003_clean_airbnb

May 17, 2025

[991]:
```
# Celda 1: Importación de Librerías
import pandas as pd
import psycopg2
from sqlalchemy import create_engine, text
import os
import logging
from dotenv import load_dotenv
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[992]:
```
# Celda 2: Configuraciones Básicas
# El log se guardará en el mismo directorio que este notebook.
LOG_FILE_PATH = '/home/nicolas/Escritorio/proyecto ETL/develop/Notebooks/airbnb/
 ↪003_clean_airbnb.log'

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler(LOG_FILE_PATH), # Nombre de archivo de log␣
 ↪actualizado
        logging.StreamHandler()
    ]
)

logging.info("Inicio del notebook de Limpieza de Datos (003_data_cleaning.
 ↪ipynb).")
print(f"Logging configurado. Los logs se guardarán en {LOG_FILE_PATH}")

# Configuraciones de Pandas para mejor visualización
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 100)
pd.set_option('display.float_format', lambda x: '%.3f' % x)
pd.set_option('display.width', 1000)

logging.info("Configuraciones de Pandas para visualización aplicadas.")
```

```
print("Celda 2: Configuraciones básicas de logging y Pandas realizadas.")
```

```
2025-05-17 02:01:14,620 - INFO - Inicio del notebook de Limpieza de Datos
(003_data_cleaning.ipynb).
2025-05-17 02:01:14,621 - INFO - Configuraciones de Pandas para visualización
aplicadas.

Logging configurado. Los logs se guardarán en /home/nicolas/Escritorio/proyecto
ETL/develop/Notebooks/airbnb/003_clean_airbnb.log
Celda 2: Configuraciones básicas de logging y Pandas realizadas.
```

[993]:
```
# Celda 3: Carga de Variables de Entorno para PostgreSQL
ENV_FILE_PATH = '/home/nicolas/Escritorio/proyecto ETL/develop/env/.env'
TABLE_NAME_RAW = 'raw_airbnb' # Tabla de donde leeremos los datos brutos

logging.info(f"Ruta del archivo .env: {ENV_FILE_PATH}")
logging.info(f"Nombre de la tabla en PostgreSQL a leer: {TABLE_NAME_RAW}")

# Cargar variables de entorno
if os.path.exists(ENV_FILE_PATH):
    load_dotenv(ENV_FILE_PATH)
    logging.info(f"Archivo .env encontrado y cargado desde {ENV_FILE_PATH}")
else:
    logging.error(f"Archivo .env NO encontrado en {ENV_FILE_PATH}. Asegúrate de␣
 ↪que la ruta es correcta.")
    raise FileNotFoundError(f"Archivo .env no encontrado en {ENV_FILE_PATH}")

POSTGRES_USER = os.getenv('POSTGRES_USER')
POSTGRES_PASSWORD = os.getenv('POSTGRES_PASSWORD')
POSTGRES_HOST = os.getenv('POSTGRES_HOST')
POSTGRES_PORT = os.getenv('POSTGRES_PORT')
POSTGRES_DATABASE = os.getenv('POSTGRES_DATABASE')

if not all([POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_HOST, POSTGRES_PORT,␣
 ↪POSTGRES_DATABASE]):
    logging.error("Una o más variables de entorno de PostgreSQL no están␣
 ↪definidas. Revisa tu archivo .env y la carga.")
    raise ValueError("Faltan credenciales de PostgreSQL. Verifica el archivo .
 ↪env y su carga.")
else:
    logging.info("Variables de entorno para PostgreSQL cargadas correctamente.")
```

```
2025-05-17 02:01:14,631 - INFO - Ruta del archivo .env:
/home/nicolas/Escritorio/proyecto ETL/develop/env/.env
2025-05-17 02:01:14,632 - INFO - Nombre de la tabla en PostgreSQL a leer:
raw_airbnb
2025-05-17 02:01:14,634 - INFO - Archivo .env encontrado y cargado desde
/home/nicolas/Escritorio/proyecto ETL/develop/env/.env
```

```
2025-05-17 02:01:14,635 - INFO - Variables de entorno para PostgreSQL cargadas
correctamente.
2025-05-17 02:01:14,632 - INFO - Nombre de la tabla en PostgreSQL a leer:
raw_airbnb
2025-05-17 02:01:14,634 - INFO - Archivo .env encontrado y cargado desde
/home/nicolas/Escritorio/proyecto ETL/develop/env/.env
2025-05-17 02:01:14,635 - INFO - Variables de entorno para PostgreSQL cargadas
correctamente.
```

[994]:
```python
# Celda 4: Extracción de Datos desde PostgreSQL
df_raw = pd.DataFrame() # DataFrame para los datos brutos
engine = None

try:
    logging.info("Intentando conectar a la base de datos PostgreSQL y extraer␣
 ↪datos.")
    DATABASE_URL = f"postgresql+psycopg2://{POSTGRES_USER}:
 ↪{POSTGRES_PASSWORD}@{POSTGRES_HOST}:{POSTGRES_PORT}/{POSTGRES_DATABASE}"
    engine = create_engine(DATABASE_URL)

    logging.info(f"Consultando la tabla completa '{TABLE_NAME_RAW}'...")
    df_raw = pd.read_sql_table(TABLE_NAME_RAW, con=engine)

    logging.info(f"Datos extraídos exitosamente de la tabla '{TABLE_NAME_RAW}'.
 ↪")
    logging.info(f"El DataFrame df_raw tiene {df_raw.shape[0]} filas y {df_raw.
 ↪shape[1]} columnas.")
    print(f"Celda 4: Datos extraídos de '{TABLE_NAME_RAW}'. Filas: {df_raw.
 ↪shape[0]}, Columnas: {df_raw.shape[1]}")

except Exception as e:
    logging.error(f"Error al conectar a PostgreSQL o extraer datos de la tabla␣
 ↪'{TABLE_NAME_RAW}': {e}")
    print(f"Celda 4: Error al conectar a PostgreSQL o extraer datos - {e}")
finally:
    if engine:
        engine.dispose()
        logging.info("Conexiones del motor de SQLAlchemy dispuestas (cerradas).
 ↪")
        print("Conexiones de DB dispuestas.")

# Crear una copia para trabajar, manteniendo df_raw intacto
df_cleaning = pd.DataFrame() # Predefinir por si df_raw está vacío
if not df_raw.empty:
    df_cleaning = df_raw.copy()
    logging.info("Copia de df_raw creada como df_cleaning para el proceso de␣
 ↪limpieza.")
```

```
    print("Copia df_cleaning creada a partir de df_raw.")
else:
    logging.warning("df_raw está vacío, por lo que df_cleaning también lo␣
 ↪estará. La limpieza no podrá proceder.")
    print("df_raw está vacío. No se puede proceder con la limpieza.")
```

```
2025-05-17 02:01:14,705 - INFO - Intentando conectar a la base de datos
PostgreSQL y extraer datos.
2025-05-17 02:01:14,707 - INFO - Consultando la tabla completa 'raw_airbnb'…
WARNING:  la base de datos «airbnb» tiene una discordancia de versión de
ordenamiento ("collation")
DETAIL:  La base de datos fue creada usando la versión de ordenamiento 2.31,
pero el sistema operativo provee la versión 2.35.
HINT:  Reconstruya todos los objetos en esta base de datos que usen el
ordenamiento por omisión y ejecute ALTER DATABASE airbnb REFRESH COLLATION
VERSION, o construya PostgreSQL con la versión correcta de la biblioteca.
2025-05-17 02:01:15,766 - INFO - Datos extraídos exitosamente de la tabla
'raw_airbnb'.
2025-05-17 02:01:15,766 - INFO - El DataFrame df_raw tiene 102599 filas y 26
columnas.
2025-05-17 02:01:15,767 - INFO - Conexiones del motor de SQLAlchemy dispuestas
(cerradas).
2025-05-17 02:01:15,804 - INFO - Copia de df_raw creada como df_cleaning para el
proceso de limpieza.

Celda 4: Datos extraídos de 'raw_airbnb'. Filas: 102599, Columnas: 26
Conexiones de DB dispuestas.
Copia df_cleaning creada a partir de df_raw.
```

[995]:
```
# Celda 5: Eliminación de Filas Duplicadas
logging.info("Celda 5: Iniciando eliminación de filas duplicadas.")
if not df_cleaning.empty:
    initial_rows = len(df_cleaning)
    logging.info(f"Número de filas antes de eliminar duplicados:␣
 ↪{initial_rows}")

    # Verificar cuántos duplicados hay (esto ya lo hicimos en EDA, pero es␣
 ↪bueno reconfirmar)
    num_duplicados_actual = df_cleaning.duplicated().sum()
    logging.info(f"Número de filas duplicadas encontradas en df_cleaning:␣
 ↪{num_duplicados_actual}")

    if num_duplicados_actual > 0:
        # Eliminar duplicados, manteniendo la primera aparición por defecto
        df_cleaning.drop_duplicates(inplace=True)
        final_rows = len(df_cleaning)
        rows_dropped = initial_rows - final_rows
```

```
        logging.info(f"Se eliminaron {rows_dropped} filas duplicadas.")
        logging.info(f"Número de filas después de eliminar duplicados:␣
 ↪{final_rows}")
    else:
        logging.info("No se encontraron filas duplicadas para eliminar.")

