Introducción a Python

Fundamentos para la Minería de Datos

BLOQUE 1: Fundamentos del Entorno Python

Subbloque 1: Versiones y Estructuras de Instalación

- Python no es una entidad única: es un lenguaje, un ejecutable, bibliotecas, carpetas y un entorno de ejecución.
- Comprender cómo se instala, coexiste y opera es crucial antes de escribir código.
- Este bloque aborda las condiciones de ejecución: versión invocada, directorio, búsqueda de paquetes y dependencias.

Coexistencia y Compatibilidad

- **Múltiples Versiones:** Python 2 (obsoleto pero presente) y Python 3 (ramas 3.x: 3.6, 3.9, 3.11, etc.) pueden coexistir.
- **Compatibilidad:**
    - No garantizada entre versiones mayores (2 vs 3).
    - Puede romperse entre versiones menores (3.x vs 3.y).
- **Impacto:** El mismo código puede funcionar en un entorno y fallar en otro debido a diferencias sutiles en las versiones de Python o de las bibliotecas instaladas.

Ejemplo: Ruptura de Compatibilidad (`pandas`)

Código válido en Python 3.6 / pandas antiguo:

```
import pandas as pd
# Permitía ignorar filas erróneas
df = pd.read_csv("archivo.csv", error_bad_lines=False)
```

En Python >= 3.10 / pandas >= 1.3.0, el argumento `error_bad_lines` fue eliminado:

```
# TypeError: read_csv() got an unexpected keyword argument 'error_bad_lines'
df = pd.read_csv("archivo.csv", error_bad_lines=False) # ¡Error!
```

Diagnóstico:

- Identificar el tipo de error (`TypeError`).
- Verificar versiones activas (`pip list`).
- Consultar *changelogs* de la biblioteca (pandas en este caso).
  El error reside en el **entorno de ejecución**, no solo en el código.

Métodos de Instalación de Python

Cada método genera estructuras y comportamientos distintos:

1. Instalación del Sistema (Linux/macOS):

- Preinstalada (`apt` , `yum`). Ubicada en `/usr/bin/python3` .
- Estable pero difícil de actualizar o tener múltiples versiones. Afecta a todo el sistema.

2. pyenv:

- Para desarrolladores. Instala y gestiona múltiples versiones aisladas (`~/.pyenv/versions/X.Y.Z/`).
- Activación por proyecto (`.python-version`). Requiere compilación y dependencias del sistema.

3. Miniconda / Anaconda:

- Distribuciones para ciencia de datos. Instalan Python por entorno (`~/miniconda3/envs/<nombre_env>/bin/python`).
- Robustas para paquetes científicos (NumPy, TensorFlow). Gestiona dependencias complejas (C/C++). Anaconda es más pesada.

Ubicación de Ejecutables y su Impacto

La ruta del binario `python` determina el entorno activo. Diferentes comandos pueden invocar distintas versiones:

```
$ which python
/usr/bin/python # Podría ser Python 2 o 3 del sistema

$ which python3
/home/usuario/.pyenv/shims/python3 # Gestionado por pyenv

$ which python
/home/usuario/miniconda3/envs/dev/bin/python # De un entorno Conda
```

Verificación desde código: Es fundamental saber qué Python se está usando, especialmente en entornos compartidos o complejos.

```
import sys
print("Ruta del ejecutable:", sys.executable)
print("Versión de Python:", sys.version)
```

Variables de Entorno y Orden de Resolución

El sistema y Python usan variables para encontrar ejecutables y módulos:

- `PATH`: Lista de directorios donde el sistema busca ejecutables (e.g., `python`).
- `PYTHONPATH`: Lista de directorios *adicionales* donde Python busca módulos (`import`).
- `sys.path`: La lista *efectiva* de rutas que Python usa en tiempo real para buscar módulos.

```
import sys
print("sys.path (orden de búsqueda de módulos):")
for p in sys.path:
    print(f" → {p}")
# sys.path.insert(0, "/ruta/a/mis/modulos") # Para añadir rutas
```

Errores comunes por resolución incorrecta: Importar versión equivocada, importar archivo local en lugar de librería estándar, no encontrar módulo fuera de `sys.path`.

Detección de Errores por Entorno Mal Configurado

Error Típico	Causa Probable	Solución Sugerida
`ModuleNotFoundError: No module named 'pandas'`	Ejecutable de Python incorrecto o entorno no activado.	Verificar `sys.executable`, activar entorno (`source venv/bin/activate` o `conda activate <env>`).
`TypeError: función() got an unexpected keyword argument 'arg_inesperado'`	Diferencias entre versiones de la librería usada.	Revisar *changelog* de la librería, verificar versión activa (`pip list`), ajustar código o versión.
`python` invoca versión 2.x en lugar de 3.x	`PATH` prioriza `/usr/bin/python` sobre `python3`.	Usar explícitamente `python3`, o corregir alias/ `PATH` en la configuración del shell.
Script funciona en máquina A pero falla en B	Diferencias sutiles en versiones de Python o librerías.	Usar entornos virtuales (`venv`, `conda`), congelar dependencias (`pip freeze > requirements.txt`), asegurar reproducibilidad.

BLOQUE 1: Fundamentos del Entorno Python

Subbloque 2: venv vs conda

- **Necesidad:** Aislar dependencias por proyecto para evitar conflictos y asegurar reproducibilidad.
- **Entorno Virtual:** Copia aislada del ejecutable de Python con su propio conjunto de paquetes (`site-packages`).
- **Beneficios:**
    - Permite diferentes versiones de librerías por proyecto.
    - Evita interferencias con el Python del sistema.
    - Facilita compartir y desplegar el entorno exacto.
- **Estrategias Principales:** `venv` (estándar) y `conda` (ciencia de datos).

venv: Estructura y Lógica

- **Inclusión:** Parte de la biblioteca estándar de Python (>= 3.3). No requiere instalación adicional.
- **Creación:** `python -m venv nombre_entorno` (e.g., `venv`)
- **Activación:** `source nombre_entorno/bin/activate` (Linux/macOS) o `nombre_entorno\Scripts\activate` (Windows)
- **Desactivación:** `deactivate`
- **Estructura Típica (Dentro del proyecto):**

```
mi_proyecto/
└── venv/
    ├── bin/      # Ejecutables (python, pip, activate)
    ├── lib/      # Bibliotecas instaladas (site-packages)
    └── include/  # Headers C (si se compilan extensiones)
```

- **Características:** Ligero (5-10 MB inicial), acoplado al proyecto (carpeta `venv/` local), ideal para scripts y proyectos simples. **No incluir `venv/` en Git.**

conda: Estructura y Lógica

- **Origen:** Gestor de paquetes y entornos (Miniconda/Anaconda).
- **Creación:** `conda create -n nombre_entorno python=3.x`
- **Activación:** `conda activate nombre_entorno`
- **Desactivación:** `conda deactivate`
- **Estructura Típica (Global, fuera del proyecto):**

```
~/miniconda3/envs/
└── nombre_entorno/
    ├── bin/
    ├── lib/
    └── ... (otros directorios de Conda)
```

- **Características:** Más pesado (300-600 MB inicial), entornos globales reutilizables, excelente para paquetes científicos (compilados C/C++/Fortran/CUDA), robusto gestor de dependencias. Preferido en ciencia de datos.

