# Pandas para Procesamiento Industrial

Trazabilidad, Control de Calidad y Análisis de Producción
Agroindustria Alimenticia 4.0

Curso de IA Aplicada al Agro
Universidad del Valle

Enero 2026

**Resumen**

Este manual introduce Pandas desde la perspectiva de la ingeniería de datos aplicada a la industria alimenticia. A diferencia del enfoque tradicional basado en análisis exploratorio genérico, aquí abordamos problemas reales de trazabilidad de lotes, control estadístico de procesos (SPC), cumplimiento normativo (HACCP, FDA) y optimización de líneas de producción. Los estudiantes aprenderán a procesar datasets heterogéneos (fechas, categorías, mediciones numéricas) con eficiencia computacional y rigor científico.

## Índice general

## Prefacio: Del Campo a la Mesa

En la Semana 02 trabajaste con NumPy procesando matrices numéricas homogéneas (humedad del suelo en 365 días × 100 zonas). Ese enfoque funciona para sensores agrícolas, pero **la agroindustria moderna genera datos más complejos**:

- **Trazabilidad**: Cada lote de café procesado tiene ID (string), timestamp de entrada/-salida, temperatura de tostado (float), operario responsable (categoría), resultado QA (booleano).
- **Series temporales irregulares**: Sensores IoT envían datos cada 30 segundos, pero fallan aleatoriamente.
- **Relaciones entre tablas**: Para rastrear un recall de producto, debes cruzar 3 datasets: `lotes_producidos`, `pruebas_laboratorio`, `despachos_clientes`.

NumPy no está diseñado para esto. **Pandas sí**.

**Contexto Industrial**

Imagina una planta procesadora de alimentos que opera 24/7 en 3 turnos, con 5 líneas de producción y 1200 lotes/mes. Cada lote genera:

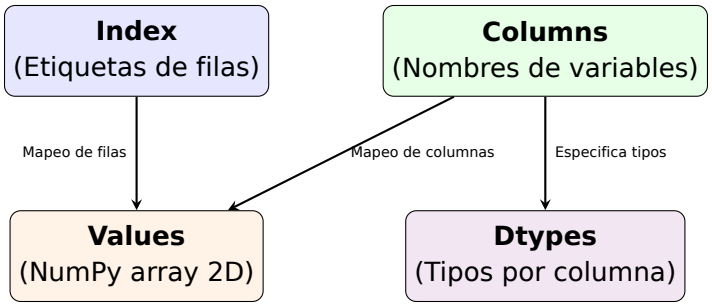
- 8 variables de proceso (temperatura, presión, pH, humedad, tiempo)
- 12 pruebas de laboratorio (microbiología, físico-químicas)
- Metadatos de trazabilidad (proveedor, lote de materia prima, destino)

**Total**: 14,400 lotes/año × 20 variables = 288,000 datos/año.
*No puedes analizar esto en Excel. Necesitas código profesional.*

# 1 Capítulo I: Fundamentos — DataFrame como Base de Datos en Memoria

## 1.1 La Anatomía de un DataFrame

Un DataFrame es una **tabla en memoria RAM** con índice explícito y columnas etiquetadas. A diferencia de NumPy (donde accedes por posición), Pandas permite consultas tipo SQL.




**Diferencia clave con NumPy**:

- NumPy: `array[0, 3]` → Posición absoluta (fila 0, columna 3)
- Pandas: `df.loc["2026-01-01", "Temperatura"]` → Etiqueta semántica

## 1.2 Series vs DataFrame

| Característica | Series | DataFrame |
|---|---|---|
| Dimensionalidad | 1D (columna única) | 2D (tabla) |
| Tipo de datos | Homogéneo (un dtype) | Heterogéneo (dtype por columna) |
| Index | Sí | Sí |
| Operaciones | Vectorizadas | Por columna/fila |
| Uso típico | Una medición | Dataset completo |