    # Resetear el índice después de eliminar filas
    df_cleaning.reset_index(drop=True, inplace=True)
    logging.info("Índice del DataFrame df_cleaning reseteado.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se pueden eliminar␣
 ↪duplicados.")
```

```
2025-05-17 02:01:15,812 - INFO - Celda 5: Iniciando eliminación de filas
duplicadas.
2025-05-17 02:01:15,813 - INFO - Número de filas antes de eliminar duplicados:
102599
2025-05-17 02:01:15,813 - INFO - Número de filas antes de eliminar duplicados:
102599
2025-05-17 02:01:16,073 - INFO - Número de filas duplicadas encontradas en
df_cleaning: 541
2025-05-17 02:01:16,363 - INFO - Se eliminaron 541 filas duplicadas.
2025-05-17 02:01:16,363 - INFO - Número de filas después de eliminar duplicados:
102058
2025-05-17 02:01:16,364 - INFO - Índice del DataFrame df_cleaning reseteado.
```

[996]:
```
# Celda 4.1: Normalización de Nombres de Columnas
logging.info("Celda 4.1: Iniciando normalización de nombres de columnas en␣
 ↪df_cleaning.")

if not df_cleaning.empty:
    original_columns = df_cleaning.columns.tolist()
    logging.info(f"Nombres de  columnas  originales: {original_columns}")

    # Normalizar: convertir a minúsculas y reemplazar espacios con guiones bajos
    normalized_columns = [col.lower().replace(' ', '_') for col in␣
 ↪original_columns]

    # También es buena idea reemplazar múltiples guiones bajos por uno solo,␣
 ↪por si acaso
    normalized_columns = [col.replace('__', '_') for col in normalized_columns]

    # Renombrar las columnas en el DataFrame
    df_cleaning.columns = normalized_columns

    logging.info(f"Nombres de columnas normalizados: {df_cleaning.columns.
 ↪tolist()}")
```

```
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se pueden␣
 ↪normalizar los nombres de las columnas.")
    print("El DataFrame df_cleaning está vacío. No se pueden normalizar los␣
 ↪nombres de las columnas.")
```

```
2025-05-17 02:01:16,370 - INFO - Celda 4.1: Iniciando normalización de nombres
de columnas en df_cleaning.
2025-05-17 02:01:16,371 - INFO - Nombres de  columnas  originales: ['id',
'NAME', 'host id', 'host_identity_verified', 'host name', 'neighbourhood group',
'neighbourhood', 'lat', 'long', 'country', 'country code', 'instant_bookable',
'cancellation_policy', 'room type', 'Construction year', 'price', 'service fee',
'minimum nights', 'number of reviews', 'last review', 'reviews per month',
'review rate number', 'calculated host listings count', 'availability 365',
'house_rules', 'license']
2025-05-17 02:01:16,372 - INFO - Nombres de columnas normalizados: ['id',
'name', 'host_id', 'host_identity_verified', 'host_name', 'neighbourhood_group',
'neighbourhood', 'lat', 'long', 'country', 'country_code', 'instant_bookable',
'cancellation_policy', 'room_type', 'construction_year', 'price', 'service_fee',
'minimum_nights', 'number_of_reviews', 'last_review', 'reviews_per_month',
'review_rate_number', 'calculated_host_listings_count', 'availability_365',
'house_rules', 'license']
2025-05-17 02:01:16,371 - INFO - Nombres de  columnas  originales: ['id',
'NAME', 'host id', 'host_identity_verified', 'host name', 'neighbourhood group',
'neighbourhood', 'lat', 'long', 'country', 'country code', 'instant_bookable',
'cancellation_policy', 'room type', 'Construction year', 'price', 'service fee',
'minimum nights', 'number of reviews', 'last review', 'reviews per month',
'review rate number', 'calculated host listings count', 'availability 365',
'house_rules', 'license']
2025-05-17 02:01:16,372 - INFO - Nombres de columnas normalizados: ['id',
'name', 'host_id', 'host_identity_verified', 'host_name', 'neighbourhood_group',
'neighbourhood', 'lat', 'long', 'country', 'country_code', 'instant_bookable',
'cancellation_policy', 'room_type', 'construction_year', 'price', 'service_fee',
'minimum_nights', 'number_of_reviews', 'last_review', 'reviews_per_month',
'review_rate_number', 'calculated_host_listings_count', 'availability_365',
'house_rules', 'license']
```

[997]:
```
logging.info("Analizando valores únicos para columnas con 2 o menos valores␣
 ↪únicos en df_cleaning.")

if not df_cleaning.empty:
    for column in df_cleaning.columns:
        unique_values = df_cleaning[column].unique()
        num_unique_values = df_cleaning[column].nunique(dropna=False) # Contar␣
 ↪NaN como un valor único si está presente
```

```
        if num_unique_values == 3:
            print(f"\nColumna: '{column}'")
            print(f"  Número de valores únicos (incluyendo NaN si existe):␣
 ↪{num_unique_values}")
            print(f"  Valores únicos: {unique_values}")
            print("  Conteo de cada valor único (incluyendo NaN):")
            print(df_cleaning[column].value_counts(dropna=False).to_markdown())
            logging.info(f"Valores únicos para '{column}': {unique_values}␣
 ↪(Total: {num_unique_values})")
        # else:
            # logging.debug(f"Columna '{column}' tiene {num_unique_values}␣
 ↪valores únicos. No se muestra.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede analizar␣
 ↪valores únicos.")
```

```
2025-05-17 02:01:16,392 - INFO - Analizando valores únicos para columnas con 2 o
menos valores únicos en df_cleaning.
2025-05-17 02:01:16,455 - INFO - Valores únicos para 'host_identity_verified':
['unconfirmed' 'verified' None] (Total: 3)
2025-05-17 02:01:16,602 - INFO - Valores únicos para 'instant_bookable':
['false' 'true' None] (Total: 3)


Columna: 'host_identity_verified'
  Número de valores únicos (incluyendo NaN si existe): 3
  Valores únicos: ['unconfirmed' 'verified' None]
  Conteo de cada valor único (incluyendo NaN):
| host_identity_verified   |   count |
|:-------------------------|--------:|
| unconfirmed              |   50944 |
| verified                 |   50825 |
|                          |     289 |

Columna: 'instant_bookable'
  Número de valores únicos (incluyendo NaN si existe): 3
  Valores únicos: ['false' 'true' None]
  Conteo de cada valor único (incluyendo NaN):
| instant_bookable   |   count |
|:-------------------|--------:|
| false              |   51186 |
| true               |   50767 |
|                    |     105 |
```

[998]:
```
# Celda 7: Transformación de Tipos de Datos - BOOLEANAS
```

```
logging.info("Celda 7: Iniciando transformación de tipos de datos - Columnas␣
 ↪Booleanas.")

if not df_cleaning.empty:
    try:

        # --- host_identity_verified ---
        # Columna original: 'host_identity_verified' (puede ser 'verified',␣
 ↪'unconfirmed', NaN, u otros strings)
        if 'host_identity_verified' in df_cleaning.columns:
            logging.info("Procesando 'host_identity_verified'.")

            # Nueva columna: 'is_host_unconfirmed'
            # True si host_identity_verified es 'unconfirmed', False para␣
 ↪'verified' o NaN/otros.
            df_cleaning['is_host_unconfirmed'] =␣
 ↪(df_cleaning['host_identity_verified'].astype(str).str.lower() ==␣
 ↪'unconfirmed')
            logging.info("Columna 'is_host_unconfirmed' creada. True si␣
 ↪original es 'unconfirmed', False en otros casos.")

            # Nueva columna: 'is_host_verified'
            # True si host_identity_verified es 'verified', False para␣
 ↪'unconfirmed' o NaN/otros.
            df_cleaning['is_host_verified'] =␣
 ↪(df_cleaning['host_identity_verified'].astype(str).str.lower() == 'verified')
            logging.info("Columna 'is_host_verified' creada. True si original␣
 ↪es 'verified', False en otros casos.")

        else:
            logging.warning("Columna 'host_identity_verified' no encontrada. No␣
 ↪se crearon 'is_host_unconfirmed' ni 'is_host_verified'.")

        # --- instant_bookable ---
        # Columna original: 'instant_bookable' (puede ser 'TRUE', 'FALSE', NaN,␣
 ↪u otros strings)
        if 'instant_bookable' in df_cleaning.columns:
            logging.info("Procesando 'instant_bookable'.")

            # Nueva columna: 'is_instant_bookable_false' (Tu lógica: si␣
 ↪original es 'FALSE' -> True, sino False)
            # Aquí "False" se refiere al valor string 'FALSE' en la columna␣
 ↪original.
            df_cleaning['is_instant_bookable_false_policy'] =␣
 ↪(df_cleaning['instant_bookable'].astype(str).str.upper() == 'FALSE')
```

```
            logging.info("Columna 'is_instant_bookable_false_policy' creada. True si original es 'FALSE', False en otros casos.")

            # Nueva columna: 'is_instant_bookable_true' (Tu lógica: si original es 'TRUE' -> True, sino False)
            df_cleaning['is_instant_bookable_true_policy'] = (df_cleaning['instant_bookable'].astype(str).str.upper() == 'TRUE')
            logging.info("Columna 'is_instant_bookable_true_policy' creada. True si original es 'TRUE', False en otros casos.")

        else:
            logging.warning("Columna 'instant_bookable' no encontrada. No se crearon 'is_instant_bookable_false_policy' ni 'is_instant_bookable_true_policy'.")