Comparación Sistemática: venv vs conda

Aspecto	venv	conda
Ubicación del Entorno	Dentro del proyecto (`./venv/`)	Global en el sistema (`~/miniconda3/envs/`)
Peso Inicial	Ligero (5–10 MB)	Pesado (300–600 MB)
Gestor de Paquetes	`pip` (desde PyPI)	`conda` (canales Conda, binarios) y también `pip`
Especialización	General, Web, Scripts	Ciencia de Datos, ML, HPC (maneja dependencias no-Python)
Acoplamiento Proyecto	Alto (entorno local al proyecto)	Bajo (entorno global, reutilizable)
Portabilidad Entorno	Limitada (reinstalación vía `requirements.txt`)	Media (exportación vía `environment.yml` , menos recompilación)
Control de Versiones	Manual (`pip freeze`)	Explícito (`conda list` , `conda env export -f environment.yml`)

Elección: `conda` suele ser mejor para proyectos científicos colaborativos por su robustez con dependencias complejas. `venv` es más directo y ligero para desarrollos web, scripts o tareas simples.

Estructura Recomendada de Proyectos

Independiente del gestor de entornos, una estructura clara facilita la organización:

```
/mi_proyecto/
├── data/           # Datos crudos, externos, procesados (.csv, .pkl)
├── src/            # Código fuente del proyecto (.py)
│   ├── __init__.py # Marca como paquete (opcional pero bueno)
│   ├── main.py     # Script principal / Orquestador
│   ├── utils.py    # Funciones auxiliares reutilizables
│   └── config.py   # Configuraciones, rutas, parámetros
├── notebooks/      # Notebooks para exploración y análisis (.ipynb)
├── outputs/        # Resultados generados (gráficos, logs, modelos)
├── .gitignore      # Archivos/carpetas a ignorar por Git
└── requirements.txt / environment.yml # Dependencias
```

Importante: Incluir `venv/`, `__pycache__/`, `*.pkl`, `outputs/`, `*.log`, `*.ipynb_checkpoints/` en `.gitignore`.

Ejercicio Práctico Sugerido

1. Crear una carpeta `proyecto_prueba`.
2. **Usando `venv`:**
   - `cd proyecto_prueba`
   - `python -m venv venv_prueba`
   - `source venv_prueba/bin/activate`
   - `pip install requests`
   - `which python` (observar ruta dentro de `venv_prueba`)
   - `pip freeze`
   - `deactivate`
3. **Usando `conda`:**
   - `conda create -n conda_prueba python=3.9 requests -y`
   - `conda activate conda_prueba`
   - `which python` (observar ruta dentro de `miniconda3/envs/`)
   - `conda list requests`
   - `conda deactivate`
4. Comparar la ubicación de los ejecutables y la estructura de las carpetas `venv_prueba` y

BLOQUE 1: Fundamentos del Entorno Python

Subbloque 3: Instalación de Paquetes con pip

- **pip:** Gestor de paquetes estándar de Python para instalar librerías desde PyPI (Python Package Index).
- **Funcionamiento:** Cuando se ejecuta `pip install` dentro de un entorno virtual activado, los paquetes se instalan *localmente* en ese entorno.

Comandos básicos:

```
# Instalar un paquete (última versión)
pip install numpy

# Instalar una versión específica
pip install pandas==1.5.3

# Actualizar un paquete
pip install --upgrade requests

# Desinstalar un paquete
pip uninstall numpy

# Listar paquetes instalados y sus versiones
```

`pip freeze` y `requirements.txt`

- **Propósito:** Registrar las dependencias exactas de un proyecto para asegurar la reproducibilidad.
- **Generación:** Guarda la lista de paquetes instalados en el entorno actual (con versiones) en un archivo.

```
pip freeze > requirements.txt
```

- **Contenido Ejemplo (`requirements.txt`):**

```
numpy==1.23.5
pandas==1.5.3
requests==2.28.1
scikit-learn==1.2.2
```

- **Reconstrucción:** Instala exactamente las mismas versiones en otro entorno (nuevo o en otra máquina).

```
pip install -r requirements.txt
```

- **Importancia:** Crucial para evitar errores "funciona en mi máquina" y para la colaboración y el despliegue.

Ejemplo de Conflicto de Versiones

Instalar paquetes en un orden específico puede revelar incompatibilidades:

1. Instalar una versión antigua de `pandas`:

```
pip install pandas==0.25.0
```

2. Intentar instalar una versión reciente de `scikit-learn` que requiere un `pandas` más nuevo:

```
pip install scikit-learn==1.4.1
```

Resultado (Error o Advertencia de pip):

```
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflicts.
scikit-learn 1.4.1 requires pandas>=1.0.5, but you have pandas 0.25.0 which is incompatible.
```

Soluciones Posibles:

- Actualizar `pandas`: `pip install "pandas>=1.0.5"`
- Usar una versión anterior de `scikit-learn` compatible con `pandas==0.25.0`: `pip install "scikit-learn<1.0"`

¿Por qué Ocurren Conflictos?

- **Dependencias Transitivas:** Paquetes dependen de otros paquetes, que a su vez dependen de otros. (`A` necesita `B>=1.0` , `C` necesita `B<1.0`).
- **Resolución de `pip` :** Intenta encontrar un conjunto de versiones que satisfaga todos los requisitos, pero puede fallar o instalar versiones conflictivas si no se especifica cuidadosamente. `conda` tiene un resolvedor más avanzado.
- **Errores Resultantes:**
    - `ERROR: Could not find a version that satisfies the requirement...` (No se encuentra versión compatible).
    - `ImportError: cannot import name 'X' from 'paquete'` (La función/clase esperada no está en la versión instalada).
    - `AttributeError: module 'paquete' has no attribute 'Y'` (Se intenta usar un atributo/método que cambió o fue eliminado entre versiones).

Orden de Resolución de Imports y *Shadowing*

Python busca módulos (`import nombre_modulo`) en este orden:

1. Directorio del script actual.
2. Directorios en `PYTHONPATH` .
3. Directorios en `sys.path` (incluye estándar y `site-packages`).

Problema de *Shadowing*: Crear un archivo local con el mismo nombre que una librería estándar o instalada.

- Ejemplo: Se crea `json.py` en el proyecto.

```
import json # Python importa tu json.py local!
data = json.loads('{"key": "value"}') # Falla!
```

- **Error Típico:** `AttributeError: partially initialized module 'json' has no attribute 'loads'` (porque `json.py` local no tiene la función `loads`).

