



**GUSTAVO CORONEL**  
DESARROLLA SOFTWARE

## **CIENCIA DE DATOS: TALLER DE FUNDAMENTOS DE MACHINE LEARNING CON PYTHON**



### **Módulo 03 ANÁLISIS DE DATOS ESTRUCTURADOS CON PANDAS: Series y DataFrames**

Dr. Eric Gustavo Coronel Castillo  
Docente UNI  
[gcoronel@uni.edu.pe](mailto:gcoronel@uni.edu.pe)



# ÍNDICE

<b>1</b>	<b>PRESENTACIÓN .....</b>	<b>6</b>
<b>2</b>	<b>OBJETIVO .....</b>	<b>7</b>
2.1	OBJETIVO GENERAL .....	7
2.2	OBJETIVOS ESPECÍFICOS.....	7
<b>3</b>	<b>INTRODUCCIÓN A PANDAS Y SU ECOSISTEMA .....</b>	<b>8</b>
<b>4</b>	<b>DATA SERIES .....</b>	<b>9</b>
4.1	¿QUÉ ES UNA PANDAS SERIES? .....	9
4.1.1	<i>Estructura de una Series.....</i>	9
4.2	INSTALACIÓN E IMPORTACIÓN DE PANDAS.....	10
4.3	CREACIÓN DE UNA SERIES .....	11
4.3.1	<i>Desde una lista de Python .....</i>	11
4.3.2	<i>Desde una lista con índice personalizado.....</i>	12
4.3.3	<i>Desde un diccionario de Python .....</i>	13
4.3.4	<i>Desde un escalar (valor constante) .....</i>	14
4.3.5	<i>Desde un arreglo NumPy.....</i>	14
4.4	ATRIBUTOS ESENCIALES DE UNA SERIES .....	15
4.5	ACCESO E INDEXACIÓN .....	17
4.5.1	<i>Acceso por etiqueta con .loc[] .....</i>	17
4.5.2	<i>Acceso por posición con .iloc[].....</i>	18
4.5.3	<i>Indexación booleana (filtrado condicional) .....</i>	19
4.6	OPERACIONES CON SERIES .....	20
4.6.1	<i>Operaciones aritméticas vectorizadas .....</i>	20
4.6.2	<i>Alineación automática por índice.....</i>	20
4.6.3	<i>Métodos estadísticos descriptivos.....</i>	21
4.7	MANEJO DE VALORES FALTANTES (NAN) .....	21
4.8	MÉTODOS Y TRANSFORMACIONES ÚTILES.....	22



4.8.1	<i>Método .apply() — Aplicar funciones personalizadas</i> .....	22
4.8.2	<i>Método .map() — Transformaciones con diccionario o función</i> .....	22
4.8.3	<i>Ordenamiento</i> .....	23
4.8.4	<i>Conteo de valores únicos</i> .....	23
4.9	CONVERSIÓN DE TIPOS DE DATOS .....	23
4.10	VISUALIZACIÓN RÁPIDA DE UNA SERIES .....	24
4.11	RESUMEN: REFERENCIA RÁPIDA DE LA SERIES .....	25
4.12	DETECCIÓN Y MANEJO DE DATOS ATÍPICOS (OUTLIERS) .....	25
4.12.1	<i>Método 1: Rango Intercuartílico (IQR)</i> .....	26
4.12.2	<i>Método 2: Puntuación Z (Z-Score)</i> .....	27
4.12.3	<i>Estrategias de Tratamiento de Outliers</i> .....	30
4.12.4	<i>Comparación de Métodos y Cuándo Usar Cada Uno</i> .....	32
4.12.5	<i>Caso Especial: Valores Centinela en Datos de Sensores</i> .....	32
4.13	EJERCICIOS PROPUESTOS .....	37
4.13.1	<i>Ejercicio 1 — Creación y atributos</i> .....	37
4.13.2	<i>Ejercicio 2 — Acceso e indexación</i> .....	38
4.13.3	<i>Ejercicio 3 — Operaciones y estadísticas</i> .....	38
4.13.4	<i>Ejercicio 4 — Valores faltantes</i> .....	38
4.13.5	<i>Ejercicio 5 — Caso integrador</i> .....	39
5	<b>DATAFRAME</b> .....	40
5.1	INTRODUCCIÓN: DEL CONCEPTO DE TABLA AL DATAFRAME.....	40
5.2	ESTRUCTURA DE UN DATAFRAME.....	40
5.3	CREACIÓN DE UN DATAFRAME .....	41
5.3.1	<i>Desde un diccionario de listas</i> .....	41
5.3.2	<i>Desde una lista de diccionarios</i> .....	42
5.3.3	<i>Desde un diccionario de Series</i> .....	42
5.3.4	<i>Desde un archivo CSV (caso más frecuente en la práctica)</i> .....	43
5.4	INSPECCIÓN INICIAL DE UN DATAFRAME .....	44
5.4.1	<i>Métodos de vista rápida</i> .....	44



5.4.2	<i>Estructura y tipos de datos</i> .....	44
5.4.3	<i>4.3 Estadísticas descriptivas</i> .....	45
5.5	ACCESO Y SELECCIÓN DE DATOS.....	45
5.5.1	<i>Selección de columnas</i> .....	45
5.5.2	<i>Selección de filas y celdas con .loc[] e .iloc[]</i> .....	46
5.5.3	<i>Filtrado condicional (indexación booleana)</i> .....	46
5.6	MANIPULACIÓN DE COLUMNAS Y FILAS .....	48
5.6.1	<i>Agregar nuevas columnas</i> .....	48
5.6.2	<i>Eliminar columnas y filas</i> .....	49
5.6.3	<i>Renombrar columnas</i> .....	52
5.6.4	<i>Estandarizar nombres de columnas</i> .....	52
5.7	MANEJO DE VALORES FALTANTES EN UN DATAFRAME .....	53
5.7.1	<i>Detección</i> .....	53
5.7.2	<i>Eliminación y relleno</i> .....	54
5.8	OPERACIONES DE ANÁLISIS ESENCIALES .....	58
5.8.1	<i>Ordenamiento</i> .....	58
5.8.2	<i>Agrupamiento con groupby</i> .....	58
5.8.3	<i>Tablas de frecuencia con value_counts()</i> .....	59
5.8.4	<i>Tabla pivote con pivot_table()</i> .....	59
5.9	TRANSFORMACIÓN DE TIPOS DE DATOS.....	60
5.10	EJERCICIOS PROPUESTOS .....	62
5.10.1	<i>Ejercicio 1 — Creación e inspección</i> .....	62
5.10.2	<i>Ejercicio 2 — Selección y filtrado</i> .....	62
5.10.3	<i>Ejercicio 3 — Manipulación de columnas</i> .....	63
5.10.4	<i>Ejercicio 4 — Agrupamiento y agregación</i> .....	63
5.10.5	<i>Ejercicio 5 — Caso integrador</i> .....	64
6	REFERENCIAS .....	65



---

## 1 Presentación

---



---

## **2 Objetivo**

---

### **2.1 Objetivo general**

### **2.2 Objetivos específicos**



### 3 Introducción a Pandas y su Ecosistema

Pandas es una librería de código abierto para el lenguaje Python, diseñada específicamente para la manipulación y el análisis de datos estructurados. Fue creada por Wes McKinney en 2008, mientras trabajaba en AQR Capital Management, con el propósito de disponer de una herramienta flexible y eficiente para trabajar con datos relacionales y con series temporales en Python (McKinney, 2022).

El nombre **pandas** proviene del término econométrico "panel data", que hace referencia a conjuntos de datos multidimensionales que involucran mediciones en el tiempo (McKinney, 2022). Actualmente, pandas es una de las librerías más utilizadas en el ecosistema de ciencia de datos con Python, junto con NumPy, Matplotlib y Scikit-learn.

*Nota: pandas se construye sobre NumPy, por lo que los arreglos ndarray subyacen en las estructuras de datos de pandas. Esto le permite aprovechar operaciones vectorizadas de alto rendimiento (pandas Development Team, 2024).*

La librería ofrece dos estructuras de datos principales:

- Series: estructura unidimensional, etiquetada, similar a un arreglo o un diccionario.
- DataFrame: estructura bidimensional tabular con filas y columnas etiquetadas.



## 4 Data Series

### 4.1 ¿Qué es una pandas Series?

Una Series es un arreglo unidimensional que contiene una secuencia de valores y un índice asociado. Cada valor de la Series está vinculado a una etiqueta de índice, lo que permite acceder a los datos de forma flexible y expresiva (McKinney, 2022).

Formalmente, se puede pensar en una Series como una columna de una hoja de cálculo o como un diccionario ordenado de Python. Sin embargo, a diferencia de un diccionario, la Series mantiene el orden y admite operaciones numéricas vectorizadas.

Según VanderPlas (2023, p. 97), la Series puede entenderse como una generalización de los arreglos de NumPy, en la que el índice no necesita ser un número entero, sino que puede ser cualquier tipo de dato hashable (cadenas de texto, fechas, etc.).

