, comparación, lógicos
- **Control de flujo**: `if` / `else`, `for`, `while`
- **Funciones**: `def`, parámetros, `return`
- **Strings**: f-strings para formateo

¿Por qué Hoy es Importante?

Hasta ahora: **Python básico** (variables, bucles, funciones)

Hoy: **Trabajar con datos reales**

- Las listas y diccionarios son fundamentales para organizar información
- Pandas es la librería #1 para análisis de datos en Python
- Lo que aprendáis hoy usará **todos los días** en análisis de datos
- Hoy es el puente entre "saber programar" y "trabajar con datos"

Estructuras de Datos en Python

Listas: Colecciones Ordenadas

Una **lista** es una secuencia ordenada de elementos.

```
# Crear una lista
precios = [250000, 320000, 180000, 450000]

# Acceder a elementos (índice empieza en 0)
primer_precio = precios[0]  # 250000
ultimo_precio = precios[-1]  # 450000

# Longitud de la lista
num_precios = len(precios)  # 4
```

Operaciones con Listas: Añadir y Eliminar

```
ciudades = ["Barcelona", "Madrid", "Valencia"]

# Añadir elementos
ciudades.append("Sevilla")  # Al final
ciudades.insert(0, "Bilbao")  # En posición específica

# Eliminar elementos
ciudades.remove("Madrid")  # Por valor
elemento = ciudades.pop()  # Último elemento (y lo devuelve)
```

```python
# Verificar si existe
existe = "Barcelona" in ciudades  # True o False
```

Operaciones con Listas: Ordenar

```
ciudades = ["Barcelona", "Madrid", "Valencia"]

# Ordenar
ciudades.sort()  # Modifica la lista original
ciudades_ordenadas = sorted(ciudades)  # Devuelve nueva lista
```

Slicing: Obtener Sublistas

Slicing permite extraer partes de una lista.

```
precios = [250000, 320000, 180000, 450000, 290000, 410000]

# Sintaxis: lista[inicio:fin:paso]
primeros_tres = precios[0:3]  # [250000, 320000, 180000]
primeros_tres = precios[:3]   # Equivalente (inicio por defecto = 0)

ultimos_dos = precios[-2:]    # [290000, 410000]
todos_menos_primero = precios[1:]  # [320000, 180000, ...]
```

```python
# Con paso
cada_dos = precios[::2]  # [250000, 180000, 290000]
invertida = precios[::-1]  # Lista al revés
```

Cuidado: El índice final NO se incluye → `precios[0:3]` devuelve elementos 0, 1, 2

Truco útil: `[::-1]` invierte la lista

Slicing de listas funciona igual que `range(start, stop, step)`

Listas de Listas (Matrices)

Podéis anidar listas para representar datos tabulares:

```
# Cada fila es una propiedad: [barrio, tipo, precio]
propiedades = [
    ["Eixample", "Piso", 350000],
    ["Gràcia", "Ático", 420000],
    ["Sants", "Piso", 280000]
]
```

```python
# Acceder a elementos
primera_propiedad = propiedades[0]  # ["Eixample", "Piso", 350000]
barrio = propiedades[0][0]  # "Eixample"
precio = propiedades[1][2]  # 420000

# Iterar
for propiedad in propiedades:
    barrio, tipo, precio = propiedad  # Desempaquetar
    print(f"{tipo} en {barrio}: {precio}€")
```

Nota: Las listas de listas son una forma simple de representar tablas, pero pandas lo hace mucho mejor.

Diccionarios: Pares Clave-Valor

Un **diccionario** almacena pares clave-valor (como un JSON).

```
# Crear diccionario
propiedad = {
    "barrio": "Eixample",
    "tipo": "Piso",
    "precio": 350000,
    "habitaciones": 3,
    "metros": 85
}
```

```python
# Acceder a valores
barrio = propiedad["barrio"]  # "Eixample"
precio = propiedad.get("precio")  # 350000 (método seguro)
precio_deposito = propiedad.get("deposito", 0)  # 0 (valor por defecto)

# Modificar o añadir
propiedad["precio"] = 340000  # Modificar
propiedad["terraza"] = True   # Añadir nueva clave
```

