

12

Introducción a numpy y pandas

En este capítulo exploraremos cómo utilizar **numpy** y **pandas**, dos de las bibliotecas más esenciales en el ecosistema de Python para la ciencia de datos y el análisis numérico. La biblioteca **numpy** permite trabajar con vectores y matrices multidimensionales de forma eficiente, realizando operaciones matemáticas y estadísticas avanzadas sobre grandes volúmenes de datos. **pandas**, construida sobre **numpy**, facilita la manipulación de datos estructurados (especialmente tabulares) y la importación/exportación de los formatos más habituales (CSV, JSON, Excel, Parquet, etc.). Ambas son herramientas fundamentales en inteligencia artificial, aprendizaje máquina y análisis de datos, y aparecen continuamente en los flujos de trabajo habituales de la ciencia de datos (análisis de registros, detección de anomalías, perfilado de series, etc.). La ciberseguridad es uno de los dominios donde estas técnicas resultan útiles, por ejemplo el análisis de *logs*, la detección de anomalías o el perfilado de tráfico.

12.1. Biblioteca numpy

La biblioteca **numpy** (*Numerical Python*) proporciona una estructura de datos eficiente para trabajar con grandes volúmenes de datos numéricos. Su objeto central es el vector multidimensional **ndarray**. A diferencia de las listas de Python, que son heterogéneas y almacenan referencias a objetos Python (con la sobrecarga que ello supone), los **ndarray** son **homogéneos** (todos los elementos son del mismo tipo) y se almacenan en un bloque contiguo de memoria con la representación nativa C del tipo. Esto permite operaciones **vectorizadas** – que actúan sobre todo el array de una vez – implementadas en C optimizado, lo que es órdenes de magnitud más rápido que un bucle Python equivalente.

12.1.1. El objeto ndarray

Para crear un array a partir de una lista usamos `np.array`:

```
import numpy as np

vector = np.array([1, 2, 3, 4])
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(vector * 2)          # [2 4 6 8]      operacion vectorizada
print(matriz[1, 2])        # 6              acceso fila 1, columna
↪      2
print(np.mean(vector))     # 2.5
```

Cada `ndarray` tiene tres atributos clave:

- `arr.shape`: tupla con las dimensiones (e.g., (3, 4) para una matriz 3×4).
- `arr.dtype`: tipo de los elementos (`int64`, `float64`, etc.).
- `arr.ndim`: número de dimensiones.

12.1.2. Tipos de datos (dtype)

Los tipos de `numpy` extienden los tipos de Python con anchuras fijas, por motivos de rendimiento y memoria. Esta es la conexión natural con lo discutido en el capítulo 8: los modelos de aprendizaje máquina suelen entrenarse en `float32` o `float16` por motivos de velocidad y memoria.

Tipo	Tamaño	Ejemplos de uso
<code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>	1, 2, 4, 8 bytes	Contadores, índices, identificadores
<code>uint8</code> , <code>uint16</code> , ...	idem	Bytes (<code>uint8</code>), píxeles, paquetes de red
<code>float16</code> , <code>float32</code> , <code>float64</code>	2, 4, 8 bytes	Cálculo numérico (entrenamiento, inferencia)
<code>complex64</code> , <code>complex128</code>	8, 16 bytes	Procesado de señal, FFT
<code>bool_</code>	1 byte	Máscaras booleanas
<code>datetime64</code> , <code>timedelta64</code>	8 bytes	Series temporales

Tabla 1: Tipos de datos más comunes en `numpy`.

El `dtype` se puede especificar al crear el array:

```
import numpy as np

a = np.array([1, 2, 3], dtype=np.int32)
b = np.zeros(1000, dtype=np.float32)      # 4 KiB en lugar
↪      de 8 KiB
print(a.dtype, a.nbytes)                  # int32 12
print(b.dtype, b.nbytes)                  # float32 4000
```

12.1.3. Operaciones y broadcasting

Las operaciones aritméticas se aplican *elemento a elemento* y devuelven un nuevo array:

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
print(a + b)      # [11 22 33]
print(a * b)      # [10 40 90]
print(a ** 2)     # [1 4 9]
```

El *broadcasting* es la regla que permite combinar arrays de formas distintas sin replicar datos en memoria. El caso más simple es operar un array con un escalar: el escalar se “difunde” a todos los elementos:

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(m + 10)
# [[11 12 13]
#  [14 15 16]
#  [17 18 19]]
```