        # Eliminar las columnas originales después de la transformación
        cols_to_drop_after_bool_transform = []
        if 'host_identity_verified' in df_cleaning.columns:
            cols_to_drop_after_bool_transform.append('host_identity_verified')
        if 'instant_bookable' in df_cleaning.columns:
            cols_to_drop_after_bool_transform.append('instant_bookable')

        if cols_to_drop_after_bool_transform:
            df_cleaning.drop(columns=cols_to_drop_after_bool_transform, inplace=True)
            logging.info(f"Columnas originales {cols_to_drop_after_bool_transform} eliminadas después de crear sus derivados booleanos.")
        logging.info("Transformaciones de tipo booleano completadas.")

    except Exception as e:
        logging.error(f"Ocurrió un error durante la transformación de tipos booleanos: {e}")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se pueden realizar transformaciones de tipo booleano.")
```

```
2025-05-17 02:01:16,757 - INFO - Celda 7: Iniciando transformación de tipos de datos - Columnas Booleanas.
2025-05-17 02:01:16,759 - INFO - Procesando 'host_identity_verified'.
2025-05-17 02:01:16,759 - INFO - Procesando 'host_identity_verified'.
2025-05-17 02:01:16,797 - INFO - Columna 'is_host_unconfirmed' creada. True si original es 'unconfirmed', False en otros casos.
2025-05-17 02:01:16,836 - INFO - Columna 'is_host_verified' creada. True si original es 'verified', False en otros casos.
2025-05-17 02:01:16,837 - INFO - Procesando 'instant_bookable'.
2025-05-17 02:01:16,878 - INFO - Columna 'is_instant_bookable_false_policy'
```

```
creada. True si original es 'FALSE', False en otros casos.
2025-05-17 02:01:16,921 - INFO - Columna 'is_instant_bookable_true_policy'
creada. True si original es 'TRUE', False en otros casos.
2025-05-17 02:01:16,971 - INFO - Columnas originales ['host_identity_verified',
'instant_bookable'] eliminadas después de crear sus derivados booleanos.
2025-05-17 02:01:16,971 - INFO - Transformaciones de tipo booleano completadas.
```

[999]:
```python
logging.info("Analizando valores únicos para columnas con 2 o menos valores␣
 ↪únicos en df_cleaning.")

if not df_cleaning.empty:
    for column in df_cleaning.columns:
        unique_values = df_cleaning[column].unique()
        num_unique_values = df_cleaning[column].nunique(dropna=False) # Contar␣
 ↪NaN como un valor único si está presente

        if num_unique_values >= 3 and num_unique_values <=20:
            print(f"\nColumna: '{column}'")
            print(f"  Número de valores únicos (incluyendo NaN si existe):␣
 ↪{num_unique_values}")
            print(f"  Valores únicos: {unique_values}")
            print("  Conteo de cada valor único (incluyendo NaN):")
            print(df_cleaning[column].value_counts(dropna=False).to_markdown())
            logging.info(f"Valores únicos para '{column}': {unique_values}␣
 ↪(Total: {num_unique_values})")
        # else:
            # logging.debug(f"Columna '{column}' tiene {num_unique_values}␣
 ↪valores únicos. No se muestra.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede analizar␣
 ↪valores únicos.")
```

```
2025-05-17 02:01:16,978 - INFO - Analizando valores únicos para columnas con 2 o
menos valores únicos en df_cleaning.
2025-05-17 02:01:17,069 - INFO - Valores únicos para 'neighbourhood_group':
['Brooklyn' 'Manhattan' 'brookln' 'manhatan' 'Queens' None 'Staten Island'
 'Bronx'] (Total: 8)
2025-05-17 02:01:17,172 - INFO - Valores únicos para 'cancellation_policy':
['strict' 'moderate' 'flexible' None] (Total: 4)
2025-05-17 02:01:17,225 - INFO - Valores únicos para 'room_type': ['Private
room' 'Entire home/apt' 'Shared room' 'Hotel room'] (Total: 4)


Columna: 'neighbourhood_group'
  Número de valores únicos (incluyendo NaN si existe): 8
  Valores únicos: ['Brooklyn' 'Manhattan' 'brookln' 'manhatan' 'Queens' None
'Staten Island'
 'Bronx']
```

  Conteo de cada valor único (incluyendo NaN):
neighbourhood_group	count
Manhattan	43557
Brooklyn	41630
Queens	13197
Bronx	2694
Staten Island	949
	29
manhatan	1
brookln	1

Columna: 'cancellation_policy'
  Número de valores únicos (incluyendo NaN si existe): 4
  Valores únicos: ['strict' 'moderate' 'flexible' None]
  Conteo de cada valor único (incluyendo NaN):
cancellation_policy	count
moderate	34162
strict	33929
flexible	33891
	76

Columna: 'room_type'
  Número de valores únicos (incluyendo NaN si existe): 4
  Valores únicos: ['Private room' 'Entire home/apt' 'Shared room' 'Hotel room']
  Conteo de cada valor único (incluyendo NaN):
room_type	count
Entire home/apt	53429
Private room	46306
Shared room	2208
Hotel room	115

2025-05-17 02:01:17,300 - INFO - Valores únicos para 'review_rate_number': [4.
5. 3. nan 2. 1.] (Total: 6)

Columna: 'review_rate_number'
  Número de valores únicos (incluyendo NaN si existe): 6
  Valores únicos: [4. 5. 3. nan 2. 1.]
  Conteo de cada valor único (incluyendo NaN):
review_rate_number	count
5	23251
4	23200
3	23130
2	22972
1	9186

| nan | 319 |

[1000]:
```
# Celda 8: Transformación de Tipos de Datos - CATEGÓRICAS
logging.info("Celda 8: Iniciando transformación de tipos de datos - Columnas␣
 ↪Categóricas.")

if not df_cleaning.empty:
    try:
        cols_to_category = [
            'neighbourhood_group',
            'cancellation_policy',
            'room_type',
            'review_rate_number',
            'neighbourhood'
        ]

        logging.info(f"Columnas a convertir a categórico: {cols_to_category}")

        for col_name in cols_to_category:
            if col_name in df_cleaning.columns:
                original_dtype = df_cleaning[col_name].dtype
                if pd.api.types.is_object_dtype(original_dtype) or pd.api.types.
 ↪is_string_dtype(original_dtype):
                    df_cleaning[col_name] = df_cleaning[col_name].astype(str).
 ↪str.strip().replace({'nan': pd.NA, '': pd.NA})
                elif pd.api.types.is_numeric_dtype(original_dtype) and col_name␣
 ↪== 'review_rate_number':
                    pass

                # Convertir a tipo 'category'
                df_cleaning[col_name] = df_cleaning[col_name].astype('category')

                logging.info(f"Columna '{col_name}' (tipo original:␣
 ↪{original_dtype}) convertida a category.")
                if df_cleaning[col_name].isna().any():
                    logging.info(f"La columna '{col_name}' contiene valores NaN/
 ↪pd.NA además de sus categorías.")

            else:
                logging.warning(f"Columna '{col_name}' no encontrada en␣
 ↪df_cleaning. No se pudo convertir a category.")

        logging.info("Transformaciones de tipo categórico completadas.")

    except Exception as e:
        logging.error(f"Ocurrió un error durante la transformación de tipos␣
 ↪categóricos: {e}")
```

```
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se pueden realizar␣
 ↪transformaciones de tipo categórico.")
```

```
2025-05-17 02:01:17,338 - INFO - Celda 8: Iniciando transformación de tipos de
datos - Columnas Categóricas.
2025-05-17 02:01:17,338 - INFO - Columnas a convertir a categórico:
['neighbourhood_group', 'cancellation_policy', 'room_type',
'review_rate_number', 'neighbourhood']
2025-05-17 02:01:17,338 - INFO - Columnas a convertir a categórico:
['neighbourhood_group', 'cancellation_policy', 'room_type',
'review_rate_number', 'neighbourhood']
2025-05-17 02:01:17,434 - INFO - Columna 'neighbourhood_group' (tipo original:
object) convertida a category.
2025-05-17 02:01:17,523 - INFO - Columna 'cancellation_policy' (tipo original:
object) convertida a category.
2025-05-17 02:01:17,615 - INFO - Columna 'room_type' (tipo original: object)
convertida a category.
2025-05-17 02:01:17,617 - INFO - Columna 'review_rate_number' (tipo original:
float64) convertida a category.
2025-05-17 02:01:17,618 - INFO - La columna 'review_rate_number' contiene
valores NaN/pd.NA además de sus categorías.
2025-05-17 02:01:17,710 - INFO - Columna 'neighbourhood' (tipo original: object)
convertida a category.
2025-05-17 02:01:17,711 - INFO - Transformaciones de tipo categórico
completadas.
```

[1001]:
```
# Celda 8.1: Conteo de Valores Únicos para Columnas No Categóricas y No␣
 ↪Booleanas
logging.info("Celda 8.1: Calculando el conteo de valores únicos para columnas␣
 ↪no categóricas y no booleanas.")
print("\n--- Conteo de Valores Únicos Totales (Excluyendo Categóricas y␣
 ↪Booleanas) ---")

if not df_cleaning.empty:
    unique_counts_summary = []

    columns_to_check = df_cleaning.select_dtypes(exclude=['category', 'bool',␣
 ↪'boolean']).columns

    if not columns_to_check.empty:
        for column in columns_to_check:
            try:
                # nunique(dropna=False) cuenta NaN/NaT como un valor único si␣
 ↪está presente
                num_unique = df_cleaning[column].nunique(dropna=False)
```

```
                unique_counts_summary.append({'Columna': column,␣
↪'Total_Valores_Unicos': num_unique, 'Tipo_Dato': df_cleaning[column].dtype})
            except Exception as e:
                logging.error(f"Error al calcular valores únicos para la␣
↪columna '{column}': {e}")
                unique_counts_summary.append({'Columna': column,␣
↪'Total_Valores_Unicos': 'Error', 'Tipo_Dato': df_cleaning[column].dtype})

        if unique_counts_summary:
            df_unique_summary = pd.DataFrame(unique_counts_summary)
            df_unique_summary_sorted = df_unique_summary.
↪sort_values(by='Total_Valores_Unicos', ascending=False) # Opcional: ordenar

            print(df_unique_summary_sorted.to_markdown(index=False))
            logging.info("Tabla resumen de conteo de valores únicos generada y␣
↪mostrada.")
        else:
            print("No se procesaron columnas para el resumen de valores únicos␣
↪(podría ser un error o no hay columnas que cumplan el criterio).")
            logging.info("No se procesaron columnas para el resumen de valores␣
↪únicos.")

    else:
        print("No hay columnas que no sean de tipo 'category' o 'boolean'/
↪'bool' para analizar.")
        logging.info("No se encontraron columnas no categóricas/booleanas para␣
↪el conteo de únicos.")

else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede generar␣
↪el resumen de valores únicos.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:17,718 - INFO - Celda 8.1: Calculando el conteo de valores
únicos para columnas no categóricas y no booleanas.
2025-05-17 02:01:17,897 - INFO - Tabla resumen de conteo de valores únicos
generada y mostrada.