Cuadro 1: Comparación Series-DataFrame

Listing 1: Crear Series y DataFrame desde código

```python
import pandas as pd
import numpy as np

# Series: Una columna de temperaturas
temps = pd.Series([72.5, 73.1, 72.8, 74.0],
                  index=['Lote_A', 'Lote_B', 'Lote_C', 'Lote_D'],
                  name='Temperatura_Pasteurizacion')

print(temps['Lote_B'])  # Acceso por etiqueta → 73.1

# DataFrame: Tabla completa de un turno
data = {
    'id_lote': ['L001', 'L002', 'L003'],
    'temp_C': [72.5, 73.1, 71.9],
    'presion_bar': [2.8, 2.9, 2.7],
```

```
    'resultado_QA': ['Aprobado', 'Aprobado', 'Rechazado']
}

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

## 1.3 Carga de Datos Industriales

En la industria, los datos vienen de múltiples fuentes:

- **SCADA** (sistemas de control): CSV/Excel con timestamps
- **LIMS** (laboratorio): Resultados en archivos Excel con hojas múltiples
- **ERP** (SAP/Oracle): Exportaciones CSV con separadores raros
- **Sensores IoT**: JSON desde APIs REST

Listing 2: Carga robusta de datos industriales

```
import pandas as pd

# 1. CSV con problemas comunes
df_scada = pd.read_csv(
    'datos_scada.csv',
    sep=';',                      # Separador europeo
    decimal=',',                  # Decimales con coma
    encoding='latin1',            # Codificación Windows
    parse_dates=['timestamp'],    # Convertir a datetime automáticamente
    na_values=['error', 'offline', '-'],  # Valores nulos personalizados
    dtype={'id_lote': str}        # Forzar ID como texto (evita 001 → 1)
)

# 2. Excel con múltiples hojas
df_lab = pd.read_excel(
    'resultados_laboratorio.xlsx',
    sheet_name='Microbiologia',   # Hoja específica
    header=2,                      # La fila 3 tiene los títulos
    usecols='A:F'                  # Solo columnas A-F
)

# 3. JSON desde API de sensor IoT
import requests
response = requests.get('https://api.sensores.com/temperatura')
df_temp = pd.DataFrame(response.json()['data'])
```

# 2 Capítulo II: Indexación y Selección — El Fundamento de Todo

## 2.1 Los 3 Métodos de Acceso

**⚠ Error #1 más común en Pandas**

Confundir `.loc[]` (etiquetas) con `.iloc[]` (posiciones). Esto causa bugs silenciosos cuando el índice no es secuencial.

| Método | Qué usa | Ejemplo industrial |
|---|---|---|
| `.loc[]` | Etiquetas (labels) | `df.loc["2026-01-15", "pH"]` |
| `.iloc[]` | Posición (enteros) | `df.iloc[0, 3]` (primera fila, cuarta columna) |
| `df[]` | Columnas (principalmente) | `df["Temperatura"]` |

Cuadro 2: Métodos de indexación en Pandas

Listing 3: Ejemplos de indexación

```python
import pandas as pd

# Dataset simulado: Control de calidad de leche
data = {
    'id_lote': ['L001', 'L002', 'L003', 'L004'],
    'fecha': ['2026-01-10', '2026-01-10', '2026-01-11', '2026-01-11'],
    'temp_pasteurizacion': [72.5, 73.0, 71.8, 74.2],
    'ph': [6.7, 6.6, 6.5, 6.9],
    'resultado': ['Aprobado', 'Aprobado', 'Rechazado', 'Aprobado']
}

df = pd.DataFrame(data)

# 1. SELECCIÓN DE COLUMNAS
temps = df['temp_pasteurizacion']  # Retorna Series
subset = df[['id_lote', 'ph']]     # Retorna DataFrame (nota el [[ ]])

# 2. SELECCIÓN POR ETIQUETA (.loc)
# Sintaxis: df.loc[filas, columnas]
primera_fila = df.loc[0]                    # Primera fila completa
ph_L002 = df.loc[1, 'ph']                   # Celda específica: 6.6
rango = df.loc[0:2, 'temp_pasteurizacion']  # Filas 0-2, una columna