Cuando los arrays tienen formas distintas, *numpy* alinea las dimensiones por la derecha y, en cada eje, exige que coincidan o que una de ellas valga 1. Si se cumple la regla, la dimensión de tamaño 1 se “expande” virtualmente. Por ejemplo:

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) # (3, 3)
v = np.array([1, 0, -1])                       # (3,)

print(m + v)  # v se aplica a cada fila
# [[2 2 2]
#  [5 5 5]
#  [8 8 8]]

c = np.array([[10], [20], [30]]) # (3, 1)
print(m + c)  # c se aplica como columna
# [[11 12 13]
#  [24 25 26]
#  [37 38 39]]
```

El *broadcasting* no copia datos: el array más pequeño se utiliza con un patrón de *strides* adecuado, lo que es muy eficiente.

12.1.4. Indexación, rebanado y máscaras

Los *ndarray* admiten indexación y rebanado igual que las listas, pero generalizado a múltiples dimensiones. Además soportan dos formas avanzadas: **indexación con arrays de índices** y **máscaras booleanas**.

```
a = np.array([10, 20, 30, 40, 50])

# Indexacion clasica
print(a[1])          # 20
print(a[1:4])        # [20 30 40]
print(a[::-1])       # [50 40 30 20 10]

# Indexacion con array de indices
indices = [0, 2, 4]
print(a[indices])    # [10 30 50]

# Mascara booleana
print(a[a > 25])     # [30 40 50]
```

Para arrays bidimensionales, separamos los índices con coma:

```
m = np.arange(12).reshape(3, 4)    # 3 filas x 4 columnas
print(m[1, 2])                    # 6
print(m[:, 0])                    # primera columna
print(m[0, :])                    # primera fila
print(m[m > 5])                   # [6 7 8 9 10 11]
```

12.1.5. Reorganización: reshape, transpose, flatten

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(m.reshape(9))              # [1 2 3 4 5 6 7 8 9]
print(m.flatten())               # idem (siempre devuelve copia)
print(m.T)                       # transpuesta
print(m.reshape(-1, 3))         # -1 = "deduce esta dimension" ->
↪ (3, 3)
```

Para combinar arrays existen `np.concatenate`, `np.hstack` (horizontal), `np.vstack` (vertical) y `np.stack` (apila por una nueva dimensión):

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(np.concatenate((a, b)))    # [1 2 3 4 5 6]
print(np.vstack((a, b)))         # [[1 2 3]
                                #   [4 5 6]]
```

12.1.6. Funciones vectorizadas útiles

Más allá de las operaciones aritméticas, **numpy** ofrece muchas funciones que evitan bucles explícitos. Las más usadas:

- ↪ ■ Estadísticas: `np.mean`, `np.median`, `np.std`, `np.var`, `np.sum`, `np.prod`, `np.min`, `np.max`, todas con parámetro `axis=` para operar a lo largo de una dimensión.
- Matemáticas: `np.log`, `np.exp`, `np.sqrt`, `np.sin`, `np.cos`, etc.
- Búsqueda: `np.argmax`, `np.argmin` (índices del extremo), `np.argsort` (índices que ordenan), `np.where` (operador condicional vectorizado), `np.unique` (valores únicos, opcionalmente con frecuencias).
- Álgebra lineal en el submódulo `np.linalg`: `inv` (inversa), `det` (determinante), `eig` (autovalores), `solve` (sistemas), `norm` (normas).

```

a = np.array([3, 1, 4, 1, 5, 9, 2, 6])

print(np.argmax(a))                # 5 (índice del 9)
print(np.unique(a, return_counts=True))
# (array([1, 2, 3, 4, 5, 6, 9]), array([2, 1, 1, 1, 1, 1, 1]))
↪ )

print(np.where(a > 3, a, 0))        # ceros donde no, valor
↪ original donde si
# [0 0 4 0 5 9 0 6]

```

Vectoriza siempre que puedas

`np.apply_along_axis(funcion_python, eje, arr)` existe pero ejecuta la función en Python una vez por fila/columna, lo que pierde casi toda la ventaja de *numpy*. Antes de recurrir a él, comprueba si tu cálculo se puede expresar con operaciones vectorizadas, `np.where`, máscaras o álgebra lineal: la diferencia de rendimiento puede ser de uno o dos órdenes de magnitud.

12.1.7. Eficiencia: vistas vs copias

Algunas operaciones devuelven una **vista** (los datos no se duplican; modificar la vista modifica el original); otras devuelven una **copia**. Es importante saber distinguirlas:

```

a = np.array([1, 2, 3, 4, 5])

vista = a[1:4]          # Slicing basico -> VISTA
vista[0] = 99
print(a)                # [ 1 99  3  4  5]    <- afectado!

copia = a[[0, 2, 4]]    # Indexacion con array -> COPIA
copia[0] = -1
print(a)                # [ 1 99  3  4  5]    <- no afectado

```

Si necesitas una copia explícita: `a.copy()`. Si no estás seguro, comprueba con `arr.base is None` (es `None` si `arr` es propietario de los datos).

12.1.8. Generación de números pseudoaleatorios

numpy ofrece la API moderna basada en `np.random.default_rng`, que es la recomendada desde numpy 1.17. Crea un *generador* con una semilla y se usan sus métodos para producir números:

```
import numpy as np

rng = np.random.default_rng(seed=42)    # Generador
reproducible

print(rng.random(5))                    # 5 floats en [0, 1)
print(rng.integers(0, 100, size=5))      # 5 enteros en [0,
100)
print(rng.normal(loc=5, scale=2, size=5)) # normal mu=5
sigma=2
print(rng.choice([10, 20, 30], size=4)) # muestreo con
reemplazo
```

La API antigua (`np.random.seed`, `np.random.rand`, `np.random.randint`, `np.random.normal`) sigue funcionando pero no se recomienda en código nuevo: la nueva API tiene mejor calidad estadística (PCG64 vs Mersenne Twister) y aísla el estado en el objeto `rng` (importante en programas concurrentes).

numpy.random no es criptográficamente seguro

Igual que el módulo `random` del cap. 7, `numpy.random` usa generadores deterministas pensados para simulación, no para criptografía. Para claves, tokens o cualquier valor que deba permanecer impredecible, usa `secrets`.

12.1.9. Interoperabilidad con bytes y ficheros binarios

Dado que a menudo trabajamos con datos textuales y registros, conviene mencionar tres funciones de `numpy` que conectan con el cap. 11:

- `np.frombuffer(data, dtype=...)`: interpreta un objeto `bytes` como un `ndarray` del tipo dado, **sin copiar**. Ideal para parsear paquetes de red u otras estructuras binarias.
- `np.fromfile(path, dtype=..., count=...)`: lee directamente un fichero binario.
- `arr.tobytes()`: serializa un array a una cadena de bytes.

```
# Interpretar 4 bytes como 4 muestras uint8
datos = b"\x01\x02\x03\x04"
muestras = np.frombuffer(datos, dtype=np.uint8)
print(muestras)           # [1 2 3 4]

# Volver a bytes
print(muestras.tobytes()) # b'\x01\x02\x03\x04'
```

Para guardar y cargar arrays preservando el `dtype` y la forma, `np.save` y `np.load` usan un formato propio (`«.npy»`) más seguro y eficiente que `pickle` (recuerda la advertencia del cap. 11: pasa siempre `allow_pickle=False` a `np.load` si los datos no son tuyos).

12.1.10. Otros tipos: complejos, fechas, estructurados

Números complejos. Soportados nativamente en `numpy` con sintaxis a `bj+`:

```
z = np.array([1 + 2j, 3 + 4j, 5 + 6j])
print(np.real(z))    # [1. 3. 5.]
print(np.imag(z))    # [2. 4. 6.]
print(np.abs(z))     # [2.236..., 5., 7.810...]    modulo
print(np.angle(z))   # argumento (radianes)
```

```
fechas = np.array(['2025-03-01', '2025-03-02', '2025-03-03'],
                  dtype='datetime64[D]')
print(fechas + np.timedelta64(10, 'D'))
# ['2025-03-11' '2025-03-12' '2025-03-13']
```

Arrays estructurados. Cada elemento es un registro con varios campos nombrados, cada uno con su propio `dtype`:

```
↪ tipo = np.dtype([('nombre', 'U10'), ('edad', 'i4'), ('altura',
↪      , 'f4')])
personas = np.array([('Alice', 25, 1.70), ('Bob', 30, 1.80)],
dtype=tipo)

print(personas['nombre']) # ['Alice' 'Bob']
print(personas['edad'])  # [25 30]
```

Para datos verdaderamente tabulares y heterogéneos, `pandas` es más cómodo y potente, como vemos a continuación.

12.2. Biblioteca pandas

La biblioteca `pandas` está construida sobre `numpy` y proporciona dos estructuras principales:

- **Series**: “lista etiquetada” unidimensional. Como un `ndarray` 1D, pero con un *índice* explícito.
- **DataFrame**: tabla bidimensional con filas (índice) y columnas (cada una de un `dtype` propio). Cada columna es una **Series**.

Vamos a verlas en orden.

12.2.1. El tipo Series

Creación

```
import pandas as pd

s = pd.Series([10, 20, 30, 40])           # Índice
    automatico 0..3
s = pd.Series([10, 20, 30, 40],
              index=['a', 'b', 'c', 'd']) # Índice
    personalizado
s = pd.Series([1.5, 'Python', True])     # Heterogenea ->
    dtype=object
```

Acceso, operaciones y funciones

```
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

print(s['b'])           # 20 por etiqueta
print(s.iloc[1])        # 20 por posición
print(s + 5)            # vectorizado
print(s.mean(), s.sum(), s.max())
print(s[s > 20])        # Filtrado por máscara

# Aplicar una función a cada elemento
print(s.apply(lambda x: x ** 2))
```

Frecuencias: `value_counts`

Para contar ocurrencias de los valores de una **Series**, `value_counts()` es más natural que `collections.Counter`:

```

s = pd.Series(['admin', 'user', 'admin', 'guest', 'user', 'admin'])
print(s.value_counts())
# admin      3
# user       2
# guest      1
# Name: count, dtype: int64

```

Fechas y cadenas

pandas ofrece accesores `.dt` y `.str` para operar vectorizadamente sobre series de fechas y de cadenas:

```

fechas = pd.to_datetime(pd.Series(['2025-01-01', '2025-02-15', '2025-03-31']))
print(fechas.dt.month)           # 1, 2, 3
print(fechas.dt.day_name())     # Wednesday, Saturday, Monday

texto = pd.Series(['apple', 'BANANA', 'Cherry'])
print(texto.str.upper())        # APPLE, BANANA, CHERRY
print(texto.str.contains('an', case=False)) # True, True, False

```

Alineación de índices

Cuando se operan dos *Series*, *pandas* alinea por índice antes de operar. Los índices que no coincidan dan *NaN*:

```

s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], index=['b', 'c', 'd'])
print(s1 + s2)
# a      NaN
# b      6.0
# c      8.0
# d      NaN

```

Valores nulos: *NaN*

Las series suelen contener valores ausentes (*NaN*, *Not a Number*). Las operaciones más útiles:

- `s.isna()`: máscara booleana de nulos.
- `s.dropna()`: descarta los nulos.

- `s.fillna(valor)`: rellena con un valor fijo.
- `s.ffill()` / `s.bfill()`: rellenan con el último valor anterior/siguiente válido (útil en series temporales con huecos).

```
s = pd.Series([1, 2, None, 4, 5])
print(s.isna())           # mascara
print(s.fillna(0))        # rellena con 0
print(s.fillna(s.mean())) # rellena con la media
```

12.2.2. El tipo DataFrame

Un `DataFrame` es una tabla con filas (índice) y columnas (cada una con su `dtype`).

Creación

```
1 import pandas as pd
2
3 df = pd.DataFrame({
4     'Nombre': ['Alice', 'Bob', 'Charlie'],
5     'Edad': [25, 30, 35],
6     'Ciudad': ['New York', 'Los Angeles', 'Chicago']
7 })
8 print(df)
9
10 #      Nombre  Edad      Ciudad
11 # 0    Alice   25    New York
12 # 1     Bob   30  Los Angeles
13 # 2  Charlie   35    Chicago
```

Lectura y escritura

`pandas` ofrece funciones `pd.read_*` y métodos `df.to_*` para los formatos más habituales: CSV, JSON, Excel, Parquet, HDF5, SQL... Las opciones más útiles de `read_csv` son:

- `dtype={...}`: forzar el tipo de cada columna (evita inferencias incorrectas).
- `parse_dates=[...]`: convierte directamente columnas de fechas.
- `chunksize=N`: itera sobre el fichero en bloques de N filas (para ficheros que no caben en memoria).

