Introducción a la Programación en Python para Aprendizaje por Refuerzo

(Apuntes del curso)

Dr. Darío Ezequiel Díaz

16 de marzo de 2025

Índice

1. Fundamentos de Python

1.1. Historia, características y aplicaciones del lenguaje

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, cuya filosofía de diseño enfatiza la legibilidad del código. Fue creado a finales de los años 80 por Guido van Rossum en los Países Bajos, como sucesor del lenguaje ABC. El nombre **Python** proviene del gusto de su creador por los humoristas británicos *Monty Python*. Python soporta múltiples paradigmas de programación (orientación a objetos, programación imperativa y funcional), lo que le da gran flexibilidad. Es **multiplataforma** (ejecutable en distintos sistemas operativos) y **multipropósito**: se utiliza en desarrollo web, ciencia de datos, automatización de tareas, inteligencia artificial, aplicaciones de escritorio, entre otros campos. Además, es software de **código abierto** mantenido por la Python Software Foundation, y ha ganado enorme popularidad a nivel mundial.

En sus primeras versiones, Python ya incluía características avanzadas como manejo de excepciones, módulos, tipos de datos básicos (str, list, dict, etc.) y programación orientada a objetos con soporte de herencia. Gracias a su sintaxis sencilla y clara, Python es considerado un lenguaje ideal para principiantes, a la vez que poderoso para expertos. Hoy en día es uno de los lenguajes más utilizados en el mundo, con una amplia comunidad y un rico ecosistema de bibliotecas para prácticamente cualquier tarea.

1.2. Sintaxis básica y estructura de un programa en Python

La sintaxis de Python es famosa por su **simplicidad y legibilidad**. A diferencia de lenguajes que usan llaves {} o palabras reservadas para delimitar bloques, Python usa **indentación** (espacios o tabuladores al inicio de la línea) para definir la estructura del código. Esto hace que el código tenga un formato uniforme y fácil de leer. Por ejemplo, una función simple en Python se define así:

```
def saludar(nombre):
    # Esta función imprime un saludo personalizado
    print("Hola,", nombre)

# Llamada a la función
saludar("Mundo")
```

En este fragmento, la función saludar está definida con la palabra clave def, seguida del nombre de la función y sus parámetros entre paréntesis, y terminando la cabecera con dos puntos :. El cuerpo de la función está indentado (por convención, se usan 4 espacios). Dentro, usamos la función incorporada print para mostrar un mensaje en pantalla. Al finalizar la indentación, la función termina. Fuera de ella, mostramos cómo llamar a la función con un argumento. La **salida** de este programa sería:

```
Hola, Mundo
```

Un programa Python típico suele consistir en declaraciones de variables, definiciones de funciones o clases, y un bloque principal opcional. Si el archivo Python se ejecuta directamente, se puede utilizar la condición if __name__ == "__main__": para encapsular el código que debe correr (esto previene que ese código se ejecute al importarse el archivo como módulo). En general, la estructura es lineal y se ejecuta secuencialmente, salvo que el flujo sea alterado por funciones o estructuras de control.

1.3. Tipos de datos primitivos

Python ofrece varios **tipos de datos integrados** básicos:

- **Enteros (*int*)**: Números enteros (ej. x = 5). Python los maneja con precisión arbitraria (no hay límite práctico de magnitud).
- **Flotantes (*float*)**: Números de coma flotante (ej. y = 3.14). Representan valores con parte decimal en doble precisión.
- **Cadenas de texto (*str*)**: Secuencias de caracteres unicode (ej. mensaje = "Hola"). Se escriben entre comillas simples o dobles.
- **Booleanos (*bool*)**: Valores lógicos True o False (verdadero o falso). Suelen resultar de comparaciones o expresiones lógicas.

Podemos utilizar la función type() para comprobar el tipo de una variable. Ejemplo:

```
a = 42
b = 3.1415
c = "Python"
d = True

print(type(a), type(b), type(c), type(d))
# Salida: <class 'int'> <class 'float'> <class 'str'> <class
    'bool'>
```

Estos tipos primitivos admiten operaciones aritméticas y lógicas habituales. Por ejemplo, enteros y flotantes se pueden sumar, restar, multiplicar, etc., las cadenas se pueden concatenar con + o repetir con *, y los booleanos soportan operaciones lógicas (and, or, not). Python es de **tipado dinámico**, lo que significa que no es necesario declarar el tipo de una variable; el tipo se asigna automáticamente en tiempo de ejecución según el valor, y una misma variable puede reasignarse a datos de distinto tipo.

1.4. Estructuras de datos avanzadas

Además de los tipos básicos, Python tiene **estructuras de datos integradas** muy flexibles:

- **Listas (*list*)**: Colecciones ordenadas y mutables de elementos. Se definen con corchetes, ej. mi_lista = [1, "dos", 3.0]. Pueden contener elementos de cualquier tipo (inclusive mezclados) y su tamaño es dinámico. Permiten operaciones de *slicing* (rebanado) para obtener subconjuntos.
- **Tuplas (*tuple*)**: Secuencias ordenadas inmutables. Se definen con paréntesis, ej. mi_tupla = (10, 20, 30). Una vez creada, no se pueden añadir, quitar o modificar elementos (son de solo lectura). Útiles para representar grupos de valores que no deben cambiar y para usar como claves en diccionarios.
- **Diccionarios (*dict*)**: Estructuras de mapeo que almacenan pares clave-valor. Se definen con llaves, ej. mi_dict = {"nombre": .^Alice", .^edad": 30}. Permiten el acceso a cada valor mediante su clave (que suele ser una cadena u otro tipo inmutable). Son mutables: se pueden añadir, modificar o eliminar pares.
- **Conjuntos (*set*)**: Colecciones no ordenadas de elementos únicos (sin repeticiones). Se definen con llaves o la función set(), ej. mi_set = {1, 2, 3}. Son útiles para operaciones de teoría de conjuntos (uniones, intersecciones, etc.) y para eliminar duplicados de una secuencia.

Estas estructuras vienen con numerosas funciones y métodos útiles. Por ejemplo, las listas tienen métodos como append (para agregar elementos al final), remove (para eliminar la primera ocurrencia de un valor), sort (ordenar la lista in-place), etc. Los diccionarios permiten consultar todas sus claves con dict.keys() o todos sus valores con dict.values(). Los conjuntos soportan operaciones algebraicas: setA.union(setB), setA.intersection(setB), etc. Veamos algunas operaciones básicas:

```python
frutas = ["manzana", "banana", "cereza"]
frutas.append("durazno")  # Agregar elemento a la lista
print(frutas[1])  # Acceder al segundo elemento (índice 1)
# Salida: banana

punto = (10, 20)
# punto[0] = 5  # Esto daría error, las tuplas no permiten asignación

persona = {"nombre": "Bob", "edad": 25}
persona["edad"] = 26  # Modificar valor bajo la clave "edad"
print(persona.get("nombre"))  # Obtener valor de "nombre"
# Salida: Bob

nums = {1, 2, 2, 3, 4}
print(nums)  # Los duplicados se eliminan en un set
# Salida: {1, 2, 3, 4}
```

En el ejemplo anterior, mostramos cómo manipular cada estructura: agregando una fruta a la lista, intentando (sin éxito) modificar una tupla, actualizando un diccionario y demostrando la unicidad de elementos en un conjunto.

1.5. Control de flujo: condicionales y bucles

Para dotar de lógica a los programas, Python ofrece **estructuras de control** condicionales y repetitivas:

- **Condicional if/elif/else**: Permite ejecutar bloques de código solo si se cumple una condición. Por ejemplo:

```python
x = 10
if x > 0:
    print("x es positivo")
elif x < 0:
    print("x es negativo")
else:
    print("x es cero")
```

Aquí, según el valor de x, se imprimirá una u otra línea. Solo se ejecuta un bloque de los anteriores: el primero cuyo condicional sea verdadero (o el else si ninguno de los anteriores lo fue). Las condiciones se pueden combinar con operadores lógicos (and, or, not) y comparar con operadores relacionales (==, !=, ¡, ¿, ¡=, ¿=). Python admite sintaxis abreviada para if en una sola línea (operador ternario) y también el match-case (similar a switch-case en otros lenguajes) a partir de Python 3.10, aunque para principiantes el if/elif/else es suficiente.

- **Bucle for**: Sirve para iterar sobre elementos de una secuencia (lista, tupla, cadena, rango, etc.). Por ejemplo, para iterar del 1 al 5:

```
for i in range(1, 6):
    print(i)
# Salida: 1 2 3 4 5
```

La función range(n) produce una secuencia de 0 a n-1. En este caso usamos range(1, 6) para obtener 1,2,3,4,5. Dentro del bucle, i toma cada valor sucesivamente y se imprime. Podemos usar for para recorrer listas directamente: for fruta in frutas: .... También es común usar enumerate(lista) para obtener el índice y valor a la vez, o zip(lista1, lista2) para iterar en paralelo dos listas.

- **Bucle while**: Repite un bloque mientras una condición sea verdadera. Ejemplo:

```
count = 0
while count < 3:
    print("Iteración", count)
    count += 1
# Salida:
# Iteración 0
# Iteración 1
# Iteración 2
```

Aquí el bloque while se ejecuta hasta que count ¡3 deje de ser verdadero (es decir, cuando count alcanza 3, el bucle termina). Es importante asegurarse de que la condición eventualmente sea falsa para evitar bucles infinitos. Dentro de los bucles (for o while) se pueden usar break para salir inmediatamente del bucle, o continue para saltar a la siguiente iteración.

Estas estructuras permiten controlar el flujo de ejecución de forma muy expresiva. La sintaxis, de nuevo, se basa en los dos puntos : y la indentación para definir qué código pertenece a cada bloque.

1.6. Buenas prácticas de codificación (PEP8)

Python cuenta con una guía oficial de estilo llamada **PEP 8** (Python Enhancement Proposal #8). Esta guía establece convenciones para escribir código Python de forma clara y consistente. Algunas recomendaciones principales de PEP8 son:

- **Indentación**: Usar 4 espacios por nivel de indentación (no usar tabulaciones mezcladas con espacios). Esto mejora la legibilidad y evita inconsistencias entre editores.

- **Longitud de línea**: Limitar las líneas a un máximo de ∼79 caracteres para facilitar la lectura en pantallas o ventanas divididas. Si una expresión es muy larga, se puede dividir en múltiples líneas usando paréntesis o el carácter \ al final de la línea.
- **Nombres de identificadores**: Usar *snake_case* para funciones y variables (ej. mi_variable), *PascalCase* para nombres de clases (ej. MiClase), y MAYÚSCULAS para constantes. Los módulos también suelen tener nombres en minúsculas.
- **Espacios en blanco**: Usar espacios alrededor de operadores y después de comas para separar elementos, pero evite espacios extra innecesarios dentro de paréntesis, corchetes o llaves. Ejemplo: preferir func(a, b) en lugar de func(a , b).
- **Importaciones**: Realizar las importaciones en la parte superior del archivo, en secciones separadas (primero librerías estándar, luego externas, luego módulos propios), y una por línea.
- **Comentarios**: Incluir comentarios descriptivos y útiles, especialmente para bloques de código no triviales. Mantenerlos actualizados si el código cambia. PEP8 sugiere usar comentarios en texto claro (inglés en la comunidad global, pero en proyectos en español podría usarse español) y docstrings para documentar módulos, funciones y clases.

Seguir estas convenciones hace que el código sea más *"pythonic"* (idiomático de Python) y facilita su lectura y mantenimiento por parte de otros desarrolladores (y de uno mismo en el futuro). Muchas herramientas como linters (por ejemplo flake8 o pylint) pueden analizar el código y advertir sobre violaciones a PEP8 para ayudar a mantener el estilo consistente.