#### 4.1.1 Estructura de una Series

La estructura de una Series consta de componentes esenciales:

Componente	Descripción
Índice (index)	Etiquetas asociadas a cada elemento. Por defecto son enteros 0, 1, 2, ... pero pueden personalizarse.
Valores (values)	Los datos almacenados, que corresponden a un arreglo NumPy subyacente.
dtype	El tipo de dato de los valores (int64, float64, object, bool, datetime64, etc.).
name	Atributo opcional que identifica a la Serie (útil cuando forma parte de un DataFrame.)



## 4.2 Instalación e Importación de pandas

Si aún no tiene pandas instalado, puede hacerlo desde la terminal o desde una celda de Jupyter Notebook:

```
# Instalación desde terminal (con Anaconda)
conda install pandas

# Instalación con pip
pip install pandas

Para importar pandas en su script o notebook, la convención estándar de la
comunidad es usar el alias pd (pandas Development Team, 2024):
import pandas as pd
import numpy as np # Se recomienda también importar NumPy

# Verificar la versión instalada
print(pd.__version__) # Ejemplo de salida: 2.2.1
```



## 4.3 Creación de una Series

La función constructora es `pd.Series(data, index, dtype, name)`, donde todos los parámetros, excepto data, son opcionales (pandas Development Team, 2024). A continuación, se detallan los métodos de creación más frecuentes.

### 4.3.1 Desde una lista de Python

La forma más directa de crear una Series es a partir de una lista. El índice se asigna automáticamente como 0, 1, 2, ...

```
import pandas as pd

# Crear una Series desde una lista
notas = pd.Series([15, 18, 12, 20, 14])
print(notas)
```

#### Resultado

```
0    15
1    18
2    12
3    20
4    14
dtype: int64
```

**Nota:** El `dtype int64` indica que pandas infirió automáticamente que los valores son enteros de 64 bits. Este comportamiento es consistente con NumPy (McKinney, 2022).



#### 4.3.2 Desde una lista con índice personalizado

Es posible asignar etiquetas descriptivas al índice para hacer más legible el acceso a los datos:

```
alumnos = ['Ana', 'Luis', 'María', 'Carlos', 'Elena']
notas = pd.Series([15, 18, 12, 20, 14], index=alumnos, name='Notas_Finales')
print(notas)
```

#### Resultado

```
Ana      15
Luis     18
María    12
Carlos   20
Elena   14
Name: Notas_Finales, dtype: int64
```



### 4.3.3 Desde un diccionario de Python

Cuando se crea una Series desde un diccionario, las claves se convierten en el índice y los valores se convierten en los datos (VanderPlas, 2023):

```
poblacion = {  
    'Lima':      10900000,  
    'Arequipa':  1080000,  
    'Trujillo':   990000,  
    'Chiclayo':  600000  
}  
  
serie_poblacion = pd.Series(poblacion, name='Población_2023')  
print(serie_poblacion)
```

### Resultado

```
Lima      10900000  
Arequipa  1080000  
Trujillo   990000  
Chiclayo  600000  
Name: Población_2023, dtype: int64
```



#### 4.3.4 Desde un escalar (valor constante)

Cuando se pasa un escalar como dato, pandas replica el valor para cada elemento del índice especificado:

```
# Crear una Series donde todos los valores son 0.0
serie_cero = pd.Series(0.0, index=['x', 'y', 'z', 'w'])
print(serie_cero)
```

#### Resultado

```
x    0.0
y    0.0
z    0.0
w    0.0
dtype: float64
```

#### 4.3.5 Desde un arreglo NumPy

Dado que pandas se construye sobre NumPy, es posible crear una Series directamente desde un ndarray:

```
import numpy as np

arr = np.array([3.5, 7.1, 2.8, 9.3])
serie_np = pd.Series(arr, index=['a', 'b', 'c', 'd'], name='Mediciones')
print(serie_np)
```

#### Resultado

```
a    3.5
b    7.1
c    2.8
d    9.3
Name: Mediciones, dtype: float64
```



## 4.4 Atributos Esenciales de una Series

Los atributos permiten inspeccionar las propiedades de una Series sin modificarla. Los más relevantes son (pandas Development Team, 2024):

Atributo	Tipo de retorno	Descripción
.values	numpy.ndarray	Arreglo NumPy con los datos.
.index	pandas.Index	Etiquetas del índice.
.dtype	dtype	Tipo de dato de los valores.
.size	int	Número total de elementos.
.shape	tuple	Tupla de dimensiones, p.ej. (5,).
.name	str / None	Nombre asignado a la Serie.
.is_unique	bool	True si todos los valores son únicos.
.hasnans	bool	True si existen valores NaN.



## Ejemplo

```
s = pd.Series([10, 20, 30, 40, 50],  
              index=['a', 'b', 'c', 'd', 'e'],  
              name='ejemplo')  
  
print('Valores: ', s.values)      # ndarray de NumPy  
print('Índice: ', s.index)        # RangeIndex o Index  
print('Tipo: ', s.dtype)          # tipo de dato  
print('Tamaño: ', s.size)         # número de elementos  
print('Forma: ', s.shape)          # tupla (n,)  
print('Nombre: ', s.name)          # nombre de la Serie
```

## Resultado

```
Valores: [10 20 30 40 50]  
Índice: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')  
Tipo: int64  
Tamaño: 5  
Forma: (5,)  
Nombre: ejemplo
```



## 4.5 Acceso e Indexación

pandas ofrece múltiples mecanismos de acceso a los elementos de una Series. McKinney (2022) distingue tres formas principales.

### 4.5.1 Acceso por etiqueta con .loc[]

.loc[] permite seleccionar elementos por su etiqueta de índice. Es el método recomendado cuando se trabaja con índices con nombres:

```
precios = pd.Series({'manzana': 2.5, 'pera': 3.0,
                     'uva': 5.5, 'mango': 4.0})

# Acceso a un solo elemento
print(precios.loc['pera'])          # 3.0
print("-" * 20)

# Acceso a múltiples elementos (lista de etiquetas)
print(precios.loc[['manzana', 'mango']])
print("-" * 20)

# Slicing por etiqueta (incluye extremo final)
print(precios.loc['manzana':'uva'])
```

### Resultado

```
3.0
-----
manzana    2.5
mango      4.0
dtype: float64
-----
manzana    2.5
pera       3.0
uva        5.5
dtype: float64
```



#### 4.5.2 Acceso por posición con .iloc[]

.iloc[] permite acceder a los elementos por su posición numérica (0-based), independientemente del índice:

```
precios = pd.Series({'manzana': 2.5, 'pera': 3.0,
                     'uva': 5.5, 'mango': 4.0})

# Acceso a un solo elemento
print(precios.loc['pera'])          # 3.0
print("-" * 20)

# Acceso a múltiples elementos (lista de etiquetas)
print(precios.loc[['manzana', 'mango']])
print("-" * 20)

# Slicing por etiqueta (incluye extremo final)
print(precios.loc['manzana':'uva'])
```

#### Resultado

```
3.0
-----
manzana    2.5
mango      4.0
dtype: float64
-----
manzana    2.5
pera       3.0
uva        5.5
dtype: float64
```

**Nota:** Diferencia clave: `.Loc[]` incluye el extremo final en el slicing, mientras que `.iloc[]` lo excluye, siguiendo la convención de Python (McKinney, 2022).



### 4.5.3 Indexación booleana (filtrado condicional)

La indexación booleana permite filtrar elementos que cumplan una condición lógica:

```
# Seleccionar frutas con precio mayor a 3.0
print(precios[precios > 3.0])
print("-"*20)

# Condición compuesta: precio entre 3.0 y 5.0
print(precios[(precios >= 3.0) & (precios <= 5.0)])
```

### Resultado

```
uva      5.5
mango    4.0
dtype: float64
-----
pera     3.0
mango    4.0
dtype: float64
```



## 4.6 Operaciones con Series

### 4.6.1 Operaciones aritméticas vectorizadas

Una de las ventajas más importantes de pandas es que las operaciones aritméticas se aplican de forma vectorizada a todos los elementos, sin necesidad de bucles (VanderPlas, 2023):

```
import numpy as np

ventas = pd.Series([1500, 2300, 1800, 3100], index=['Q1', 'Q2', 'Q3', 'Q4'])
print(ventas)
print("-"*20)

# Multiplicar todos los valores por 1.1 (incremento del 10%)
ventas_nuevas = ventas * 1.1
print(ventas_nuevas)
print("-"*20)

# Aplicar función matemática
print(np.sqrt(ventas))
```

### 4.6.2 Alineación automática por índice

Pandas alinea automáticamente los elementos por su índice al realizar operaciones entre dos Series. Si un índice no existe en alguna de las series, el resultado es NaN (Not a Number) (McKinney, 2022):

```
s1 = pd.Series({'a': 10, 'b': 20, 'c': 30})
s2 = pd.Series({'b': 5, 'c': 15, 'd': 25})

print(s1 + s2)
```

---

**Nota:** La alineación automática es análoga al comportamiento de un JOIN en bases de datos relacionales: se alinean los registros por la clave común y se producen nulos donde no hay correspondencia (McKinney, 2022).