# 3. SELECCIÓN POR POSICIÓN (.iloc)
primera_celda = df.iloc[0, 0]     # 'L001'
subcuadro = df.iloc[0:2, 1:3]     # 2 filas × 2 columnas
```

## 2.2 Filtrado Booleano (Máscaras)

El poder real de Pandas está en las **consultas vectorizadas**. No uses bucles `for` — usa máscaras booleanas.

Listing 4: Filtrado avanzado para control de calidad

```python
import pandas as pd

# Cargar datos de producción
df = pd.read_csv('produccion_cafe_enero.csv')
```

```python

# 1. CONSULTA SIMPLE: Lotes rechazados
rechazados = df[df['resultado'] == 'Rechazado']

# 2. CONSULTAS COMPUESTAS: Temperatura fuera de spec Y presión baja
# Rango de pasteurización: 72–76°C, Presión mínima: 2.5 bar
problemas_criticos = df[
    ((df['temp_C'] < 72) | (df['temp_C'] > 76)) &
    (df['presion_bar'] < 2.5)
]

# 3. FILTRO POR LISTA (isin): Solo líneas L1 y L3
lineas_foco = df[df['linea'].isin(['L1', 'L3'])]

# 4. FILTRO POR STRING (contiene): Lotes de turno nocturno
nocturnos = df[df['id_lote'].str.contains('NOCHE')]

# 5. QUERY (sintaxis SQL-like)
# Nota: Solo funciona si nombres de columnas no tienen espacios
criticos = df.query('temp_C > 76 and resultado == "Rechazado"')
```

### Complejidad Computacional de Máscaras

Una máscara booleana `df['temp'] > 72` tiene complejidad $O(n)$ donde $n$ es el número de filas. Internamente:

1. Pandas delega la comparación a NumPy (código C optimizado)
2. Se crea un array booleano en memoria del mismo tamaño que la columna
3. El filtrado `df[mask]` usa fancy indexing de NumPy

Para un DataFrame de 1M filas, esto toma $\sim$10ms. Un bucle `for` equivalente tomaría $\sim$2 segundos (200x más lento).

# 3 Capítulo III: Limpieza de Datos — Fail Fast en Producción

## 3.1 El Problema de los Tipos Incorrectos

**⚠ Tipo `object`**

Si una columna numérica aparece como `dtype: object`, significa que Pandas la leyó como texto. No podrás hacer operaciones matemáticas hasta convertirla.

**Causas comunes**:

- Un solo valor con texto (”Error”, ”N/A”, ”-”) contamina toda la columna
- Formato de número europeo: ”3,14” en lugar de ”3.14”
- Espacios en blanco: ” 25.5 ” no se convierte automáticamente

Listing 5: Diagnóstico y corrección de tipos

```python
import pandas as pd

df = pd.read_csv('sensores_planta.csv')

# 1. DIAGNÓSTICO
print(df.dtypes)
print(df.info())  # Muestra tipos y valores no-nulos

# Ejemplo de salida problemática:
# temperatura    object ←⚠  Debería ser float64
# presion        object ←⚠  Debería ser float64

# 2. INSPECCIÓN MANUAL
print(df['temperatura'].unique())  # Ver valores únicos
# Output: ['25.5', '26.1', 'Error', '24.8', ...] ← "Error" causa el problema

# 3. CONVERSIÓN FORZADA (errores → NaN)
df['temperatura'] = pd.to_numeric(df['temperatura'], errors='coerce')
df['presion'] = pd.to_numeric(df['presion'], errors='coerce')

# 4. VERIFICACIÓN
print(df.dtypes)
# temperatura    float64 ←✓  Corregido
# presion        float64 ←✓  Corregido

print(df['temperatura'].isna().sum())  # Contar cuántos NaN se generaron
```

## 3.2 Tratamiento de Valores Faltantes

En la industria alimenticia, **un dato faltante puede significar un fallo crítico**. No siempre es correcto rellenar con el promedio.

Listing 6: Imputación contextual para sensores