```

import pandas as pd

# Lectura
df = pd.read_csv('usuarios.csv', dtype={'ID': 'int32'},
                 parse_dates=['FechaRegistro'])

print(df.head())    # primeras 5 filas

# Escritura
df.to_csv('salida.csv', index=False)
df.to_parquet('salida.parquet')    # mas eficiente que CSV
df.to_json('salida.json', orient='records', force_ascii=False)

```

Otros formatos. `read_excel`, `read_json`, `read_sql`, `read_parquet` leen formatos correspondientes. *Parquet* es un formato columnar binario muy eficiente para datasets grandes – es lo que se usa habitualmente en el ecosistema de datos hoy.

Selección: `loc`, `iloc`, máscaras

- `df['col']` o `df.col`: selecciona una columna como `Series`.
- `df[['col1', 'col2']]`: selecciona varias columnas como `DataFrame`.
- `df.iloc[i]`, `df.iloc[i, j]`: por posición entera.
- `df.loc[etiqueta]`, `df.loc[etiqueta, 'col']`: por etiquetas del índice.
- `df[mascara_booleana]`: filtra filas.

```

df = pd.DataFrame({'Nombre': ['Alice', 'Bob', 'Charlie'],
                  'Edad': [25, 30, 35],
                  'Ciudad': ['NY', 'LA', 'Chicago']})

print(df['Edad'])                # Serie
print(df.iloc[0])                # Primera fila
print(df.loc[df['Edad'] > 28, 'Nombre']) # Nombres con edad >
28

```

SettingWithCopyWarning: usa loc para asignar

Asignar a una selección encadenada **no funciona de forma fiable**:

```
df[df['Edad'] > 28]['Ciudad'] = 'desconocida' # MAL:  
puede no escribir
```

Esto suele lanzar `SettingWithCopyWarning` porque `df[df['Edad'] > 28]` devuelve, según el caso, una vista o una copia, y la siguiente asignación no garantiza modificar el original. La forma correcta es asignar a `.loc` con la condición y la columna en una sola operación:

```
df.loc[df['Edad'] > 28, 'Ciudad'] = 'desconocida' #  
BIEN
```

Modificación y eliminación

```
df['Edad'] = df['Edad'] + 1 # Recalcular  
columna  
df['Categoria'] = df['Edad'].apply( # Crear columna  
nueva  
lambda x: 'Joven' if x < 30 else 'Mayor')  
  
df.drop('Categoria', axis=1, inplace=True) # Eliminar columna  
df.drop(0, axis=0, inplace=True) # Eliminar fila  
por etiqueta
```

Resumen estadístico: describe

`df.describe()` devuelve un resumen estadístico (count, mean, std, min, max, percentiles 25/50/75) de las columnas numéricas. Con `include='all'` extiende a todas las columnas.

```
df = pd.DataFrame({'Edad': [25, 30, 35, 40, 45],  
                  'Sesiones': [10, 15, 12, 18, 20]})  
print(df.describe())
```

Valores nulos

Igual que en Series: `isna`, `dropna`, `fillna`, `ffill`, `bfill`.

```

df = pd.DataFrame({
    'Nombre': ['Alice', 'Bob', None, 'Charlie'],
    'Edad': [25, None, 35, 40],
    'Ciudad': ['NY', 'LA', None, 'Chicago']
})

print(df.isna().sum())      # Conteo de nulos por columna
df.dropna(axis=0)           # Filas sin nulos
df.fillna({'Nombre': 'Desconocido', 'Edad': df['Edad'].mean()
})

# Forward / backward fill (modernos: metodo, no parametro
# method=)
df.ffill()
df.bfill()

```

Agrupación: groupby

`groupby` divide el `DataFrame` en grupos y aplica una función de agregación. Es una de las herramientas más útiles de *pandas*:

```

df = pd.DataFrame({
    'Nombre': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Edad': [25, 30, 35, 40, 28],
    'Ciudad': ['NY', 'LA', 'NY', 'Chicago', 'Chicago'],
    'Departamento': ['Ventas', 'TI', 'Ventas', 'TI', 'Ventas']
})

# Edad media por ciudad
print(df.groupby('Ciudad')['Edad'].mean())

# Múltiples agregaciones a la vez
print(df.groupby('Ciudad')['Edad'].agg(['mean', 'std', 'count'])

# Agrupación jerárquica
print(df.groupby(['Ciudad', 'Departamento'])['Edad'].mean())

```

Combinación de tablas: merge

`pd.merge` realiza un *join* estilo SQL entre dos `DataFrame`. El parámetro `on=` indica la columna(s) común(es); `how=` controla el tipo (`'inner'`, `'left'`, `'right'`, `'outer'`).