---



#### 4.6.3 Métodos estadísticos descriptivos

La Series incluye métodos integrados para el cálculo de estadísticas descriptivas:

```
s = pd.Series([85, 92, 78, 95, 88, 76, 91, 83])

print(s.describe())      # Resumen estadístico completo
print("-" * 20)

print('Media: ', s.mean())
print('Mediana:', s.median())
print('Std:     ', s.std())
print('Min:     ', s.min())
print('Max:     ', s.max())
```

#### 4.7 Manejo de Valores Faltantes (NaN)

En el análisis de datos real, es frecuente encontrar valores faltantes. pandas representa estos valores con NaN (del inglés Not a Number), que es en realidad un valor de punto flotante especial de IEEE 754 (McKinney, 2022).

```
import numpy as np

s = pd.Series([10.0, np.nan, 30.0, np.nan, 50.0])

print(s.isnull())          # Detectar NaN (True donde hay NaN)
print(s.notnull())         # Detectar no-NaN
print(s.dropna())          # Eliminar NaN
print(s.fillna(0))         # Reemplazar NaN con 0
print(s.fillna(s.mean()))  # Reemplazar NaN con la media
```



## 4.8 Métodos y Transformaciones Útiles

### 4.8.1 Método .apply() — Aplicar funciones personalizadas

El método `.apply()` permite aplicar cualquier función a cada elemento de la Series:

```
import pandas as pd

notas = pd.Series([15, 18, 12, 20, 9, 14])

# Clasificar notas con una función lambda
clasificacion = notas.apply(
    lambda x: 'Aprobado' if x >= 11 else 'Desaprobado')
print(clasificacion)

print("-" * 20)
print("Juntar como columnas")
df = pd.concat([notas, clasificacion], axis=1)
print(df)

print("-" * 20)
print("Creando un DataFrame")
df = pd.DataFrame({
    'Nota': notas,
    'Estado': clasificacion
})
print(df)
```

### 4.8.2 Método .map() — Transformaciones con diccionario o función

`.map()` es útil para transformar valores de una Series usando un diccionario de mapeo o una función:

```
genero = pd.Series(['M', 'F', 'M', 'F', 'M'])
mapa = {'M': 'Masculino', 'F': 'Femenino'}
print(genero.map(mapa))
```



#### 4.8.3 Ordenamiento

```
ventas = pd.Series({'norte': 850, 'sur': 1200, 'este': 630, 'oeste': 980})  
  
print(ventas.sort_values())                      # Ascendente (default)  
print("-" * 20)  
print(ventas.sort_values(ascending=False))    # Descendente  
print("-" * 20)  
print(ventas.sort_index())                     # Ordenar por índice
```

#### 4.8.4 Conteo de valores únicos

```
departamentos = pd.Series(['Lima', 'Cusco', 'Lima', 'Arequipa', 'Cusco', 'Lima'])  
  
print(departamentos.unique())                  # Array de valores únicos  
print("-" * 20)  
print(departamentos.nunique())                # Cantidad de valores únicos  
print("-" * 20)  
print(departamentos.value_counts())           # Frecuencia de cada valor
```

### 4.9 Conversión de Tipos de Datos

Es frecuente necesitar convertir el tipo de dato de una Series mediante el método `.astype()` (pandas Development Team, 2024):

```
s = pd.Series(['1', '2', '3', '4'])  # dtype: object (texto)  
s_int = s.astype(int)                 # Convertir a entero  
s_float = s.astype(float)            # Convertir a decimal  
  
print(s.dtype)          # object  
print(s_int.dtype)        # int64  
print(s_float.dtype)      # float64
```



## 4.10 Visualización Rápida de una Series

Pandas integra una interfaz de trazado basada en [Matplotlib](#) que permite generar gráficos rápidos directamente desde una Series con el método `.plot()` (pandas Development Team, 2024):

```
import matplotlib.pyplot as plt

ventas = pd.Series(
    [1500, 2300, 1800, 3100, 2700],
    index=['Ene', 'Feb', 'Mar', 'Abr', 'May'],
    name='Ventas Mensuales (S/..)'
)

# Gráfico de línea (default)
ventas.plot(kind='line', title='Ventas Mensuales', marker='o', color='steelblue')
plt.ylabel('Ventas (S/.)')
plt.tight_layout()
plt.show()

# Gráfico de barras
ventas.plot(kind='bar', title='Ventas por Mes', color='coral')
plt.tight_layout()
plt.show()
```

---

**Nota:** Los tipos de gráfico disponibles con `.plot()` incluyen: 'line', 'bar', 'barh', 'hist', 'pie', 'box', 'area', entre otros. Se explorará a fondo en el Módulo 5.

---



## 4.11 Resumen: Referencia Rápida de la Series

Operación	Código
Crear desde lista	pd.Series([1, 2, 3])
Crear desde dict	pd.Series({'a':1, 'b':2})
Con índice	pd.Series(data, index=[...])
Acceso por etiqueta	s.loc['etiqueta']
Acceso por posición	s.iloc[0]
Filtro booleano	s[s > 5]
Estadísticas	s.describe() / s.mean()
Detectar NaN	s.isnull() / s.notnull()
Eliminar NaN	s.dropna()
Rellenar NaN	s.fillna(valor)
Aplicar función	s.apply(lambda x: ...)
Mapear valores	s.map({'a': 1})
Ordenar valores	s.sort_values()
Contar únicos	s.value_counts()
Convertir tipo	s.astype(float)
Graficar	s.plot(kind='bar')

## 4.12 Detección y Manejo de Datos Atípicos (Outliers)

Un dato atípico (outlier) es una observación que se aleja considerablemente del patrón general de los datos. Su presencia puede distorsionar medidas estadísticas como la media y la desviación estándar, e impactar negativamente en modelos de análisis posteriores (Tukey, 1977, citado en McKinney, 2022). Por ello, su detección y tratamiento son pasos obligatorios en cualquier proceso de limpieza de datos.



Es importante diferenciar los outliers de los valores faltantes: mientras que un NaN indica ausencia de información, un outlier es un valor presente pero extremo. Ambos requieren estrategias de tratamiento distintas (pandas Development Team, 2024).

#### 4.12.1 Método 1: Rango Intercuartílico (IQR)

El método IQR (Interquartile Range) es la técnica robusta más utilizada para detectar outliers, propuesta originalmente por Tukey (1977). Se basa en los cuartiles de la distribución y no se ve tan afectado por los valores extremos como la media. Se definen los siguientes límites (VanderPlas, 2023):

- $IQR = Q3 - Q1$  (diferencia entre el tercer y primer cuartil)
- Límite inferior:  $Q1 - 1.5 \times IQR$
- Límite superior:  $Q3 + 1.5 \times IQR$
- Todo valor fuera de ese rango se considera outlier.

#### Ejemplo

```
import pandas as pd

# Datos de salarios mensuales (S.) con valores extremos
salarios = pd.Series([1800, 2100, 1950, 2300, 1700, 15000, 2050, 1900, 2400, 200])

# Calcular Q1, Q3 e IQR
Q1 = salarios.quantile(0.25)
Q3 = salarios.quantile(0.75)
IQR = Q3 - Q1

# Definir límites
lim_inf = Q1 - 1.5 * IQR
lim_sup = Q3 + 1.5 * IQR

print(f"Q1={Q1}, Q3={Q3}, IQR={IQR}")
print(f"Limite inf={lim_inf:.1f}, Limite sup={lim_sup:.1f}")

# Detectar outliers
outliers = salarios[(salarios < lim_inf) | (salarios > lim_sup)]
print("Outliers detectados:\n", outliers)
```



#### 4.12.2 Método 2: Puntuación Z (Z-Score)

El Z-Score mide cuántas desviaciones estándar se aleja un valor de la media. La convención más extendida es considerar como outlier todo valor con  $|z| > 3$ , lo que corresponde a una probabilidad de ocurrencia inferior al 0.3% bajo una distribución normal (VanderPlas, 2023).

Su fórmula es:

$$Z = \frac{x - \mu}{\sigma}$$

Donde:

$Z$  Es el Z-score o puntaje estándar.

$x$  Es el valor actual.

$\mu$  Valor promedio.

$\sigma$  Desviación estandar

#### Ejemplo

```
import pandas as pd

# Datos de salarios mensuales (S./.) con valores extremos
salarios = pd.Series([1800, 2100, 1950, 2300, 1700, 15000, 2050, 1900, 2400, 200])

print(f"Promedio: {salarios.mean()}")
print(f"Desviacion estandar: {salarios.std()}")

# Z-Score calculado directamente con pandas (sin scipy)
z_scores = (salarios - salarios.mean()) / salarios.std()

# Identificar outliers con |z| > 3
outliers_z = salarios[z_scores.abs() > 3]
print('Z-Scores:\n', z_scores.round(2))
print('Outliers por Z-Score:\n', outliers_z)
```



#### 4.12.2.1 Limitación del Z-Score: el efecto de enmascaramiento

El Z-Score presenta una limitación estructural que es importante comprender antes de aplicarlo: utiliza la media ( $\mu$ ) y la desviación estándar ( $\sigma$ ) calculadas sobre el conjunto completo de datos, incluyendo los propios valores atípicos que se pretende detectar. Esto genera un efecto denominado enmascaramiento (masking effect), descrito por Barnett y Lewis (1994) y citado en la literatura estadística aplicada a ciencia de datos (VanderPlas, 2023).