```python
import pandas as pd

df = pd.read_csv('temperatura_camara_fria.csv', parse_dates=['timestamp'])
df = df.set_index('timestamp')

```

| Método | Cuándo usarlo | Riesgo |
|---|---|---|
| `fillna(0)` | Contadores (eventos) | 0 puede ser válido en agro |
| `ffill()` | Series temporales (sensores) | Oculta fallos prolongados |
| `interpolate()` | Datos continuos (temperatura) | Inventa datos inexistentes |
| `dropna()` | QA crítico | Pierdes información |

Cuadro 3: Estrategias de imputación de datos faltantes

```
# CASO 1: Interpolación limitada (máximo 2 valores consecutivos)
# Si faltan >2 valores, algo falló y no deberíamos inventar datos
df['temp'] = df['temp'].interpolate(method='time', limit=2)

# CASO 2: Forward fill con límite temporal
# Rellenar con el último valor conocido, pero solo por 10 minutos
df['humedad'] = df['humedad'].fillna(method='ffill', limit=20)  # 20 registros = 10 min

# CASO 3: Marcar como fallo en lugar de imputar
df['sensor_falla'] = df['temp'].isna()  # Columna booleana de alertas

# CASO 4: Eliminar filas con datos críticos faltantes
df_limpio = df.dropna(subset=['ph', 'acidez'])  # Solo si faltan variables críticas
```

## 3.3 Detección de Outliers

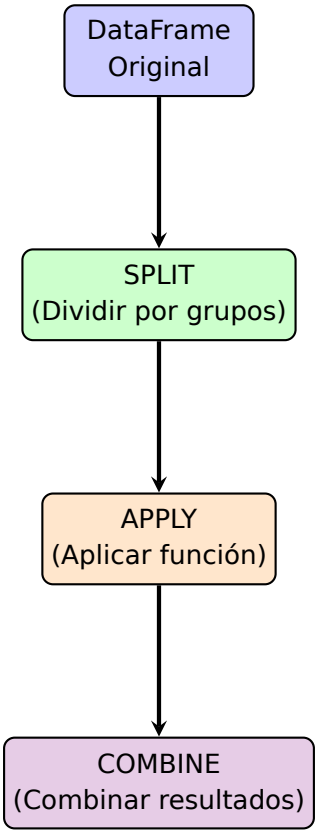
Listing 7: Detección estadística de anomalías

```
import pandas as pd
import numpy as np

df = pd.read_csv('temperatura_pasteurizacion.csv')

# MÉTODO 1: Rango intercuartílico (IQR) — Robusto a valores extremos
Q1 = df['temp'].quantile(0.25)
Q3 = df['temp'].quantile(0.75)
IQR = Q3 - Q1

limite_inferior = Q1 - 1.5 * IQR
limite_superior = Q3 + 1.5 * IQR

outliers = df[(df['temp'] < limite_inferior) | (df['temp'] > limite_superior)]
print(f"Detectados {len(outliers)} outliers")

# MÉTODO 2: Z-score (asume distribución normal)
mean = df['temp'].mean()
std = df['temp'].std()
df['z_score'] = (df['temp'] - mean) / std

# Outliers: |z| > 3 (regla de 3 sigmas)
outliers_zscore = df[np.abs(df['z_score']) > 3]

# MÉTODO 3: Límites físicos (conocimiento del dominio)
# La temperatura de pasteurización NUNCA puede ser > 100°C
errores_sensor = df[df['temp'] > 100]
df.loc[df['temp'] > 100, 'temp'] = np.nan  # Marcar como faltante
```

# 4 Capítulo IV: GroupBy — El Motor de Agregación

## 4.1 El Paradigma Split-Apply-Combine

`.groupby()` es la operación más importante en Pandas. Implementa el patrón *split-apply-combine*:




Listing 8: Análisis de productividad por línea

```python
import pandas as pd

# Dataset: 2000 lotes de café procesados en enero
df = pd.read_csv('produccion_cafe_enero.csv', parse_dates=['timestamp_inicio', 'timestamp_fin'])

# Calcular duración de cada lote
df['duracion_min'] = (df['timestamp_fin'] - df['timestamp_inicio']).dt.total_seconds() / 60