Por ejemplo, observe la diferencia en legibilidad:

```
# Mal estilo (no sigue PEP8)
def Sumar(a,b):return(a+b)
resultado=Sumar(5,3);print(resultado)

# Buen estilo (siguiendo PEP8)
def sumar(a, b):
    """Retorna la suma de a y b."""
    return a + b

resultado = sumar(5, 3)
print(resultado)  # Imprime 8
```

En el segundo fragmento, usamos minúsculas para el nombre de la función, agregamos un docstring que explica qué hace, separamos las operaciones

en líneas distintas y dejamos espacios alrededor del operador + y después de las comas. Esto hace el código mucho más entendible.

2. Programación Funcional y Orientada a Objetos

2.1. Definición y uso de funciones en Python

Las **funciones** son bloques de código reutilizables que realizan una tarea específica. En Python, se definen con la sintaxis def nombre_funcion(parametros): seguida de un bloque indentado con el cuerpo de la función. Ya vimos un ejemplo sencillo con la función saludar. Las funciones pueden aceptar **parámetros** (argumentos de entrada) y opcionalmente **devolver** un valor usando la sentencia return. Si no se especifica un return, la función retorna None (un tipo especial que indica ausencia de valor).

Ejemplo de una función con cálculo y retorno:

```
def elevar_al_cuadrado(numero):
    """Devuelve el cuadrado del número proporcionado."""
    resultado = numero ** 2  # ** es el operador de potencia
    return resultado

x = 4
y = elevar_al_cuadrado(x)
print(f"{x} al cuadrado es {y}")
# Salida: 4 al cuadrado es 16
```

Aquí la función elevar_al_cuadrado toma un número, calcula su cuadrado y lo retorna. La variable y recibe el resultado. Las **f-strings** (f"{x} ... {y}") nos permiten incrustar variables dentro de una cadena formateada para imprimir.

Python también soporta **parámetros opcionales** con valores por defecto. Por ejemplo:

```
def saludar(nombre, saludo="Hola"):
    print(f"{saludo}, {nombre}")

saludar("Ana")  # Usa el saludo por defecto "Hola"
saludar("Luis", saludo="Hi")  # Especifica un saludo
    diferente
# Salida:
# Hola, Ana
# Hi, Luis
```

Las funciones son **ciudadanas de primera clase** en Python, lo que significa que pueden asignarse a variables, pasarse como argumentos a otras

funciones o retornarse como resultados. Esto habilita un estilo de programación funcional. Por ejemplo, se puede hacer:

```
def doble(n):
    return n*2

aplicar = doble  # Asignamos la función 'doble' a la variable 'aplicar'
print(aplicar(5))  # Llamamos a 'aplicar' (que refiere a 'doble')
# Salida: 10
```

Además, Python permite funciones **anónimas** (también conocidas como *funciones lambda*). Son funciones pequeñas y sin nombre que se definen en una sola línea con la sintaxis lambda parametros: expresion. Por ejemplo: lambda x, y: x + y define una función anónima que suma dos valores. Las lambdas se usan comúnmente para funciones simples de corto recorrido, por ejemplo en parámetros de funciones de orden superior (como veremos a continuación).

2.2. Decoradores y funciones de orden superior

En Python, las **funciones de orden superior** son funciones que reciben otras funciones como argumentos o devuelven funciones como resultado. Ejemplos integrados son map, filter y reduce, que provienen de la programación funcional tradicional:

- map(func, secuencia) aplica la función func a cada elemento de la secuencia, devolviendo un iterador con los resultados. Ejemplo: map(lambda x: x**2, [1,2,3]) produce [1,4,9].
- filter(func_condicion, secuencia) filtra los elementos de la secuencia para los cuales func_condicion (una función que retorna True/False) devuelve True. Ejemplo: filter(lambda x: x%2==0, [1,2,3,4]) produce [2,4] (números pares).
- reduce(func_combinacion, secuencia) aplica acumulativamente func_combinacion a los elementos de la secuencia, reduciéndolos a un solo valor. (Esta función se debe importar desde functools). Ejemplo: reduce(lambda a,b: a+b, [1,2,3,4]) devuelve 10 (suma todos los elementos).

Veamos un ejemplo usando estas funciones para ilustrar su uso con funciones lambda:

```python
from functools import reduce

nums = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x**2, nums))
impares = list(filter(lambda x: x % 2 == 1, nums))
suma_total = reduce(lambda a, b: a + b, nums)

print("Cuadrados:", cuadrados)
print("Impares:", impares)
print("Suma total:", suma_total)
# Salida:
# Cuadrados: [1, 4, 9, 16, 25]
# Impares: [1, 3, 5]
# Suma total: 15
```

En el código anterior, map genera la lista de cuadrados, filter extrae los impares, y reduce suma todo.

Otra poderosa característica de Python son los **decoradores**. Un *decorador* es esencialmente una función de orden superior que toma una función y devuelve una versión modificada de la misma. Se usan para añadir funcionalidad a funciones o métodos existentes de manera compacta, sin modificar su código original (por ejemplo, para logging, control de acceso, medición de tiempo de ejecución, etc.). La sintaxis de decorador utiliza el símbolo @ antes de la definición de una función para envolverla con otra.

Ejemplo sencillo de un decorador que imprime un mensaje antes y después de ejecutar una función dada:

```python
def anuncio(func):
    def funcion_decorada(*args, **kwargs):
        print("-> Voy a ejecutar", func.__name__)
        resultado = func(*args, **kwargs)
        print("<- Terminé de ejecutar", func.__name__)
        return resultado
    return funcion_decorada

@anuncio
def saludar(nombre):
    print(f"Hola, {nombre}!")

saludar("Carlos")
# Salida:
# -> Voy a ejecutar saludar
# Hola, Carlos!
# <- Terminé de ejecutar saludar
```

Aquí, anuncio es un decorador que envuelve cualquier función para imprimir mensajes antes y después de su ejecución. La sintaxis @anuncio sobre la definición de saludar es equivalente a hacer saludar = anuncio(saludar).

Al llamar saludar(Çarlos"), en realidad se está ejecutando funcion_decorada dentro del decorador, que realiza las acciones adicionales y luego llama a la función original. Los decoradores permiten añadir esta lógica extra de forma declarativa y reusable.

2.3. Clases y objetos en Python (POO)

La **Programación Orientada a Objetos (POO)** es un paradigma en el cual estructuramos el código en torno a **clases** (planos o moldes que definen comportamiento y datos) y **objetos** (instancias concretas de esas clases). Python soporta totalmente la POO, permitiendo definir clases, crear objetos, y aplicar principios como encapsulación, herencia y polimorfismo. De hecho, en Python *todo* es un objeto (incluso tipos básicos como enteros, cadenas, funciones, etc., son objetos de alguna clase).

Una clase en Python se define con la palabra clave class seguida del nombre de la clase y dos puntos. Dentro, se definen atributos (variables) y métodos (funciones) que describen el estado y comportamiento de los objetos de esa clase. Por convención, usamos *PascalCase* para nombrar clases. Ejemplo básico:

```
class Persona:
    """Modelo sencillo de una persona."""

    def __init__(self, nombre, edad):
        # Método constructor: inicializa atributos del objeto
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        # Método de instancia: comportamiento usando
    atributos del objeto
        print(f"Hola, me llamo {self.nombre} y tengo {self.
    edad} años.")
```

En esta clase Persona, el método especial __init__ es el **constructor**: una función que Python invoca al crear una nueva instancia de la clase. Sus parámetros (aparte de self) son los datos necesarios para inicializar el objeto; aquí asignamos self.nombre y self.edad. El parámetro self representa al propio objeto (instancia) y es obligatorio en la definición de métodos de instancia (Python lo pasa automáticamente al llamar al método). La clase también tiene un método regular saludar que imprime una presentación usando los atributos del objeto.

Veamos cómo usar esta clase:

```
# Crear instancias (objetos) de la clase Persona
```

```
p1 = Persona("Alice", 30)
p2 = Persona("Bob", 25)

# Llamar métodos de los objetos
p1.saludar()
p2.saludar()
# Salida:
# Hola, me llamo Alice y tengo 30 años.
# Hola, me llamo Bob y tengo 25 años.
```

Cada objeto (p1, p2) mantiene su propio estado interno (nombre y edad distintos) y el método saludar actúa según ese estado. Esto ejemplifica la **encapsulación**: los datos (nombre, edad) y los métodos que operan sobre ellos (saludar) están reunidos en una misma entidad lógica (la instancia de Persona).

Python soporta **herencia de clases**, lo que permite crear nuevas clases basadas en clases existentes, reutilizando su código. Una clase hija hereda atributos y métodos de la padre, y puede agregar nuevos o sobreescribir (*override*) los existentes. Además, Python admite **herencia múltiple** (una clase puede heredar de varias clases base). Ejemplo: definamos una subclase Estudiante que hereda de Persona y agrega un atributo de grado académico:

```
class Estudiante(Persona):  # Hereda de Persona
    def __init__(self, nombre, edad, carrera):
        # Llamamos al constructor de la clase base (Persona)
        super().__init__(nombre, edad)
        self.carrera = carrera

    def saludar(self):
        # Sobrescribimos el método saludar para incluir la carrera
        print(f"Hola, soy {self.nombre}, estudiante de {self.carrera}.")
```

Aquí Estudiante extiende Persona. Usamos super().__init__(nombre, edad) para invocar el constructor original de Persona (así no reimplementamos la asignación de nombre y edad) y luego definimos el nuevo atributo carrera. También sobrescribimos saludar para que muestre la carrera en la presentación. Ahora, un objeto de Estudiante responderá a saludar con el nuevo comportamiento, pero aún puede usar métodos de Persona no sobrescritos. Por ejemplo:

```
e = Estudiante("Carlos", 22, "Ingeniería")
e.saludar()  # Llama al saludar() de Estudiante
# Salida: Hola, soy Carlos, estudiante de Ingeniería.

print(e.edad)  # Podemos acceder al atributo heredado edad
# Salida: 22
```

La instancia e posee tanto los atributos de Persona (nombre, edad) como el nuevo carrera. Este mecanismo muestra el **polimorfismo**: Estudiante.saludar reemplaza la versión de Persona.saludar, adecuándose al nuevo tipo de objeto.

Python también ofrece **métodos de clase** (definidos con @classmethod) que reciben la clase como parámetro en lugar de instancia (convención: cls en vez de self), útiles por ejemplo para constructores alternativos. Y **métodos estáticos** (@staticmethod) que no reciben ni instancia ni clase y actúan como funciones normales definidas dentro de la clase (simplemente para organización). Un ejemplo de método de clase podría ser un constructor que cree un Estudiante a partir de una cadena con datos separados por coma; un método estático podría ser una función auxiliar de la clase, como validar si una edad es válida.

En Python no existe *encapsulamiento estricto* (todos los atributos son accesibles públicamente por defecto), pero se usan convenciones: un atributo con guión bajo (e.g. _secreto) indica que es ïnternoz no debería accederse desde fuera de la clase. Para verdaderos atributos privados, Python "manglea"los nombres con doble guión bajo, e.g. __privado se renombra internamente para que no sea fácilmente accesible desde fuera.