El mecanismo del enmascaramiento opera de la siguiente manera: cuando existen uno o más outliers de gran magnitud en el conjunto de datos, estos elevan la media e inflan la desviación estándar. Como consecuencia, la distancia del propio outlier a la media —medida en unidades de desviación estándar— se reduce artificialmente, haciendo que su puntuación Z no supere el umbral de detección. En otras palabras, el valor extremo “contamina” los parámetros estadísticos que sirven para detectarlo, volviendo el método ciego ante su propia presencia.

**Concepto clave — Efecto de enmascaramiento:** *un outlier de gran magnitud desplaza la media y amplifica la desviación estándar, reduciendo su propio Z-Score. Cuanto más extremo es el valor, mayor es su influencia sobre los parámetros, y menor resulta aparentemente su puntuación Z.*

Este efecto se intensifica bajo las siguientes condiciones, donde el Z-Score pierde confiabilidad como detector de outliers (McKinney, 2022; VanderPlas, 2023):

- Tamaño de muestra pequeño: con pocos datos, un solo valor extremo tiene un peso proporcional mayor sobre la media y la desviación estándar.
- Outliers múltiples: la presencia de varios valores atípicos acumula su efecto sobre los parámetros, enmascarando a todos ellos simultáneamente.
- Distribuciones asiéticas: cuando los datos no siguen una distribución normal, el umbral  $|Z| > 3$  pierde su justificación estadística y puede resultar demasiado permisivo o demasiado restrictivo según el caso.
- Outliers unilaterales muy extremos: un valor de magnitud muy superior al resto produce una desviación estándar tan grande que ningún otro valor del conjunto logra superar el umbral.

En contraste, el método IQR no se ve afectado por el enmascaramiento porque los cuartiles Q1 y Q3 son estadísticos robustos: su valor no cambia significativamente ante la presencia de valores extremos, ya que dependen de la posición relativa de los



datos y no de su magnitud absoluta. Por esta razón, el IQR es el método preferido cuando no se puede garantizar normalidad o cuando el tamaño de muestra es reducido (McKinney, 2022).

La recomendación práctica es aplicar ambos métodos de forma complementaria: si el IQR detecta outliers que el Z-Score no detecta, ello no significa que uno de los métodos esté equivocado, sino que probablemente el efecto de enmascaramiento está actuando y el IQR está siendo más sensible. La decisión final sobre qué valores tratar debe sustentarse también en el criterio de dominio del negocio o campo de estudio.



### 4.12.3 Estrategias de Tratamiento de Outliers

Una vez detectados los outliers, existen cuatro estrategias principales de tratamiento. La elección depende del contexto del negocio y de la causa raíz del valor extremo (McKinney, 2022):

#### 4.12.3.1 Estrategia A: Eliminación

Se elimina el registro completo cuando se tiene certeza de que el valor es un error de captura o un registro irrelevante:

```
import pandas as pd

# Datos de salarios mensuales (S./.) con valores extremos
salarios = pd.Series([1800, 2100, 1950, 2300, 1700,
                      15000, 2050, 1900, 2400, 200])

# Calcular Q1, Q3 e IQR
Q1 = salarios.quantile(0.25)
Q3 = salarios.quantile(0.75)
IQR = Q3 - Q1

# Definir límites
lim_inf = Q1 - 1.5 * IQR
lim_sup = Q3 + 1.5 * IQR

# Conservar solo los valores dentro de los límites IQR
s_limpio = salarios[(salarios >= lim_inf) & (salarios <= lim_sup)]

# Reporte
print(f"Original:\n{salarios}")
print(f"Limpios:\n{s_limpio}")
```



#### 4.12.3.2 Estrategia B: Imputación por la mediana

Se prefiere la mediana sobre la media para la imputación, porque la mediana es robusta a los propios outliers que se están tratando (McKinney, 2022):

```
mediana = salarios.median()
mascara_outlier = (salarios < lim_inf) | (salarios > lim_sup)
s_imputado = salarios.where(~mascara_outlier, other=mediana)

#Reporte
print(f"Mediana: {mediana}")
print(f"Mascara:\n{s_imputado}")
print(f"Imputado:\n{s_imputado}")
```

#### 4.12.3.3 Estrategia C: Recorte (Winsorization / Capping)

El recorte reemplaza los valores extremos por los límites del rango aceptable, preservando el número de registros. pandas ofrece el método `.clip()` para este propósito (pandas Development Team, 2024):

```
# .clip() reemplaza valores fuera del rango por el límite correspondiente
s_recortado = salarios.clip(lower=lim_inf, upper=lim_sup)
print(s_recortado)
```

#### 4.12.3.4 Estrategia D: Conversión a NaN para tratamiento posterior

Una práctica habitual es convertir los outliers en NaN para luego aplicarles las técnicas de la sección 4.7 (`.fillna()`, `.dropna()`), unificando el flujo de limpieza de datos:

```
import numpy as np
s_nan_outliers = salarios.where(~mascara_outlier, other=np.nan)
print(s_nan_outliers)
```



#### 4.12.4 Comparación de Métodos y Cuándo Usar Cada Uno

La siguiente tabla sintetiza cuándo es más apropiado usar cada método de detección, de acuerdo con McKinney (2022) y VanderPlas (2023):

Criterio	Método IQR	Método Z-Score
Supuesto distribucional	Ninguno (no paramétrico)	Distribución normal
Robustez ante outliers	Alta (usa mediana/cuartiles)	Baja (media influenciada por outliers)
Umbral por defecto	$Q1/Q3 \pm 1.5 \times IQR$	$ Z  > 3$
Recomendado cuando	Datos asimétricos o con outliers múltiples	Datos aproximadamente normales
Implementación pandas	.quantile() + filtro booleano	Cálculo manual con .mean() y .std()

#### 4.12.5 Caso Especial: Valores Centinela en Datos de Sensores

En entornos industriales, de salud, telecomunicaciones o cualquier sistema que dependa de medición física, los datos provienen frecuentemente de sensores. Estos dispositivos presentan un tipo particular de valor atípico que es distinto en su naturaleza a los outliers estadísticos estudiados en las secciones anteriores: el denominado valor centinela (sentinel value).

Un **valor centinela** es un valor numérico predefinido que el sensor o el sistema de adquisición de datos consigna automáticamente cuando ocurre una falla en la lectura, una desconexión, una saturación del instrumento u otro tipo de error operacional. Valores típicos son -999, 9999, 0, -1, 99999 o cualquier número que esté fuera del rango físicamente posible de la variable medida (McKinney, 2022).

**Diferencia clave:** un outlier estadístico es un valor presente pero extremo, de causa ambigua, que requiere detección mediante IQR o Z-Score. Un valor centinela es un código de error conocido de antemano, de causa determinista, que no necesita detección estadística sino identificación directa y reemplazo inmediato.



#### 4.12.5.1 Por qué los métodos IQR y Z-Score no son adecuados para valores centinela

Aplicar IQR o Z-Score a datos que contienen valores centinela produce dos problemas graves. En primer lugar, el valor centinela contamina los cálculos estadísticos: la media, la desviación estándar y los propios cuartiles se ven distorsionados por un valor que no representa ningún fenómeno real. En segundo lugar, el método podría no detectar el centinela si, por coincidencia, su magnitud no supera los umbrales definidos, dejándolo silenciosamente en el conjunto de datos y contaminando cualquier análisis posterior.

La estrategia correcta es actuar antes de cualquier análisis estadístico: identificar los valores centinela por su valor exacto conocido y convertirlos a NaN, que es la representación estándar de ausencia de dato en pandas (pandas Development Team, 2024).

#### 4.12.5.2 Reemplazo con .replace()

El método `.replace()` permite identificar y sustituir valores específicos de forma directa, sin necesidad de calcular ningún parámetro estadístico previo.

##### Ejemplo

```
import pandas as pd
import numpy as np

# Lecturas de temperatura (grados C) donde -999 es código de error del sensor
temperatura = pd.Series([22.5, 23.1, -999, 22.8, 23.4, -999, 22.9, 23.0])
print(temperatura)
print("-" * 20)

# Paso 1: reemplazar el código de error por NaN
temperatura = temperatura.replace(-999, np.nan)
print(temperatura)
```



Si el sistema utiliza múltiples códigos de error según el tipo de falla, `.replace()` acepta una lista de valores o un diccionario.