```
usuarios = pd.DataFrame({
    'id': [1, 2, 3],
    'nombre': ['Alice', 'Bob', 'Charlie']
})

eventos = pd.DataFrame({
    'usuario_id': [1, 1, 2, 4],
    'evento': ['login', 'logout', 'login', 'login']
})

# Join: une eventos con datos del usuario
combinado = pd.merge(eventos, usuarios,
                     left_on='usuario_id', right_on='id', how
                     ='left')
↪ print(combinado)
```

	usuario_id	evento	id	nombre
# 0	1	login	1.0	Alice
# 1	1	logout	1.0	Alice
# 2	2	login	2.0	Bob
# 3	4	login	NaN	NaN

Reorganización: pivot y melt

`pivot` pasa de “formato largo” a “formato ancho”; `melt` es la operación inversa. Es habitual en análisis de datos cuando los datos vienen como series largas (una fila por observación) y se quieren mostrar como tabla.

```
ventas = pd.DataFrame({
    'fecha': ['2025-01', '2025-01', '2025-02', '2025-02'],
    'producto': ['A', 'B', 'A', 'B'],
    'unidades': [10, 5, 12, 7]
})

# Largo -> ancho: filas=fechas, columnas=productos, valores=
↪ unidades
↪ print(ventas.pivot(index='fecha', columns='producto', values=
    'unidades'))
```

	A	B
fecha		
2025-01	10	5
2025-02	12	7

```

# Ancho -> largo (operacion inversa)
ancho = ventas.pivot(index='fecha', columns='producto',
    ↪     values='unidades')
largo = ancho.reset_index().melt(id_vars='fecha',
    ↪                               var_name='producto',
                               value_name='unidades')

```

Series temporales

pandas destaca en series temporales. Si el índice es de tipo `datetime64`, se desbloquean operaciones como `resample` (cambio de frecuencia), `shift` (desplazamiento) y `rolling` (ventana móvil).

```

data = pd.DataFrame({
    ↪     'fecha': pd.date_range('2025-01-01', periods=6, freq='D')
    ,
    'valor': [10, 20, 30, 40, 50, 60]
})
data = data.set_index('fecha')

# Remuestreo diario -> mensual (sumar)
print(data.resample('ME').sum())    # 'ME' = Month End (pandas
    ↪    2.x)

# Diferencia con el día anterior
data['delta'] = data['valor'] - data['valor'].shift(1)

# Media movil de 3 días
data['mm3'] = data['valor'].rolling(3).mean()
print(data)

```

Componentes de la fecha mediante el accesor `.dt`:

```

df = pd.DataFrame({'fecha': pd.date_range('2025-01-01',
    ↪     periods=3, freq='D')})
df['anyo'] = df['fecha'].dt.year
df['mes'] = df['fecha'].dt.month
df['dia'] = df['fecha'].dt.day
df['dia_semana'] = df['fecha'].dt.day_name()

```

12.3. Aplicación: análisis de registros de actividad

Cerramos el capítulo ilustrando cómo se combinan `numpy` y `pandas` en escenarios típicos de seguridad y análisis de *logs*. Los ejemplos son intencionadamente compactos;

en producción, querríamos añadir robustez (manejo de errores, validación de entradas, etc.) pero las ideas son las mismas.

12.3.1. Detección de anomalías por z-score

Un patrón muy común para detectar valores atípicos es el *z-score*: para cada observación, calcular cuántas desviaciones estándar la separan de la media. Valores con $|z| > 3$ se consideran sospechosos:

```
import numpy as np
import pandas as pd

# Tiempos de respuesta de un servicio (ms)
tiempos = pd.Series(np.random.default_rng(0).normal(50, 10,
    1000))
tiempos.iloc[42] = 500    # Inyectamos un outlier

z = (tiempos - tiempos.mean()) / tiempos.std()
anomalias = tiempos[z.abs() > 3]
print(f"Detectadas {len(anomalias)} anomalias")
```

12.3.2. Top de IPs en un fichero de logs

Combinando pandas y `value_counts`, sustituimos el código manual con `collections.Counter` del cap. 10 por una sola línea:

```
import pandas as pd
import re

ip_re = re.compile(r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}")

with open("access.log", encoding="utf-8") as f:
    ips = pd.Series(m.group(1) for line in f if (m := ip_re.
        match(line)))

print(ips.value_counts().head(10))    # Top 10 de IPs mas
    activas
```

12.3.3. Eventos por hora con resample

Para detectar picos de actividad sospechosa, agrupamos eventos por hora y los contamos. La media móvil ayuda a localizar desviaciones:

```

eventos = pd.read_csv("eventos.csv", parse_dates=['timestamp'])
eventos = eventos.set_index('timestamp')

por_hora = eventos.resample('h').size()           # eventos por
hora                                           # hora
mm = por_hora.rolling(24).mean()               # media movil
de 24h
picos = por_hora[por_hora > mm * 3]             # horas con 3
x la media
print(picos)

```

12.3.4. Cruzar IPs con una lista negra

`merge` permite cruzar dos tablas por una clave común; con `how='inner'` obtenemos solo las coincidencias. Útil para enriquecer eventos con información de *threat intelligence*:

```

import pandas as pd

eventos = pd.read_csv("eventos.csv")           # con
columna 'ip'
malas = pd.read_csv("ioc_malas.csv")           # con
columnas 'ip', 'campaign'

sospechosas = pd.merge(eventos, malas, on='ip', how='inner')
print(f"Eventos coincidentes con IOCs: {len(sospechosas)}")
print(sospechosas.groupby('campaign').size().sort_values(
ascending=False))

```

12.3.5. Análisis de frecuencias para criptoanálisis clásico

Combinando `numpy` y `pandas`, contar frecuencias de letras en un texto cifrado es trivial:

```

texto = "WKL V D VHFUHW PHVVDJH HQFRGHG ZLWK FDHVDU"
letras = pd.Series([c for c in texto if c.isalpha()])
print(letras.value_counts(normalize=True).head(5))

```

Con esto cerramos los fundamentos de manipulación de datos. En el capítulo siguiente (13) veremos cómo presentar visualmente lo que hemos analizado, con `matplotlib` y `seaborn`, y en el capítulo 14 daremos el salto al aprendizaje automático con `scikit-learn`.

12.4. Ejercicios

EJERCICIO 12.1

Escribe un programa en Python que, usando `numpy`, simule 1000 registros de tráfico de red, cada uno con tres características: (1) duración del paquete (entre 1 y 5000 ms), (2) tamaño del paquete (entre 64 y 1500 bytes) e (3) intervalo entre paquetes (entre 1 y 3000 ms). El programa debe identificar los paquetes anómalos, definidos como aquellos cuyos valores en alguna de las tres características estén a más de una vez y media desviaciones estándar por encima o por debajo de la media.

Ejemplo de salida esperada

```
Total de paquetes anómalos detectados: 339
```

```
Primeros 5 paquetes anómalos:
```

```
[[3093  138 1693]
```

```
 [2920  410  106]
```

```
 [ 131  703 1444]
```

```
 [2392  939    2]
```

```
 [4118   88 1287]]
```

Solución (girar para leer)

```

1 import numpy as np
2
3 # Fijar semilla para reproducibilidad
4 np.random.seed(42)
5
6 # Generar 1000 paquetes de red con 3 características
7 # duración (ms), tamaño (bytes), intervalo (ms)
8 trafico = np.column_stack([
9     np.random.randint(1, 5000, 1000),
10    np.random.randint(64, 1500, 1000),
11    np.random.randint(1, 3000, 1000)
12 ])
13
14 # Calcular media y desviación estándar por columna
15 media = np.mean(trafico, axis=0)
16 desviacion = np.std(trafico, axis=0)
17
18 # Definir umbrales superior e inferior (1.5
19 # desviaciones)
20 umbral_superior = media + 1.5 * desviacion
21 umbral_inferior = media - 1.5 * desviacion
22 # Detectar anomalías
23 anomalos = (trafico > umbral_superior) | (trafico <
24 umbral_inferior)
25 paquetes_anomalos = np.any(anomalos, axis=1)
26 # Mostrar resultados
27 print(f"Total de paquetes anomalos detectados: {np.sum(
28     paquetes_anomalos)}")
29 print("Primeros 5 paquetes anomalos:\n", trafico[
30     paquetes_anomalos[:5]])

```

Código 12.1: Solución al Ejercicio 12.1.