# AGREGACIÓN 1: Productividad por línea
productividad = df.groupby('linea').agg({
    'kg_procesados': 'sum',         # Total de kilos
    'duracion_min': 'mean',          # Duración promedio
    'id_lote': 'count'               # Cantidad de lotes
})

print(productividad)
# Output:
#       kg_procesados  duracion_min  id_lote
# linea
# L1            45000          87.2      650
# L2            38000          92.1      520
# L3            42000          89.5      600

```

```
# AGREGACIÓN 2: Rechazos por turno
rechazos = df.groupby(['turno', 'resultado']).size().unstack(fill_value=0)
print(rechazos)

# AGREGACIÓN 3: Múltiples estadísticas
stats = df.groupby('linea')['duracion_min'].agg(['mean', 'std', 'min', 'max'])
```

## 4.2 GroupBy con Transformaciones

A veces no quieres reducir el DataFrame, sino **agregar columnas calculadas por grupo**.

Listing 9: Normalización por grupo

```
import pandas as pd

df = pd.read_csv('lotes_produccion.csv')

# CASO 1: Calcular % de productividad de cada lote respecto a su línea
df['kg_promedio_linea'] = df.groupby('linea')['kg_procesados'].transform('mean')
df['performance_relativo'] = (df['kg_procesados'] / df['kg_promedio_linea']) * 100

# CASO 2: Ranking dentro de cada turno
df['ranking_turno'] = df.groupby('turno')['kg_procesados'].rank(ascending=False)

# CASO 3: Detectar lotes atípicos (> 2 std de su grupo)
df['media_linea'] = df.groupby('linea')['duracion_min'].transform('mean')
df['std_linea'] = df.groupby('linea')['duracion_min'].transform('std')
df['es_atipico'] = (df['duracion_min'] - df['media_linea']).abs() > (2 * df['std_linea'])
```

**Complejidad de GroupBy**

Internamente, `.groupby()` usa un algoritmo de hashing para agrupar filas:

1. Calcula hash de cada valor en la columna de agrupación: $O(n)$
2. Ordena los índices por hash: $O(n \log n)$
3. Aplica función a cada grupo: $O(n)$

Complejidad total: $O(n \log n)$. Para 1M filas, esto toma $\sim$100ms en un CPU moderno. **Comparación**: Un bucle manual con diccionarios tomaría $\sim$5 segundos (50x más lento).

# 5 Capítulo V: Series Temporales — El Corazón de la Industria

## 5.1 Datetime como Index

En la industria, **el tiempo es el índice natural** de los datos. Convertir el DataFrame a índice temporal desbloquea operaciones avanzadas.

Listing 10: Configurar índice temporal

```python
import pandas as pd

# Cargar datos de sensor con timestamps
df = pd.read_csv('temperatura_camara.csv')

# PASO 1: Convertir columna a datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# PASO 2: Establecer como índice
df = df.set_index('timestamp')

# PASO 3: Ordenar por tiempo (¡importante!)
df = df.sort_index()

# Ahora puedes hacer selección por rangos de fecha:
enero = df['2026-01-01':'2026-01-31']
primera_semana = df['2026-01-01':'2026-01-07']
```

## 5.2 Resampling — Cambiar la Frecuencia

**Resampling vs Rolling**

- **Resample**: Cambia la frecuencia temporal. Ejemplo: datos cada 30 seg → promedio diario.
- **Rolling**: Ventana deslizante. Mantiene la frecuencia original pero suaviza con promedios móviles.

Listing 11: Resampling para reportes diarios

```python
import pandas as pd

# Datos de temperatura cada 30 segundos
df = pd.read_csv('temp_pasteurizacion.csv', parse_dates=['timestamp'], index_col='timestamp')

# RESAMPLE 1: Promedio diario
temp_diaria = df['temperatura'].resample('D').mean()

# RESAMPLE 2: Máximo por hora
temp_horaria_max = df['temperatura'].resample('H').max()

# RESAMPLE 3: Múltiples agregaciones
stats_diarias = df.resample('D').agg({
    'temperatura': ['mean', 'min', 'max', 'std'],
    'presion': 'mean'
})