## Ejemplo

```
# Lecturas de temperatura (grados C) donde -999 es código de error del sensor
temperatura = pd.Series([22.5, 23.1, -999, 22.8, 23.4, 0, 22.9, 23.0, 999])
print('Temperaturas:',temperatura.values)

# Multiples códigos de error -> todos a NaN
temp1 = temperatura.replace([-999, 999, 0], np.nan)
print('temp1:',temp1.values)

# Con diccionario cuando cada código tiene un significado distinto
temp2 = temperatura.replace({-999: np.nan, 999: np.nan, 0: np.nan})
print('temp2:',temp2.values)
```

## Resultado

```
Temperaturas: [22.5  23.1 -999.   22.8  23.4    0.   22.9  23.   999. ]
temp1: [22.5 23.1  nan 22.8 23.4  nan 22.9 23.  nan]
temp2: [22.5 23.1  nan 22.8 23.4  nan 22.9 23.  nan]
```

### 4.12.5.3 Imputación: interpolación vs. mediana

Una vez convertidos los centinelas a NaN, es necesario decidir cómo imputar los valores ausentes. En el caso de datos de sensores, hay que considerar que las lecturas tienen continuidad temporal: la temperatura a las 10:02 debería estar cerca de la temperatura a las 10:01 y a las 10:03, porque la temperatura no salta abruptamente de un instante al siguiente.

Si se implica con la mediana global, se coloca el valor promedio de toda la serie en la posición faltante, sin considerar qué valores tiene alrededor. En cambio, la interpolación lineal estima el valor faltante como un punto intermedio entre su vecino anterior y su vecino posterior, respetando la tendencia local de la señal. Por esa razón, la interpolación produce resultados más coherentes con la naturaleza física de los datos medidos por sensores.



## Ejemplo

```
print('temp1:',temp1.values)
print('Mediana:',temp1.median())

# Opcion A: imputacion con la mediana (ignora la continuidad temporal)
temp_mediana = temp1.fillna(temp1.median())
print('Con mediana:', temp_mediana.values)

# Opcion B: interpolacion lineal (respeta la tendencia local del sensor)
temp_interp = temp1.interpolate(method='linear')
print('Interpolado:', temp_interp.values)
```

## Resultado

```
temp1: [22.5 23.1  nan 22.8 23.4  nan 22.9 23.  nan]
Mediana: 22.95
Con mediana: [22.5 23.1 22.95 22.8 23.4 22.95 22.9 23. 22.95]
Interpolado: [22.5 23.1 22.95 22.8 23.4 23.15 22.9 23. 23. ]
```

**Nota:** En la interpolación lineal, el NaN en la posición 2 se estima como el promedio entre 23.1 (posición 1) y 22.8 (posición 3), resultando 22.95. El NaN en la posición 5 se estima entre 23.4 (posición 4) y 22.9 (posición 6), resultando 23.15. Cada valor ausente se reconstruye a partir de sus vecinos inmediatos, lo que produce resultados más coherentes con la naturaleza continua de las señales físicas.



#### 4.12.5.4 Verificación del proceso de limpieza

Es una buena práctica documentar cuantos valores centinela fueron encontrados y reemplazados antes y después del proceso, como parte del registro de calidad del datos.

```
import pandas as pdimpo
import numpy as np

# Serie original con códigos de error
temperatura_raw = pd.Series([22.5, 23.1, -999, 22.8, 23.4, -999, 22.9, 23.0])
print('Temperaturas:',temperatura_raw.values)

# Contar centinelas antes de limpiar
n_centinelas = (temperatura_raw == -999).sum()
print(f'Valores centinela encontrados: {n_centinelas}')

# Limpiar y verificar
temp_limpia = temperatura_raw.replace(-999, np.nan)
print('Temp. limpias:',temp_limpia.values)
temp_limpia = temp_limpia.interpolate(method='linear')
print('Temp. interpoladas:',temp_limpia.values)
print(f'Valores NaN restantes: {temp_limpia.isnull().sum()}')
print('Serie final:', temp_limpia.values)
```

#### Resultado

```
Temperaturas: [22.5  23.1 -999.    22.8  23.4 -999.    22.9  23. ]
Valores centinela encontrados: 2
Temp. limpias: [22.5 23.1  nan 22.8 23.4  nan 22.9 23. ]
Temp. interpoladas: [22.5 23.1  22.95 22.8  23.4  23.15 22.9  23. ]
Valores NaN restantes: 0
Serie final: [22.5 23.1  22.95 22.8  23.4  23.15 22.9  23. ]
```



#### 4.12.5.5 Tabla resumen: tipos de valor problemático y su tratamiento

La siguiente tabla unifica los tres tipos de valor problemático que puede encontrar una Series en un contexto real de análisis de datos.

Tipo de valor atípico	Origen	Detección	Tratamiento recomendado
Outlier estadístico	Variabilidad natural o error de captura ambiguo	IQR o Z-Score	Eliminar, imputar con mediana o aplicar <code>.clip()</code>
Valor centinela de sensor	Código de error predefinido por el sistema	Identificación directa por valor conocido	Reemplazar con <code>.replace() &gt; NaN &gt;</code> imputar o interpolar
Dato faltante (NaN)	Ausencia de lectura, fallo de transmisión	Detección con <code>.isnull()</code>	Imputar con <code>.fillna()</code> o <code>.interpolate()</code>

## 4.13 Ejercicios Propuestos

Los siguientes ejercicios tienen como propósito consolidar los conceptos desarrollados en esta unidad. Se recomienda resolverlos en Jupyter Notebook, documentando cada paso con celdas Markdown.

### 4.13.1 Ejercicio 1 — Creación y atributos

Cree una Series llamada temperaturas con los siguientes datos de temperatura promedio mensual (°C) para la ciudad de Lima, utilizando los nombres de los meses (Ene, Feb, ..., Jun) como índice:

Ene	Feb	Mar	Abr	May	Jun
23.5	24.1	22.8	20.3	18.1	16.5

Luego, imprima: (a) los valores como ndarray, (b) el índice, (c) el tipo de dato y (d) el tamaño.



#### 4.13.2 Ejercicio 2 — Acceso e indexación

Con la Series temperaturas del ejercicio anterior:

1. Acceda a la temperatura de Marzo usando `.loc[]`.
2. Obtenga las temperaturas de Febrero a Mayo usando slicing con `.loc[]`.
3. Seleccione todos los meses con temperatura mayor a 20 °C usando indexación booleana.
4. Use `.iloc[]` para obtener los últimos dos elementos.

#### 4.13.3 Ejercicio 3 — Operaciones y estadísticas

Se tienen los siguientes datos de ventas (en soles) de una tienda durante la primera semana del mes:

```
ventas = pd.Series([450, 380, 520, 610, 480, 700, 550],  
                   index=['Lun', 'Mar', 'Mié', 'Jue', 'Vie', 'Sáb', 'Dom'])
```

1. Calcule el promedio, la mediana y la desviación estándar.
2. Incremente todas las ventas en un 8% (simular IGV incluido).
3. Identifique qué días la venta superó el promedio.
4. Ordene los valores de mayor a menor.

#### 4.13.4 Ejercicio 4 — Valores faltantes

Cree la siguiente Series y realice las tareas indicadas:

```
import numpy as np  
calificaciones = pd.Series([18, np.nan, 15, np.nan, 12, 20, np.nan, 14])
```

1. Determine cuántos valores faltantes hay.
2. Calcule la media ignorando los NaN.
3. Rellene los NaN con la media de los valores existentes.
4. Elimine los NaN y guarde el resultado en una nueva Series.



#### 4.13.5 Ejercicio 5 — Caso integrador

Un investigador de mercado recopila datos sobre el precio (S./.) de cinco productos en tres tiendas distintas. Cree tres Series (una por tienda) y realice lo siguiente:

1. Calcule la diferencia de precios entre la Tienda A y la Tienda B para cada producto.
2. Determine en qué productos la Tienda C tiene el precio más bajo.
3. Genere un gráfico de barras comparando las tres tiendas para el mismo producto.

Datos sugeridos:

- manzana (A:2.5, B:2.8, C:2.3)
- pera (A:3.0, B:2.9, C:3.2)
- uva (A:5.5, B:5.0, C:5.8)
- mango (A:4.0, B:4.5, C:3.9)
- naranja (A:1.5, B:1.6, C:1.4).



## 5 DataFrame

### 5.1 Introducción: del concepto de tabla al DataFrame

En la sección anterior se estudió la Series, estructura unidimensional fundamental de pandas. El [DataFrame](#) es la extensión natural hacia dos dimensiones: una estructura tabular compuesta por filas y columnas, en la que cada columna es internamente una Series que comparte un índice común con las demás (McKinney, 2022).

Conceptualmente, el [DataFrame](#) es análogo a una hoja de cálculo de Excel, a una tabla de base de datos relacional o a un [data.frame](#) del lenguaje R. Sin embargo, a diferencia de esas herramientas, el [DataFrame](#) de pandas permite operar sobre los datos de forma vectorizada, combinando el rendimiento de NumPy con la expresividad de etiquetas descriptivas en filas y columnas (VanderPlas, 2023).