2.4. Manejo de excepciones en POO

El manejo de excepciones (errores en tiempo de ejecución) también es parte de la robustez de un diseño orientado a objetos. En Python, las excepciones son objetos de clases que heredan de BaseException. Pueden ser lanzadas (raise) y capturadas (try/except). En contexto de POO, es común definir **excepciones personalizadas** creando una subclase de Exception. Por ejemplo:

```
class EdadInvalidaError(Exception):
    pass

def set_edad(persona, edad):
    if edad < 0:
        raise EdadInvalidaError("La edad no puede ser
   negativa")
    persona.edad = edad

# Uso
p = Persona("Ana", 30)
try:
    set_edad(p, -5)
except EdadInvalidaError as e:
    print("Error al asignar edad:", e)
```

```
15 # Salida: Error al asignar edad: La edad no puede ser
       negativa
```

Aquí definimos una excepción EdadInvalidaError (vacía, usamos pass ya que no añadimos comportamiento extra, solo la creamos para identificar este tipo de error). La función set_edad lanza esa excepción si se proporciona una edad inválida. Luego, al usar try/except, capturamos específicamente EdadInvalidaError y manejamos el error imprimiendo un mensaje. Este patrón es útil en métodos de clases también: una clase podría validar sus atributos y lanzar excepciones significativas si algo anda mal. Por ejemplo, podríamos incorporar esta lógica dentro de un método setter de la clase Persona.

Con esto terminamos la sección de POO. Python permite escribir código en estilos multiparadigma: podemos mezclar programación funcional (con funciones de orden superior, lambdas) y orientación a objetos en el mismo programa según convenga. Esta flexibilidad es parte del poder de Python.

3. Manipulación de Datos con NumPy y Pandas

En el ecosistema científico y de análisis de datos en Python, dos bibliotecas destacan: **NumPy** (Numerical Python) y **Pandas**. NumPy proporciona un objeto eficiente para arrays (arreglos) multidimensionales y operaciones matemáticas vectorizadas, mientras que Pandas ofrece estructuras de datos de alto nivel (principalmente DataFrames) para manipulación tabular de datos, similar a hojas de cálculo o tablas SQL. Aprender a usar ambas es fundamental para el manejo de datos previo a aplicar algoritmos de *Machine Learning* o análisis más avanzados.

3.1. Lectura y escritura de datos desde CSV, Excel, JSON y SQL

Una de las primeras tareas en un flujo de trabajo de datos es **cargar datos desde fuentes externas**. Pandas simplifica enormemente esto mediante funciones de entrada/salida (I/O) que leen archivos comunes directamente en estructuras DataFrame. Por ejemplo:

- pd.read_csv(.ªrchivo.csv") carga un archivo CSV (valores separados por comas) en un DataFrame. Pandas detecta automáticamente cabeceras de columna, tipos de datos, etc., aunque se pueden especificar parámetros para ajustar la lectura (separador, codificación, filas a saltar, columnas a usar, tipos de datos, manejo de fechas, etc.).

- pd.read_excel(.ªrchivo.xlsx", sheet_name="Hoja1") lee una hoja de un archivo Excel (formatos .xls o .xlsx) en un DataFrame. Requiere tener instalado *openpyxl* o *xlrd* según la versión de Excel, pero Pandas lo integra transparentemente.

- pd.read_json("datos.json") carga datos en formato JSON. Pandas espera que el JSON tenga una estructura tabular (por ejemplo una lista de registros/diccionarios). También existe pd.read_html para leer tablas en páginas HTML, pd.read_sql para ejecutar consultas SQL y obtener DataFrames (requiere una conexión a base de datos), etc. En general, **Pandas soporta múltiples formatos**: CSV, JSON, HTML, Excel, SQL, Parquet, entre otros.

Del mismo modo, para **escribir o exportar** datos, Pandas ofrece métodos simétricos: DataFrame.to_csv("salida.csv"), DataFrame.to_excel("salida.xlsx"), DataFrame.to_json("salida.json"), DataFrame.to_sql("tabla", conexion) etc., que guardan el contenido del DataFrame en archivos o bases de datos. Estos métodos facilitan mover datos entre sistemas sin esfuerzo manual.

Como ejemplo básico, supongamos que tenemos un archivo CSV con datos de personas (personas.csv):

```
nombre,edad,ciudad
Ana,34,Madrid
Bob,45,Barcelona
Carlos,29,Valencia
```

Podemos leerlo y luego escribir otro formato así:

```python
import pandas as pd

df = pd.read_csv("personas.csv")
print(df.head())
# Salida (DataFrame de ejemplo):
# nombre edad ciudad
# 0 Ana 34 Madrid
# 1 Bob 45 Barcelona
# 2 Carlos 29 Valencia

df.to_json("personas.json", orient="records", lines=True)
```

En este código, df.head() muestra las primeras filas del DataFrame (por defecto 5). Luego usamos to_json para guardar el DataFrame en formato JSON, donde cada línea del archivo JSON será un registro (orientación por registros). Esta versatilidad hace que Pandas sea un *hub* central para datos de distintas fuentes.

3.2. Manipulación y limpieza de DataFrames en Pandas

Una vez los datos están en un DataFrame de Pandas, podemos realizar una variedad de **operaciones de limpieza y transformación** de forma muy eficiente y con sintaxis concisa:

- **Selección y filtrado**: Podemos seleccionar columnas usando df[çol"] o df[[çol1",çol2"]] (esta última devuelve otro DataFrame con solo esas columnas). Para filtrar filas según una condición, podemos usar expresiones booleanas: df[df[."edad"] ¿30] devolverá solo las filas donde la columna *edad* sea mayor que 30. También existen métodos como df.loc[filas, columnas] para selección por etiquetas, y df.iloc[fila_idx, col_idx] para selección por índices numéricos.

- **Agregar o eliminar columnas**: Asignar a una columna no existente la crea. Ej: df[."edad_10años"] = df[."edad"] + 10 crearía una nueva columna calculada. Para eliminar, df.drop(çolumna", axis=1) devuelve un nuevo DataFrame sin esa columna (axis=1 indica columna; axis=0 sería para filas). Igualmente df.drop([lista_de_columnas], axis=1) quita múltiples. Para eliminar filas, se usa df.drop(etiqueta_fila, axis=0).

- **Detectar y manejar valores nulos**: Con df.isnull() obtenemos un DataFrame booleano del mismo tamaño indicando True donde hay valores NaN (nulos). Sumando booleans en Pandas (True=1, False=0) se puede contar nulos: df.isnull().sum() da el conteo de nulos por columna. Para eliminar nulos, df.dropna() quita filas con cualquier NaN (o columnas, según parámetros). Alternativamente, df.fillna(valor) reemplaza los NaN por un valor dado (o usando métodos como forward-fill/ffill). Un buen flujo es primero identificar la cantidad de datos faltantes y luego decidir si eliminar filas/columnas o imputar valores.

- **Eliminar duplicados**: A veces el dataset contiene filas duplicadas. Pandas ofrece df.duplicated() que marca duplicados, y df.drop_duplicates() que elimina las filas duplicadas, conservando solo la primera ocurrencia (por defecto). Se pueden especificar subset de columnas para considerar duplicados y si conservar primera o última ocurrencia.

- **Renombrar columnas**: Con df.rename(columns={."old": "new", ...}, inplace=True) podemos cambiar nombres de columnas para hacerlos más manejables.

- **Tipo de datos**: Cada columna en Pandas tiene un tipo (df.dtypes los lista). Podemos convertir tipos, p. ej., si una columna de fechas está como texto, usar pd.to_datetime(df[”fecha”]) para parsearla a tipo datetime. Igualmente df[çol”] = df[çol”].astype(int) convertiría una columna a entero (si es posible).

- **Operaciones vectorizadas**: Muchas operaciones aritméticas o lógicas pueden aplicarse directamente a columnas. Ya se vio un ejemplo sumando 10 a la columna edad. Similarmente df[ïngresos_anuales”] = df[ïngresos_mensuales”] * 12 multiplicaría elemento a elemento. Esto está altamente optimizado en Pandas (usando NumPy por debajo).

Veamos un pequeño ejemplo de **limpieza**: supongamos que nuestro DataFrame df de personas tiene algunos valores faltantes y duplicados:

```
# Ejemplo de DataFrame con valores faltantes y duplicados
import numpy as np

df = pd.DataFrame({
    "nombre": ["Ana", "Bob", "Cara", "Bob"],
    "edad": [34, 45, np.nan, 45],
    "ciudad": ["Madrid", "Barcelona", "Sevilla", "Barcelona"]
})

print("Original:\n", df, "\n")

# 1. Eliminar filas duplicadas
df = df.drop_duplicates()

# 2. Manejar valores nulos: reemplazar NaN en 'edad' por la
   media de edades
media_edad = df["edad"].mean(skipna=True)
df["edad"].fillna(media_edad, inplace=True)

print("Limpio:\n", df)
```

Imaginemos que la salida de este código es:

```
Original:
   nombre   edad     ciudad
0     Ana   34.0     Madrid
1     Bob   45.0  Barcelona
2    Cara    NaN    Sevilla
3     Bob   45.0  Barcelona

Limpio:
```

```
   nombre   edad     ciudad
0     Ana   34.0     Madrid
1     Bob   45.0  Barcelona
2    Cara   39.5    Sevilla
```

Explicando: teníamos una fila duplicada (Bob, 45, Barcelona repetido) que fue eliminada por drop_duplicates(); luego teníamos la edad de Çaraçomo NaN (dato faltante), que reemplazamos por la media de las edades existentes ((34+45+45)/3 = 124/3 ≈ 41.33, aunque en la salida aparece 39.5 lo cual sugiere quizás otro cálculo, pero imaginemos que es la media). Ahora el DataFrame "limpioza no tiene duplicados ni faltantes.

Esta capacidad de transformar datos rápidamente es una de las fortalezas de Pandas para preparar datasets antes del modelado.

3.3. Operaciones avanzadas con Pandas (groupby, merge, nulos y duplicados)

Además de las operaciones básicas, Pandas soporta operaciones relacionales y de agregación similares a las de SQL:

- **Group By (agrupación)**: Permite agrupar las filas por valores de una o varias columnas, y luego aplicar funciones de agregación a cada grupo (suma, promedio, conteo, etc.). Es la operación de "split-apply-combine": dividir el DataFrame en grupos según criterios, aplicar cálculos en cada grupo y combinar los resultados. Por ejemplo: df.groupby(çiudad")[.edad"].mean() agruparía por ciudad y calcularía la edad media en cada ciudad. Podemos agrupar por múltiples campos y aplicar múltiples agregaciones simultáneamente usando .agg({...}).

- **Merge/JOIN (combinación de DataFrames)**: Para combinar datos de diferentes fuentes en base a una clave común, Pandas ofrece pd.merge(df1, df2, on=çlave"), análogo a un JOIN de SQL. Podemos especificar el tipo de join: interno (por defecto), externo (how=.outer"), left, right, etc., y también manejar situaciones de claves duplicadas o nombres de columnas conflictivos. Alternativamente, df1.join(df2) puede combinar DataFrames por índice. Y pd.concat([...]) puede concatenar DataFrames verticalmente (apilando filas) u horizontalmente (añadiendo columnas si comparten índice). Pandas provee métodos para detectar problemas durante merges, como comprobar duplicados inesperados en las claves.

- **Manejo de nulos y duplicados avanzado**: Ya cubrimos funciones básicas. En escenarios complejos, podríamos querer, por ejemplo, rellenar nulos con métodos de propagación (ej. fillna(method="ffill") propaga el último valor válido hacia adelante, útil en series temporales). O eliminar duplicados manteniendo cierto criterio (ej. conservar la última aparición: df.drop_duplicates(keep="last")).