# RESAMPLE 4: Contar eventos por turno (8 horas)
eventos_turno = df.resample('8H').count()
```

## 5.3 Rolling Windows — Suavizar Ruido

Listing 12: Ventanas móviles para control de procesos

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('temperatura_real_time.csv', parse_dates=['timestamp'], index_col='timestamp')

# Media móvil de 10 minutos (window=20 si los datos son cada 30 seg)
df['temp_suavizada'] = df['temperatura'].rolling(window=20).mean()

# Desviación estándar móvil (detectar variabilidad)
df['temp_std_movil'] = df['temperatura'].rolling(window=20).std()

# Detectar derivas: si la std móvil supera 2°C, el proceso está inestable
df['proceso_inestable'] = df['temp_std_movil'] > 2.0

# Visualización
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['temperatura'], alpha=0.3, label='Datos crudos')
plt.plot(df.index, df['temp_suavizada'], linewidth=2, label='Media móvil 10 min')
plt.legend()
plt.title('Control de Temperatura – Pasteurización')
plt.savefig('control_temperatura.png', dpi=150)
```

# 6 Capítulo VI: Merge y Trazabilidad — Conectar las Piezas

## 6.1 El Problema de la Trazabilidad

En agroindustria alimenticia, la trazabilidad es **un requisito legal** (HACCP, ISO 22000, FDA). Debes poder responder:

- ¿Qué clientes recibieron lotes de un proveedor contaminado?
- ¿Qué lotes fueron procesados por un operario específico en una fecha?
- ¿Qué materia prima se usó en un lote con defecto?

Esto requiere **cruzar múltiples tablas**.

## 6.2 Tipos de Merge

| Tipo | Comportamiento |
|---|---|
| `inner` | Solo filas con match en ambas tablas (intersección) |
| `left` | Todas las filas de la tabla izquierda + matches de la derecha |
| `right` | Todas las filas de la tabla derecha + matches de la izquierda |
| `outer` | Todas las filas de ambas tablas (unión) |

Cuadro 4: Tipos de merge en Pandas

Listing 13: Caso HACCP: Rastreo de lote contaminado

```
import pandas as pd

# PASO 1: Cargar las 3 tablas
lotes = pd.read_csv('lotes_producidos.csv')
# Columnas: id_lote, fecha_produccion, kg, linea

pruebas = pd.read_csv('pruebas_laboratorio.csv')
# Columnas: id_lote, ph, acidez, resultado

despachos = pd.read_csv('despachos.csv')
# Columnas: id_lote, cliente, fecha_envio, destino

# PASO 2: Primera unión (lotes + pruebas)
lotes_con_qa = pd.merge(lotes, pruebas, on='id_lote', how='left')

# PASO 3: Segunda unión (agregar despachos)
trazabilidad_completa = pd.merge(lotes_con_qa, despachos, on='id_lote', how='left')

# PASO 4: Identificar lotes problemáticos (pH < 4.3)
lotes_problematicos = trazabilidad_completa[trazabilidad_completa['ph'] < 4.3]

# PASO 5: Listar clientes afectados
clientes_afectados = lotes_problematicos[['id_lote', 'cliente', 'destino', 'fecha_envio']]
print(clientes_afectados)

```

```
# PASO 6: Exportar para reporte de recall
clientes_afectados.to_csv('recall_lista_clientes.csv', index=False)
```

**⚠ Error Común: Claves con tipos diferentes**

Si `lotes['id_lote']` es string y `pruebas['id_lote']` es int, el merge fallará silenciosamente (0 matches).
**Solución**: Siempre verificar tipos antes de merge:

```
print(lotes['id_lote'].dtype)
print(pruebas['id_lote'].dtype)

# Si son diferentes, convertir:
lotes['id_lote'] = lotes['id_lote'].astype(str)
pruebas['id_lote'] = pruebas['id_lote'].astype(str)
```