- **Solución:** Nunca nombrar archivos propios igual que módulos estándar (`math` , `random` , `os` , `sys` , `json` , `csv` , `datetime` , etc.) o librerías comunes (`requests` , `pandas` , etc.).

Actividad Sugerida: Explorar Conflictos

1. Crear un entorno virtual limpio (`venv` o `conda`).
2. Activar el entorno.
3. Instalar una versión antigua de `pandas` :

```
pip install pandas==0.25.0
```

4. Intentar instalar una versión reciente de `scikit-learn` :

```
pip install scikit-learn==1.4.1
```

5. **Observar** el mensaje de error/advertencia de `pip` sobre la incompatibilidad.
6. Resolver el conflicto (elegir una opción):
    - Opción A: Mantener `sklearn` reciente, actualizar `pandas` : `pip install "pandas>=1.0.5"`
    - Opción B: Mantener `pandas` antiguo, bajar `sklearn` : `pip install "scikit-`

BLOQUE 1: Fundamentos del Entorno Python

Subbloque 4: Lectura de Datos Inicial (con pandas)

- **Punto de Partida:** La mayoría de los análisis comienzan cargando datos desde archivos externos.
- **Formatos Comunes:** CSV (Comma-Separated Values), JSON, TXT, Excel, Bases de Datos.
- **Herramienta Principal:** `pandas` es la librería estándar *de facto* para manipulación de datos tabulares en Python.

Lectura básica de CSV:

```
import pandas as pd

# Asume que 'clientes.csv' está en el mismo directorio
df = pd.read_csv("clientes.csv")

# Inspección inicial rápida
print("Primeras 5 filas:")
print(df.head())
print("\nInformación estructural:")
df.info()
print("\nTipos de datos por columna:")
print(df.dtypes)
```

El resultado `df` es un **DataFrame**: una tabla en memoria optimizada para análisis.

Errores Comunes: Codificación (Encoding)

- **Problema:** Archivos de texto creados en sistemas diferentes (e.g., Windows) pueden usar codificaciones no estándar (como `latin1`, `cp1252`) en lugar de `utf-8`.
- **Error Típico al Leer con `pandas` (default `utf-8`):**

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe1 in position 10: invalid continuation byte
```

- **Solución:** Especificar la codificación correcta durante la lectura.

```
# Intentar con codificaciones comunes si utf-8 falla
df = pd.read_csv("clientes.csv", encoding="latin1")
# Otras opciones: 'cp1252', 'iso-8859-1'
```

- **Manejo Robusto:** Usar `try-except` para probar varias codificaciones.

```
encodings_to_try = ['utf-8', 'latin1', 'cp1252']
df = None
for enc in encodings_to_try:
    try:
        df = pd.read_csv("clientes.csv", encoding=enc)
        print(f"Archivo leído con encoding: {enc}")
        break # Salir si funciona
    except UnicodeDecodeError:
        print(f"Encoding {enc} falló, intentando siguiente...")
if df is None:
    print("Error: No se pudo leer el archivo con las codificaciones probadas.")
```

Estructura de un DataFrame (pandas)

Un DataFrame es más que una simple tabla:

- **Componentes:**
    - **Índice (Index):** Etiquetas para las filas (usualmente números 0, 1, 2...).
    - **Columnas (Columns):** Etiquetas para las columnas (nombres de variables).
    - **Datos:** Valores organizados en filas y columnas, usualmente de tipos `numpy`.
- **Analogía:** Similar a un diccionario donde las claves son nombres de columna y los valores son **Series**.
- **Serie (Series):** Cada columna es una Serie: un array unidimensional etiquetado (como un array NumPy con un índice).

Inspección básica:

```
print("Nombres de columnas:", df.columns) # Muestra las etiquetas de columna
print("Dimensiones (filas, columnas):", df.shape)
print("Tipo de dato de la columna 'edad':", df['edad'].dtype)
```

Acceso a columnas:

```
columna_nombre = df["nombre"]  # Devuelve una Serie
columna_edad = df.edad        # Sintaxis alternativa (si el nombre es válido como identificador)
```

Archivos Serializados: Pickle (`.pkl`)

- **¿Qué es?** Pickle es el formato nativo de Python para **serializar** (convertir a bytes) y **deserializar** (reconstruir desde bytes) objetos Python completos (incluyendo DataFrames, listas, diccionarios, modelos ML, etc.).

- **Uso con `pandas`:**

```
# Guardar un DataFrame a un archivo pickle
df.to_pickle("mi_dataframe.pkl")

# Cargar un DataFrame desde un archivo pickle
df_cargado = pd.read_pickle("mi_dataframe.pkl")
```

- **Ventajas:**
    - Rápido: Más eficiente que leer/escribir CSV.
    - Preserva Tipos de Datos: Guarda la estructura y tipos exactos del DataFrame.

- **Desventajas:**
    - No legible por humanos.
    - Específico de Python (no portable a otros lenguajes).
    - Puede tener problemas de compatibilidad entre versiones muy distintas de Python o `pandas`.
    - **Seguridad:** No cargar archivos pickle de fuentes no confiables (pueden contener código malicioso).

Lectura Manual (Sin pandas)

Entender cómo se leen archivos con Python base ayuda a comprender lo que `pandas` abstrae:

1. Leer línea por línea y dividir:

```
datos_lista = []
with open("clientes.csv", encoding="utf-8") as f:
    encabezado = f.readline().strip().split(',') # Leer encabezado
    for linea in f:
        valores = linea.strip().split(',')
        datos_lista.append(valores) # Lista de listas

# print(datos_lista[:2]) # [['Ana', '33', '1800'], ['Luis', '55', '950']]
```

2. Usando el módulo `csv` (más robusto):

```
import csv

datos_dict = []
with open("clientes.csv", newline='', encoding='utf-8') as f:
    lector = csv.DictReader(f) # Lee filas como diccionarios
    for fila in lector:
        datos_dict.append(fila)

# print(datos_dict[:2]) # [{'nombre': 'Ana', 'edad': '33', 'gasto_mensual': '1800'}, ...]
```

Aunque posible, `pandas` es mucho más eficiente y ofrece más funcionalidades para datos tabulares.

Exploración Estructural Rápida (Post-carga)

Una vez cargado el DataFrame (`df`), las primeras validaciones son clave:

1. **Valores Faltantes (NaN/Null):**

- Identificar valores nulos:

```
df.isnull().sum()  # Cuenta nulos por columna
df.isna().sum()    # Alternativa equivalente
```

- Verificar tipos de datos:

```
df.dtypes  # Muestra tipos de cada columna
```

- Reemplazar nulos:

```
df.fillna(valor)  # Reemplaza todos los nulos con 'valor'
df.fillna(method='ffill')  # Forward fill: usa valor anterior
df.fillna(method='bfill')  # Backward fill: usa valor siguiente
```

- Eliminar filas con nulos:

```
df.dropna()  # Elimina filas con cualquier nulo
df.dropna(subset=['col1', 'col2'])  # Solo elimina si nulos en columnas específicas
```

Actividad Sugerida: Carga y Exploración

1. Conseguir o crear un archivo `clientes.csv` (puede incluir errores de formato o codificación si se desea experimentar).
    - Ejemplo: `nombre,edad,ciudad\nAna,33,Rosario\nLuis,55,Madrid\nEva,NA,Barcelona` (con un valor `NA`).
2. **Intentar Cargar con `pandas`:**
    - `import pandas as pd`
    - `df = pd.read_csv("clientes.csv")`
    - Si hay `UnicodeDecodeError`, probar con `encoding='latin1'` u otros.
    - Si hay valores como `NA`, `pandas` podría interpretarlos como NaN (verificar con `df.isnull().sum()`).
3. **Exploración Inicial:**
    - `df.head()`
    - `df.info()`
    - `df.dtypes`
    - `df.isnull().sum()`
    - `df['columna_relevante'].value_counts()`

BLOQUE 2: Notebooks y Estructura de Proyectos

Subbloque 1: Anatomía Técnica de un Notebook (`.ipynb`)

- **No es solo código:** Un archivo `.ipynb` es un documento **JSON** estructurado.
- **Componentes Principales:**
    - Metadatos (versión del kernel, lenguaje).
    - Una lista ordenada de **celdas**.
- **Celdas:** Bloques individuales con contenido y atributos.
    - Pueden contener código, texto formateado (Markdown), o datos crudos.
    - Almacenan su propio `source` (contenido) y `outputs` (resultados de ejecución).
- **Ejecución:** Depende del **estado del kernel**, no solo del orden visual de las celdas.

Tipos de Celdas en un Notebook

Cada celda tiene un `"cell_type"` que define su comportamiento:

1. **`"code"`**:
    - Contiene código ejecutable (e.g., Python).
    - Se ejecuta a través del kernel asociado.
    - Genera `outputs` (resultados, gráficos, errores) que se guardan en el JSON.
    - Tiene un `execution_count` (número de ejecución).
2. **`"markdown"`**:
    - Contiene texto con sintaxis Markdown.
    - Se renderiza como texto formateado (títulos, listas, negritas, etc.).
    - Usada para documentación, explicaciones, títulos de sección.
3. **`"raw"`**:
    - Contiene texto plano sin formato especial.
    - No es interpretada por el kernel ni renderizada como Markdown.
    - Uso menos común, a veces para incluir contenido que será procesado por herramientas externas (e.g., `nbconvert`).

La elección del tipo de celda estructura el notebook y define qué partes son ejecutables y cuáles son descriptivas.

Estructura Interna (JSON Simplificado)

Un archivo `.ipynb` visto como texto revela su estructura JSON:

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "# Análisis Exploratorio\n",
    "Carga de datos y visualización inicial."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 1, // Número de ejecución
   "metadata": {},
   "outputs": [], // Sin salida visible directa
   "source": [
    "import pandas as pd\n",
    "df = pd.read_csv('datos.csv')"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 2,
   "metadata": {},
   "outputs": [ // Salida de la ejecución
    {
     "data": { // Datos de salida (puede ser texto, html, png...)
      "text/plain": [
       "   Col1  Col2\n",
       "0     1     A\n",
       "1     2     B"
      ]
     },
     "execution_count": 2,
     "metadata": {},
     "output_type": "execute_result" // Tipo de salida
    }
   ],
   "source": [ "df.head(2)" ]
  }
 ],
 "metadata": { /* Info del kernel, lenguaje... */ },
 "nbformat": 4, // Versión del formato
 "nbformat_minor": 5
}
```

El Kernel y el Estado Oculto

- **Kernel:** Proceso que ejecuta el código de las celdas (`code`). Mantiene en memoria todas las variables, funciones y objetos creados durante la sesión.
- **Estado Persistente:** El estado del kernel **no** depende del orden visual de las celdas en el archivo `.ipynb` .
- **Ejecución No Lineal:** Es posible ejecutar celdas en cualquier orden. Una variable definida en una celda posterior puede usarse en una celda anterior si la celda de definición se ejecutó primero.

```
# Celda 1 (ejecutada después de Celda 2)
print(x * 2) # Funciona si la Celda 2 ya se ejecutó

# Celda 2 (ejecutada primero)
x = 10
```

Este comportamiento es flexible para exploración pero peligroso para la **reproducibilidad**.

Riesgos del Uso No Controlado de Notebooks

La flexibilidad de los notebooks puede llevar a problemas si no se gestiona:

- **Estado Oculto:** Variables o imports de celdas eliminadas pueden seguir activos en el kernel, haciendo que el notebook funcione *solo en esa sesión*.
- **Ejecución No Lineal:** El resultado final puede depender del orden *histórico* de ejecución, no del orden visual, dificultando la depuración y reproducción.
- **Outputs Desactualizados:** Las salidas (gráficos, tablas) guardadas en el `.ipynb` pueden no corresponder al código actual si este se modificó sin re-ejecutar.
- **Dependencias Implícitas:** Dependencia de archivos locales, variables de entorno no declaradas, o paquetes no listados en `requirements.txt`.
- **Dificultad para Versionar:** Los `.ipynb` (JSON con outputs) generan diffs grandes y poco legibles en Git.

Consecuencia: Notebooks que "funcionan en mi máquina" pero son irreproducibles en otros entornos o incluso en una nueva sesión del mismo usuario.

Herramientas para Gestión de Notebooks

Existen herramientas para mitigar los riesgos y mejorar la integración:

- **`nbconvert`:**
  - Convierte notebooks (`.ipynb`) a otros formatos: scripts Python (`.py`), HTML, PDF, Markdown.
  - Permite ejecutar el notebook desde la línea de comandos y exportar resultados o solo el código.
  - `jupyter nbconvert --to script mi_notebook.ipynb` (Extrae solo el código a `mi_notebook.py`)
- **`nbstripout`:**
  - Elimina las salidas (`outputs`) y contadores de ejecución del archivo `.ipynb`.
  - Útil para limpiar notebooks antes de guardarlos en Git, reduciendo el tamaño y los conflictos. Se puede integrar como hook de Git.
- **`jupytext`:**
  - Permite sincronizar un `.ipynb` con una versión en formato de texto plano (e.g., `.py` con sintaxis especial o `.md`).
  - Facilita la edición en IDEs, el control de versiones y la colaboración.

Estas herramientas ayudan a mantener los notebooks limpios, versionables y más integrados en flujos de trabajo robustos.

Subbloque 2: Estructuración de Proyectos y Separación del Código

- **Limitación de Notebooks:** A medida que un proyecto crece, usar solo notebooks se vuelve insostenible para:
    - Organización del código.
    - Reutilización de funciones.
    - Testing sistemático.
    - Colaboración eficiente.
    - Trazabilidad de cambios.
- **Solución:** Integrar notebooks en una **arquitectura de proyecto modular**, separando el código en archivos `.py` con responsabilidades definidas.

