Transformación y Limpieza de Datos en Pandas (sin imputación)

Este documento presenta técnicas **sin imputación** para preparar y transformar datos usando pandas: detección/eliminación de nulos, normalización de datos semi-estructurados (strings, listas y fechas).

---

1. Manejo de valores faltantes (sin imputación)

En esta ayudantía **no haremos imputación** de valores (nada de promedios/medianas). Solo detectaremos y eliminaremos casos problemáticos, o los **marcaremos** para procesarlos después.

Dataframe Inicial

```
import pandas as pd

df = pd.DataFrame({
    'id': [' 001','002','003','004','003 '],
    'nombre_completo': ['ana GÓmez','  LUIS  pérez ','Sofía  Díaz', None,'sofia  DÍAZ'],
    'emails': ['ana@uc.cl; ana@gmail.com', 'l.perez@uc.cl', 'sdiaz@uc.cl;SOFIA@MAIL.COM',
'pedro@uc.cl', None],
    'fecha_atencion': ['12/03/2024','31-04-2024','2024-05-10','2024-06-15','10-05-24'],
    'monto': ['10000', '12.500', '4500', '15.750', 'USD 3,000']
    'fono': ['123456789', '9-8765-4321', '(+56) 9 1111 2222', '987654321', 'fono: 456789123']

})

df
```

	id	nombre_completo	emails	fecha_atencion	monto	fono
0	001	ana GÓmez	ana@uc.cl; ana@gmail.com	12/03/2024	10000	123456789
1	002	LUIS pérez	l.perez@uc.cl	31-04-2024	12.500	9-8765-4321
2	003	Sofía Díaz	sdiaz@uc.cl;SOFIA@MAIL.COM	2024-05-10	4500	(+56) 9 1111 2222
3	004	None	pedro@uc.cl	2024-06-15	15.750	987654321
4	003	sofia DÍAZ	None	10-05-24	USD 3,000	fono: 456789123

Detección

```
df.info() # Vista general de tipos y nulos
df.isna().sum()  # Conteo de NaN por columna
df[df['col'].isna()] # Filas donde 'col' es NaN
```

Eliminación de duplicados

```
# Eliminar filas duplicadas según todas las columnas
df_sin_duplicados = df.drop_duplicates()

# Eliminar duplicados solo considerando la columna 'id'
df_sin_duplicados = df.drop_duplicates(subset=['id'])
```

```
# Ejemplo: eliminar duplicados considerando 'id' y 'nombre_completo'
df_sin_duplicados = df.drop_duplicates(subset=['id', 'nombre_completo'])
```

Eliminación de nulos

```
df_limpio = df.dropna()                       # Elimina filas con al menos un NaN
df_limpio = df.dropna(subset=['colA','colB']) # Solo si A o B son NaN
df_limpio = df.dropna(how='all')              # Elimina filas completamente vacías
```

```
# Ejemplo: eliminar filas donde 'nombre_completo' es nulo
df = df.dropna(subset=['nombre_completo'])
```

Resultado tras limpieza y eliminación de nulos

	id	nombre_completo	emails	fecha_atencion	monto	fono
0	001	ana GÓmez	ana@uc.cl;ana@gmail.com	12/03/2024	10000	123456789
1	002	LUIS pérez	l.perez@uc.cl	31-04-2024	12.500	9876544321
2	003	Sofía Díaz	sdiaz@uc.cl;SOFIA@MAIL.COM	2024-05-10	4500	911112222
4	003	sofia DÍAZ	None	10-05-24	USD 3,000	456789123

Es necesario hacer `df.reset_index(drop=True, inplace=True)`

	id	nombre_completo	emails	fecha_atencion	monto	fono
0	001	ana GÓmez	ana@uc.cl;ana@gmail.com	12/03/2024	10000	123456789
1	002	LUIS pérez	l.perez@uc.cl	31-04-2024	12.500	9876544321
2	003	Sofía Díaz	sdiaz@uc.cl;SOFIA@MAIL.COM	2024-05-10	4500	911112222
3	003	sofia DÍAZ	None	10-05-24	USD 3,000	456789123

> Se eliminó la fila con `nombre_completo` nulo (índice 3).