--- Conteo de Valores Únicos Totales (Excluyendo Categóricas y Booleanas) ---
| Columna                        |   Total_Valores_Unicos | Tipo_Dato   |
|:-------------------------------|-----------------------:|:------------|
| id                             |                 102058 | int64       |
| host_id                        |                 102057 | int64       |
| name                           |                  61282 | object      |
| lat                            |                  21992 | float64     |
| long                           |                  17775 | float64     |
```

```
| host_name                          |                 13191 | object      |
| last_review                        |                  2478 | object      |
| house_rules                        |                  1977 | object      |
| price                              |                  1152 | object      |
| reviews_per_month                  |                  1017 | float64     |
| number_of_reviews                  |                   477 | float64     |
| availability_365                   |                   439 | float64     |
| service_fee                        |                   232 | object      |
| minimum_nights                     |                   154 | float64     |
| calculated_host_listings_count     |                    79 | float64     |
| construction_year                  |                    21 | float64     |
| country                            |                     2 | object      |
| country_code                       |                     2 | object      |
| license                            |                     2 | object      |
```

[1002]:
```python
# Celda 9: Transformación de Tipos de Datos - Numéricos y Fecha
logging.info("Celda 9: Iniciando transformación de tipos de datos - Numéricos y␣
 ↪Fecha.")

if not df_cleaning.empty:
    try:
        # --- Transformación de Fecha ---
        col_last_review = 'last_review' # Nombre normalizado
        if col_last_review in df_cleaning.columns:
            logging.info(f"Convirtiendo '{col_last_review}' a datetime.")
            original_non_nat = df_cleaning[col_last_review].notna().sum()
            df_cleaning[col_last_review] = pd.
 ↪to_datetime(df_cleaning[col_last_review], format='%m/%d/%Y', errors='coerce')
            coerced_nat_count = df_cleaning[col_last_review].isna().sum() -␣
 ↪(len(df_cleaning) - original_non_nat)
            if coerced_nat_count > 0:
                logging.warning(f"Columna '{col_last_review}':␣
 ↪{coerced_nat_count} valores no pudieron ser convertidos a datetime y se␣
 ↪establecieron como NaT.")
        else:
            logging.warning(f"Columna '{col_last_review}' no encontrada.")

        # --- Transformaciones Numéricas (desde Object a Float) ---
        cols_object_to_float = {
            'price': 'price', # El nombre normalizado ya es 'price'
            'service_fee': 'service_fee' # El nombre normalizado ya es␣
 ↪'service_fee'
        }

        for original_col_name, target_col_name in cols_object_to_float.items():
            if original_col_name in df_cleaning.columns:
```

```
                logging.info(f"Limpiando y convirtiendo '{original_col_name}' a␣
↪float como '{target_col_name}'.")
                original_non_nan_text = df_cleaning[original_col_name].notna().
↪sum()

                # Limpieza de strings
                cleaned_series = df_cleaning[original_col_name].astype(str) \
                    .str.replace('$', '', regex=False) \
                    .str.replace(',', '', regex=False) \
                    .str.strip()

                # Convertir a numérico
                df_cleaning[target_col_name] = pd.to_numeric(cleaned_series,␣
↪errors='coerce')

                coerced_nan_count_text = df_cleaning[target_col_name].isna().
↪sum() - (len(df_cleaning) - original_non_nan_text)
                if coerced_nan_count_text > 0:
                     logging.warning(f"Columna '{target_col_name}':␣
↪{coerced_nan_count_text} valores no pudieron ser convertidos a float y se␣
↪establecieron como NaN después de la limpieza.")
            else:
                logging.warning(f"Columna original '{original_col_name}' no␣
↪encontrada para convertir a float.")

        # --- Transformaciones Numéricas (desde Float64 a Int64) ---
        cols_float_to_int64 = [
            'number_of_reviews',
            'availability_365',
            'minimum_nights',
            'calculated_host_listings_count',
            'construction_year'
        ]

        for col_name in cols_float_to_int64:
            if col_name in df_cleaning.columns:
                if pd.api.types.is_float_dtype(df_cleaning[col_name]) or pd.api.
↪types.is_integer_dtype(df_cleaning[col_name]): # Aceptar si ya es Int64
                    logging.info(f"Convirtiendo '{col_name}' a Int64 (nullable␣
↪integer).")
                    df_cleaning[col_name] = df_cleaning[col_name].
↪astype('Int64')
                else:
                    logging.warning(f"Columna '{col_name}' no es de tipo float␣
↪o int. Tipo actual: {df_cleaning[col_name].dtype}. No se convirtió a Int64.")
            else:
```

```
                logging.warning(f"Columna '{col_name}' no encontrada para␣
 ↪convertir a Int64.")

        logging.info("Transformaciones de tipos numéricos y de fecha␣
 ↪completadas.")

    except Exception as e:
        logging.error(f"Ocurrió un error durante la transformación de tipos␣
 ↪numéricos y de fecha: {e}")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se pueden realizar␣
 ↪transformaciones numéricas ni de fecha.")
```

```
2025-05-17 02:01:17,908 - INFO - Celda 9: Iniciando transformación de tipos de
datos - Numéricos y Fecha.
2025-05-17 02:01:17,909 - INFO - Convirtiendo 'last_review' a datetime.
2025-05-17 02:01:17,909 - INFO - Convirtiendo 'last_review' a datetime.
2025-05-17 02:01:17,947 - INFO - Limpiando y convirtiendo 'price' a float como
'price'.
2025-05-17 02:01:18,048 - INFO - Limpiando y convirtiendo 'service_fee' a float
como 'service_fee'.
2025-05-17 02:01:18,149 - INFO - Convirtiendo 'number_of_reviews' a Int64
(nullable integer).
2025-05-17 02:01:18,155 - INFO - Convirtiendo 'availability_365' a Int64
(nullable integer).
2025-05-17 02:01:18,161 - INFO - Convirtiendo 'minimum_nights' a Int64 (nullable
integer).
2025-05-17 02:01:18,166 - INFO - Convirtiendo 'calculated_host_listings_count' a
Int64 (nullable integer).
2025-05-17 02:01:18,171 - INFO - Convirtiendo 'construction_year' a Int64
(nullable integer).
2025-05-17 02:01:18,177 - INFO - Transformaciones de tipos numéricos y de fecha
completadas.
```

[1003]:
```
# Celda 9: Eliminación de la Columna 'license'
logging.info("Celda 6: Intentando eliminar la columna 'license' (normalizada).")
if not df_cleaning.empty:

    column_to_drop_normalized = 'license'

    if column_to_drop_normalized in df_cleaning.columns:
        df_cleaning.drop(columns=[column_to_drop_normalized], inplace=True)
        logging.info(f"Columna '{column_to_drop_normalized}' eliminada␣
 ↪exitosamente.")
        logging.info(f"Dimensiones de df_cleaning después de eliminar␣
 ↪'{column_to_drop_normalized}': {df_cleaning.shape}")
    else:
```

```
        logging.warning(f"La columna '{column_to_drop_normalized}' no se␣
 ↪encontró en df_cleaning. No se realizó ninguna eliminación.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede eliminar␣
 ↪la columna.")
```

```
2025-05-17 02:01:18,184 - INFO - Celda 6: Intentando eliminar la columna
'license' (normalizada).
2025-05-17 02:01:18,217 - INFO - Columna 'license' eliminada exitosamente.
2025-05-17 02:01:18,218 - INFO - Dimensiones de df_cleaning después de eliminar
'license': (102058, 27)
```

[1004]:
```
# Celda 10.1: Identificación de Valores Fuera de Rango en 'availability_365'
logging.info("Celda 10.1: Identificando valores fuera de rango en␣
 ↪'availability_365'.")

if not df_cleaning.empty:
    col_name = 'availability_365' # Nombre ya normalizado

    if col_name in df_cleaning.columns:
        # Verificar si la columna es numérica, si no, intentar convertir␣
 ↪(aunque ya debería serlo por Celda 9)
        if not pd.api.types.is_numeric_dtype(df_cleaning[col_name]):
            logging.warning(f"La columna '{col_name}' no es de tipo numérico␣
 ↪(tipo: {df_cleaning[col_name].dtype}). Intentando convertir a Int64.")
            try:
                df_cleaning[col_name] = pd.to_numeric(df_cleaning[col_name],␣
 ↪errors='coerce').astype('Int64')
                logging.info(f"Columna '{col_name}' convertida a Int64 para el␣
 ↪análisis.")
            except Exception as e:
                logging.error(f"No se pudo convertir '{col_name}' a Int64.␣
 ↪Omitiendo análisis para esta columna. Error: {e}")

        # Proceder solo si la columna es numérica (o se pudo convertir)
        if pd.api.types.is_numeric_dtype(df_cleaning[col_name]):
            logging.info(f"Analizando columna '{col_name}' (Tipo actual:␣
 ↪{df_cleaning[col_name].dtype}).")