**Nota:** Cada columna de un [DataFrame](#) es una pandas Series. Todas las columnas comparten el mismo índice de filas, lo que garantiza la alineación automática de los datos (Pandas Development Team, 2024).

### 5.2 Estructura de un DataFrame

Un DataFrame se organiza en componentes esenciales que es importante distinguir desde el inicio.

Componente	Tipo	Descripción
index	pandas.Index	Etiquetas de las filas. Por defecto 0, 1, 2, ... pero pueden ser texto, fechas, etc.
columns	pandas.Index	Etiquetas de las columnas, equivalen a los nombres de los campos.
values	numpy.ndarray	Los datos en sí, representados como un arreglo NumPy bidimensional.
dtypes	pandas.Series de dtype	Tipo de dato de cada columna. Cada columna puede tener un dtype distinto.



## 5.3 Creación de un DataFrame

La función constructora es:

```
pd.DataFrame(data, index, columns, dtype)
```

Donde data es el único parámetro obligatorio (pandas Development Team, 2024). A continuación, se presentan los métodos de creación más frecuentes.

### 5.3.1 Desde un diccionario de listas

Es la forma más intuitiva. Cada clave del diccionario se convierte en el nombre de una columna, y cada lista en los valores de esa columna:

```
import pandas as pd

data = {
    'nombre': ['Ana', 'Luis', 'María', 'Carlos', 'Elena'],
    'edad': [25, 30, 28, 35, 22],
    'ciudad': ['Lima', 'Cusco', 'Lima', 'Arequipa', 'Lima'],
    'salario': [3500, 4200, 3800, 5100, 3200]
}

df = pd.DataFrame(data)
print(df)
```

### Resultado

	nombre	edad	ciudad	salario
0	Ana	25	Lima	3500
1	Luis	30	Cusco	4200
2	María	28	Lima	3800
3	Carlos	35	Arequipa	5100
4	Elena	22	Lima	3200



### 5.3.2 Desde una lista de diccionarios

Cuando los datos provienen de una API o de registros individuales, cada diccionario representa una fila:

```
registros = [
    {'producto': 'Laptop', 'precio': 2500, 'stock': 15},
    {'producto': 'Monitor', 'precio': 850, 'stock': 30},
    {'producto': 'Teclado', 'precio': 120, 'stock': 80},
]

df_prod = pd.DataFrame(registros)
print(df_prod)
```

#### Resultado

```
   producto  precio  stock
0    Laptop     2500      15
1  Monitor      850      30
2   Teclado     120      80
```

### 5.3.3 Desde un diccionario de Series

Cuando ya se tienen Series independientes, pandas las alinea automáticamente por su índice al construir el DataFrame (McKinney, 2022):

```
poblacion = pd.Series({'Lima': 10900000, 'Arequipa': 1080000, 'Trujillo': 990000})
superficie = pd.Series({'Lima': 2672, 'Arequipa': 63345, 'Trujillo': 1769})

df_ciudades = pd.DataFrame({'poblacion': poblacion, 'superficie_km2': superficie})
print(df_ciudades)
```

#### Resultado

```
        poblacion  superficie_km2
Lima      10900000            2672
Arequipa    1080000            63345
Trujillo    990000             1769
```



### 5.3.4 Desde un archivo CSV (caso más frecuente en la práctica)

En proyectos reales, los datos casi siempre provienen de archivos externos. La función `pd.read_csv()` es la puerta de entrada más utilizada (pandas Development Team, 2024).

#### Ejemplo 1

```
import pandas as pd

# Leer el archivo CSV local
df = pd.read_csv('ventas.csv')

# Paso 1: Ver todas las columnas disponibles
print(df.columns.tolist())

# Paso 2: Mostrar solo las columnas de interés
col_seleccionadas = ['fecha', 'producto', 'categoria', 'cantidad', 'total']
print(df[col_seleccionadas].head())
```

#### Ejemplo 2

```
import pandas as pd

# Leer directamente desde una URL
url = 'https://raw.githubusercontent.com/gcorone1c/CEPSUNI-PYTHON-70160/refs/heads/main/Data/iris.csv'
df_iris = pd.read_csv(url)
print(df_iris.head())
```

**Nota:** `pd.read_csv()` admite parámetros adicionales muy útiles: `sep` para el separador (';' en archivos europeos), `encoding` para la codificación del texto ('utf-8', 'latin-1'), `index_col` para designar una columna como índice, y `na_values` para definir qué valores representan datos faltantes (pandas Development Team, 2024).



## 5.4 Inspección Inicial de un DataFrame

Antes de cualquier análisis, es indispensable conocer la estructura y el contenido del DataFrame. Los siguientes métodos y atributos forman parte del flujo estándar de inspección inicial (McKinney, 2022, pp. 140–143):

### 5.4.1 Métodos de vista rápida

```
df = pd.read_csv('ventas.csv')

columnas = ['fecha', 'producto', 'categoria', 'cantidad', 'total']

print(df[columnas].head())      # Primeras 5 filas (por defecto)
print("-" * 70)
print(df[columnas].head(10))    # Primeras 10 filas
print("-" * 70)
print(df[columnas].tail())      # Últimas 5 filas
print("-" * 70)
print(df[columnas].sample(5))   # 5 filas aleatorias
```

### 5.4.2 Estructura y tipos de datos

```
print(df.shape)      # (filas, columnas) como tupla
print(df.dtypes)     # Tipo de dato de cada columna
print(df.columns)    # Nombres de columnas
print(df.index)      # Etiquetas del índice
df.info()           # Resumen completo: dtype, nulls, memoria
```

**Nota:** `df.info()` es especialmente valioso porque muestra de un vistazo cuántos valores no nulos tiene cada columna, lo que permite identificar rápidamente las columnas con datos faltantes sin necesidad de calcular `df.isnull().sum()` por separado.



### 5.4.3 4.3 Estadísticas descriptivas

```
df.describe()          # Estadísticas de columnas numéricas
df.describe(include='all') # Incluye columnas de texto
```

## 5.5 Acceso y Selección de Datos

pandas ofrece múltiples mecanismos para seleccionar subconjuntos de un DataFrame. McKinney (2022) los clasifica en tres categorías principales.

### 5.5.1 Selección de columnas

#### Ejemplo 1

```
# Una columna → devuelve una Series
df['producto']
df.producto    # Notación de atributo (solo si el nombre es válido en Python)
```

#### Ejemplo 2

```
# Múltiples columnas → devuelve un DataFrame
df[['producto', 'cantidad']]
df[['vendedor', 'categoria', 'total']]
```

**Nota:** Se recomienda usar siempre la notación con corchetes `df['columna']` en lugar de `df.columna`, porque la notación de atributo falla cuando el nombre de la columna contiene espacios, coincide con un método de pandas (como 'count' o 'mean'), o se asigna un valor nuevo (VanderPlas, 2023).



### 5.5.2 Selección de filas y celdas con `.loc[]` e `.iloc[]`

`.loc[]` selecciona por etiquetas; `.iloc[]` selecciona por posición numérica. Ambos admiten la sintaxis `[filas, columnas]` para seleccionar simultáneamente filas y columnas.

#### Ejemplo 1

```
# .loc[]: por etiqueta de fila y nombre de columna
df.loc[0]                      # Fila con índice 0
df.loc[2:5]                     # Filas 2, 3, 5 y 5 (incluye extremo final)
df.loc[0, 'producto']           # Celda: fila 0, columna 'producto'
df.loc[0:3, ['producto', 'fecha']]# Filas 0-3, columnas producto y fecha
```

#### Ejemplo 2

```
# .iloc[]: por posición numérica
df.iloc[0]                      # Primera fila
df.iloc[0:3]                     # Filas en posiciones 0, 1, 2
df.iloc[0, 1]                     # Celda: posición fila 0, columna 1
df.iloc[:, 0:2]                  # Todas las filas, primeras 2 columnas
```

### 5.5.3 Filtrado condicional (indexación booleana)

Es la operación de selección más frecuente en análisis de datos. Permite extraer filas que cumplen una o varias condiciones lógicas.

#### Ejemplo 1

```
# Condición simple
df[df['region'] == 'Arequipa']
```

#### Ejemplo 2

```
# Condición numérica
df[df['total'] > 15000]
```



## Ejemplo 3

```
# Condiciones múltiples con & (y) y | (o)
df[(df['region'] == 'Arequipa') & (df['total'] < 1000)]
df[(df['cantidad'] < 5) | (df['total'] < 1000)]
```

## Ejemplo 4

```
# Método .query(): sintaxis más legible para condiciones
print("Cantidad:",len(df.query('region == "Arequipa" and total < 1000')))

df.query('region == "Arequipa" and total < 1000')
```

## Ejemplo 5

```
# Filtrar por lista de valores con .isin()
regiones_norte = ['Trujillo', 'Piura']
df[df['region'].isin(regiones_norte)]
```

**Nota:** Al combinar condiciones booleanas en pandas se deben usar los operadores **&** (y), **|** (o) y **~** (no), nunca los operadores **and**, **or**, **not** de Python estándar. Además, cada condición debe ir entre paréntesis. De lo contrario se producirá un error de precedencia de operadores (McKinney, 2022).