> 💡 Sugerencia práctica: cuando elimines, **registra cuántas filas** se van, para justificar tus decisiones en la I1/Tarea.

---

2. Normalización de datos semi-estructurados

Muchos datasets llegan con texto “sucio” o comprimido en una sola columna. Usaremos el API de strings de pandas (`.str`) y algunas transformaciones.

2.0 `str`

La función `str` en pandas se utiliza para realizar operaciones vectorizadas sobre datos de tipo cadena (strings) en una Serie o columna de un DataFrame. Esto significa que puedes aplicar métodos de manipulación de cadenas de manera eficiente a todos los elementos de una columna sin necesidad de usar bucles explícitos.

Por ejemplo, puedes usar `str` para convertir texto a minúsculas, eliminar espacios en blanco, buscar patrones, reemplazar texto, dividir cadenas, entre otras operaciones.

- La función `str` solo funciona con datos de tipo cadena. Si la columna contiene valores nulos (NaN) o tipos diferentes, es posible que necesites manejar esos casos antes de usarla.

- Es una herramienta poderosa para limpiar y transformar datos textuales en proyectos de análisis de datos.

Ejemplo práctico de uso de `.str`

Supón que tienes una columna con nombres desordenados y quieres estandarizarlos:

```
df = pd.DataFrame({'nombre': ['  ana GÓmez', 'LUIS   PÉREZ', 'soFía   díAz  ']})

# Usando .str para limpiar espacios y normalizar mayúsculas/minúsculas
df['nombre_limpio'] = (df['nombre']
                       .str.strip()                # Elimina espacios al inicio/final
                       .str.title()                # Convierte a Title Case
                       .str.replace(r'\s+', ' ', regex=True))  # Elimina espacios extra

print(df)
```

Resultado:

	nombre	nombre_limpio
0	ana GÓmez	Ana Gómez
1	LUIS PÉREZ	Luis Pérez
2	soFía díAz	Sofía Díaz

2.1. Limpieza de strings

`strip()` elimina los espacios en blanco al inicio y al final de un string.
Por ejemplo: `" hola ".strip()` devuelve `"hola"`.

También puedes usar:

- `lstrip()` para quitar solo los espacios a la izquierda.
- `rstrip()` para quitar solo los espacios a la derecha.

```
# Estandarizar mayúsculas/minúsculas y espacios

df['nombres'] = df['nombres'].str.strip().str.title()
df['emails']  = df['emails'].str.strip().str.lower()
```

`replace()` es un método que permite reemplazar partes de un string según un patrón. Documentación

```
df['fono'] = (df['fono']
              .str.replace(r'\D+', '', regex=True)  # Dejar solo dígitos
              .str.replace(r'^56', '', regex=True))  # Quitar código país si viene repetido
```

- `.str.replace(r'\D+', '', regex=True)` elimina todo lo que **no es dígito** (`\D`). El signo `+` indica que se eliminarán **uno o más** caracteres no numéricos consecutivos.
- `.str.replace(r'^56', '', regex=True)` elimina el prefijo `56` si está al inicio (`^`), quitando el código de país chileno.

Así, se limpian los números de teléfono para que queden solo los dígitos relevantes.

Alternativa con `apply` para reemplazos

En vez de usar `.str.replace`, puedes aplicar una función personalizada con `.apply` y el método nativo de Python `replace`:

```
# Ejemplo para limpiar 'fono' usando apply y una función definida
def limpiar_fono(x):
    solo_digitos = ''.join([c for c in str(x) if c.isdigit()])
    if solo_digitos.startswith('56'):
        solo_digitos = solo_digitos[2:]
    return solo_digitos

df['fono'] = df['fono'].apply(limpiar_fono)

# Para reemplazar espacios en 'emails' (como en el pipeline)
def limpiar_emails(x):
    return str(x).replace(' ', '') if pd.notnull(x) else x

df['emails'] = df['emails'].apply(limpiar_emails)
```

Esto permite mayor flexibilidad si necesitas lógica más compleja que los patrones regulares.