            # Identificar valores < 0
            valores_menores_a_cero = df_cleaning[df_cleaning[col_name] <␣
 ↪0][col_name]
            count_menores_a_cero = len(valores_menores_a_cero)

            # Límite superior realista (365, ya que el máximo original era muy␣
 ↪alto)
```

```
            limite_superior_realista = 365
            valores_mayores_limite = df_cleaning[df_cleaning[col_name] >␣
 ↪limite_superior_realista][col_name]
            count_mayores_limite = len(valores_mayores_limite)

            logging.info(f"Para '{col_name}':")
            logging.info(f"  - Número de valores < 0: {count_menores_a_cero}")
            if count_menores_a_cero > 0:
                unique_negatives = valores_menores_a_cero.unique().tolist()
                logging.info(f"    Valores únicos encontrados < 0:␣
 ↪{unique_negatives}")

            logging.info(f"  - Número de valores > {limite_superior_realista}:␣
 ↪{count_mayores_limite}")
            if count_mayores_limite > 0:
                unique_positives_outliers = valores_mayores_limite.unique()[:
 ↪10].tolist() # Mostrar hasta 20 únicos
                logging.info(f"    Valores únicos encontrados >␣
 ↪{limite_superior_realista} (hasta 20): {unique_positives_outliers}")
        else:
            pass

    else:
        logging.warning(f"Columna '{col_name}' no encontrada en df_cleaning.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede procesar␣
 ↪'availability_365'.")
```

```
2025-05-17 02:01:18,226 - INFO - Celda 10.1: Identificando valores fuera de
rango en 'availability_365'.
2025-05-17 02:01:18,228 - INFO - Analizando columna 'availability_365' (Tipo
actual: Int64).
2025-05-17 02:01:18,228 - INFO - Analizando columna 'availability_365' (Tipo
actual: Int64).
2025-05-17 02:01:18,240 - INFO - Para 'availability_365':
2025-05-17 02:01:18,241 - INFO -   - Número de valores < 0: 431
2025-05-17 02:01:18,242 - INFO -     Valores únicos encontrados < 0: [-9, -10,
-2, -1, -6, -4, -8, -5, -3, -7]
2025-05-17 02:01:18,243 - INFO -   - Número de valores > 365: 2754
2025-05-17 02:01:18,244 - INFO -     Valores únicos encontrados > 365 (hasta
20): [374, 375, 372, 383, 411, 416, 405, 393, 400, 417]
```

[1005]:
```
# Celda 10.2: Aplicación de Corrección (Clipping) a 'availability_365' y␣
 ↪Verificación
logging.info("Celda 10.2: Aplicando corrección (clipping) a 'availability_365'.
 ↪")
```

```
if not df_cleaning.empty:
    col_name = 'availability_365'

    if col_name in df_cleaning.columns and pd.api.types.
 ↪is_numeric_dtype(df_cleaning[col_name]):
        limite_superior_realista = 365

        df_cleaning[col_name] = df_cleaning[col_name].clip(lower=1,␣
 ↪upper=limite_superior_realista)

        logging.info(f"Valores en '{col_name}' corregidos (clipping entre 1 y␣
 ↪{limite_superior_realista}).")

        new_min = df_cleaning[col_name].min()
        new_max = df_cleaning[col_name].max()

        logging.info(f"Nuevo rango para '{col_name}' después de la corrección:␣
 ↪Min={new_min}, Max={new_max}")

    elif col_name not in df_cleaning.columns:
        logging.warning(f"Columna '{col_name}' no encontrada en df_cleaning␣
 ↪para aplicar corrección.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_name]):
        logging.warning(f"Columna '{col_name}' no es de tipo numérico. No se␣
 ↪aplicó la corrección de clipping.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede aplicar␣
 ↪corrección a 'availability_365'.")
```

```
2025-05-17 02:01:18,251 - INFO - Celda 10.2: Aplicando corrección (clipping) a
'availability_365'.
2025-05-17 02:01:18,256 - INFO - Valores en 'availability_365' corregidos
(clipping entre 1 y 365).
2025-05-17 02:01:18,259 - INFO - Nuevo rango para 'availability_365' después de
la corrección: Min=1, Max=365
2025-05-17 02:01:18,256 - INFO - Valores en 'availability_365' corregidos
(clipping entre 1 y 365).
2025-05-17 02:01:18,259 - INFO - Nuevo rango para 'availability_365' después de
la corrección: Min=1, Max=365
```

[1006]:
```
# Celda 11.1: Identificación de Valores Fuera de Rango en 'minimum_nights'
logging.info("Celda 11.1: Identificando valores fuera de rango en␣
 ↪'minimum_nights'.")

if not df_cleaning.empty:
    col_name_mn = 'minimum_nights' # Nombre ya normalizado
```

```
  if col_name_mn in df_cleaning.columns:
      # Verificar si la columna es numérica (ya debería ser Int64 por Celda 9)
      if not pd.api.types.is_numeric_dtype(df_cleaning[col_name_mn]):
          logging.warning(f"La columna '{col_name_mn}' no es de tipo numérico␣
↪(tipo: {df_cleaning[col_name_mn].dtype}). Intentando convertir a Int64.")
          try:
              df_cleaning[col_name_mn] = pd.
↪to_numeric(df_cleaning[col_name_mn], errors='coerce').astype('Int64')
              logging.info(f"Columna '{col_name_mn}' convertida a Int64 para␣
↪el análisis.")
          except Exception as e:
              logging.error(f"No se pudo convertir '{col_name_mn}' a Int64.␣
↪Omitiendo análisis para esta columna. Error: {e}")

      # Proceder solo si la columna es numérica
      if pd.api.types.is_numeric_dtype(df_cleaning[col_name_mn]):
          logging.info(f"Analizando columna '{col_name_mn}' (Tipo actual:␣
↪{df_cleaning[col_name_mn].dtype}).")

          limite_inferior_mn = 1
          valores_menores_limite_inf = df_cleaning[df_cleaning[col_name_mn] <␣
↪limite_inferior_mn][col_name_mn]
          count_menores_limite_inf = len(valores_menores_limite_inf)

          limite_superior_mn = 365
          valores_mayores_limite_sup = df_cleaning[df_cleaning[col_name_mn] >␣
↪limite_superior_mn][col_name_mn]
          count_mayores_limite_sup = len(valores_mayores_limite_sup)

          logging.info(f"Para '{col_name_mn}':")
          logging.info(f"  - Número de valores < {limite_inferior_mn}:␣
↪{count_menores_limite_inf}")
          if count_menores_limite_inf > 0:
              unique_negatives_or_zero = valores_menores_limite_inf.unique().
↪tolist()
              logging.info(f"    Valores únicos encontrados <␣
↪{limite_inferior_mn}: {unique_negatives_or_zero}")

          logging.info(f"  - Número de valores > {limite_superior_mn}:␣
↪{count_mayores_limite_sup}")
          if count_mayores_limite_sup > 0:
              unique_positives_outliers_mn = valores_mayores_limite_sup.
↪unique()[:10].tolist() # Mostrar hasta 20 únicos
              logging.info(f"    Valores únicos encontrados >␣
↪{limite_superior_mn} (hasta 20): {unique_positives_outliers_mn}")
      else:
```

```
            pass # Mensaje de error ya manejado en el bloque try-except de conversión

    else:
        logging.warning(f"Columna '{col_name_mn}' no encontrada en df_cleaning.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede procesar '{col_name_mn}'.")
```

```
2025-05-17 02:01:18,269 - INFO - Celda 11.1: Identificando valores fuera de
rango en 'minimum_nights'.
2025-05-17 02:01:18,270 - INFO - Analizando columna 'minimum_nights' (Tipo
actual: Int64).
2025-05-17 02:01:18,273 - INFO - Para 'minimum_nights':
2025-05-17 02:01:18,274 - INFO -   - Número de valores < 1: 13
2025-05-17 02:01:18,276 - INFO -     Valores únicos encontrados < 1: [-10, -5,
-1, -12, -2, -3, -1223, -365, -200, -125]
2025-05-17 02:01:18,277 - INFO -   - Número de valores > 365: 35
2025-05-17 02:01:18,270 - INFO - Analizando columna 'minimum_nights' (Tipo
actual: Int64).
2025-05-17 02:01:18,273 - INFO - Para 'minimum_nights':
2025-05-17 02:01:18,274 - INFO -   - Número de valores < 1: 13
2025-05-17 02:01:18,276 - INFO -     Valores únicos encontrados < 1: [-10, -5,
-1, -12, -2, -3, -1223, -365, -200, -125]
2025-05-17 02:01:18,277 - INFO -   - Número de valores > 365: 35
2025-05-17 02:01:18,277 - INFO -     Valores únicos encontrados > 365 (hasta
20): [371, 366, 399, 452, 3455, 398, 370, 1000, 1250, 500]
```

[1007]:
```
# Celda 11.2: Aplicación de Corrección (Clipping) a 'minimum_nights' y Verificación
logging.info("Celda 11.2: Aplicando corrección (clipping) a 'minimum_nights'.")

if not df_cleaning.empty:
    col_name_mn = 'minimum_nights' # Nombre ya normalizado

    if col_name_mn in df_cleaning.columns and pd.api.types.is_numeric_dtype(df_cleaning[col_name_mn]):
        limite_inferior_mn = 1
        limite_superior_mn = 365

        # Aplicar clip:
        df_cleaning[col_name_mn] = df_cleaning[col_name_mn].clip(lower=limite_inferior_mn, upper=limite_superior_mn)

        logging.info(f"Valores en '{col_name_mn}' corregidos (clipping entre {limite_inferior_mn} y {limite_superior_mn}).")
```

```
        # Verificar después de la corrección
        new_min_mn = df_cleaning[col_name_mn].min()
        new_max_mn = df_cleaning[col_name_mn].max()

        logging.info(f"Nuevo rango para '{col_name_mn}' después de la␣
 ↪corrección: Min={new_min_mn}, Max={new_max_mn}")

    elif col_name_mn not in df_cleaning.columns:
        logging.warning(f"Columna '{col_name_mn}' no encontrada en df_cleaning␣
 ↪para aplicar corrección.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_name_mn]): # Este␣
 ↪caso debería ser cubierto por la Celda 11.1 si la conversión falló
        logging.warning(f"Columna '{col_name_mn}' no es de tipo numérico. No se␣
 ↪aplicó la corrección de clipping.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede aplicar␣
 ↪corrección a '{col_name_mn}'.")
```

```
2025-05-17 02:01:18,286 - INFO - Celda 11.2: Aplicando corrección (clipping) a
'minimum_nights'.
2025-05-17 02:01:18,294 - INFO - Valores en 'minimum_nights' corregidos
(clipping entre 1 y 365).
2025-05-17 02:01:18,295 - INFO - Nuevo rango para 'minimum_nights' después de la
corrección: Min=1, Max=365
2025-05-17 02:01:18,294 - INFO - Valores en 'minimum_nights' corregidos
(clipping entre 1 y 365).
2025-05-17 02:01:18,295 - INFO - Nuevo rango para 'minimum_nights' después de la
corrección: Min=1, Max=365
```

[1008]:
```
# Mostrar el resumen estadístico de las columnas numéricas de df_cleaning como␣
 ↪tabla markdown
numeric_summary = df_cleaning.select_dtypes(include=np.number).describe().T
print(numeric_summary.to_markdown())
```