Ejemplo de *groupby* y *merge*: supongamos tenemos un DataFrame ventas con columnas producto y cantidad_vendida, y otro DataFrame precios con producto y precio_unitario. Queremos saber ingresos totales por producto. Podríamos agrupar ventas por producto, sumar cantidades, luego unir con precios y calcular ingresos:

```
ventas_por_prod = ventas.groupby("producto")["
    cantidad_vendida"].sum().reset_index()
# reset_index() para convertir el índice de grupo en columna
    normal
ventas_por_prod = ventas_por_prod.merge(precios, on="producto
    ")
ventas_por_prod["ingreso_total"] = ventas_por_prod["
    cantidad_vendida"] * ventas_por_prod["precio_unitario"]
```

Aquí ventas_por_prod tendría columnas producto, cantidad_vendida, precio_unitario e ingreso_total para cada producto. Hemos realizado un *split-apply-combine* con groupby + sum, luego un merge para añadir información de precio, y finalmente una operación vectorizada para obtener el ingreso.

3.4. Operaciones con arrays en NumPy (slicing, vectorización, broadcasting)

NumPy es la biblioteca fundamental para cálculo numérico eficiente en Python. Proporciona el tipo ndarray (N-dimensional array) que es una cuadrícula de valores (todos del mismo tipo) indexados por tuplas de enteros no negativos. A diferencia de las listas de Python, los arrays de NumPy ocupan un bloque contiguo de memoria y se procesan mediante operaciones de bajo nivel optimizadas en C, lo que los hace **mucho más rápidos** para computaciones numéricas intensivas.

Creación de arrays: Podemos convertir listas a arrays usando np.array(lista). También NumPy ofrece funciones para generar arrays comunes, como np.arange(ini, fin, paso) (similar a range pero devuelve array), np.zeros(shape) (array de zeros), np.ones(shape), np.linspace(inicio, fin, num_puntos) (números equidistantes), etc. Ejemplo:

```
import numpy as np
```

```
2
3 a = np.array([1, 2, 3, 4])  # array 1D
4 b = np.array([[1, 2], [3, 4]])  # array 2D (matriz 2x2)
5
6 print(a.shape, b.shape)
7 # Salida: (4,) (2, 2)
```

Aquí a.shape es (4,) indicando un vector de longitud 4, y b.shape es (2,2) indicando 2 filas x 2 columnas.

Indexación y slicing: Los arrays NumPy soportan indexación similar a listas (enteros, slicing con :). Ejemplo: a[0] es 1, a[-1] es 4 (último elemento). Para matrices, b[0,1] accede a fila 0, columna 1 (índice base 0, así que valor 2). Podemos rebanar subarrays: a[:2] da los primeros 2 elementos [1,2], b[:,0] da la primera columna de la matriz [1,3]. Estas operaciones **no crean copias** sino *vistas* de los datos originales (hasta que modifiquemos valores). Por ejemplo,

```
1 sub = a[1:3]  # sub es [2, 3]
2 sub[0] = 99
3 print(a)  # a ahora es [1, 99, 3, 4] debido a la vista
    compartida
```

Conviene tener esto en cuenta para evitar efectos colaterales inesperados; si queremos una copia independiente, podemos usar a[1:3].copy().

Operaciones vectorizadas: Una de las mayores ventajas de NumPy es poder aplicar operaciones aritméticas elemento a elemento sin escribir bucles explícitos en Python. Por ejemplo, a * 2 devuelve [2, 198, 6, 8] si a es el array modificado anterior (multiplica cada elemento por 2). Similarmente, podemos hacer np.sqrt(a) para obtener la raíz cuadrada de cada elemento, o a + a para sumar arrays del mismo tamaño. Estas operaciones están implementadas en C y son muy rápidas, esto es lo que se llama **vectorización**. Al evitar bucles en Python y delegarlos a código compilado, se obtienen grandes mejoras de rendimiento.

Broadcasting: NumPy permite operar arrays de distintas formas en una misma expresión bajo ciertas reglas de compatibilidad, extendiendo implícitamente dimensiones cuando es seguro. Por ejemplo, sumar un escalar a un array (a + 3) suma 3 a cada elemento. O sumar un array 1D a cada fila de un array 2D:

```
1 A = np.array([[1,2,3],
2                [4,5,6]])
3 v = np.array([10, 20, 30])
4 C = A + v
5
6 print(C)
7 # Salida:
8 # [[11 22 33]
```

```
#  [14 25 36]]
```

Aquí A es 2x3, v es de longitud 3. NumPy *broadcast* v para que actúe como dos filas (replicando v en cada fila) y luego realiza la suma. El broadcasting es un mecanismo que **vectoriza operaciones en arrays de diferentes tamaños aprovechando las dimensiones compatibles**, sin hacer copias innecesarias de datos.

Las reglas de broadcasting, en resumen, son: se alinean las formas de los arrays de derecha a izquierda, y se considera que una dimensión de tamaño 1 se puede estirar (replicar) para igualar la del otro array, o una que no existe se toma como 1. Si alguna dimensión no coincide y no es 1 en ninguno de los dos, da error. En el ejemplo, las formas eran (2,3) y (3,) que se considera (1,3); la primera dimensión se estira de 1 a 2, resultando en (2,3) contra (2,3) para la suma.

Esta característica elimina la necesidad de escribir explícitamente bucles de anidación en la mayoría de cálculos con matrices. Además, NumPy ofrece multitud de **funciones matemáticas** universales (*ufuncs* como np.sum, np.mean, np.std, np.dot, etc.) optimizadas. Por ejemplo, np.sum(A) sumará todos los elementos de A, np.mean(A, axis=0) dará el promedio por columnas, np.dot(A, A_transpuesta) computa el producto matricial, etc. También permite generar números aleatorios (np.random), ordenar arrays (np.sort), concatenar, hacer comparaciones vectoriales, y mucho más.

Un breve ejemplo demostrando vectorización y broadcasting juntos:

```
import numpy as np

# Creamos una cuadrícula de coordenadas (x,y)
x = np.linspace(0, 5, 6)  # [0. 1. 2. 3. 4. 5.]
y = np.linspace(0, 5, 6)[:, np.newaxis]  # convertimos y a
    columna 6x1 usando newaxis

distancias = np.sqrt((x - 2)**2 + (y - 3)**2)
print(distancias)
```

Aquí x es (6,) y y es (6,1). Al restar el escalar 2 de x, obtenemos array de forma (6,) representando las diferencias en x. Al restar 3 de y, obtenemos (6,1). Luego, (x-2)**2 tiene forma (6,) y (y-3)**2 forma (6,1). Sumarlos activa broadcasting: el término de x se estira a (6,6) replicando la fila en 6 filas, y el de y se estira a (6,6) replicando la columna en 6 columnas. El resultado distancias será un array 6x6 con la distancia euclídea desde el punto (2,3) para cada combinación de coordenadas x,y de la cuadrícula 0-5. Este tipo de cálculo vectorizado sería mucho más complejo y lento con bucles Python anidados.

3.5. Análisis estadístico básico

NumPy y Pandas incluyen métodos para **estadística descriptiva** rápida:

- Con NumPy: funciones como np.mean(array), np.median(array), np.std(array), np.min(array), np.max(array), percentiles np.percentile(array, q), etc., operan sobre arrays completos o sobre ejes específicos (argumento axis).
- Con Pandas: un DataFrame tiene métodos similares: df.mean(), df.describe(), etc. Por ejemplo, df[."edad"].mean() da la edad promedio, df.describe() proporciona un resumen con count, mean, std, min, quartiles, max para cada columna numérica, de forma conveniente.

Por ejemplo:

```
import numpy as np

datos = np.array([10, 20, 20, 30, 40])
print("Media:", np.mean(datos))
print("Desviación estándar:", np.std(datos))
print("Mediana:", np.median(datos))
print("Percentil 75:", np.percentile(datos, 75))
# Salida:
# Media: 24.0
# Desviación estándar: 11.313708498984761
# Mediana: 20.0
# Percentil 75: 30.0
```

En este caso, la media de los datos es 24, la desviación ∼11.31, la mediana 20, y el 75º percentil es 30. Estas medidas resumen la distribución de valores de forma rápida.

En DataFrames, podríamos usar por ejemplo df["ingreso_total"].sum() para sumar una columna, o df.groupby("ciudad")[."edad"].mean() para obtener la edad promedio por ciudad (combinando agrupación y media, como vimos).

En resumen, NumPy proporciona los fundamentos para manipular y calcular eficientemente sobre datos numéricos en bruto, y Pandas construye sobre NumPy para brindar estructuras tabulares de alto nivel con las que es muy cómodo trabajar en tareas de análisis de datos. Dominar ambas bibliotecas nos permite preparar y entender los datos antes de entrar al mundo de *Machine Learning*. En la siguiente sección, veremos cómo aplicar estos datos limpios y estructurados a algoritmos de aprendizaje automático usando scikit-learn.

4. Visualización de Datos con Matplotlib y Seaborn

Una imagen vale más que mil palabras, y en análisis de datos la **visualización** es clave para entender tendencias, distribuciones y outliers. Python cuenta con múltiples librerías de visualización. La base es **Matplotlib**, que ofrece bajo nivel para crear todo tipo de gráficos estáticos 2D (y 3D simple). Sobre Matplotlib existen librerías de más alto nivel como **Seaborn** (especializada en visualización estadística) y **Plotly** (gráficos interactivos). Aquí nos centraremos en Matplotlib y Seaborn para gráficos estáticos, e introduciremos Plotly para gráficos interactivos.

4.1. Creación de gráficos básicos con Matplotlib

Matplotlib tiene una API de bajo nivel y otra de alto nivel estilo MATLAB a través del módulo matplotlib.pyplot. La forma más común es importar import matplotlib.pyplot as plt y usar funciones como plt.plot, plt.scatter, plt.hist, etc., que actúan sobre una figura y ejes .ªctuales".

- **Gráfico de líneas**: útil para series temporales o funciones matemáticas continuas. Se crea con plt.plot(x, y, formato). Si no se especifican x, usa índices de y. El parámetro formato permite definir color y estilo (ej. 'r–o' significa línea roja punteada con marcadores circulares).
- **Gráfico de dispersión (scatter plot)**: muestra puntos x-y, útil para visualizar relación entre dos variables. Se usa plt.scatter(x, y). Admite opciones como tamaño o color de puntos dependiendo de terceros valores.
- **Histograma**: para distribuciones univariadas. plt.hist(datos, bins=numero_de_bins) agrupa los datos en bins y grafica la frecuencia en cada uno.
- **Gráfico de barras**: ideal para datos categóricos. plt.bar(lista_categorias, valores) dibuja barras para cada categoría. También hay plt.barh para horizontales.

Cada uno de estos gráficos se puede personalizar y combinar. Matplotlib funciona tanto en modo *script* (llamando funciones pyplot) como en una interfaz orientada a objetos (creando objetos Figure y Axes y llamando métodos). Aquí usaremos la primera, más sencilla al inicio.

Ejemplo: supongamos que queremos graficar el crecimiento de una inversión en varios años. Tenemos años en el eje x y capital en el eje y:

```python
import matplotlib.pyplot as plt

anios = [2018, 2019, 2020, 2021, 2022]
capital = [10000, 12000, 14000, 18000, 22000]

plt.plot(anios, capital, 'bo-')  # 'bo-' = círculos azules con línea sólida
plt.xlabel("Año")
plt.ylabel("Capital ($)")
plt.title("Crecimiento de la Inversión")
plt.show()
```

Al ejecutar esto, se abrirá una ventana con un gráfico de línea: eje X de 2018 a 2022, eje Y con el capital, y puntos marcados en azul. La llamada a plt.show() muestra la figura (en entornos interactivos como Jupyter no siempre es necesario, pero en scripts sí). El gráfico tendría puntos en (2018,10000), (2019,12000), ... unidos por segmentos.