Resultado

	id	nombre_completo	emails	fecha_atencion	monto	fono
0	001	ana GÓmez	ana@uc.cl;ana@gmail.com	12/03/2024	10000	123456789
1	002	LUIS pérez	l.perez@uc.cl	31-04-2024	12.500	9876544321
2	003	Sofía Díaz	sdiaz@uc.cl;sofia@mail.com	2024-05-10	4500	911112222
3	003	sofia DÍAZ	None	10-05-24	USD 3,000	456789123

- Se eliminaron espacios extra en todas las columnas con `.str.strip()`.
- En la columna `fono`, se dejaron solo los dígitos relevantes usando `.str.replace(r'\D+', '', regex=True)` y se quitó el prefijo `56` si estaba presente.
- En la columna `emails`, se eliminaron los espacios con `.str.replace(' ', '', regex=True)` y se normalizó a minúsculas.

Normalizar ID

```
# quitar espacios y ceros a la izquierda → Int64 (permite NA)
clean['id'] = (clean['id'].str.strip()
                          .str.replace(r'^0+', '', regex=True)
                          .replace('', None)
                          .astype('Int64'))

clean
```

	id	nombre_completo	emails	fecha_atencion	monto	fono
0	1	ana GÓmez	ana@uc.cl;ana@gmail.com	12/03/2024	10000	123456789
1	2	LUIS pérez	l.perez@uc.cl	31-04-2024	12.500	9876544321
2	3	Sofía Díaz	sdiaz@uc.cl;sofia@mail.com	2024-05-10	4500	911112222
3	3	sofia DÍAZ	None	10-05-24	USD 3,000	456789123

2.2. Separar columnas con `str.split`

```
# Columna "nombre_completo" -> "nombre" y "apellido"
df[['nombre', 'apellido']] = df['nombre_completo'].str.strip().str.split(' ', n=1, expand=True)

# - str.split(' ', n=1, expand=True): Divide cada cadena en la columna en dos partes usando el
espacio (' ') como separador.
#   - n=1: Limita la división a un máximo de 1 separación, obteniendo como resultado dos partes.
#   - expand=True: Devuelve el resultado como un DataFrame, lo que permite asignar directamente
las columnas "nombre" y "apellido".


# Columna "region|comuna" separada por '|'
df[['region','comuna']] = df['region_comuna'].str.split('|', expand=True)

# - str.split('|', expand=True): Divide cada cadena en la columna "region_comuna" en dos partes
usando el carácter '|' como separador.
#   - expand=True: Devuelve el resultado como un DataFrame, lo que permite asignar directamente
las columnas "region" y "comuna".
```

	id	nombre_completo	nombre	apellido	emails	fecha_atencion	monto	fono
0	1	ana GÓmez	Ana	Gómez	ana@uc.cl;ana@gmail.com	12/03/2024	10000	123456789
1	2	LUIS pérez	Luis	Pérez	l.perez@uc.cl	31-04-2024	12.500	9876544321
2	3	Sofía Díaz	Sofía	Díaz	sdiaz@uc.cl;sofia@mail.com	2024-05-10	4500	911112222
3	3	sofia DÍAZ	Sofia	Díaz	None	10-05-24	USD 3,000	456789123