Características: Claves únicas, muy rápidos para búsquedas

Nota: `get()` es más seguro que `[]` porque no da error si falta la clave

Operaciones con Diccionarios

```
propiedad = {"barrio": "Gràcia", "precio": 320000, "metros": 75}

# Obtener claves, valores, pares
claves = propiedad.keys()  # dict_keys(['barrio', 'precio', 'metros'])
valores = propiedad.values()  # dict_values(['Gràcia', 320000, 75])
pares = propiedad.items()  # dict_items([('barrio', 'Gràcia'), ...])

# Verificar si existe una clave
tiene_precio = "precio" in propiedad  # True
```

```python
# Iterar
for clave, valor in propiedad.items():
    print(f"{clave}: {valor}")

# Eliminar clave
del propiedad["metros"]
valor_eliminado = propiedad.pop("precio")  # Devuelve 320000
```

Listas vs Diccionarios

Característica	Lista	Diccionario
Acceso	Por índice (posición)	Por clave (nombre)
Orden	Sí, importante	Sí (desde Python 3.7+)
Uso típico	Secuencias, colecciones	Registros, configuraciones
Ejemplo	`precios[0]`	`propiedad["precio"]`

Cuándo usar cada una:

- **Lista**: Secuencia de elementos similares (lista de precios, nombres)
- **Diccionario**: Datos estructurados con campos nombrados (un registro)
- En la práctica, combinaréis ambas (lista de diccionarios)

Listas de Diccionarios (Común)

Estructura muy frecuente en análisis de datos:

```
# Lista de propiedades (cada propiedad = diccionario)
propiedades = [
    {"barrio": "Eixample", "precio": 350000, "metros": 85},
    {"barrio": "Gràcia", "precio": 420000, "metros": 95},
    {"barrio": "Sants", "precio": 280000, "metros": 70}
]
```

```python
# Calcular precio medio
precios = [prop["precio"] for prop in propiedades]
precio_medio = sum(precios) / len(precios)

# Filtrar por condición
baratas = [prop for prop in propiedades if prop["precio"] < 300000]
```

Esto es exactamente lo que hace pandas internamente → pero con mucha más potencia

Introducción a pandas

¿Qué es pandas?

pandas = Librería de Python para análisis de datos

Pensad en pandas como **"Excel con superpoderes"**:

- Tablas de datos (DataFrames)
- Operaciones rápidas sobre millones de filas
- Funciones estadísticas integradas
- Filtrado, agrupamiento, unión de datos
- Lectura/escritura de CSV, Excel, SQL, etc.

```python
import pandas as pd  # Convención estándar
```

Nombre: "Panel Data" (datos de panel en economía)

Creador: Wes McKinney (2008) → ahora es estándar de facto

El DataFrame: Concepto Central

	barrio	tipo	precio	metros
0	Eixample	Piso	350000	85
1	Gràcia	Ático	420000	95
2	Sants	Piso	280000	70

Componentes:

- **Filas**: Registros individuales (cada propiedad)
- **Columnas**: Características (barrio, precio, etc.)
- **Índice**: Etiquetas de fila (por defecto: 0, 1, 2, ...)

```python
# Crear DataFrame desde diccionario
data = {
    "barrio": ["Eixample", "Gràcia", "Sants"],
    "tipo": ["Piso", "Ático", "Piso"],
    "precio": [350000, 420000, 280000],
    "metros": [85, 95, 70]
}
df = pd.DataFrame(data)
```

Nota: El índice no es una columna - es metadata de las filas

Cargar Datos desde CSV

La forma más común de obtener datos:

```
import pandas as pd

# Cargar CSV desde URL (funciona en Google Colab)
url = "https://raw.githubusercontent.com/.../Houses_Barcelona_samp.csv"
df = pd.read_csv(url)

# O desde archivo local (si lo tenéis descargado)
# df = pd.read_csv("Houses_Barcelona_samp.csv")
```

`pd.read_csv()` es muy potente:

- Detecta automáticamente separadores y tipos
- Maneja miles de filas en segundos
- Opciones para parsear fechas, manejar valores faltantes, etc.
- pandas soporta muchos formatos: CSV, Excel, JSON, SQL, Parquet...