```
|                               |   count |           mean |           std |
min |           25% |           50% |           75% |           max |
|:------------------------------|--------:|---------------:|--------------:|---
------------:|--------------:|--------------:|--------------:|--------------
:|
| id                            |  102058 |    2.91844e+07 |   1.62717e+07 |
1.00125e+06 |    1.50929e+07 |    2.91844e+07 |    4.32759e+07 |    5.73674e+07
|
| host_id                       |  102058 |    4.92674e+10 |   2.85374e+10 |
1.23601e+08 |    2.45992e+10 |    4.91287e+10 |    7.40062e+10 |    9.87631e+10
|
| lat                           |  102050 |   40.7281      |   0.0558524   |
```

```
40.4998      |   40.6887      |   40.7223      |   40.7628      |   40.917
|
| long                          |  102050 |  -73.9497      |   0.0495016   |
-74.2498      |  -73.9826      |  -73.9544      |  -73.9323      |  -73.7052
|
| construction_year             |  101844 | 2012.49        |   5.76584     |
2003          | 2007           | 2012           | 2017           | 2022
|
| price                         |  101811 |  625.356       | 331.673       |
50            |  340           |  625           |  913           | 1200
|
| service_fee                   |  101785 |  125.039       |  66.3259      |
10            |   68           |  125           |  183           |  240
|
| minimum_nights                |  101658 |    7.96793     |  18.3028      |
1             |    2           |    3           |    5           |  365
|
| number_of_reviews             |  101875 |   27.5179      |  49.5717      |
0             |    1           |    7           |   31           | 1024
|
| reviews_per_month             |   86240 |    1.37541     |   1.74802     |
0.01          |    0.22        |    0.74        |    2.01        |   90
|
| calculated_host_listings_count |  101739 |    7.93694     |  32.2664      |
1             |    1           |    1           |    2           |  332
|
| availability_365              |  101610 |  140.439       | 133.191       |
1             |    3           |   96           |  268           |  365
|
```

[1009]:
```python
# Celda 12.1: Imputación de Nulos en 'lat' y 'long'
logging.info("Celda 12.1: Iniciando imputación de nulos para 'lat' y 'long'.")

if not df_cleaning.empty:
    cols_geo = ['lat', 'long']

    for col_geo in cols_geo:
        if col_geo not in df_cleaning.columns:
            logging.warning(f"Columna '{col_geo}' no encontrada. Omitiendo␣
 ↪imputación para esta columna.")
            continue

        nulos_antes = df_cleaning[col_geo].isnull().sum()
        logging.info(f"Nulos en '{col_geo}' ANTES de la imputación:␣
 ↪{nulos_antes}")

        if nulos_antes == 0:
```

```
            logging.info(f"No hay nulos que imputar en '{col_geo}'.")
            continue

        for index in df_cleaning[df_cleaning[col_geo].isnull()].index:
            current_neighbourhood = df_cleaning.loc[index, 'neighbourhood']
            current_neighbourhood_group = df_cleaning.loc[index,␣
↪'neighbourhood_group']

            imputed_value = pd.NA

            # Estrategia 1: Usar 'neighbourhood'
            if pd.notna(current_neighbourhood):
                mean_val_neighbourhood = df_cleaning[
                    (df_cleaning['neighbourhood'] == current_neighbourhood) &
                    (df_cleaning.index != index) &
                    (df_cleaning[col_geo].notna())
                ][col_geo].mean()

                if pd.notna(mean_val_neighbourhood):
                    imputed_value = mean_val_neighbourhood
                    logging.debug(f"Imputando '{col_geo}' para índice {index}␣
↪con media de neighbourhood '{current_neighbourhood}': {imputed_value}")
                else:
                    logging.warning(f"No se pudo calcular la media de␣
↪'{col_geo}' para neighbourhood '{current_neighbourhood}' (índice {index}).␣
↪Intentando con neighbourhood_group.")

            # Estrategia 2: Usar 'neighbourhood_group' (si la Estrategia 1␣
↪falló o 'neighbourhood' era nulo)
            if pd.isna(imputed_value) and pd.notna(current_neighbourhood_group):
                mean_val_group = df_cleaning[
                    (df_cleaning['neighbourhood_group'] ==␣
↪current_neighbourhood_group) &
                    (df_cleaning.index != index) &
                    (df_cleaning[col_geo].notna())
                ][col_geo].mean()

                if pd.notna(mean_val_group):
                    imputed_value = mean_val_group
                    logging.debug(f"Imputando '{col_geo}' para índice {index}␣
↪con media de neighbourhood_group '{current_neighbourhood_group}':␣
↪{imputed_value}")
                else:
                    logging.warning(f"No se pudo calcular la media de␣
↪'{col_geo}' para neighbourhood_group '{current_neighbourhood_group}' (índice␣
↪{index}).")
```

```
            # Aplicar el valor imputado (si se encontró alguno)
            if pd.notna(imputed_value):
                df_cleaning.loc[index, col_geo] = imputed_value
            else:
                logging.warning(f"No se pudo imputar '{col_geo}' para el índice␣
↪{index} usando neighbourhood o neighbourhood_group. El valor permanece nulo.
↪")

        nulos_despues = df_cleaning[col_geo].isnull().sum()
        logging.info(f"Nulos en '{col_geo}' DESPUÉS de la imputación:␣
↪{nulos_despues}")
        if nulos_antes > nulos_despues:
            print(f"Se imputaron {nulos_antes - nulos_despues} valores en␣
↪'{col_geo}'.")
        elif nulos_antes == nulos_despues and nulos_antes > 0:
            print(f"No se pudieron imputar los nulos restantes en '{col_geo}'␣
↪con la estrategia actual.")

    # Verificar rangos de lat/long después de la imputación (opcional, pero␣
↪bueno)
    if 'lat' in df_cleaning.columns and df_cleaning['lat'].notna().any():
        print(f"\nNuevo rango para 'lat': Min={df_cleaning['lat'].min():.4f},␣
↪Max={df_cleaning['lat'].max():.4f}")
    if 'long' in df_cleaning.columns and df_cleaning['long'].notna().any():
        print(f"Nuevo rango para 'long': Min={df_cleaning['long'].min():.4f},␣
↪Max={df_cleaning['long'].max():.4f}")

else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
↪la imputación de 'lat'/'long'.")
```

```
2025-05-17 02:01:18,411 - INFO - Celda 12.1: Iniciando imputación de nulos para
'lat' y 'long'.
2025-05-17 02:01:18,413 - INFO - Nulos en 'lat' ANTES de la imputación: 8
2025-05-17 02:01:18,435 - INFO - Nulos en 'lat' DESPUÉS de la imputación: 0
2025-05-17 02:01:18,436 - INFO - Nulos en 'long' ANTES de la imputación: 8
2025-05-17 02:01:18,456 - INFO - Nulos en 'long' DESPUÉS de la imputación: 0

Se imputaron 8 valores en 'lat'.
Se imputaron 8 valores en 'long'.

Nuevo rango para 'lat': Min=40.4998, Max=40.9170
Nuevo rango para 'long': Min=-74.2498, Max=-73.7052
```

[1010]:
```
# Celda 12.4: Imputación de Nulos en 'country' y 'country_code'
```

```
logging.info("Celda 12.4: Iniciando imputación de nulos para 'country' y␣
 ↪'country_code'.")
print("\n--- Celda 12.4: Imputación de Nulos en 'country' y 'country_code' ---")

if not df_cleaning.empty:
    cols_to_impute_mode = ['country', 'country_code'] # Nombres ya normalizados

    for col_name in cols_to_impute_mode:
        if col_name not in df_cleaning.columns:
            logging.warning(f"Columna '{col_name}' no encontrada. Omitiendo␣
 ↪imputación para esta columna.")
            print(f"Advertencia: Columna '{col_name}' no encontrada. Se omite␣
 ↪su imputación.")
            continue

        nulos_antes_cc = df_cleaning[col_name].isnull().sum()
        logging.info(f"Nulos en '{col_name}' ANTES de la imputación:␣
 ↪{nulos_antes_cc}")
        print(f"\nNulos en '{col_name}' ANTES de la imputación:␣
 ↪{nulos_antes_cc}")

        if nulos_antes_cc == 0:
            logging.info(f"No hay nulos que imputar en '{col_name}'.")
            print(f"No hay nulos que imputar en '{col_name}'.")
            continue

        # Calcular la moda de la columna (excluyendo NaNs para el cálculo de la␣
 ↪moda)
        moda_col = df_cleaning[col_name].dropna().mode()

        if not moda_col.empty:
            valor_moda = moda_col.iloc[0] # Tomar la primera moda si hay varias
            logging.info(f"La moda para '{col_name}' es: '{valor_moda}'")
            print(f"La moda para '{col_name}' es: '{valor_moda}'")

            # Imputar los valores nulos con la moda
            df_cleaning[col_name].fillna(valor_moda, inplace=True)
            imputed_count_cc = nulos_antes_cc - df_cleaning[col_name].isnull().
 ↪sum() # Contar cuántos se imputaron

            nulos_despues_cc = df_cleaning[col_name].isnull().sum()
            logging.info(f"Nulos en '{col_name}' DESPUÉS de la imputación:␣
 ↪{nulos_despues_cc}")
            print(f"Nulos en '{col_name}' DESPUÉS de la imputación:␣
 ↪{nulos_despues_cc}")
            if imputed_count_cc > 0:
```

```
                print(f"Se imputaron {imputed_count_cc} valores en '{col_name}'␣
↪con '{valor_moda}'.")