Ahora vamos a normalizar la columna `nombre_completo` usando `.str.strip()`, `.str.title()` y `.str.replace(r'\s+', ' ', regex=True)` para dejar los nombres en formato estándar:

```
df['nombre_completo'] = (df['nombre_completo'].str.strip().str.title().str.replace(r'\s+', ' ',
regex=True))
```

	id	nombre_completo	nombre	apellido	emails	fecha_atencion	monto	fono
0	1	Ana Gómez	Ana	Gómez	ana@uc.cl;ana@gmail.com	12/03/2024	10000	123456789
1	2	Luis Pérez	Luis	Pérez	l.perez@uc.cl	31-04-2024	12.500	9876544321
2	3	Sofía Díaz	Sofía	Díaz	sdiaz@uc.cl;sofia@mail.com	2024-05-10	4500	911112222
3	3	Sofia Díaz	Sofia	Díaz	None	10-05-24	USD 3,000	456789123

2.3. Listas y `explode`

```
# Columna 'emails' con "correo1;correo2;correo3"
df['emails_list'] = df['emails'].str.split(';') # Transforma emails en una lista con los emails
df = df.drop(columns=['emails'])
df = df.explode('emails_list', ignore_index=True) # después de "explotar" la columna el índice
del DataFrame se reasigna de forma secuencial empezando desde 0.
df = df.rename(columns={'emails_list':'email'}) # La renombramos
```

	id	nombre_completo	nombre	apellido	email	fecha_atencion	monto	fono
0	1	Ana Gómez	Ana	Gómez	ana@uc.cl	12/03/2024	10000	123456789
1	1	Ana Gómez	Ana	Gómez	ana@gmail.com	12/03/2024	10000	123456789
2	2	Luis Pérez	Luis	Pérez	l.perez@uc.cl	31-04-2024	12.500	9876544321
3	3	Sofía Díaz	Sofía	Díaz	sdiaz@uc.cl	2024-05-10	4500	911112222
4	3	Sofía Díaz	Sofía	Díaz	sofia@mail.com	2024-05-10	4500	911112222
5	3	Sofia Díaz	Sofia	Díaz	None	10-05-24	USD 3,000	456789123

2.4. Fechas con `to_datetime`

```
# Convertir strings a fechas, marcando errores como NaT (sin rellenar)
df['fecha_atencion'] = pd.to_datetime(df['fecha_atencion'], format='%d/%m/%Y', errors='coerce') #
Si algo no se puede normalizar, queda como `NaT`.
```

Esto solo funciiona si todas las fechas están en el mismo formato

Fechas en distinto formato con `to_datatime`

```
# Supongamos que tu DF se llama df
col = df['fecha_atencion'].astype(str)

# Lista de formatos que queremos intentar
formatos = ["%d/%m/%Y", "%d-%m-%Y", "%Y-%m-%d", "%d-%m-%y"]

# Lista para guardar los resultados parciales
parsed_list = []

for fmt in formatos:
    parsed = pd.to_datetime(col, format=fmt, errors="coerce")
    parsed_list.append(parsed)

# Concatenar todos los resultados
df_concat = pd.concat(parsed_list, axis=1)

# Tomar el primer valor no nulo por fila
df['fecha_atencion'] = df_concat.bfill(axis=1).iloc[:, 0]

# Normalizar el formato
df['fecha_atencion'] = df['fecha_atencion'].dt.strftime("%Y-%m-%d")
df
```

Fechas en distinto formato con `apply`

```
from datetime import datetime

def parse_fecha(fecha):
    formatos = ["%d/%m/%Y", "%d-%m-%Y", "%Y-%m-%d", "%d-%m-%y"]
    for fmt in formatos:
        try:
            return datetime.strptime(str(fecha), fmt) # Convierte una fecha en un objeto datetime
```

```
de Python usando el formato especificado en fmt
        except ValueError:
            continue
    return pd.NaT  # si no calza con ninguno

df['fecha_atencion'] = df['fecha_atencion'].apply(parse_fecha)

# Normalizar a YYYY-MM-DD
df['fecha_atencion'] = df['fecha_atencion'].dt.strftime("%Y-%m-%d")

print(df[['nombre_completo','fecha_atencion']])
```