🔬 Ahora al notebook

Ejercicio Listas y Diccionarios (15 minutos)

Trabajando con DataFrames

Inspeccionar Datos: Primeros Vistazo

Siempre que cargáis datos, **inspeccionadlos primero**:

```
# Primeras y últimas filas
df.head()
df.tail()
df.head(10)  # Primeras N filas
```

Recordad: NUNCA asumir la estructura de los datos sin mirar primero

Inspeccionar Datos: Forma y Columnas

```
# Forma del DataFrame (filas, columnas)
df.shape  # (1000, 20) → 1000 filas, 20 columnas

# Nombres de columnas
df.columns
```

Información General: info()

`info()` os da un resumen completo del DataFrame:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 20 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id              1000 non-null   int64
 1   reference       1000 non-null   int64
 2   neighborhood    1000 non-null   object
 3   price           850 non-null    float64
 4   rooms           950 non-null    float64
...
```

Estadísticas Descriptivas: describe()

`describe()` calcula estadísticas para columnas numéricas:

```
df.describe()
```

	price	rooms	sqrmts
count	850	950	920
mean	315000	2.8	82.5
std	95000	1.1	28.3
min	93000	1.0	33.0
25%	250000	2.0	65.0
50%	295000	3.0	78.0
75%	380000	3.0	95.0
max	895000	50.0	5000.0

Nota: `describe()` es muy útil para detectar outliers y valores raros

Acceso a Columnas

```
# Una columna (devuelve una Serie)
precios = df["price"]
barrios = df["neighborhood"]

# Múltiples columnas (devuelve DataFrame)
subset = df[["neighborhood", "price", "rooms"]]

# Atributo (solo funciona si el nombre es válido como variable)
precios = df.price  # Equivalente a df["price"]
```

Serie vs DataFrame:

- `df["price"]` → Serie (una dimensión, como una columna de Excel)
- `df[["price"]]` → DataFrame (dos dimensiones, tabla con 1 columna)

Nota: Preferir `df["price"]` si el nombre tiene espacios o caracteres especiales

Filtrado de Filas: Condiciones Booleanas

Seleccionar filas que cumplen una condición:

```
# Propiedades caras (precio > 400000)
caras = df[df["price"] > 400000]

# Propiedades en Eixample
eixample = df[df["neighborhood"] == "Eixample"]

# Múltiples condiciones: & (and), | (or)
caras_eixample = df[(df["price"] > 400000) & (df["neighborhood"] == "Eixampl
baratas_o_gracia = df[(df["price"] < 200000) | (df["neighborhood"] == "Gràci
```

Importante: Usar `&` y `|` (no `and` / `or`). Paréntesis obligatorios.

Paso a paso:

1. `mask = df["price"] > 400000` → Serie de True/False
2. `df[mask]` → Filtra filas donde True

Indexación: loc (por etiquetas)

loc: Acceso por etiquetas (nombres de filas/columnas)

```
# loc: [filas, columnas] por etiqueta
df.loc[0, "price"]  # Fila con índice 0, columna "price"
df.loc[0:5, ["neighborhood", "price"]]  # Filas 0-5, columnas específicas
df.loc[df["price"] > 400000, "neighborhood"]  # Barrios de propiedades caras
```

Nota: `loc` incluye el extremo final del slice

Indexación: iloc (por posición)

iloc: Acceso por posición (índices numéricos)

```
# iloc: [filas, columnas] por posición
df.iloc[0, 3]  # Primera fila, cuarta columna
df.iloc[0:5, 0:3]  # Primeras 5 filas, primeras 3 columnas
df.iloc[:, -1]  # Todas las filas, última columna
```

Nota: `iloc` NO incluye el extremo final (como slicing de listas)

Preferir `loc` cuando sea posible (más legible con nombres de columnas)

Resumen: Flujo de Trabajo Básico

Cuando cargáis un dataset nuevo:

1. **Cargar**: `df = pd.read_csv(url)`
2. **Inspeccionar**:
   - `df.head()` → primeras filas
   - `df.info()` → tipos y valores faltantes
   - `df.describe()` → estadísticas descriptivas

3. **Explorar**:
   - Acceder a columnas: `df["columna"]`
   - Filtrar filas: `df[df["columna"] > valor]`
   - Seleccionar subconjuntos: `df.loc[...]`, `df.iloc[...]`

Próximos pasos (Día 3): Limpiar datos, transformar, visualizar

Ahora al notebook

Ejercicio pandas: Inspección y Filtrado (45 minutos)

Recursos Adicionales

Documentación oficial pandas:

- pandas.pydata.org

Tutoriales recomendados:

- "10 minutes to pandas" (official quickstart)
- Kaggle Learn: Pandas course

Gracias