## 5.6 Manipulación de Columnas y Filas

### 5.6.1 Agregar nuevas columnas

Crear una nueva columna es tan simple como asignarle un valor. Si se asigna una expresión vectorizada, pandas la aplica elemento a elemento.

#### Configuración previa (opcional)

```
import pandas as pd
import numpy as np

# Configuracion
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.expand_frame_repr', False)
```

#### Ejemplo 1

```
df = pd.read_csv('rrhh.csv')
pd.set_option('display.width', None)
print(df)
# Columna calculada
df['ingreso_anual'] = df['salario'] * 12
print(df)
```

#### Ejemplo 2

```
# Columna condicional con .apply()
df['nivel'] = df['salario'].apply(
    lambda x: 'Alto' if x > 4500 else ('Medio' if x > 3500 else 'Básico')
)
```



## Ejemplo 3

```
# Columna con numpy (más eficiente en datasets grandes)
import numpy as np
df['bono'] = np.where(df['salario'] > 4000, df['salario'] * 0.10, 0)
print(df[['nombre', 'salario', 'nivel', 'bono']].head())
```

## Resultado

```
    nombre  salario    nivel      bono
0     Ana      3500  Básico      0.0
1   Luis      4200  Medio     420.0
2  María      3800  Medio      0.0
3  Carlos      5100   Alto     510.0
4  Elena      3200  Básico      0.0
```

## 5.6.2 Eliminar columnas y filas

### Ejemplo 1

```
# df completo
print(df)
# Eliminar columna (axis=1 indica columnas)
df_sin_bono = df.drop('bono', axis=1)
print(df_sin_bono)
```

### Ejemplo 2

```
# Eliminar múltiples columnas
df_temp = df.drop(['bono', 'ingreso_anual'], axis=1, inplace=False)
df_temp
```



### Ejemplo 3

```
# Eliminar filas por índice (axis=0 es el default)
df_temp = df.drop([0, 2], axis=0)
print("Filas df:",len(df))
print("Filas df_temp:",len(df_temp))
```

### Ejemplo 4

```
# DataFrame local
data = {
    'id_empleado': [1, 2, 3, 4, 2, 5, 3],
    'nombre': ['Ana', 'Luis', 'María', 'Carlos', 'Luis', 'Elena', 'María'],
    'salario': [3500, 4200, 3800, 5100, 4200, 3200, 3800]
}

dfl = pd.DataFrame(data)

print("DataFrame local:")
print(dfl)

print("Detectar filas duplicadas")
print("Cantidad:",dfl.duplicated().sum())
print(dfl.duplicated())

# Eliminar filas duplicadas
df_sin_duplicados = dfl.drop_duplicates()

print("DataFrame sin duplicados:")
print(df_sin_duplicados)
```



## Ejemplo 5

DataFrame sin filas duplicadas:

```
data = {  
    'id_empleado': [1, 2, 3, 4, 2, 5, 3],  
    'nombre': ['Ana', 'Luis', 'María', 'Carlos', 'Luis', 'Elena', 'María'],  
    'salario': [3500, 4200, 3800, 5100, 5200, 3200, 4800]  
}  
  
df1 = pd.DataFrame(data)  
  
print("DataFrame local:")  
print(df1)  
  
print("Detectar filas duplicadas")  
print("Cantidad:", df1.duplicated().sum())  
print(df1.duplicated())
```

Pero si se considera solo el `id_empleado` o el `nombre` si encontramos filas duplicadas.

```
# Eliminar filas duplicadas  
df1.drop_duplicates(subset=['id_empleado'], inplace=True)  
  
print("DataFrame sin duplicados:")  
print(df1)
```

**Nota:** El parámetro `inplace=True` modifica el DataFrame original directamente, sin necesidad de reasignar. Sin embargo, muchos equipos de trabajo prefieren evitar `inplace=True` por razones de legibilidad y trazabilidad del código, creando siempre nuevos objetos en su lugar (VanderPlas, 2023).



### 5.6.3 Renombrar columnas

```
print("Columnas originales:", df.columns)
# Renombrar columnas específicas con diccionario
df.rename(columns={'nombre': 'empleado', 'ciudad': 'sede'}, inplace=True)
print("Nuevas columnas:", df.columns)
```

### 5.6.4 Estandarizar nombres de columnas

#### Definir el DataFrame

```
df1 = pd.DataFrame({
    'Nombre Empleado': ['Ana Torres', 'Luis Gómez', 'María Pérez'],
    'Edad Persona ': [25, 30, 28],
    'Ciudad Trabajo': ['Lima', 'Cusco', 'Arequipa'],
    'Salario Base': [3500, 4200, 3800]
})

print("Columnas originales:")
print(df1.columns)
```

#### Técnica 1: Reemplazar todos los nombres manualmente

```
df1.columns = ['Empleado', 'Edad', 'Sede', 'Salario']

print("Después de reemplazar manualmente:")
print(df1.columns)
```

#### Técnica 2: Estandarizar automáticamente

```
df1.columns = df1.columns.str.strip().str.lower().str.replace(' ', '_')

print("Después de estandarizar:")
print(df1.columns)
```



### Técnica 3: Estandarizar automáticamente

```
df1.columns = (df1.columns
    .str.strip()
    .str.lower()
    .str.replace(' ', '_')
)

print("Después de estandarizar:")
print(df1.columns)
```

## 5.7 Manejo de Valores Faltantes en un DataFrame

El tratamiento de valores faltantes en un DataFrame sigue la misma lógica que en la Series, pero ahora opera sobre múltiples columnas. Cada columna puede tener distintos patrones de nulos y requerir estrategias de tratamiento independientes (McKinney, 2022).

### 5.7.1 Detección

```
df = pd.read_csv('rrhh.csv')

# Mapa de nulos (True donde hay NaN)
#df.isnull()

# Conteo de nulos por columna
df.isnull().sum()

# Porcentaje de nulos por columna
df.isnull().sum() / len(df) * 100

# Filas que tienen al menos un nulo
df[df.isnull().any(axis=1)]

# Cantidad de filas con al menos un valor nulo
print(f"Filas totales: {len(df)}")
print(f"Filas con al menos un nulo: {len(df[df.isnull().any(axis=1)])}")
```



## 5.7.2 Eliminación y relleno

### Ejemplo 1

```
# Eliminar filas con cualquier nulo
df_new = df.dropna()
df_new
```

### Ejemplo 2

```
# Eliminar filas solo si TODAS las columnas son nulas
df_new = df.dropna(how='all')
df_new
```

### Ejemplo 3

```
# Paso 1
# Crear el DataFrame de ejemplo
data = {
    'estudiante': ['Ana', 'Luis', 'María', 'Carlos', 'Rosa', 'Jorge'],
    'nota_1': [15, 12, None, 14, None, 11],
    'nota_2': [18, None, 16, None, None, 13],
    'nota_3': [None, None, None, None, None, None],
    'nota_4': [14, 15, 13, 16, 12, None],
}

df_notas = pd.DataFrame(data)
print("DataFrame original:")
print(df_notas)
print()

# Paso 2
# Ver el porcentaje de nulos por columna
print("Porcentaje de nulos por columna:")
print((df_notas.isnull().sum() / len(df_notas) * 100).round(1).astype(str) + "%")
print()
```



```
# Paso 3

# Aplicar dropna con thresh al 50%
# filas x 0.5 = 3 valores válidos mínimos
thresh = len(df_notas) * 0.5
df_new = df_notas.dropna(thresh=thresh, axis=1)

print(f"thresh = len(df_notas) * 0.5 = {len(df_notas)} x 0.5 = {thresh}")
print()
print("DataFrame después de eliminar columnas con más del 50% de nulos:")
print(df_new)
```

## Ejemplo 4

```
# Paso 1

# Crear el DataFrame de ejemplo
data = {
    'estudiante': ['Ana', 'Luis', 'María', 'Carlos', 'Rosa', 'Jorge'],
    'nota_1': [15, 12, None, 14, None, 11],
    'nota_2': [18, None, 16, None, None, 13],
    'nota_3': [None, None, None, None, None, None],
    'nota_4': [14, 15, 13, 16, 12, None],
}

df_notas = pd.DataFrame(data)
print("DataFrame original:")
print(df_notas)
print()

# Paso 2

# Rellenar nulos con cero
cols_notas = ['nota_1', 'nota_2', 'nota_3', 'nota_4']
df_notas[cols_notas] = df_notas[cols_notas].apply(pd.to_numeric, errors='coerce')
df_notas[cols_notas] = df_notas[cols_notas].fillna(0)
df_notas.fillna(0)
```



## Ejemplo 5

```
# Rellenar columna específica con su mediana

# Crear DataFrame local
df_local = pd.DataFrame({
    'empleado': ['Ana', 'Luis', 'María', 'Carlos', 'Rosa'],
    'salario': [1500, 1600, np.nan, 1700, 50000]
})

print("DataFrame original:")
print(df_local)

mediana_salario = df_local['salario'].median()
print("Mediana:", mediana_salario)

df_local['salario'] = df_local['salario'].fillna(mediana_salario)

print("DataFrame después de imputar:")
print(df_local)

# Forma mas compacta
# df_local['salario'] = df_local['salario'].fillna(df_local['salario'].median())
```