	id	nombre_completo	nombre	apellido	email	fecha_atencion	monto	fono
0	1	Ana Gómez	Ana	Gómez	ana@uc.cl	2024-03-12	10000	123456789
1	1	Ana Gómez	Ana	Gómez	ana@gmail.com	2024-03-12	10000	123456789
2	2	Luis Pérez	Luis	Pérez	l.perez@uc.cl	NaT	12.500	9876544321
3	3	Sofía Díaz	Sofía	Díaz	sdiaz@uc.cl	2024-05-10	4500	911112222
4	3	Sofía Díaz	Sofía	Díaz	sofia@mail.com	2024-05-10	4500	911112222
5	3	Sofia Díaz	Sofia	Díaz	None	2024-05-10	USD 3,000	456789123

Puedes encontrar más funciones y utilidades de la librería datatime en este link

También está la documentación de python link

---

3. Transformación de tipos

Caso 1: Si es que no tienen comas ni puntos

```
df['id'] = df['id'].str.replace(r'\D+', '', regex=True)
df['monto'] = df['monto'].str.replace(r'\D+', '', regex=True)

df['id'] = df['id'].astype('Int64') # Entero con soporte de NaN
df['monto'] = pd.to_numeric(df['monto'], errors='coerce')
df['es_titular'] = df['es_titular'].astype('boolean') # boolean con NA
```

> Usa `errors="coerce"` para marcar entradas inválidas como NaN/NaT y **no las rellenes** aquí.

Caso 3: Con comas y puntos, además de formatos distintos

```
def monto_simple(valor):
    if pd.isna(valor):
        return None
    s = str(valor)

    # dejar solo dígitos, comas y puntos con replace
    for ch in s:
        if not (ch.isdigit() or ch in [',','.']):
            s = s.replace(ch, '')

    # si hay más de un separador -> el último es decimal
    if s.count('.') + s.count(',') > 1:
```

```
        # buscamos el último separador
        idx = max(s.rfind('.'), s.rfind(','))
        # quitamos todos los separadores anteriores
        s = s[:idx].replace('.', '').replace(',', '') + '.' + s[idx+1:].replace(',',
'').replace('.', '')
    else:
        # si hay solo coma, la usamos como decimal
        s = s.replace(',', '.')

    try:
        return float(s)
    except:
        return None

df['monto'] = df['monto'].apply(monto_simple)
```

	id	nombre_completo	nombre	apellido	email	fecha_atencion	monto	fono
0	1	Ana Gómez	Ana	Gómez	ana@uc.cl	2024-03-12	10000	123456789
1	1	Ana Gómez	Ana	Gómez	ana@gmail.com	2024-03-12	10000	123456789
2	2	Luis Pérez	Luis	Pérez	l.perez@uc.cl	NaT	12500	9876544321
3	3	Sofía Díaz	Sofía	Díaz	sdiaz@uc.cl	2024-05-10	4500	911112222
4	3	Sofía Díaz	Sofía	Díaz	sofia@mail.com	2024-05-10	4500	911112222
5	3	Sofia Díaz	Sofia	Díaz	None	2024-05-10	3000	456789123

4. Luego podemos hacer conexión de DataFrames coo ya sabemos

4.1. concat

```
# Apilar por filas (mismas columnas)
df_all = pd.concat([df_2023, df_2024], axis=0, ignore_index=True)

# Unir por columnas (mismo índice o reindexar)
df_cols = pd.concat([dfA.set_index('id'), dfB.set_index('id')], axis=1).reset_index()
```

4.2. merge

```
# En clave común
df_ab = pd.merge(a, b, on='id', how='inner')      # inner/left/right/outer
# Nombres distintos
df_ab = pd.merge(a, b, left_on='id_afiliado', right_on='id_paciente', how='left')
# Varias claves
df_ab = pd.merge(a, b, on=['id','fecha'], how='outer')
```

4.3. join

```
df_join = a.set_index('id').join(b.set_index('id'), how='left').reset_index()
```

> Consejo: al unir, **verifica duplicados** en las claves y el **cardinal** (1-1, 1-N, N-1, N-N).