Estructura de Carpetas Recomendada

Un esquema robusto y extensible:

```
/mi_proyecto_analisis/
├── data/
│   ├── raw/        # Datos originales sin modificar
│   └── processed/  # Datos limpios o transformados
├── src/
│   ├── __init__.py
│   ├── data_processing.py # Funciones para limpiar/transformar datos
│   ├── modeling.py        # Funciones para entrenar/evaluar modelos
│   ├── visualization.py   # Funciones para generar gráficos
│   ├── config.py          # Constantes, rutas, parámetros
│   └── main.py            # Script orquestador del pipeline
├── notebooks/             # Notebooks exploratorios, no críticos
│   ├── 01_data_exploration.ipynb
│   └── 02_model_prototyping.ipynb
├── outputs/
│   ├── figures/           # Gráficos generados
│   ├── models/            # Modelos entrenados guardados (.pkl, etc.)
│   └── reports/           # Reportes generados (.md, .html)
├── .gitignore
└── requirements.txt / environment.yml
```

Cada componente tiene un propósito claro, facilitando la navegación y el mantenimiento.

Ventajas de la Modularización

Separar el código en archivos (`src/`) y carpetas estructuradas ofrece beneficios clave:

- **Ejecución Centralizada:** El proyecto completo puede ejecutarse con un solo comando (e.g., `python src/main.py`).
- **Reutilización:** Funciones definidas en `src/` (e.g., `data_processing.py`) pueden ser importadas y usadas tanto en `main.py` como en los `notebooks/` .
- **Testing:** Facilita la escritura de pruebas unitarias para funciones específicas en `src/` .
- **Colaboración:** Diferentes personas pueden trabajar en módulos distintos (`src/`) sin interferencias constantes.
- **Mantenibilidad:** Es más fácil encontrar, entender y modificar partes específicas del código.
- **Trazabilidad:** Los cambios en archivos `.py` son mucho más fáciles de seguir con Git que los cambios en `.ipynb` .
- **Reducción de Errores:** Aísla la lógica de negocio de las exploraciones interactivas, minimizando el riesgo de estado oculto o ejecución no lineal en el código crítico.

El Rol de `.gitignore`

- **Propósito:** Especifica archivos y carpetas que **no** deben ser rastreados ni subidos al repositorio Git.
- **¿Por qué?** Para evitar subir archivos:
    - Grandes (datos crudos, modelos pesados).
    - Específicos del entorno local (`venv/`, `__pycache__/`).
    - Generados automáticamente (`outputs/`, logs, checkpoints).
    - Sensibles (credenciales, claves API - aunque es mejor usar otros métodos).

Ejemplo de `.gitignore` para proyecto Python/DS:

```
# Entornos virtuales
venv/
*.env
env/

# Archivos compilados y caché de Python
__pycache__/
*.pyc
*.pyo
*.pyd

# Archivos de IDEs y editores
.vscode/
.idea/
*.swp

# Archivos generados y datos pesados
data/raw/       # Si los datos crudos son muy grandes
```

Transición desde un Notebook a Estructura Modular

Pasos para refactorizar un análisis hecho inicialmente en un notebook:

1. **Exportar Código:** Usar `jupyter nbconvert --to script mi_notebook.ipynb` para obtener un archivo `mi_notebook.py`.
2. **Identificar Bloques Lógicos:** Revisar el script `.py` y agrupar líneas de código que realizan tareas específicas (e.g., carga de datos, limpieza, cálculo de métrica, visualización).
3. **Crear Funciones:** Encapsular cada bloque lógico en una función dentro de archivos apropiados en `src/` (e.g., `cargar_datos()` en `data_processing.py`, `plot_histograma()` en

Ejemplo: `src/main.py` Orquestador

Este script importa lógica de otros módulos y define el flujo principal:

```python
# src/main.py

# Importar funciones de otros módulos en src/
from data_processing import cargar_datos, limpiar_columna_edad
from modeling import entrenar_modelo_simple
from visualization import graficar_distribucion
# Importar configuraciones
from config import RUTA_DATOS_RAW, RUTA_DATOS_PROCESADOS, RUTA_MODELO

def run_pipeline():
    """Ejecuta el pipeline completo de análisis."""
    print("Iniciando pipeline...")

    # 1. Carga de datos
    df_raw = cargar_datos(RUTA_DATOS_RAW)
    print(f"Datos crudos cargados: {df_raw.shape[0]} filas.")

    # 2. Procesamiento
    df_processed = limpiar_columna_edad(df_raw)
    # ... más pasos de limpieza ...
    df_processed.to_csv(RUTA_DATOS_PROCESADOS, index=False)
    print("Datos procesados y guardados.")

    # 3. Modelado (ejemplo)
    modelo = entrenar_modelo_simple(df_processed)
    # Guardar modelo (ejemplo con pickle)
    import pickle
    with open(RUTA_MODELO, 'wb') as f:
        pickle.dump(modelo, f)
    print("Modelo entrenado y guardado.")

    # 4. Visualización (ejemplo)
    # graficar_distribucion(df_processed['edad'], 'distribucion_edad.png')
    # print("Gráfico generado.")

    print("Pipeline completado.")

# Punto de entrada estándar para scripts ejecutables
if __name__ == "__main__":
    run_pipeline()
```

Subbloque 3: Namespaces, Imports y Resolución de Módulos

- **Namespace (Espacio de Nombres):** Contenedor que mapea nombres (variables, funciones, clases) a objetos. Cada archivo `.py` (módulo) tiene su propio namespace aislado.
- `import`**:** Instrucción para acceder a nombres definidos en otro módulo. Es la forma de conectar namespaces.
- **Modularidad:** Separar código en módulos (`.py`) requiere entender cómo importar y gestionar estos namespaces para evitar conflictos y mantener la claridad.

Formas de Importar Módulos

Python ofrece varias sintaxis, cada una con implicaciones:

1. **Importar Módulo Completo:**

```
import utils
resultado = utils.mi_funcion(param) # Requiere prefijo del módulo
```

   - Claro, evita colisiones de nombres. Bueno para módulos grandes.

2. **Importar Nombres Específicos:**

```
from utils import mi_funcion, OtraClase
resultado = mi_funcion(param) # Se usa directamente
obj = OtraClase()
```

   - Más conciso. Riesgo de colisión si `mi_funcion` existe localmente.

3. **Importar con Alias:**

```
import numpy as np
import pandas as pd
arr = np.array([1, 2, 3]) # Usa alias corto
```

Resolución del Sistema de Importación

Cuando se ejecuta `import mi_modulo`, Python busca `mi_modulo.py` (o un paquete `mi_modulo/`) en una secuencia de directorios:

1. **Directorio del Script Principal:** La carpeta donde se inició la ejecución.
2. **Directorios en `PYTHONPATH`:** Variable de entorno que puede contener rutas adicionales.
3. **Directorios Estándar:** Ubicaciones de la instalación de Python (e.g., `lib/pythonX.Y/`).
4. **`site-packages`:** Directorio donde `pip` o `conda` instalan paquetes de terceros dentro del entorno activo.