Para ilustrar la potencia de Matplotlib, veamos otro ejemplo con más personalización: graficamos dos series en el mismo gráfico y agregamos leyenda:

```python
import numpy as np

t = np.linspace(0, 10, 100)  # 100 puntos de 0 a 10
y1 = np.sin(t)
y2 = np.cos(t)

plt.plot(t, y1, label="sin(t)", color="red")
plt.plot(t, y2, label="cos(t)", color="blue", linestyle="--")
plt.xlabel("Tiempo (s)")
plt.ylabel("Amplitud")
plt.title("Funciones Sinusoidales")
plt.legend()
plt.grid(True)
plt.show()
```

Aquí usamos arrays NumPy t, y1, y2. Dibujamos sin en rojo sólido y cos en azul punteado (linestyle=-"). plt.legend() muestra la leyenda automáticamente usando las etiquetas provistas. plt.grid(True) agrega una rejilla punteada. El resultado sería un gráfico con dos curvas sinusoidales de diferente color, claramente identificadas por la leyenda.

4.2. Personalización de gráficos (etiquetas, títulos, leyendas, estilos)

Matplotlib permite personalizar prácticamente cada aspecto de un gráfico: etiquetas de ejes (xlabel, ylabel), título (title), leyenda (legend), límites

de ejes (xlim, ylim), escala (lineal, logarítmica), anotaciones en puntos específicos (plt.text(x, y, "texto") para colocar texto en coordenadas dadas), agregar flechas, cambiar grosor de líneas, tamaño de marcadores, etc.

En la figura anterior (generada con código similar al ejemplo de sin/cos pero con datos distintos), se observa cómo la adición de **etiquetas, títulos, leyendas y anotaciones** ayuda a describir completamente la información de la gráfica. El título da contexto, los ejes indican qué miden (con unidades), la leyenda identifica cada serie y se ha colocado una anotación textual en un punto de interés (P1(2,6) en color azul). La rejilla facilita seguir los valores. Todo esto mejora la comunicabilidad del gráfico.

Matplotlib tiene estilos predefinidos (similar a plantillas estéticas) que se pueden activar con plt.style.use('nombre_estilo'). Por ejemplo 'ggplot' imita el estilo de ggplot2 de R, 'seaborn' aplica un estilo acorde a Seaborn (que por defecto ya ajusta ciertos parámetros). Probar distintos estilos puede mejorar la apariencia con mínimo esfuerzo.

Además, se pueden crear **subplots** (múltiples gráficos en una misma figura, dispuestos en cuadrícula) usando plt.subplots(nrows, ncols) que devuelve una figura y una matriz de ejes sobre los que plotear por separado. Esto es útil para comparar distintas variables o conjuntos de datos en la misma imagen.

Con Matplotlib podemos crear cualquier gráfico 2D necesario, y aunque la configuración fina puede requerir varias líneas de código, la flexibilidad es enorme. Para simplificar algunos tipos de gráficos estadísticos habituales, podemos apoyarnos en Seaborn.

4.3. Visualización avanzada con Seaborn

Seaborn es una biblioteca construida sobre Matplotlib que proporciona una interfaz de alto nivel para dibujar gráficos estadísticos atractivos con menos código. Seaborn viene con conjuntos de datos de ejemplo y estilos predefinidos que mejoran la estética por defecto (paletas de colores más agradables, estilos de tramas, etc.). Algunos tipos de gráficos que Seaborn facilita:

- **Mapas de calor (heatmaps)**: representación matricial de valores con un mapa de colores. Útil para visualizar correlaciones, matrices de confusión, etc. En Seaborn se usa sns.heatmap(data) pasando una matriz (por ejemplo un DataFrame de correlación).
- **Gráficos de violín (violin plots)**: combinan información de distribución tipo kernel density con un boxplot interno. Útiles para comparar

distribuciones de una variable numérica entre categorías. Se dibujan con sns.violinplot(x=categoria, y=valor, data=datos).

- **Gráficos de caja (box plots) y de caja extendida (boxen plots)**: resumen distribución por cuartiles y outliers. sns.boxplot(...) o sns.boxenplot(...).
- **Gráficos de dispersión con distribuciones (jointplot)**: Seaborn permite hacer gráficos combinados que muestran la dispersión entre dos variables más los histogramas de cada una en los ejes, con sns.jointplot(x, y, data=..., kind="scatter"), o incluso calculando correlaciones.
- **Grids de gráficos (FacetGrid, PairGrid)**: para datos con varias variables, Seaborn facilita trazar múltiples gráficos en cuadrículas condicionados a categorías. Por ejemplo, sns.pairplot(df) grafica pares de variables numéricas entre sí y sus distribuciones diagonales, muy útil para exploración inicial. O un FacetGrid para trazar la misma relación x vs y separada por categorías en subplots diferentes.

Como ejemplo ilustrativo, supongamos que tenemos datos de propinas (tips) que vienen en Seaborn. Podemos hacer un gráfico de violín de las propinas por día de la semana:

```
import seaborn as sns

tips = sns.load_dataset("tips")  # carga dataset ejemplo
sns.violinplot(x="day", y="total_bill", data=tips)
plt.title("Distribución de cuentas por día")
plt.show()
```

Este gráfico mostraría para cada día (Thur, Fri, Sat, Sun en el conjunto de datos) una forma de "violín"que indica la distribución de la cuenta total (total_bill). Los violines combinan la funcionalidad de un boxplot (los cuartiles aparecen como una línea interna) y la estimación de densidad (el ancho del violín refleja cuántas cuentas caen en esa cuantía). Es muy útil para ver si, por ejemplo, los sábados las cuentas suelen ser más altas que los jueves, etc., y también la simetría o asimetría de cada distribución.

Seaborn ajusta la estética automáticamente: por ejemplo, aplica un fondo con cuadrícula suave, colores por defecto agradables, etc. También facilita agregar elementos como intervalos de confianza en gráficos de línea (sns.lineplot calcula automáticamente intervalos si se le da data agrupada), o ajustar modelos de regresión en un scatter (sns.lmplot dibuja puntos y la línea de regresión con banda de confianza).

Otro ejemplo: **mapa de calor de una matriz de correlación** entre variables:

```python
import numpy as np
import seaborn as sns

# Datos simulados
data = np.random.rand(5,5)
mask = np.triu(np.ones_like(data, dtype=bool))  # máscara para triangular superior
ax = sns.heatmap(data, mask=mask, annot=True, cmap="coolwarm")
ax.set_title("Mapa de calor (ejemplo)")
plt.show()
```

Este código genera una matriz 5x5 aleatoria, luego dibuja un heatmap. La máscara mask oculta la parte triangular superior para quizás emular una matriz de correlación (que es simétrica, no necesitamos duplicar valores). annot=True muestra los valores numéricos dentro de cada celda, cmap=çoolwarm.elige un gradiente de azul a rojo pasando por blanco. El resultado es un color por celda que refleja su valor, con números anotados.

En resumen, **Seaborn** nos permite producir visualizaciones más sofisticadas con muy pocas líneas de código, especialmente cuando se trata de exploración de datos estadísticos: distribuciones, relaciones categóricas, etc. Para muchos casos, podríamos hacer lo mismo con Matplotlib, pero Seaborn nos ahorra manejar muchos detalles manualmente (colores, leyendas, cálculos estadísticos).

4.4. Uso de gráficos interactivos con Plotly

Mientras Matplotlib y Seaborn producen gráficos estáticos (imágenes), **Plotly** es una biblioteca que permite crear gráficos interactivos. Con Plotly, los gráficos pueden tener *tooltips* (cuadros emergentes al pasar el mouse mostrando valores), zoom, panning, y se pueden exportar como archivos HTML interactivos o visualizar en notebooks.

Plotly tiene una interfaz de alto nivel llamada plotly.express (px). Podemos usarla de forma similar a Seaborn. Por ejemplo:

```python
import plotly.express as px

fig = px.scatter(tips, x="total_bill", y="tip", color="day", size="size",
                 title="Propinas vs Total de la cuenta")
fig.show()
```

Este código crearía un gráfico de dispersión donde cada punto es una comida del dataset tips, en el eje x la cuenta total y en y la propina, con puntos de diferente color según el día de la semana y tamaño según el tamaño del

grupo de personas. Al renderizar fig.show(), se obtiene un gráfico interactivo: podemos pasar el cursor sobre un punto para ver detalles (total_bill, tip, day, etc.), hacer zoom en una región arrastrando, o activar/desactivar series desde la leyenda.

Plotly admite **más de 40 tipos de gráficos** (barras, líneas, scatter 2D y 3D, mapas geográficos, treemaps, violin, etc.) e incluso gráficos 3D interactivos. Además, permite animaciones (por ejemplo variando un parámetro a través de *frames* de tiempo), lo cual es útil para datos que evolucionan o para presentaciones.

Una fortaleza es que puede mezclar datos numéricos y categóricos fácilmente, generando gráficas integradas. Por ejemplo, crear un gráfico de líneas animado por años, o un diagrama de dispersión con un slider temporal, es muy directo.

Plotly también se integra con herramientas como **Dash** para construir dashboards interactivos web, pero eso escapa al alcance de esta introducción.

En conclusión, para **visualización**:

- Matplotlib: el más bajo nivel, máximo control; hay que configurar manualmente muchas cosas, pero sirve de base.
- Seaborn: alto nivel para visualización estadística, produce gráficos hermosos con mínima configuración; ideal para exploración de datos.
- Plotly: gráficos interactivos listos para la web o presentaciones, con la habilidad de involucrar al usuario en la exploración de los datos de manera dinámica.

Según la necesidad (publicar un informe estático vs. crear una herramienta interactiva), elegiremos la adecuada.

Con los datos limpios (sección 3) y la capacidad de visualizarlos (sección 4), estamos listos para adentrarnos en **Machine Learning**, donde entrenaremos modelos para extraer patrones y hacer predicciones a partir de esos datos.

5. Introducción a Machine Learning con Scikit-learn

El **Aprendizaje Automático (Machine Learning, ML)** es un campo de la inteligencia artificial que permite a las computadoras **aprender de los datos** en lugar de ser explícitamente programadas. Existen varias ramas, pero las tres categorías principales son:

- **Aprendizaje supervisado**: El modelo se entrena con **datos de entrada y salidas esperadas (etiquetas)**. El objetivo es que aprenda una función que, dada una entrada, prediga la salida correcta. Es como aprender con un maestro: se le dan ejemplos correctos. Ejemplos: clasificación (asignar etiqueta de clase, p.ej. spam vs no spam), regresión (predecir un valor numérico, p.ej. precio de una casa).

- **Aprendizaje no supervisado**: El modelo se entrena **solo con datos de entrada, sin etiquetas**. Debe descubrir estructura oculta en los datos por sí mismo. Ejemplos: clustering (agrupar datos similares, p.ej. segmentación de clientes), reducción de dimensionalidad (encontrar representaciones simplificadas de los datos), detección de anomalías.

- **Aprendizaje por refuerzo**: Un agente interactúa con un entorno tomando acciones y recibiendo **recompensas o castigos** según el resultado. El agente aprende por **ensayo y error** a maximizar la recompensa acumulada. Es como entrenar mediante incentivos: sin respuestas çorrectas"directas, sino feedback en forma de puntaje. Ejemplos: juegos (ej. un agente que aprende a jugar ajedrez), control robótico, sistemas de recomendación que se adaptan con la interacción del usuario.

Scikit-learn (sklearn) es una biblioteca poderosa y sencilla para implementar algoritmos de **aprendizaje supervisado y no supervisado** en Python. (Para aprendizaje por refuerzo se utilizan otras librerías especializadas, aunque sklearn nos preparará con fundamentos aplicables).