            # Opcional: Convertir a tipo 'category' después de la imputación si␣
↪no lo es ya
            # (Ya deberían serlo por la Celda 8 si tenían pocos valores únicos␣
↪o se incluyeron allí)
            if not pd.api.types.is_categorical_dtype(df_cleaning[col_name]):
                if df_cleaning[col_name].nunique(dropna=False) <= 10: # Umbral␣
↪de ejemplo
                    logging.info(f"Convirtiendo '{col_name}' a tipo 'category'␣
↪después de la imputación.")
                    df_cleaning[col_name] = df_cleaning[col_name].
↪astype('category')
                    print(f"Columna '{col_name}' convertida a tipo 'category'.␣
↪Nuevo tipo: {df_cleaning[col_name].dtype}")
        else:
            logging.warning(f"No se pudo determinar la moda para '{col_name}'␣
↪(la columna podría ser completamente nula o tener problemas). No se␣
↪imputaron valores.")
            print(f"Advertencia: No se pudo determinar la moda para␣
↪'{col_name}'. No se imputaron valores.")

else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
↪la imputación.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,466 - INFO - Celda 12.4: Iniciando imputación de nulos para
'country' y 'country_code'.
2025-05-17 02:01:18,477 - INFO - Nulos en 'country' ANTES de la imputación: 532
2025-05-17 02:01:18,499 - INFO - La moda para 'country' es: 'United States'
2025-05-17 02:01:18,527 - INFO - Nulos en 'country' DESPUÉS de la imputación: 0
/tmp/ipykernel_1305523/1937374830.py:43: DeprecationWarning:
is_categorical_dtype is deprecated and will be removed in a future version. Use
isinstance(dtype, pd.CategoricalDtype) instead
  if not pd.api.types.is_categorical_dtype(df_cleaning[col_name]):
2025-05-17 02:01:18,544 - INFO - Convirtiendo 'country' a tipo 'category'
después de la imputación.
2025-05-17 02:01:18,569 - INFO - Nulos en 'country_code' ANTES de la imputación:
131
2025-05-17 02:01:18,595 - INFO - La moda para 'country_code' es: 'US'


--- Celda 12.4: Imputación de Nulos en 'country' y 'country_code' ---

Nulos en 'country' ANTES de la imputación: 532
```

```
La moda para 'country' es: 'United States'
Nulos en 'country' DESPUÉS de la imputación: 0
Se imputaron 532 valores en 'country' con 'United States'.
Columna 'country' convertida a tipo 'category'. Nuevo tipo: category

Nulos en 'country_code' ANTES de la imputación: 131
La moda para 'country_code' es: 'US'

2025-05-17 02:01:18,623 - INFO - Nulos en 'country_code' DESPUÉS de la
imputación: 0
/tmp/ipykernel_1305523/1937374830.py:43: DeprecationWarning:
is_categorical_dtype is deprecated and will be removed in a future version. Use
isinstance(dtype, pd.CategoricalDtype) instead
  if not pd.api.types.is_categorical_dtype(df_cleaning[col_name]):

Nulos en 'country_code' DESPUÉS de la imputación: 0
Se imputaron 131 valores en 'country_code' con 'US'.

2025-05-17 02:01:18,637 - INFO - Convirtiendo 'country_code' a tipo 'category'
después de la imputación.

Columna 'country_code' convertida a tipo 'category'. Nuevo tipo: category
```

[1011]:
```python
# Celda 12.5: Eliminación de la Columna 'host_name'
logging.info("Celda 12.5: Eliminando la columna 'host_name'.")
print("\n--- Celda 12.5: Eliminación de la Columna 'host_name' ---")

if not df_cleaning.empty:
    col_to_drop_hostname = 'host_name' # Nombre ya normalizado

    if col_to_drop_hostname in df_cleaning.columns:
        nulos_en_hostname_antes = df_cleaning[col_to_drop_hostname].isnull().
 ↪sum()
        logging.info(f"Columna '{col_to_drop_hostname}' encontrada. Nulos antes␣
 ↪de eliminar: {nulos_en_hostname_antes}.")
        print(f"Columna '{col_to_drop_hostname}' encontrada. Nulos antes de␣
 ↪eliminar: {nulos_en_hostname_antes}.")

        df_cleaning.drop(columns=[col_to_drop_hostname], inplace=True)

        logging.info(f"Columna '{col_to_drop_hostname}' eliminada exitosamente.
 ↪")
        print(f"Columna '{col_to_drop_hostname}' eliminada exitosamente.")
        logging.info(f"Dimensiones de df_cleaning después de eliminar␣
 ↪'{col_to_drop_hostname}': {df_cleaning.shape}")
        print(f"Nuevas dimensiones de df_cleaning: {df_cleaning.shape}")
    else:
        logging.warning(f"La columna '{col_to_drop_hostname}' no se encontró en␣
 ↪df_cleaning. No se realizó ninguna eliminación.")
```

```
        print(f"La columna '{col_to_drop_hostname}' no se encontró. No se␣
 ↪eliminó nada.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede eliminar␣
 ↪la columna 'host_name'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,667 - INFO - Celda 12.5: Eliminando la columna 'host_name'.


--- Celda 12.5: Eliminación de la Columna 'host_name' ---

2025-05-17 02:01:18,679 - INFO - Columna 'host_name' encontrada. Nulos antes de
eliminar: 404.
2025-05-17 02:01:18,689 - INFO - Columna 'host_name' eliminada exitosamente.
2025-05-17 02:01:18,690 - INFO - Dimensiones de df_cleaning después de eliminar
'host_name': (102058, 26)

Columna 'host_name' encontrada. Nulos antes de eliminar: 404.
Columna 'host_name' eliminada exitosamente.
Nuevas dimensiones de df_cleaning: (102058, 26)
```

[1012]:
```
# Celda 12.7.1: Identificación de Fechas Anómalas en 'last_review' (posteriores␣
 ↪a 2024)
logging.info("Celda 12.7.1: Identificando fechas anómalas (posteriores a 2024)␣
 ↪en 'last_review'.")
print("\n--- Celda 12.7.1: Identificación de Fechas Anómalas en 'last_review'␣
 ↪---")

if not df_cleaning.empty:
    col_lr = 'last_review' # Nombre ya normalizado

    if col_lr not in df_cleaning.columns:
        logging.warning(f"Columna '{col_lr}' no encontrada. Omitiendo análisis.
 ↪")
        print(f"Advertencia: Columna '{col_lr}' no encontrada.")
    elif not pd.api.types.is_datetime64_any_dtype(df_cleaning[col_lr]):
        logging.warning(f"Columna '{col_lr}' no es de tipo datetime (Tipo:␣
 ↪{df_cleaning[col_lr].dtype}). No se pueden identificar fechas anómalas.")
        print(f"Advertencia: Columna '{col_lr}' no es de tipo datetime.")
    else:
        limite_fecha_superior = pd.Timestamp('2024-12-31 23:59:59')

        logging.info(f"Límite superior para fechas válidas en '{col_lr}':␣
 ↪{limite_fecha_superior.strftime('%Y-%m-%d')}")
        print(f"Límite superior considerado para fechas válidas en '{col_lr}':␣
 ↪{limite_fecha_superior.strftime('%Y-%m-%d')}")
```

```
        fechas_anomalas_lr = df_cleaning.loc[
            df_cleaning[col_lr].notna() & (df_cleaning[col_lr] >␣
↪limite_fecha_superior), col_lr
        ]
        count_fechas_anomalas_lr = len(fechas_anomalas_lr)

        logging.info(f"Número de fechas en '{col_lr}' posteriores a␣
↪{limite_fecha_superior.strftime('%Y-%m-%d')}: {count_fechas_anomalas_lr}")
        print(f"Número de fechas en '{col_lr}' encontradas posteriores a␣
↪{limite_fecha_superior.strftime('%Y-%m-%d')}: {count_fechas_anomalas_lr}")

        if count_fechas_anomalas_lr > 0:
            print("Algunas fechas anómalas encontradas (hasta 10 ejemplos):")
            print(fechas_anomalas_lr.unique()[:10])
            logging.info(f"Ejemplos de fechas anómalas: {fechas_anomalas_lr.
↪unique()[:10].tolist()}")

        print("\nEstadísticas descriptivas de fechas (antes de corregir␣
↪anomalías):")
        if df_cleaning[col_lr].notna().any():
            # CORRECCIÓN AQUÍ: Se elimina datetime_is_numeric=True
            print(df_cleaning[col_lr].describe().to_markdown())
        else:
            print("No hay fechas no nulas para describir.")
else:
    logging.warning("El DataFrame df_cleaning está vacío.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,699 - INFO - Celda 12.7.1: Identificando fechas anómalas
(posteriores a 2024) en 'last_review'.
2025-05-17 02:01:18,701 - INFO - Límite superior para fechas válidas en
'last_review': 2024-12-31
2025-05-17 02:01:18,703 - INFO - Número de fechas en 'last_review' posteriores a
2024-12-31: 4
2025-05-17 02:01:18,704 - INFO - Ejemplos de fechas anómalas:
[Timestamp('2025-06-26 00:00:00'), Timestamp('2058-06-16 00:00:00'),
Timestamp('2026-03-28 00:00:00'), Timestamp('2040-06-16 00:00:00')]


--- Celda 12.7.1: Identificación de Fechas Anómalas en 'last_review' ---
Límite superior considerado para fechas válidas en 'last_review': 2024-12-31
Número de fechas en 'last_review' encontradas posteriores a 2024-12-31: 4
Algunas fechas anómalas encontradas (hasta 10 ejemplos):
<DatetimeArray>
['2025-06-26 00:00:00', '2058-06-16 00:00:00', '2026-03-28 00:00:00',
'2040-06-16 00:00:00']
Length: 4, dtype: datetime64[ns]
```

```
Estadísticas descriptivas de fechas (antes de corregir anomalías):
```

	last_review
count	86226
mean	2019-06-11 02:12:22.996311808
min	2012-07-11 00:00:00
25%	2018-10-27 00:00:00
50%	2019-06-13 00:00:00
75%	2019-07-05 00:00:00
max	2058-06-16 00:00:00

[1013]:
```
# Celda 12.7.2: Corrección de Fechas Anómalas en 'last_review' (reemplazar con␣
 ↪NaT)
logging.info("Celda 12.7.2: Corrigiendo fechas anómalas en 'last_review'␣
 ↪reemplazándolas con NaT.")
print("\n--- Celda 12.7.2: Corrección de Fechas Anómalas en 'last_review' ---")

if not df_cleaning.empty:
    col_lr = 'last_review'

    if col_lr in df_cleaning.columns and pd.api.types.
 ↪is_datetime64_any_dtype(df_cleaning[col_lr]):
        limite_fecha_superior = pd.Timestamp('2024-12-31 23:59:59')

        indices_anomalos = df_cleaning[
            df_cleaning[col_lr].notna() & (df_cleaning[col_lr] >␣
 ↪limite_fecha_superior)
        ].index

        count_a_corregir = len(indices_anomalos)

        if count_a_corregir > 0:
            df_cleaning.loc[indices_anomalos, col_lr] = pd.NaT
            logging.info(f"{count_a_corregir} fechas anómalas en '{col_lr}' han␣
 ↪sido reemplazadas con NaT.")
            print(f"{count_a_corregir} fechas anómalas en '{col_lr}' han sido␣
 ↪reemplazadas con NaT.")