Verificar la ruta de búsqueda:

```
import sys
```

Conflicto Común: *Shadowing* de Módulos Estándar

- **Problema:** Se crea un archivo local con el mismo nombre que un módulo de la biblioteca estándar o una librería instalada (e.g., `math.py`, `random.py`, `requests.py`).
- **Consecuencia:** Al intentar importar el módulo real (`import math`), Python encuentra y carga el archivo local primero (debido al orden de resolución), "ocultando" (shadowing) el módulo deseado.

Ejemplo:

```
# Se crea un archivo llamado math.py en el proyecto.
# En otro archivo del mismo proyecto:
import math

# Esto falla, porque 'math' ahora se refiere a math.py local,
# que no tiene la función 'sqrt'.
print(math.sqrt(9))
# AttributeError: module 'math' has no attribute 'sqrt'
```

Diagnóstico: Verificar qué archivo se importó realmente.

```
import math
print(math.__file__) # Mostrará la ruta al archivo math.py local
```

Solución: Nunca usar nombres de archivo que coincidan con módulos importantes.

Importación Relativa vs. Absoluta

Dentro de un proyecto estructurado (e.g., con una carpeta `src/`), hay dos formas de importar módulos internos:

1. **Importación Absoluta (Recomendada):** Especifica la ruta completa desde la raíz del proyecto (o desde donde `sys.path` pueda encontrar `src`).

```
# Dentro de src/main.py o src/otro_modulo.py
from src.utils import funcion_util
from src.config import PARAMETRO
```

   - Clara, funciona independientemente de cómo se ejecute el script (siempre que `src` esté en `sys.path`). Se ejecuta desde la raíz: `python src/main.py`.

2. **Importación Relativa:** Usa puntos (`.`) para referirse a módulos dentro del mismo paquete.

```
# Dentro de src/modulo_a.py
from .utils import funcion_util  # Importa desde utils.py en el mismo directorio (src)
from ..data import cargar_raw    # Importa desde data.py en el directorio padre (no recomendado así)
```

   - Más frágil. Depende de cómo se ejecuta el script y si Python lo reconoce como parte de un paquete. Puede fallar con `ImportError: attempted relative import with no known parent package` si se ejecuta el archivo directamente.

Impacto en la Modularidad

Una gestión correcta de namespaces e imports es la base de la modularidad y trae consigo:

- **Reutilización de Código:** Definir funciones una vez en un módulo (`utils.py`) y usarlas en múltiples scripts o notebooks.
- **Aislamiento:** Cada módulo tiene su propio contexto, reduciendo efectos secundarios inesperados.
- **Testing:** Permite probar unidades de código (funciones, clases) de forma independiente.
- **Legibilidad:** El código se vuelve más fácil de entender, ya que las dependencias son explícitas (`import` statements).
- **Mantenibilidad:** Modificar o corregir una función en un módulo afecta a todos los lugares donde se usa, sin necesidad de buscar y reemplazar código duplicado.
- **Escalabilidad:** Facilita el crecimiento del proyecto añadiendo nuevos módulos con funcionalidades específicas.

Subbloque 4: Uso Riguroso de Notebooks en Entornos Reales

- **Rol Adecuado:** Los notebooks son excelentes para:
    - Exploración interactiva de datos.
    - Prototipado rápido de ideas o modelos.
    - Visualización de resultados.
    - Documentación ejecutable (combinando código, texto y gráficos).
- **Limitaciones en Proyectos Serios:** No deben ser el *único* artefacto. La lógica crítica, pipelines de producción y funciones reutilizables deben residir en módulos `.py`.
- **Integración:** El objetivo es usar notebooks como *clientes* de la lógica modular definida en `src/`, no como contenedores de código monolítico y difícil de mantener.

Comparación de Entornos de Notebook

Entorno	Modelo de Ejecución	Estado del Kernel	Modularización Externa	Exportación Limpia (.py)	Notas
Jupyter	Manual, celda a celda	Global, persistente	Posible (vía `import`)	Requiere `nbconvert`	Estándar, ecosistema maduro. Riesgo de estado oculto.
Colab	Similar a Jupyter (remoto)	Global, persistente	Posible	Requiere descarga	Gratuito, GPU/TPU. Entorno preconfigurado.
VSCode	Integrado (celdas o script .py)	Controlado por el usuario	Fácil (integrado)	Directa (guardar como)	Buena integración IDE, debug, Git.
Marimo	Reactivo, basado en dependencias	Aislado por celda	Natural (es .py)	Directa (es .py)	Evita estado oculto, enfocado en reproducibilidad. Nuevo.

Clave: Entender cómo cada entorno maneja el estado y facilita (o dificulta) la integración con código modular (`.py`) es crucial para un uso profesional.

Ejemplo: Flujo con Marimo (Alternativa Reactiva)

Marimo propone un enfoque diferente para evitar el estado oculto:

1. **Archivo `.py`:** Un notebook Marimo *es* un archivo Python (`.py`).

2. **Ejecución Reactiva:** Cada celda se ejecuta solo si sus dependencias cambian. No hay estado global compartido implícitamente. Las dependencias entre celdas deben ser explícitas.

3. **Definición de Funciones (en una celda):**

```
# celda_funciones.py (o dentro del notebook .py)
def clasificar_cliente(edad, gasto):
    # ... (misma lógica de clasificación) ...
    if gasto >= 2000: return "premium"
    # ... etc ...
    else: return "regular"
```