5.1. Preprocesamiento de datos: StandardScaler, MinMaxScaler, OneHotEncoder

Antes de entrenar modelos, es crucial **preprocesar** los datos. Esto incluye escalar características numéricas, codificar variables categóricas, normalizar distribuciones, etc., para que los algoritmos funcionen mejor. Scikit-learn ofrece un módulo sklearn.preprocessing con transformadores para estas tareas. Tres de los más usados:

- **StandardScaler**: estandariza características numéricas reescalándolas a tener **media 0 y desviación estándar 1**. Esto significa transformar cada valor x en $(x-\mu)/\sigma$. Tras este escalado, las características quedan comparables en escala (importante para algoritmos como redes neuronales, SVM, k-NN, etc. que son sensibles a las magnitudes). Ejemplo: alturas en cm ($\sim$170) y pesos en kg ($\sim$70) quedan en rangos similares tras estandarizar.

- **MinMaxScaler**: escala características numéricas a un **rango [0, 1]** (u otro rango especificado). Transforma linealmente cada valor x en $(x - min)/(max - min)$ del conjunto de entrenamiento. Es útil cuando se quiere valores en un rango acotado (por ejemplo, para algoritmos que requieren input normalizado en [0,1] o para interpretar probabilidades). A diferencia de StandardScaler, no centra en cero ni afecta la forma de la distribución, solo la traslada y escala al rango destino.

- **OneHotEncoder**: codifica variables categóricas en **vectores binarios** (*one-hot vectors*). Convierte cada categoría distinta en una nueva columna indicadora (0/1). Por ejemplo, una columna Çolorçon valores {Rojo", "Verde", .ªzul"} se convierte en tres columnas: Çolor_Rojo", Çolor_Verde", Çolor_Azul", con un 1 en la columna correspondiente al color de esa fila y 0 en las otras. Esto permite a los modelos numéricos trabajar con variables categóricas. Scikit-learn también tiene LabelEncoder (que asigna un número entero a cada categoría) pero OneHot es preferible para variables nominales no ordinales, ya que evita introducir un orden arbitrario entre categorías.

Estos transformadores funcionan ajustando los parámetros en el conjunto de entrenamiento y luego transformando consistentemente los datos de entrenamiento y prueba. Es decir: StandardScaler calcula μ y σ en entrenamiento y los usa para escalar tanto entrenamiento como prueba (y futuros datos); MinMaxScaler calcula min y max en entrenamiento; OneHotEncoder determina las categorías únicas en entrenamiento.

Ejemplo en código:

```
from sklearn.preprocessing import StandardScaler,
    MinMaxScaler, OneHotEncoder

# Datos simulados
import numpy as np

X_num = np.array([[5.1, 20.0],
                  [6.5, 25.0],
                  [5.9, 30.0]])  # Supongamos columnas: [
    longitud(cm), temperatura( grados C)]
cats = np.array([["Rojo"], ["Verde"], ["Azul"]])  # Columna
    categórica

# Escalado estándar
scaler = StandardScaler()
X_num_scaled = scaler.fit_transform(X_num)
print("Media de cada feature:", scaler.mean_)  # media
    aprendida
```

```
print("Escala (desv.std) de cada feature:", scaler.scale_)  # std aprendida

# Escalado MinMax
mms = MinMaxScaler()
X_num_mm = mms.fit_transform(X_num)
print("Minimos:", mms.data_min_, "Maximos:", mms.data_max_)

# One-hot encoding
encoder = OneHotEncoder(sparse_output=False)
cats_encoded = encoder.fit_transform(cats)
print("Categorías:", encoder.categories_)
print(cats_encoded)
```

Supongamos que la salida es algo así:

```
Media de cada feature: [5.8333, 25.0]
Escala (desv.std) de cada feature: [0.593, 5.0]
Minimos: [5.1, 20.0] Maximos: [6.5, 30.0]
Categorías: [array(['Azul', 'Rojo', 'Verde'], dtype=object)]
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
```

Esto indicaría que StandardScaler calculó media ∼5.833 y 25 para las dos columnas numéricas, y desviaciones ∼0.593 y 5; MinMaxScaler tomó min=5.1, max=6.5 para la primera col, min=20, max=30 para la segunda; OneHotEncoder detectó 3 categorías ('Azul','Rojo','Verde') y transformó 'Rojo' -¿[0,1,0], 'Verde' -¿[0,0,1], 'Azul' -¿[1,0,0]. (Notar que el orden alfabético por defecto hizo Azul index 0, Rojo 1, Verde 2, pero el orden final no importa mientras se use consistentemente).

En la práctica, solemos encadenar transformaciones de preprocesamiento junto con el modelo en un **Pipeline** de scikit-learn, para evitar fugas de datos entre entrenamiento y validación, y facilitar la reproducibilidad. Por ejemplo:

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', SomeModel())
])

pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
```

Así, el escalado se ajusta solo con X_train dentro del pipeline y se aplica a X_test correctamente antes de predecir.

5.2. Modelos básicos de clasificación y regresión

Scikit-learn implementa muchos algoritmos de ML listos para usar mediante una API consistente: cada modelo es una clase con métodos .fit(X, y) para entrenar y .predict(X) para predecir. Veamos tres modelos básicos:

- **Regresión Lineal (LinearRegression)**: modelo de regresión supervisado que asume una relación lineal entre las características de entrada y la variable objetivo. Ajusta coeficientes $w_1, ..., w_p$ y una intercepción b para minimizar el error cuadrático medio entre las predicciones y los valores reales. Es el método de *Mínimos Cuadrados Ordinarios*. Por ejemplo, para predecir precios en función de tamaño de casa, la regresión lineal ajusta una recta. Uso: from sklearn.linear_model import LinearRegression. Después de entrenar, podemos examinar model.coef_ (coeficientes) y model.intercept_. Este modelo es simple y rápido, pero puede subestimar relaciones no lineales o interacciones complejas.

- **Regresión Logística (LogisticRegression)**: pese al nombre, es usado principalmente para **clasificación binaria** (aunque scikit-learn soporta multinomial también). Modela la probabilidad de la clase "positivaüsando una función logística (sigmoide) sobre una combinación lineal de las entradas. Es equivalente a un clasificador lineal con salida probabilística. Soporta regularización para evitar sobreajuste. Por ejemplo, para clasificación de email spam/no spam, logistic regresión produce una probabilidad de spam entre 0 y 1 que luego se umbraliza. En sklearn: from sklearn.linear_model import LogisticRegression. Podemos obtener probabilidades con .predict_proba(X). Es un modelo muy utilizado como baseline por su eficiencia y buena interpretabilidad (coeficientes indican influencia de cada feature en la log-odds de la clase).

- **Árbol de Decisión (DecisionTreeClassifier)**: modelo de clasificación (o regresión, hay DecisionTreeRegressor) que aprende una estructura de árbol binario. Cada nodo del árbol aplica una condición (p.ej. "feature i ¡= umbral?") y según la respuesta va a una rama u otra, hasta llegar a una hoja que da la predicción. Los árboles pueden capturar relaciones no lineales y manejar interacciones entre features de forma natural. Son fáciles de interpretar (se pueden visualizar como reglas). Sin embargo, tienden a *sobreajustar* si no se podan, porque pueden crear reglas muy específicas para el conjunto de entrenamiento. Scikit-learn permite limitar la profundidad del árbol, el número mínimo de muestras por hoja, etc., para controlar eso. Uso: from sklearn.tree

import DecisionTreeClassifier. Tras entrenar, podemos visualizarlo o consultar feature_importances_ (una medida de importancia de cada característica en el árbol). Un árbol de decisión puede manejar tanto features numéricas como categóricas (éstas últimas codificadas, pero el árbol aprende particiones por categoría). DecisionTreeClassifier puede hacer clasificación multi-clase de forma natural.

Ejemplo de código entrenando estos modelos en un dataset sencillo:

```
from sklearn.linear_model import LinearRegression,
    LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

# Datos de ejemplo para regresión (predicción de y a partir
    de x)
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 5, 4, 5])

reg = LinearRegression().fit(X, y)
print("Coef:", reg.coef_, "Intercept:", reg.intercept_)
print("Predicción para x=6:", reg.predict([[6]]))

# Datos de ejemplo para clasificación (OR lógico)
X_clf = np.array([[0,0],[0,1],[1,0],[1,1]])
y_clf = np.array([0, 1, 1, 1])  # salida OR

logreg = LogisticRegression().fit(X_clf, y_clf)
tree = DecisionTreeClassifier(max_depth=2).fit(X_clf, y_clf)

print("LogReg predicciones:", logreg.predict(X_clf))
print("Árbol predicciones:", tree.predict(X_clf))
```

Imaginemos la salida:

```
Coef: [0.6] Intercept: 2.8
Predicción para x=6: [6.4]
LogReg predicciones: [0 1 1 1]
Árbol predicciones: [0 1 1 1]
```

Esto indica que la regresión lineal encontró la recta y = 0.6*x + 2.8 que se ajusta a esos puntos de entrenamiento (que parecían casi lineales). La predicción para x=6 es 6.4. En la clasificación OR, tanto la regresión logística como el árbol predijeron correctamente las salidas OR para los 4 casos (en realidad OR es perfectamente linealmente separable excepto el punto [1,1] vs [0,0], logistic likely got it perfecto con un umbral de 0.5 en probabilidades; el árbol también).

Estos son ejemplos muy simples, pero ilustran el uso de la API: .fit(X, y) entrenar, .predict(X) predecir. Scikit-learn unifica esta interfaz para todos sus modelos (SVM, KNN, RandomForest, etc.).

5.3. Evaluación de modelos: train_test_split, cross_val_score, matriz de confusión

Para medir el desempeño de un modelo de ML, no basta con observar qué bien aprendió el conjunto de entrenamiento (podría sobreajustar). Necesitamos evaluar con **datos no vistos**. Por eso es vital dividir el dataset en **entrenamiento** y **prueba** (train/test) o usar validación cruzada.

- **train_test_split**: función práctica de sklearn que divide aleatoriamente los datos en dos subconjuntos (por ejemplo 70 % para entrenar, 30 % para probar). Uso: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42). El parámetro random_state fija la semilla aleatoria para reproducibilidad. Tras entrenar en X_train, y_train, se evalúa el modelo con X_test, y_test calculando métricas como accuracy (para clasificación) o RMSE (para regresión).

- **cross_val_score**: función para realizar **Validación Cruzada (cross-validation)** automática. La validación cruzada consiste en dividir los datos en k pliegues (folds), entrenar el modelo k veces cada vez dejando uno de los folds como validación y entrenando con el resto, y promediando el desempeño. Esto da una estimación más robusta del rendimiento, utilizando de forma eficiente todos los datos para entrenamiento y prueba (cada punto es probado exactamente una vez). En sklearn: scores = cross_val_score(model, X, y, cv=5) entrenará y evaluará con 5-fold CV, retornando 5 scores (por defecto, la métrica model.score que suele ser accuracy para clasificación, R^2 para regresión). Podemos especificar scoring="neg_mean_squared_errorü otras métricas mediante el parámetro scoring. La validación cruzada ayuda a detectar sobreajuste y a seleccionar mejores hiperparámetros, evitando la volatilidad de una única split train/test.

- **Matriz de confusión**: Para problemas de clasificación, una manera detallada de evaluar errores es la **matriz de confusión**. Es una tabla de contingencia donde filas representan las clases reales y columnas las clases predichas (o viceversa). Cada celda (i, j) contiene el número de instancias cuyo verdadero label era la clase i que el modelo clasificó como clase j. En una matriz de confusión ideal (diagonal perfecta),

todos los aciertos están en la diagonal y los errores son las celdas fuera de ella. Por ejemplo, para un clasificador binario, la matriz de confusión es 2x2:

- TP (True Positives): predijo positivo cuando era positivo (esquina [1,1] si definimos pos=Clase1).
- TN (True Negatives): predijo negativo cuando era negativo ([0,0]).
- FP (False Positives): predijo positivo cuando era negativo ([0,1]).
- FN (False Negatives): predijo negativo cuando era positivo ([1,0]).

De aquí derivan métricas como **accuracy** = (TP+TN)/total, **precision** = TP/(TP+FP), **recall** = TP/(TP+FN), **F1** etc. Scikit-learn provee from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score para calcularlas. Visualmente, un *heatmap* de la matriz de confusión puede ayudar a ver en qué clases se confunde más el modelo.

Ejemplo: supongamos que entrenamos un modelo de clasificación multiclase en etiquetas [”gato”, ”perro”, çonejo”]. Obtenemos las siguientes predicciones vs reales:

Reales: gato, perro, conejo, gato, perro, conejo (2 de cada)
Predichos: gato, gato, conejo, conejo, perro, conejo

La matriz de confusión sería:

	Pred: Gato	Pred: Perro	Pred: Conejo
Real: Gato	1	0	1
Real: Perro	1	1	0
Real: Conejo	0	0	2

Interpretando: de 2 gatos reales, 1 fue predicho gato (acierto) y 1 mal predicho conejo; de 2 perros reales, 1 mal predicho gato y 1 acierto perro; de 2 conejos reales, los 2 predichos conejo (aciertos). Este modelo confunde algunos gatos con perros o conejos, etc. Una matriz así ayuda a ver que, por ejemplo, nunca confunde perro con conejo (casilla [Perro, Conejo] = 0), pero confunde gato con conejo.

En sklearn:

```
from sklearn.metrics import confusion_matrix

y_true = ["gato","perro","conejo","gato","perro","conejo"]
y_pred = ["gato","gato","conejo","conejo","perro","conejo"]

cm = confusion_matrix(y_true, y_pred, labels=["gato","perro",
    "conejo"])
print(cm)
# Salida:
# [[1 0 1]
```

```
#   [1  1  0]
#   [0  0  2]]
```

Que coincide con la tabla descrita (las filas siguen el orden [gato, perro, conejo]). A partir de esto, podríamos calcular accuracy = (1+1+2)/6 ≈ 0.67, etc.

En la práctica, el flujo típico es: dividir datos, entrenar modelo, predecir en test, calcular métrica(s). Por ejemplo:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2)
model = DecisionTreeClassifier(max_depth=5).fit(X_train,
    y_train)
y_pred = model.predict(X_test)
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)
```

Luego inspeccionar acc y cm. Para modelos de regresión, usamos métricas como error cuadrático medio (mean_squared_error) o R^2 (r2_score).

Finalmente, a menudo hacemos **ajuste de hiperparámetros** (parámetros del modelo que no aprende automáticamente, como max_depth de un árbol o C de una SVM) mediante búsqueda en rejilla (*GridSearchCV*) con validación cruzada. Sklearn tiene GridSearchCV y RandomizedSearchCV que automatizan entrenar múltiples modelos con distintas combinaciones de hiperparámetros y seleccionar el mejor según una métrica en validación.

Con estos fundamentos, estamos equipados para entrenar modelos simples, prepararlos correctamente y evaluar su rendimiento. Como ejemplo integrador: podríamos cargar un dataset (por ejemplo, iris de sklearn), preprocesarlo (si necesitara), entrenar un árbol de decisión para clasificar la especie de flor, y evaluar su accuracy con validación cruzada, revisando la matriz de confusión para ver qué especies confunde más.

6. Redes Neuronales y Deep Learning con TensorFlow

Los algoritmos anteriores son excelentes para muchos problemas, pero algunas tareas muy complejas (como visión por computadora, procesamiento de lenguaje natural, jugar videojuegos a nivel superhumano) han sido revolucionadas por el **Deep Learning (aprendizaje profundo)**. El aprendizaje profundo utiliza **Redes Neuronales Artificiales** con múltiples capas para aprender representaciones de alto nivel de los datos.

6.1. Conceptos fundamentales de redes neuronales artificiales

Una **red neuronal artificial (RNA)** se inspira (muy vagamente) en el cerebro humano: consiste en una colección de unidades simples interconectadas llamadas **neuronas artificiales** organizadas en capas. Cada neurona realiza una operación matemática simple: recibe entradas (valores numéricos) ya sea de datos originales o de la salida de otras neuronas, calcula una suma ponderada de esas entradas más un sesgo, y luego aplica una **función de activación** no lineal (como sigmoide, ReLU, tanh) al resultado, produciendo una salida. Esa salida alimentará a neuronas de la capa siguiente, y así sucesivamente.

Las redes suelen tener una capa de entrada (que recibe los datos iniciales), **capas ocultas** internas donde ocurre el procesamiento, y una capa de salida que entrega el resultado (por ejemplo, para clasificación, la capa de salida podría tener tantas neuronas como clases, que produzcan probabilidades). Cuando hay **múltiples capas ocultas**, se habla de *aprendizaje profundo* (deep learning), de ahí el nombre, ya que "profundo"se refiere a tener muchas capas. Una red con al menos dos capas ocultas ya se considera "profunda".

Durante el **entrenamiento**, la red aprende los **pesos** (parámetros) asociados a cada conexión entre neuronas ajustándolos iterativamente para minimizar un cierto **coste o función de pérdida** (por ejemplo, el error cuadrático medio en regresión o el entropía cruzada en clasificación). El algoritmo típico es **backpropagation** (propagación hacia atrás del error) combinado con un método de optimización como **gradiente descendente** (y variantes como Adam, RMSprop, etc.). Básicamente, se calcula el error del output vs la etiqueta deseada, y ese error se distribuye hacia atrás por la red calculando derivadas parciales (gradientes) de los pesos, para luego actualizar los pesos en la dirección que reduce el error.

Al entrenar, alimentamos muchos ejemplos (batches de datos) repetidamente (muchas **épocas** de entrenamiento). Inicialmente los pesos son aleatorios, pero con cada iteración la red va mejorando su capacidad predictiva en el conjunto de entrenamiento. Finalmente esperamos que generalice bien a datos nuevos (con suficiente regularización, cantidad de datos, etc.).

Las **redes neuronales profundas** han demostrado ser capaces de aproximar funciones muy complejas. Por ejemplo, **redes convolucionales (CNN)** especializadas en imágenes aprenden detectores de bordes en primeras capas, luego formas simples, luego patrones complejos, y finalmente pueden reconocer objetos completos. **Redes recurrentes (RNN)** o sus mejoras (LSTM, GRU, y las modernas **Transformers**) pueden modelar secuencias y han logrado enormes avances en traducción automática y modelos de lenguaje.

En resumen, una RNA es un modelo con muchos parámetros (pesos) que aprende representaciones internas de los datos mediante composición de funciones en múltiples capas. Aunque matemáticamente equivalen a componer muchas funciones lineales y no lineales, conceptualmente se pueden pensar como *cajas* que van refinando los datos paso a paso. Una red profunda con suficientes neuronas puede aproximar casi cualquier relación matemática dada suficiente entrenamiento, de ahí su poder, aunque también su riesgo de sobreajustar si es demasiado compleja para los datos disponibles.

6.2. Creación de modelos de aprendizaje profundo con Keras

TensorFlow es una de las principales bibliotecas de deep learning (desarrollada por Google). Incluye como sub-API a **Keras**, que originalmente era independiente pero ahora está integrada como la interfaz de alto nivel de TensorFlow. Keras nos permite definir y entrenar redes neuronales en muy pocas líneas, abstrayendo muchos detalles de bajo nivel. Su énfasis está en ser **fácil de usar, modular y extensible**.

Para crear un modelo en Keras, solemos seguir estos pasos:

1. **Definir la arquitectura**: usando la API Secuencial (capas apiladas linealmente) o la API Funcional (más flexible, permite grafos arbitrarios de capas). En la mayoría de casos simples, tf.keras.Sequential es suficiente.

2. **Compilar el modelo**: se especifica el optimizador (p. ej. 'adam'), la función de pérdida (p. ej. 'mse' para regresión, 'binary_crossentropy' para clasificación binaria, 'sparse_categorical_crossentropy' para multiclase con labels enteras) y opcionalmente métricas a monitorear (p. ej. 'accuracy').

3. **Entrenar el modelo**: llamar model.fit(X_train, y_train, epochs=n, batch_size=m, validation_data=(X_val, y_val)). Esto inicia el proceso iterativo de optimización. Podemos ver el progreso de la pérdida y las métricas por época, y el uso de un conjunto de validación ayuda a monitorear si el modelo generaliza o está sobreajustando (si la pérdida de validación empieza a empeorar mientras la de entrenamiento mejora, es signo de sobreajuste).

4. **Evaluar y predecir**: usar model.evaluate(X_test, y_test) para obtener métricas finales en datos de prueba, y model.predict(nuevos_datos) para obtener predicciones sobre nuevos ejemplos.

Veamos un ejemplo concreto: queremos aproximar la función $y = 2x^2$ con una red neuronal sencilla.

```
import tensorflow as tf
import numpy as np

# Generamos datos sintéticos
X = np.linspace(-10, 10, 1000)
y = 2 * (X**2)

# Definir modelo secuencial
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu', input_shape=(1,)),  # capa oculta 1 con 10 neuronas
    tf.keras.layers.Dense(10, activation='relu'),  # capa oculta 2 con 10 neuronas
    tf.keras.layers.Dense(1)  # capa de salida (regresión a 1 valor)
])

# Compilar modelo
model.compile(optimizer='adam', loss='mse')

# Entrenar modelo
model.fit(X, y, epochs=50, batch_size=32, verbose=0)  # verbose=0 para no imprimir cada epoch

# Evaluar en algunos puntos
test_X = np.array([[-3], [0], [4]])
pred = model.predict(test_X)
print("Predicciones:", pred.flatten())
```

Esta red tiene 2 capas ocultas densas (fully connected) con ReLU y 1 de salida lineal. Probablemente tras entrenamiento, las predicciones estarán cercanas a [18, 0, 32] para entradas -3, 0, 4 (ya que $2*x^2 = 18, 0, 32$). Obviamente es un problema trivial que podríamos resolver fácilmente sin red, pero sirve para ilustrar.