4.4. Cardinalidad

Cardinalidad en unión de DataFrames

La **cardinalidad** describe la relación entre las filas de los DataFrames que vas a unir. Es clave para evitar duplicados inesperados y entender el resultado de la unión.

- **1 a 1 (uno a uno):** Cada valor de la clave aparece una sola vez en ambos DataFrames.
  - Ejemplo:

```
df1 = pd.DataFrame({'id': [1,2], 'nombre': ['Ana','Luis']})
df2 = pd.DataFrame({'id': [1,2], 'edad': [23, 34]})
pd.merge(df1, df2, on='id')
```

id	nombre	edad
1	Ana	23
2	Luis	34

- **1 a N (uno a muchos):** La clave aparece una vez en el primer DataFrame y varias veces en el segundo.
  - Ejemplo:

```
df1 = pd.DataFrame({'id': [1,2], 'nombre': ['Ana','Luis']})
df2 = pd.DataFrame({'id': [1,1,2], 'email': ['ana@a.com','ana@b.com','luis@c.com']})
pd.merge(df1, df2, on='id')
```

id	nombre	email
1	Ana	ana@a.com
1	Ana	ana@b.com
2	Luis	luis@c.com

- **N a 1 (muchos a uno):** La clave aparece varias veces en el primer DataFrame y una vez en el segundo.
  - Ejemplo:

```
df1 = pd.DataFrame({'id': [1,1,2], 'email': ['ana@a.com','ana@b.com','luis@c.com']})
df2 = pd.DataFrame({'id': [1,2], 'nombre': ['Ana','Luis']})
pd.merge(df1, df2, on='id')
```

id	email	nombre
1	ana@a.com	Ana
1	ana@b.com	Ana
2	luis@c.com	Luis

- **N a N (muchos a muchos):** La clave aparece varias veces en ambos DataFrames. El resultado puede tener muchas combinaciones.
  - Ejemplo:

```
df1 = pd.DataFrame({'id': [1,1], 'fecha': ['2024-01-01','2024-01-02']})
df2 = pd.DataFrame({'id': [1,1], 'valor': [10,20]})
pd.merge(df1, df2, on='id')
```

id	fecha	valor
1	2024-01-01	10
1	2024-01-01	20
1	2024-01-02	10
1	2024-01-02	20

Ejemplo de mal uso de `merge`

Un error común es unir DataFrames sin revisar duplicados en la clave, lo que genera combinaciones inesperadas (N a N):

```
df1 = pd.DataFrame({'id': [1,1,2], 'nombre': ['Ana','Ana','Luis']})
df2 = pd.DataFrame({'id': [1,1,2], 'email': ['ana@a.com','ana@b.com','luis@c.com']})

# Mal uso: no se revisó duplicados en 'id'
df_merged = pd.merge(df1, df2, on='id')
print(df_merged)
```

Resultado:

id	nombre	email
1	Ana	ana@a.com
1	Ana	ana@b.com
1	Ana	ana@a.com
1	Ana	ana@b.com
2	Luis	luis@c.com

Aquí, para cada combinación de `id=1`, se cruzan todas las filas, generando duplicados inesperados.
Solución: Revisa duplicados antes de unir y asegúrate de la cardinalidad adecuada.

Recomendación:
Antes de unir, revisa con `value_counts()` o `duplicated()` cuántas veces aparece cada clave. Así evitas duplicados inesperados y entiendes la estructura del resultado.

---

5. Checklist limpieza y transformación para I1 / Tarea 2

- [ ] Reporté `df.info()` y `isna().sum()` inicial.
- [ ] Documenté las **reglas de limpieza** (qué eliminé y por qué).
- [ ] Normalicé strings (`strip`, `lower/title`, `replace`, `split`).
- [ ] Convertí tipos con `astype`/`to_datetime` (`errors='coerce'`).
- [ ] Desanidé listas con `explode`.
- [ ] Revisé duplicados y cardinalidad antes de `merge`.