## Ejemplo 6

```
# Crear DataFrame local
df_local = pd.DataFrame({
    'empleado': ['Ana', 'Luis', 'María', 'Carlos', 'Rosa'],
    'salario': [1500, 1600, np.nan, 1700, 50000],
    'ciudad': ['Lima', np.nan, 'Arequipa', 'Cusco', np.nan],
    'edad': [25, 30, np.nan, 28, 40]
})

print("DataFrame original:")
print(df_local)

# Rellenar cada columna con un valor adecuado
df_local = df_local.fillna({
    'salario': df_local['salario'].median(), # Mediana (robusta ante outliers)
    'ciudad': 'Desconocido', # Texto fijo para categoría
    'edad': df_local['edad'].mean() # Media para edad
})

print("\nDataFrame después de imputar:")
print(df_local)
```



## 5.8 Operaciones de Análisis Esenciales

### 5.8.1 Ordenamiento

```
# Ordenar por una columna
df.sort_values('salario')                      # Ascendente
df.sort_values('salario', ascending=False)       # Descendente

# Ordenar por múltiples columnas
df.sort_values(['ciudad', 'salario'], ascending=[True, False])

# Ordenar por el índice
df.sort_index()
```

### 5.8.2 Agrupamiento con groupby

El método `.groupby()` es una de las operaciones más poderosas de pandas. Permite dividir el DataFrame en grupos según los valores de una o más columnas, aplicar una función de agregación a cada grupo y combinar los resultados (McKinney, 2022):

```
# Promedio de salario por ciudad
df.groupby('ciudad')['salario'].mean()

# Múltiples agregaciones sobre múltiples columnas
df.groupby('ciudad').agg(
    salario_promedio=('salario', 'mean'),
    salario_maximo=('salario', 'max'),
    total_empleados=('nombre', 'count')
)

# Agrupar por múltiples columnas
df.groupby(['ciudad', 'departamento'])['salario'].mean().reset_index()
```



### 5.8.3 Tablas de frecuencia con value\_counts()

```
# Frecuencia absoluta
df['ciudad'].value_counts()

# Frecuencia relativa (proporciones)
df['ciudad'].value_counts(normalize=True).round(2)
```

### 5.8.4 Tabla pivot con pivot\_table()

El método `.pivot_table()` es el equivalente en pandas a las tablas dinámicas de Excel. Permite cruzar variables y aplicar funciones de agregación en la intersección (McKinney, 2022).

#### Sintaxis:

```
df.pivot_table(
    values='columna_a_resumir',
    index='columna_filas',
    columns='columna_columnas',
    aggfunc='mean'
)
```

#### Donde:

- **values** Columna numérica que quieras resumir
- **index** Variable que irá en las filas
- **columns** Variable que irá en las columnas
- **aggfunc** Función de agregación (mean, sum, count, etc.)

#### Ejemplo

```
df.pivot_table(
    values='salario',
    index='ciudad',
    columns='departamento',
    aggfunc='mean'
```



## 5.9 Transformación de Tipos de Datos

Es frecuente que al leer un CSV las columnas numéricas se carguen como texto (dtype object), o que las fechas lleguen como cadenas. La conversión de tipos es un paso obligatorio antes del análisis (pandas Development Team, 2024):

```
# Crear DataFrame con tipos como texto
df_local = pd.DataFrame({
    'precio': ['100', '250', 'error', '300'],
    'fecha': ['2024-01-10', '2024-02-15', '2024-03-20', '2024-04-25'],
    'ciudad': ['Lima', 'Cusco', 'Lima', 'Arequipa']
})

print("DataFrame original:")
print(df_local)
print("\nTipos originales:")
print(df_local.dtypes)

# Convertir columna de texto a numérico
df_local['precio'] = pd.to_numeric(df_local['precio'], errors='coerce')

# errors='coerce' convierte valores no convertibles a NaN
# errors='raise' lanza error (default)
# errors='ignore' deja el valor original sin convertir

# Convertir columna de texto a fecha
df_local['fecha'] = pd.to_datetime(df_local['fecha'], format='%Y-%m-%d')

# Convertir a categórico (ahorra memoria en columnas con pocos valores únicos)
df_local['ciudad'] = df_local['ciudad'].astype('category')

# Verificar los tipos resultantes
print("\nTipos finales")
print(df_local.dtypes)
```



**Nota:** Usar `pd.to_numeric()` con `errors='coerce'` es más seguro que `.astype(float)` cuando los datos pueden contener valores no numéricos, como cadenas vacías, guiones o textos mal formateados. En lugar de lanzar un error, convierte esos valores a NaN y permite continuar el procesamiento (pandas Development Team, 2024).



## 5.10 Ejercicios Propuestos

Los siguientes ejercicios tienen como propósito integrar los conceptos desarrollados en esta unidad. Se recomienda resolverlos en Jupyter Notebook, documentando cada paso con celdas Markdown y comentarios en el código.

### 5.10.1 Ejercicio 1 — Creación e inspección

Cree un DataFrame llamado estudiantes con los siguientes datos de 6 estudiantes universitarios: nombre, carrera, ciclo (1 al 10), nota\_promedio y condicion ('Regular', 'Observado' o 'Destacado'). Luego realice las siguientes tareas:

1. Muestre las primeras 3 filas con `.head()`.
2. Imprima la forma (shape), los tipos de datos (dtypes) y el resumen completo con `.info()`.
3. Obtenga las estadísticas descriptivas con `.describe()`.
4. Liste los valores únicos de la columna carrera.

### 5.10.2 Ejercicio 2 — Selección y filtrado

Usando el DataFrame estudiantes del ejercicio anterior:

5. Seleccione solo las columnas nombre y nota\_promedio.
6. Use `.loc[]` para obtener la fila del tercer estudiante y sus columnas nombre y ciclo.
7. Filtre los estudiantes con nota\_promedio mayor a 15.
8. Filtre los estudiantes cuya condición sea 'Destacado' Y estén en ciclo mayor a 5.
9. Use `.query()` para obtener estudiantes de Ingeniería con nota mayor a 14.



### 5.10.3 Ejercicio 3 — Manipulación de columnas

Cree el siguiente DataFrame de productos de una tienda:

```
productos = pd.DataFrame({  
    'producto': ['Laptop', 'Monitor', 'Teclado', 'Mouse', 'Auriculares'],  
    'precio_sin_igv': [2119, 720, 102, 51, 212],  
    'stock': [15, 30, 80, 120, 45],  
    'categoria': ['Computo', 'Computo', 'Periférico', 'Periférico', 'Audio']  
})
```

1. Cree la columna precio\_con\_igv aplicando 18% al precio sin IGV.
2. Cree la columna valor\_inventario multiplicando precio\_con\_igv por stock.
3. Cree la columna disponibilidad con valor 'Disponible' si stock > 20, 'Bajo Stock' si no.
4. Elimine la columna precio\_sin\_igv.
5. Renombre precio\_con\_igv a precio\_venta.

### 5.10.4 Ejercicio 4 — Agrupamiento y agregación

Con el DataFrame de productos del ejercicio anterior:

6. Calcule el precio\_venta promedio por categoría.
7. Para cada categoría, calcule: total de unidades en stock, precio mínimo y precio máximo.
8. Identifique qué categoría tiene el mayor valor\_inventario total.



### 5.10.5 Ejercicio 5 — Caso integrador

Se le proporciona el dataset público del Titanic, disponible en la siguiente URL:

```
url =  
'https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv'  
df = pd.read_csv(url)
```

1. Realice la inspección inicial completa: head(), shape, dtypes, info() y describe().
2. Identifique cuántos valores faltantes hay por columna y calcule el porcentaje.
3. Calcule la tasa de supervivencia (columna Survived) por clase de pasajero (Pclass).
4. Calcule la edad promedio de supervivientes y no supervivientes.
5. Cree una columna grupo\_edad con las categorías 'Niño' (< 12), 'Adulto Joven' (12–35) y 'Adulto' (> 35), ignorando los NaN de Age.
6. Exporte el DataFrame limpio (sin filas con Age o Fare nulos) a un archivo CSV llamado titanic\_limpi.csv.



## 6 Referencias

- McKinney, W. (2022). *Python for Data Analysis: Data Wrangling with pandas, NumPy & Jupyter* (3rd ed.). O'Reilly Media. <https://wesmckinney.com/book/>
- Pandas Development Team. (2026). *Pandas documentation (Version 3.0.1)*. Pandas. <https://pandas.pydata.org/docs/>
- Python Software Foundation. (2026). *The Python standard library*. Python 3.14.2 documentation. <https://docs.python.org/3/library/>
- VanderPlas, J. (2023). *Python Data Science Handbook: Essential Tools for Working with Data* (2nd ed.). O'Reilly Media.