Para un problema de **clasificación** (digamos MNIST, dígitos escritos a mano 28x28 pixeles, 10 clases), podríamos definir:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),  # convertir imagen 28x28 a vector de 784
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')  # 10 neuronas de salida con softmax
])

model.compile(optimizer='adam', loss='
```

```
    sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, validation_split=0.1)
```

Aquí usamos una capa Flatten para transformar la imagen en un vector, luego una capa oculta de 128 neuronas ReLU, y la salida de 10 neuronas con softmax que da una distribución de probabilidad sobre las 10 posibles clases. La pérdida 'sparse_categorical_crossentropy' se usa porque las etiquetas y_train son enteros 0-9 (si tuviéramos one-hot encoding usaríamos 'categorical_crossentropy'). Tras entrenar, podríamos evaluar con model.evaluate(X_test, y_test) para ver la precisión final.

Keras nos permite agregar fácilmente nuevas capas, cambiar activaciones, etc. También proporciona muchas capas especializadas: Convolucionales (Conv2D), Pooling (MaxPooling2D), Recurrentes (LSTM, GRU), Dropout (para regularización), y podemos diseñar arquitecturas complejas (ej. CNN para imágenes, RNN para texto, incluso modelos multimodales).

Además, Keras hace que el **ajuste de hiperparámetros** como número de neuronas, capas, función de activación, sea relativamente fácil: es cuestión de cambiar parámetros en la definición. Herramientas como Keras Tuner pueden automatizar la búsqueda de buenas configuraciones.

7. Entornos de Simulación para Aprendizaje por Refuerzo con OpenAI Gym

Finalmente, llegamos al **Aprendizaje por Refuerzo (RL)**, donde un agente aprende a tomar decisiones mediante interacción con un entorno para maximizar recompensas acumuladas. A diferencia del aprendizaje supervisado, en RL no se dan pares entrada-salida deseada, sino que el agente descubre por sí mismo qué acciones le rinden más recompensa a largo plazo, a través de prueba y error. Este campo ha permitido grandes logros, como agentes que superan humanos en videojuegos, Go o controlan robots complejos.

OpenAI Gym (ahora Gymnasium en versiones más recientes) es una biblioteca que proporciona una colección de **entornos simulados estándar** y una API unificada para interactuar con ellos. En otras palabras, Gym ofrece diversos "juegos.º "problemas"listos (p. ej. controlar un carrito con un palo invertido, jugar Atari, resolver laberintos, etc.) donde podemos probar algoritmos de RL sin tener que programar la física o lógica del entorno. Esto nos permite centrarnos en implementar el agente.

7.1. Instalación y configuración de OpenAI Gym

Gym se instala fácilmente via pip: pip install gym. Este comando instala el núcleo básico con entornos clásicos como control de un péndulo, CartPole, MountainCar, etc. Para entornos adicionales (Atari, robóticos) se instalan extras, e.g. pip install gym[atari]. En nuestro caso, con el básico es suficiente para empezar.

Una vez instalado, el uso típico es:

```
import gym

env = gym.make("CartPole-v1")
```

Esto crea un entorno llamado *CartPole-v1*, un problema clásico donde un poste está equilibrado sobre un carrito y debemos mover el carrito izquierda/derecha para que el poste no caiga. Gym tiene muchos IDs de entornos (como "MountainCar-v0", .Acrobot-v1", etc.).

7.2. Espacios de observación y acción en entornos simulados

Cada entorno Gym define:

- Un **espacio de observación** (*observation space*): la forma y tipo de datos que representa el estado del entorno devuelto al agente. Puede ser un vector continuo (por ejemplo, en CartPole es un vector de 4 floats: posición y velocidad del carro, ángulo y velocidad angular del palo), o una imagen pixelada (Atari envía frames de 210x160x3), o un espacio discreto (p. ej. en un laberinto podría ser la posición representada por un número). Gym usa objetos como spaces.Box (para espacios continuos, con límites) o spaces.Discrete(n) (para un entero de 0 a n-1) para formalizar esto.

- Un **espacio de acción** (*action space*): define qué acciones puede tomar el agente. Puede ser discreto (un número que representa, por ejemplo, 0 = izquierda, 1 = derecha), o continuo (por ejemplo, una fuerza real entre -1 y 1). Por ejemplo, en CartPole es Discrete(2) porque solo hay dos acciones: empujar carrito a la izquierda o a la derecha. En MountainCar, la acción también es discreta (acelerar izq, nada, o der). En Pendulum-v0, la acción es continua (un torque entre -2 y 2).

Gym provee propiedades env.observation_space y env.action_space que describan estos. Por ejemplo, print(env.observation_space, env.action_space)

en CartPole muestra algo como Box(-4.8 - Inf, 4.8 + Inf) Discrete(2) (indicando rangos de observaciones y 2 acciones posibles).

El agente en cada instante recibe una **observación** (estado) del entorno y debe devolver una **acción**. Tras ejecutar la acción, el entorno evoluciona a un nuevo estado y devuelve una **recompensa** inmediata y un indicador de si el episodio terminó (por ejemplo, en CartPole termina si el palo cae demasiado o se alcanzó cierto tiempo máximo).

7.3. Implementación de agentes básicos con políticas aleatorias

Para familiarizarnos, podemos implementar un agente aleatorio, que simplemente toma acciones al azar, y ver cómo interactúa con Gym. Esto no resolverá el entorno, pero muestra la mecánica básica:

```
import gym

env = gym.make("CartPole-v1")
obs = env.reset()  # obtener estado inicial
total_reward = 0

for t in range(1000):
    action = env.action_space.sample()  # elegir acción
    aleatoria válida
    obs, reward, done, info = env.step(action)  # ejecutar
    acción
    total_reward += reward
    if done:
        break

env.close()
print("Recompensa total obtenida:", total_reward)
```

En este código:

- env.reset() inicializa el entorno y nos da la primera observación (estado inicial). En CartPole es un vector de 4 floats.
- Dentro del bucle, env.action_space.sample() elige uniformemente una acción válida (0 o 1 aleatorio).
- env.step(action) aplica la acción al entorno. Retorna cuatro cosas: la siguiente observación (obs), la recompensa inmediata (reward), un booleano done indicando si el episodio terminó, y un diccionario info (información adicional diagnóstica que generalmente podemos ignorar al entrenar agentes).

- Acumulamos la recompensa para saber cómo fue (aunque un agente aleatorio usualmente tendrá bajo total). Si done es True, salimos del loop (episodio terminado).

En entornos episódicos como CartPole, cada episodio tiene un inicio y un fin; después de done, debemos hacer env.reset() para iniciar un nuevo episodio si queremos continuar.

El agente aleatorio en CartPole típicamente durará pocos timesteps antes de perder (palo cae), acumulando recompensas bajas (CartPole da +1 por paso de tiempo que el palo siga en pie, con un máximo de 500). Un agente entrenado puede llegar a 500 sin caer.

Podemos usar Gym también para visualizar entornos: muchos entornos tienen env.render() que muestra una animación. En CartPole abre una ventana con el carrito y el palo. (En entornos headless, no hace nada). Para propósitos educativos, a veces se visualiza el comportamiento tras entrenar para ver qué aprendió el agente.

7.4. Evaluación de desempeño de los agentes

En RL, la evaluación suele hacerse midiendo la **recompensa total promedio** que el agente consigue en un episodio (o varias), o si cumple cierta tarea. Por ejemplo, en CartPole podríamos evaluar promedio de episodios exitosos o duración promedio.

No hay una función de "score."[en] Gym dado que la métrica depende del entorno. Uno usualmente entrena el agente (ya sea con Q-learning, Policy Gradients, Deep Q Networks, etc.) y periódicamente simula episodios completos para ver la recompensa media. Si alcanza cierto umbral (ej. en CartPole, $\geq$ 475 de promedio en 100 episodios), se considera resuelto.

Para un agente aleatorio, podríamos ejecutar varias corridas del código anterior y promediar total_reward. Veríamos que es muy bajo comparado con agentes inteligentes.

7.5. Integración con TensorFlow y PyTorch para agentes más avanzados

Para resolver entornos de Gym más difíciles, se suelen usar algoritmos de RL que implican entrenar políticas o funciones de valor. Muchos de estos usan redes neuronales para aproximar la política (mapa de estado-¿acción) o el valor (estado-¿retorno esperado).

Es aquí donde entra la integración con frameworks como TensorFlow o PyTorch. Por ejemplo, para implementar **Deep Q-Network (DQN)**, usa-

mos una red neuronal (p. ej. en Keras) que recibe el estado y predice valores Q para cada acción, y entrenamos esa red usando las transiciones de experiencia recolectadas del entorno. Para **Policy Gradient** (como REINFORCE o Actor-Critic), usamos TensorFlow/PyTorch para representar la política y ajustar sus parámetros a través del gradiente de la recompensa esperada.

OpenAI Gym en sí no tiene dependencias de TF/PyTorch, es solo la interfaz del entorno. Nosotros escribimos el bucle de entrenamiento RL, y para decisiones de acción en agentes inteligentes normalmente hacemos:

```
action = policy_network.predict(obs.reshape(1,-1))
```

y luego tomamos la acción que maximice cierta probabilidad o valor. Y entrenamos policy_network con las recompensas obtenidas.

Existen librerías de más alto nivel que integran con Gym, por ejemplo **Stable Baselines3** (en PyTorch) que tiene implementaciones listas de DQN, PPO, etc. y solo le pasas env de Gym y entrena. O **TF-Agents** de TensorFlow. Pero para tener idea:

- Con TensorFlow/Keras: podríamos usar model = tf.keras.Sequential([...]) para definir la política o Q-net, y entrenarla con model.fit adaptado (o manualmente computing gradients via tape en TF2). De hecho, hay tutoriales oficiales de TensorFlow mostrando cómo entrenar un agente Actor-Critic en CartPole.

- Con PyTorch: definimos una red como clase nn.Module y entrenamos en loop calculando pérdidas y haciendo loss.backward(); optimizer.step().

Un ejemplo a alto nivel: implementar **Policy Gradient REINFORCE** para CartPole usando Keras: se definiría una red con entrada estado(4,) y salida 2 neuronas softmax (para prob de acción izq/der). En cada episodio, generaríamos una secuencia de estados, acciones tomadas (según esas probabilidades), y recompensas obtenidas. Luego calcularíamos el retorno acumulado de cada paso (discounted sum of future rewards) y ajustaríamos los pesos de la red para que las acciones que llevaron a altas recompensas incrementen su probabilidad (multiplicando log-probabilidades por retorno, etc.). Se usaría un optimizador como Adam para actualizar. Esto es un poco complejo de implementar desde cero, pero con TF es factible, siguiendo pseudocódigo de libros de RL.

Gym + TensorFlow/PyTorch nos permite entrenar agentes para entornos complejos. Por ejemplo, OpenAI publicó Gym Retro para juegos de consola clásicos, y usando redes profundas con RL, la gente ha entrenado agentes que juegan Sonic, Mario, etc. con éxito.

En esta introducción, el detalle de algoritmos RL específicos es mucho, pero al menos se muestra cómo **Gym facilita la interacción con entornos** y cómo podemos enchufar un modelo de deep learning para decidir acciones.

Vale destacar que hoy día, librerías dedicadas como Stable Baselines 3 (Python) o Ray RLlib, o dopamine (de Google) ya tienen muchos componentes listos. Pero aprender a codear un agente sencillo con Gym es muy instructivo.

Recapitulando, OpenAI Gym proporciona entornos estándares con los que podemos: resetear, dar acciones, obtener observaciones y recompensas. Con eso, implementamos el ciclo de RL. Evaluamos al agente viendo su recompensa promedio. Y si queremos lograr agentes avanzados, integramos redes neuronales de TensorFlow/PyTorch para que aprendan políticas eficientes.

Por ejemplo, un agente entrenado con Deep Q-Learning en el entorno **MountainCar-v0** aprenderá a oscilar el coche para tomar impulso y finalmente subir la colina (algo que un agente aleatorio nunca logra). O en **LunarLander-v2**, un buen agente aprenderá a controlar cohetes para aterrizar suavemente. Estos logros son fruto de combinar Gym (simulación) con algoritmos de RL (valor o política) optimizados con herramientas de deep learning.

Conclusión

Hemos recorrido desde los fundamentos de Python (para sentar bases sólidas en el lenguaje) pasando por la manipulación de datos (indispensable para preparar entradas de algoritmos), la visualización (clave para entender y presentar datos), hasta la creación de modelos de *Machine Learning* tradicionales con scikit-learn y modernos con redes neuronales profundas en TensorFlow, finalizando con un vistazo al aprendizaje por refuerzo y cómo experimentarlo en entornos simulados con OpenAI Gym.

Con este conocimiento, un principiante en Python podrá comprender códigos de análisis de datos y aprendizaje automático, desarrollar pequeños proyectos de predicción o clasificación, e incluso dar sus primeros pasos en entrenamiento de agentes inteligentes. El siguiente paso natural sería profundizar en cada área con proyectos prácticos: por ejemplo, construir un clasificador de imágenes con una CNN, o un agente que resuelva CartPole.

¡El campo de la programación en Python para ciencia de datos y aprendizaje por refuerzo es amplio y emocionante, y con estos fundamentos estás listo para explorarlo más a fondo y aprender practicando! ¡Éxitos en tu camino de aprendizaje por refuerzo y más allá!