# 7 Capítulo VII: Ingeniería de Características

## 7.1 Creación de Columnas Derivadas

Listing 14: Variables calculadas para análisis

```python
import pandas as pd

df = pd.read_csv('produccion_diaria.csv', parse_dates=['fecha'])

# 1. Duración de proceso (timedelta a minutos)
df['duracion_min'] = (df['hora_fin'] - df['hora_inicio']).dt.total_seconds() / 60

# 2. Rendimiento (kg/hora)
df['rendimiento'] = df['kg_producidos'] / (df['duracion_min'] / 60)

# 3. Categorización de turnos
def clasificar_turno(hora):
    if 6 <= hora < 14:
        return 'Mañana'
    elif 14 <= hora < 22:
        return 'Tarde'
    else:
        return 'Noche'

df['turno'] = df['hora_inicio'].dt.hour.apply(clasificar_turno)

# 4. Día de la semana (útil para detectar patrones)
df['dia_semana'] = df['fecha'].dt.day_name()

# 5. Semana del año (agrupación temporal)
df['semana'] = df['fecha'].dt.isocalendar().week
```

## 7.2 Discretización (Binning)

Listing 15: Clasificar variables continuas en categorías

```python
import pandas as pd

df = pd.read_csv('analisis_ph.csv')

# Clasificar pH en categorías
bins = [0, 6.5, 7.0, 14]
labels = ['Ácido', 'Neutro', 'Alcalino']
df['categoria_ph'] = pd.cut(df['ph'], bins=bins, labels=labels)

# Clasificar temperatura de pasteurización en zonas
bins_temp = [0, 71, 74, 77, 100]
labels_temp = ['Bajo Spec', 'Óptimo Bajo', 'Óptimo Alto', 'Sobre Spec']
df['zona_temp'] = pd.cut(df['temp'], bins=bins_temp, labels=labels_temp)
```

# 8 Capítulo VIII: Ética y Calidad de Datos

**⚖ Responsabilidad en la Limpieza de Datos**

Cada decisión de limpieza altera la realidad registrada. En la industria alimenticia, esto tiene implicaciones legales y de salud pública.
**Principios éticos**:

1. **Trazabilidad**: Guardar dataset original sin modificar (`raw/`)
2. **Documentación**: Registrar cada transformación en un log
3. **Transparencia**: Reportar cuántas filas se eliminaron y por qué
4. **Sesgo de imputación**: No ocultar fallos sistemáticos rellenando con promedios

## 8.1 Pipeline de Limpieza Documentado

Listing 16: Pipeline con logging

```python
import pandas as pd
import logging

# Configurar logging
logging.basicConfig(filename='limpieza.log', level=logging.INFO)

def limpiar_dataset(path_entrada, path_salida):
    # Cargar datos crudos
    df = pd.read_csv(path_entrada)
    filas_originales = len(df)
    logging.info(f"Dataset cargado: {filas_originales} filas")

    # 1. Eliminar duplicados
    df = df.drop_duplicates()
    duplicados = filas_originales - len(df)
    logging.info(f"Duplicados eliminados: {duplicados}")

    # 2. Convertir tipos
    df['temperatura'] = pd.to_numeric(df['temperatura'], errors='coerce')
    nulos_generados = df['temperatura'].isna().sum()
    logging.info(f"Valores no numéricos convertidos a NaN: {nulos_generados}")

    # 3. Eliminar outliers
    Q1 = df['temperatura'].quantile(0.25)
    Q3 = df['temperatura'].quantile(0.75)
    IQR = Q3 - Q1
    df_limpio = df[
        (df['temperatura'] >= Q1 - 1.5*IQR) &
        (df['temperatura'] <= Q3 + 1.5*IQR)
    ]
    outliers = len(df) - len(df_limpio)
    logging.info(f"Outliers eliminados: {outliers}")

    # Guardar dataset limpio
    df_limpio.to_csv(path_salida, index=False)
    logging.info(f"Dataset final: {len(df_limpio)} filas guardadas en {path_salida}")

    return df_limpio

```

```
# Ejecutar
df_limpio = limpiar_dataset('data/raw/sensores.csv', 'data/processed/sensores_clean.csv')
```

# 9 Capítulo IX: Talleres Prácticos

## 9.1 Taller 1: Análisis de Línea de Producción

**Caso: Planta de Procesamiento de Café**

Tienes 2000 lotes procesados en enero en 3 líneas (L1, L2, L3). Debes analizar productividad, identificar cuellos de botella y generar reporte ejecutivo.