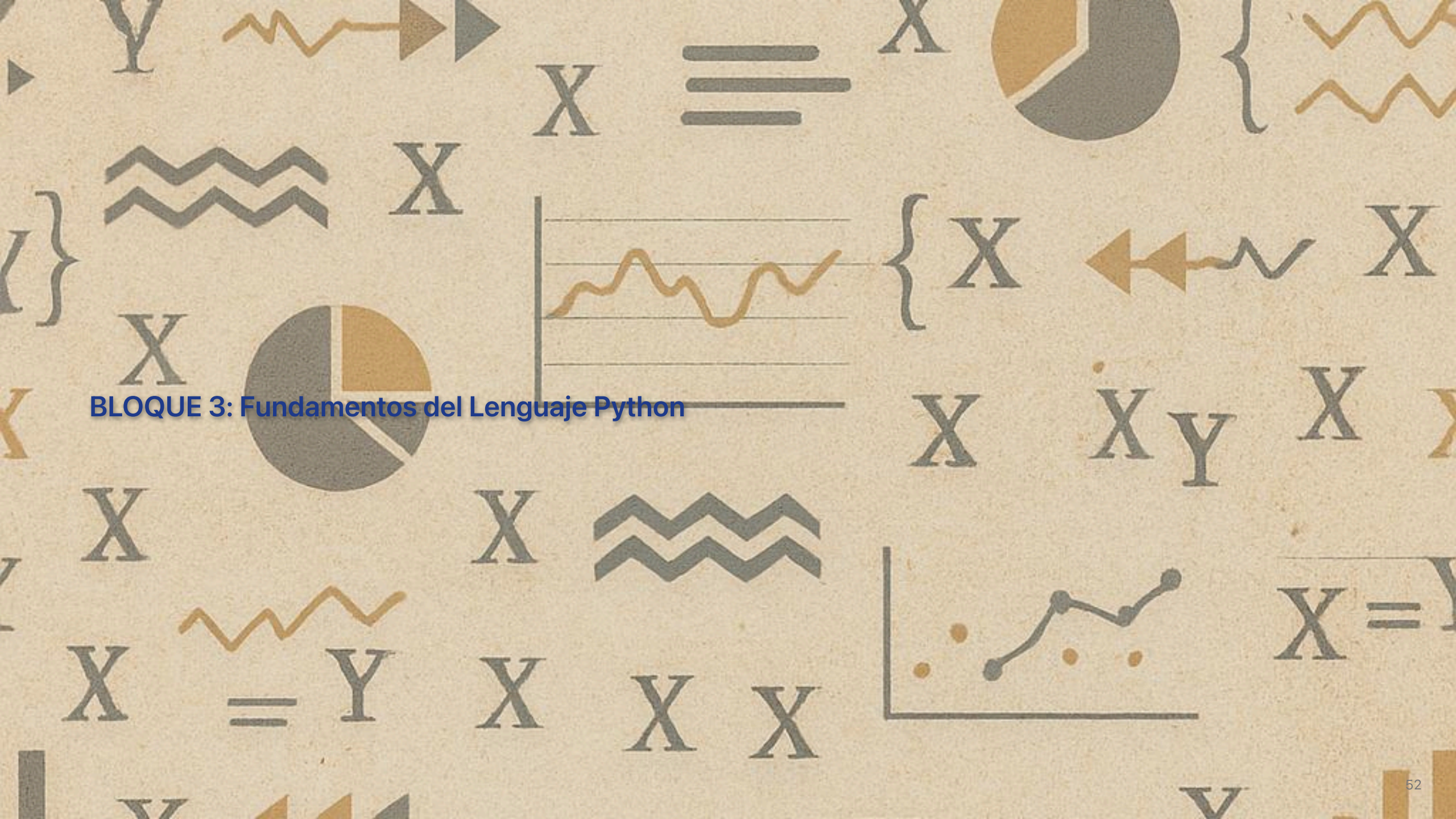
4. **Carga y Aplicación (en otra celda):**

```
# celda_proceso.py
```

Buenas Prácticas para Uso Sostenible de Notebooks

Para maximizar beneficios y minimizar riesgos:

- **Alcance Definido:** Usar notebooks para exploración, prototipado, visualización y documentación, no para lógica de producción.
- **Linealidad:** Escribir el notebook de forma que pueda ejecutarse de arriba abajo (`Kernel > Restart & Run All`) sin errores.
- **Modularidad:** Importar funciones complejas desde módulos `.py` en `src/` en lugar de definirlas dentro del notebook.
- **Documentación:** Usar celdas Markdown para explicar el propósito, los pasos y los resultados.
- **Limpieza:** Eliminar celdas obsoletas o experimentales. Usar herramientas como `nbstripout` para quitar outputs antes de versionar en Git.
- **Entorno Claro:** Asegurarse de que el notebook se ejecuta en el entorno virtual correcto (`venv` o `conda`) con las dependencias definidas en `requirements.txt` .
- **Validación Periódica:** Ejecutar `Restart & Run All` frecuentemente para detectar problemas de estado oculto o dependencia de orden.



BLOQUE 3: Fundamentos del Lenguaje Python

Subbloque 1: Tipos Básicos, Operadores y Validación

- **Tipado Dinámico:** No se declara el tipo de variable explícitamente al crearla.

```
variable = 10      # Python infiere que es int
variable = "hola"  # Ahora es str
```

- **Tipos Primitivos Fundamentales:**
  - `int`: Números enteros (e.g., `5`, `-23`, `0`).
  - `float`: Números de punto flotante (reales) (e.g., `3.14`, `-0.5`, `1.0`).
  - `str`: Cadenas de texto (e.g., `"Python"`, `'Data Mining'`).
  - `bool`: Valores lógicos booleanos (`True` o `False`).
- **Importancia:** Aunque dinámico, cada valor *tiene* un tipo interno que determina qué operaciones son válidas.

Verificación y Validación de Tipos

Es crucial poder saber y comprobar el tipo de una variable antes de operar:

- **Obtener el Tipo:** La función `type()` devuelve el tipo de un objeto.

```
x = 10.5
print(type(x))  # Salida: <class 'float'>

y = "10"
print(type(y))  # Salida: <class 'str'>
```

- **Validar el Tipo:** La función `isinstance()` verifica si un objeto es de un tipo específico (o de una subclase). Devuelve `True` o `False`.

```
valor = 42
if isinstance(valor, int):
    print("Es un entero.")
else:
    print("No es un entero.")

texto = "42"
print(isinstance(texto, int)) # Salida: False
print(isinstance(texto, str)) # Salida: True
```

`isinstance()` es preferible a `type(x) == int` porque maneja herencia correctamente.

Conversión Entre Tipos (Casting)

Transformar un valor de un tipo a otro es común, especialmente al leer datos como texto:

- **Funciones de Conversión:** `int()`, `float()`, `str()`, `bool()`

```
numero_texto = "123"
numero_entero = int(numero_texto) # 123 (int)

precio_texto = "99.95"
precio_float = float(precio_texto) # 99.95 (float)

edad = 30
edad_texto = str(edad) # "30" (str)
```

- **Posibles Errores:** La conversión puede fallar si el valor no es compatible.

```
# int("hola")    # ValueError: invalid literal for int() with base 10: 'hola'
# float("abc")   # ValueError: could not convert string to float: 'abc'
```

- **Conversión Segura:** Usar `try-except` para manejar errores de conversión (se verá más adelante).

Operadores Aritméticos

Realizan cálculos matemáticos:

- `+` : Suma (`5 + 3 = 8`)
- `-` : Resta (`5 - 3 = 2`)
- `*` : Multiplicación (`5 * 3 = 15`)
- `/` : División (siempre devuelve `float`) (`10 / 4 = 2.5`)
- `//` : División Entera (descarta la parte decimal) (`10 // 4 = 2`)
- `%` : Módulo (resto de la división entera) (`10 % 4 = 2`)
- `**` : Exponenciación (potencia) (`5 ** 2 = 25`)

```
a = 15
b = 4
print(f"{a} / {b} = {a / b}")      # 15 / 4 = 3.75
print(f"{a} // {b} = {a // b}")    # 15 // 4 = 3
print(f"{a} % {b} = {a % b}")      # 15 % 4 = 3
print(f"2 ** 10 = {2 ** 10}")      # 2 ** 10 = 1024
```

Operadores de Comparación

Comparan dos valores y devuelven un resultado booleano (`True` o `False`):

- `==` : Igual a (`5 == 5` -> `True`)
- `!=` : Distinto de (`5 != 3` -> `True`)
- `>` : Mayor que (`5 > 3` -> `True`)
- `<` : Menor que (`5 < 3` -> `False`)
- `>=` : Mayor o igual que (`5 >= 5` -> `True`)
- `<=` : Menor o igual que (`5 <= 3` -> `False`)

```
edad = 20
permiso = (edad >= 18) # permiso será True
print(f"¿Puede votar? {permiso}")

nombre1 = "Ana"
nombre2 = "ana"
print(f"¿Nombres iguales? {nombre1 == nombre2}") # False (sensible a mayúsculas)
```

Fundamentales para construir condiciones en estructuras de control (`if`).

Operadores Lógicos

Combinan expresiones booleanas:

- `and`: Devuelve `True` si **ambas** expresiones son `True`.
- `or`: Devuelve `True` si **al menos una** expresión es `True`.
- `not`: Invierte el valor booleano (`not True` es `False`, `not False` es `True`).

```
temperatura = 25
llueve = False

hace_calor = temperatura > 30       # False
buen_clima = temperatura > 20 and not llueve # True (25 > 20 es True, not False es True => True and True)

print(f"¿Hace calor? {hace_calor}")
print(f"¿Buen clima para salir? {buen_clima}")

# Evaluación Cortocircuito:
# En 'A and B', si A es False, B no se evalúa.
# En 'A or B', si A es True, B no se evalúa.
```

Permiten crear condiciones complejas para tomar decisiones.

Ejemplo Integrado: Validación y Operación

Combinar validación, conversión y operación es esencial para código robusto:

```
entrada_usuario = "50.5" # Dato leído como texto
valor_numerico = None

# 1. Validar si la entrada es convertible a número (float)
try:
    valor_numerico = float(entrada_usuario)
except ValueError:
    print("Error: La entrada no es un número válido.")

# 2. Operar solo si la conversión fue exitosa
if valor_numerico is not None:
    # 3. Validar condición antes de calcular
    if valor_numerico > 0:
        resultado = valor_numerico * 1.21 # Ejemplo: Calcular con IVA
        print(f"El valor con 21% de IVA es: {resultado:.2f}") # Formatear a 2 decimales
    else:
        print("El valor debe ser positivo para calcular el IVA.")
```

Este patrón (validar/convertir -> operar si es válido) previene errores y hace el código más seguro.

Subbloque 2: Estructuras Condicionales (`if`, `elif`, `else`)

- **Propósito:** Controlar qué bloques de código se ejecutan basándose en si ciertas condiciones son verdaderas o falsas. Es la base de la toma de decisiones en un programa.

- **Sintaxis Principal:**

```
if condicion1:
    # Bloque de código si condicion1 es True
    ...
elif condicion2:
    # Bloque de código si condicion1 es False y condicion2 es True
    ...
elif condicion3:
    # Bloque de código si c1 y c2 son False, y c3 es True
    ...
else:
    # Bloque de código si NINGUNA de las condiciones anteriores es True
    ...
```

- `elif` (else if) y `else` son opcionales. Puede haber múltiples `elif`.

Sintaxis: `if`, `elif`, `else` - Ejemplo

Clasificación simple basada en una puntuación:

```
puntuacion = 75

if puntuacion >= 90:
    calificacion = "Sobresaliente"
    mensaje = "¡Excelente trabajo!"
elif puntuacion >= 70:
    calificacion = "Notable"
    mensaje = "Buen esfuerzo."
elif puntuacion >= 50:
    calificacion = "Aprobado"
    mensaje = "Suficiente, pero puedes mejorar."
else:
    calificacion = "Suspendido"
    mensaje = "Necesitas repasar."

print(f"Puntuación: {puntuacion}")
print(f"Calificación: {calificacion}")
print(f"Mensaje: {mensaje}")

# Salida para puntuacion = 75:
# Puntuación: 75
# Calificación: Notable
# Mensaje: Buen esfuerzo.
```

Condiciones Compuestas y Anidadas

Las condiciones pueden ser más complejas:

1. Usando Operadores Lógicos (`and`, `or`, `not`):

```
edad = 25
tiene_carnet = True

if edad >= 18 and tiene_carnet:
    print("Puede conducir.")
elif edad >= 16 and not tiene_carnet:
    print("Puede obtener carnet (si aplica), pero aún no conducir.")
else:
    print("No puede conducir.")
```

2. Anidando Estructuras `if`:

```
nota = 8

if nota >= 5:
    print("Aprobado.")
    if nota >= 9:
        print("¡Con honores!")
    elif nota >= 7:
        print("Buen desempeño.")
```

Funciones con Condicionales

Encapsular lógica condicional en funciones mejora la reutilización y claridad:

```
def categorizar_edad(edad):
    """Devuelve una categoría de edad basada en el valor numérico."""
    if not isinstance(edad, (int, float)) or edad < 0:
        return "Edad inválida" # Validación inicial
    elif edad < 13:
        return "Niño/a"
    elif edad < 18:
        return "Adolescente"
    elif edad < 65:
        return "Adulto/a"
    else:
        return "Adulto/a Mayor"

# Uso de la función
print(categorizar_edad(10))  # Salida: Niño/a
print(categorizar_edad(25))  # Salida: Adulto/a
print(categorizar_edad(70))  # Salida: Adulto/a Mayor
print(categorizar_edad(-5))  # Salida: Edad inválida
print(categorizar_edad("veinte")) # Salida: Edad inválida
```

La función se puede probar de forma aislada y usarse en diferentes partes del código.

Validación con `if` Antes de Operar

Un uso fundamental de `if` es proteger el código contra errores por datos inesperados:

1. Verificar si una variable tiene valor (no es `None`):

```
# Supongamos que 'resultado_calculo' puede ser None si falla
resultado_calculo = obtener_resultado() # Puede devolver None

if resultado_calculo is not None:
    # Solo procesar si hay un resultado válido
    procesado = resultado_calculo * 2
    print(f"Resultado procesado: {procesado}")
else:
    print("No se pudo obtener un resultado válido.")
```

2. Comprobar existencia de clave en diccionario:

```
config = {"ruta_archivo": "/data/datos.csv", "modo": "lectura"}

if "ruta_archivo" in config:
    print(f"Usando ruta: {config['ruta_archivo']}")
else:
    print("Advertencia: Falta la clave 'ruta_archivo' en la configuración.")
```

Estos chequeos evitan `TypeError` (operar con `None`) o `KeyError` (acceder a clave inexistente).