**Dataset**: `produccion_cafe_enero.csv`

**Tareas**:

1. Calcular duración promedio por línea
2. Identificar el turno más lento
3. Detectar lotes con duración > 2 desviaciones estándar
4. Generar tabla resumen con productividad (kg/hora)

## 9.2 Taller 2: Control de Calidad Temporal

**Caso: Pasteurización de Leche**

Sensor registra temperatura cada 30 segundos durante una semana. Debes detectar derivas, generar alertas y producir gráficos de control.

**Dataset**: `temperatura_pasteurizacion_semana.csv`

**Tareas**:

1. Resamplear a promedios de 10 minutos
2. Calcular desviación estándar móvil (ventana 20 lecturas)
3. Detectar periodos con temperatura fuera de [72-76°C] por >5 minutos
4. Visualizar con matplotlib

## 9.3 Taller 3: Trazabilidad y Recall

**Caso HACCP: Lote Contaminado**

Se detectó contaminación microbiológica en el lote L20260115_042. Debes rastrear qué clientes lo recibieron y generar lista para recall.

**Datasets**:

- `lotes_producidos.csv`
- `pruebas_microbiologia.csv`
- `despachos_clientes.csv`

**Tareas**:

1. Merge de las 3 tablas por `id_lote`
2. Filtrar lotes con resultado ”Contaminado”
3. Generar CSV con: cliente, dirección, fecha_envío, kg_afectados
4. Crear reporte LaTeX con tabla de afectados

# 10 Capítulo X: Visualización Profesional con Matplotlib

## 10.1 Gráficos de Control Estadístico (SPC)

Los gráficos SPC son fundamentales en industria alimenticia para detectar derivas de proceso.

Listing 17: Gráfico de control

```python
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv('ph_lotes.csv')
media = df['ph'].mean()
std = df['ph'].std()
UCL = media + 3*std
LCL = media - 3*std

fig, ax = plt.subplots(figsize=(14, 6))
ax.scatter(df.index, df['ph'], alpha=0.7)
ax.axhline(y=media, color='green', linewidth=2, label='Media')
ax.axhline(y=UCL, color='red', linestyle='--', label='UCL')
ax.axhline(y=LCL, color='red', linestyle='--', label='LCL')
ax.legend()
plt.savefig('spc.png', dpi=300)
```

# 11 Capítulo XI: Estadística con SciPy

## 11.1 Pruebas de Hipótesis

Listing 18: t-test entre turnos

```python
from scipy import stats
import pandas as pd

df = pd.read_csv('defectos.csv')
diurno = df[df['turno'] == 'Diurno']['defectos']
nocturno = df[df['turno'] == 'Nocturno']['defectos']

t_stat, p_value = stats.ttest_ind(diurno, nocturno)

if p_value < 0.05:
    print("Diferencia significativa entre turnos")
```

# 12 Capítulo XII: SQL para Ciencia de Datos

## 12.1 Conexión y Queries

Listing 19: Pandas + SQL

```python
import pandas as pd
from sqlalchemy import create_engine

engine = create_engine('postgresql://user:pass@localhost/planta')

df = pd.read_sql_query(
    "SELECT linea, AVG(kg) FROM lotes GROUP BY linea",
    engine
)
```

# Referencias y Recursos

## Bibliografía Científica

1. McKinney, W. (2017). *Python for Data Analysis*. 2nd Edition. O'Reilly Media.
2. VanderPlas, J. (2016). *Python Data Science Handbook*. O'Reilly Media.
3. Pandas Development Team (2024). *Pandas Documentation*. `https://pandas.pydata.org/docs/`

## Estándares Industriales

- ISO 22000:2018 — Food Safety Management Systems
- FDA 21 CFR Part 11 — Electronic Records and Signatures
- Codex Alimentarius — HACCP Principles

## Datasets de Práctica

- Kaggle: Food Safety Inspections
- UCI Machine Learning Repository: Wine Quality Dataset
- Open Food Facts: Global food products database