            fechas_aun_anomalas_indices = df_cleaning[
                df_cleaning[col_lr].notna() & (df_cleaning[col_lr] >␣
 ↪limite_fecha_superior)
            ].index

            if len(fechas_aun_anomalas_indices) == 0: # Comprobar si la␣
 ↪longitud de los índices es 0
```

```
                logging.info("Verificación: No quedan fechas anómalas␣
↪(posteriores al límite).")
                print("Verificación: No quedan fechas anómalas (posteriores al␣
↪límite).")
            else:
                logging.warning("Advertencia: Todavía se detectan fechas␣
↪anómalas después de la corrección. Revisar lógica.")
                print("Advertencia: Todavía se detectan fechas anómalas después␣
↪de la corrección. Revisar lógica.")
        else:
            logging.info(f"No se encontraron fechas anómalas en '{col_lr}' para␣
↪corregir (posteriores a {limite_fecha_superior.strftime('%Y-%m-%d')}).")
            print(f"No se encontraron fechas anómalas en '{col_lr}' para␣
↪corregir.")

        print("\nEstadísticas descriptivas de fechas (después de corregir␣
↪anomalías y antes de imputar NaTs):")
        if df_cleaning[col_lr].notna().any():
             # CORRECCIÓN AQUÍ: Se elimina datetime_is_numeric=True
             print(df_cleaning[col_lr].describe().to_markdown())
        else:
            print(f"Todos los valores en '{col_lr}' son ahora NaT o la columna␣
↪está vacía.")

    elif col_lr not in df_cleaning.columns:
        logging.warning(f"Columna '{col_lr}' no encontrada.")
        print(f"Columna '{col_lr}' no encontrada.")
    else: # No es datetime
        logging.warning(f"Columna '{col_lr}' no es de tipo datetime.")
        print(f"Columna '{col_lr}' no es de tipo datetime.")
else:
    logging.warning("El DataFrame df_cleaning está vacío.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,718 - INFO - Celda 12.7.2: Corrigiendo fechas anómalas en
'last_review' reemplazándolas con NaT.
2025-05-17 02:01:18,723 - INFO - 4 fechas anómalas en 'last_review' han sido
reemplazadas con NaT.
2025-05-17 02:01:18,725 - INFO - Verificación: No quedan fechas anómalas
(posteriores al límite).


--- Celda 12.7.2: Corrección de Fechas Anómalas en 'last_review' ---
4 fechas anómalas en 'last_review' han sido reemplazadas con NaT.
Verificación: No quedan fechas anómalas (posteriores al límite).

Estadísticas descriptivas de fechas (después de corregir anomalías y antes de
```

imputar NaTs):

	last_review
count	86222
mean	2019-06-10 18:47:53.419776768
min	2012-07-11 00:00:00
25%	2018-10-27 00:00:00
50%	2019-06-13 00:00:00
75%	2019-07-05 00:00:00
max	2024-08-15 00:00:00

[1014]:
```
# Celda 12.7.3: Imputación de NaT's en 'last_review' con la Media (Corregida)
logging.info("Celda 12.7.3: Iniciando imputación de NaT's para 'last_review' con la media (corregida).")
print("\n--- Celda 12.7.3: Imputación de NaT's en 'last_review' con Media Corregida ---")

if not df_cleaning.empty:
    col_lr = 'last_review'

    if col_lr in df_cleaning.columns and pd.api.types.is_datetime64_any_dtype(df_cleaning[col_lr]):
        nulos_antes_lr_imputacion = df_cleaning[col_lr].isnull().sum()
        logging.info(f"Total de NaT's en '{col_lr}' ANTES de la imputación con media: {nulos_antes_lr_imputacion}")
        print(f"Total de NaT's en '{col_lr}' ANTES de la imputación con media: {nulos_antes_lr_imputacion}")

        if nulos_antes_lr_imputacion == 0:
            logging.info(f"No hay NaT's que imputar en '{col_lr}'.")
            print(f"No hay NaT's que imputar en '{col_lr}'.")
        else:
            mean_date_lr_corrected = df_cleaning[col_lr].dropna().mean()

            if pd.notna(mean_date_lr_corrected):
                logging.info(f"La fecha media (corregida) calculada para '{col_lr}' es: {mean_date_lr_corrected}")
                print(f"La fecha media (corregida) calculada para '{col_lr}' es: {mean_date_lr_corrected.strftime('%Y-%m-%d %H:%M:%S')}")

                df_cleaning[col_lr].fillna(mean_date_lr_corrected, inplace=True)
                imputed_count_lr_final = nulos_antes_lr_imputacion - df_cleaning[col_lr].isnull().sum()

                nulos_despues_lr_imputacion = df_cleaning[col_lr].isnull().sum()
                logging.info(f"NaT's en '{col_lr}' DESPUÉS de la imputación con media: {nulos_despues_lr_imputacion}")
```

```
                print(f"NaT's en '{col_lr}' DESPUÉS de la imputación con media:␣
↪{nulos_despues_lr_imputacion}")
                if imputed_count_lr_final > 0:
                    print(f"Se imputaron {imputed_count_lr_final} valores NaT␣
↪en '{col_lr}' con la fecha media (corregida).")
            else:
                logging.warning(f"No se pudo calcular la fecha media␣
↪(corregida) para '{col_lr}'. No se imputaron valores.")
                print(f"Advertencia: No se pudo calcular la fecha media␣
↪(corregida) para '{col_lr}'. No se imputaron valores.")

            if df_cleaning[col_lr].notna().any():
                min_date_final = df_cleaning[col_lr].min()
                max_date_final = df_cleaning[col_lr].max()
                print(f"\nNuevo rango FINAL para '{col_lr}' después de la␣
↪imputación:")
                print(f"  Fecha mínima: {min_date_final.strftime('%Y-%m-%d')}")
                print(f"  Fecha máxima: {max_date_final.strftime('%Y-%m-%d')}")
                print("\nEstadísticas descriptivas FINALES para 'last_review':")
                # CORRECCIÓN AQUÍ: Se elimina datetime_is_numeric=True
                print(df_cleaning[col_lr].describe().to_markdown())

    elif col_lr not in df_cleaning.columns:
        logging.warning(f"Columna '{col_lr}' no encontrada.")
        print(f"Columna '{col_lr}' no encontrada.")
    else: # No es datetime
        logging.warning(f"Columna '{col_lr}' no es de tipo datetime.")
        print(f"Columna '{col_lr}' no es de tipo datetime.")
else:
    logging.warning("El DataFrame df_cleaning está vacío.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,739 - INFO - Celda 12.7.3: Iniciando imputación de NaT's
para 'last_review' con la media (corregida).
2025-05-17 02:01:18,742 - INFO - Total de NaT's en 'last_review' ANTES de la
imputación con media: 15836
2025-05-17 02:01:18,744 - INFO - La fecha media (corregida) calculada para
'last_review' es: 2019-06-10 18:47:53.419777024
2025-05-17 02:01:18,745 - INFO - NaT's en 'last_review' DESPUÉS de la imputación
con media: 0


--- Celda 12.7.3: Imputación de NaT's en 'last_review' con Media Corregida ---
Total de NaT's en 'last_review' ANTES de la imputación con media: 15836
La fecha media (corregida) calculada para 'last_review' es: 2019-06-10 18:47:53
NaT's en 'last_review' DESPUÉS de la imputación con media: 0
Se imputaron 15836 valores NaT en 'last_review' con la fecha media (corregida).
```

```
Nuevo rango FINAL para 'last_review' después de la imputación:
  Fecha mínima: 2012-07-11
  Fecha máxima: 2024-08-15

Estadísticas descriptivas FINALES para 'last_review':
|       | last_review                 |
|:------|:----------------------------|
| count | 102058                      |
| mean  | 2019-06-10 18:47:53.419776768 |
| min   | 2012-07-11 00:00:00         |
| 25%   | 2019-01-02 00:00:00         |
| 50%   | 2019-06-10 18:47:53.419777024 |
| 75%   | 2019-07-01 00:00:00         |
| max   | 2024-08-15 00:00:00         |
```

[1015]:
```python
if 'house_rules' in df_cleaning.columns:
    df_cleaning.drop(columns=['house_rules'], inplace=True)
    print("Columna 'house_rules' eliminada.")
else:
    print("La columna 'house_rules' ya no existe en df_cleaning.")
```

```
Columna 'house_rules' eliminada.
```

[1016]:
```python
# Celda 12.8: Imputación de Nulos en 'reviews_per_month' con Estrategia␣
 ↪Personalizada
logging.info("Celda 12.8: Iniciando imputación de nulos para␣
 ↪'reviews_per_month' con estrategia personalizada.")
print("\n--- Celda 12.8: Imputación de Nulos en 'reviews_per_month' ---")

if not df_cleaning.empty:
    col_rpm = 'reviews_per_month' # Nombre ya normalizado

    if col_rpm not in df_cleaning.columns:
        logging.warning(f"Columna '{col_rpm}' no encontrada. Omitiendo␣
 ↪imputación.")
        print(f"Advertencia: Columna '{col_rpm}' no encontrada. Se omite su␣
 ↪imputación.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_rpm]): # Ya debería␣
 ↪ser float64
        logging.warning(f"Columna '{col_rpm}' no es de tipo numérico (Tipo:␣
 ↪{df_cleaning[col_rpm].dtype}). No se puede imputar.")
        print(f"Advertencia: Columna '{col_rpm}' no es de tipo numérico. No se␣
 ↪puede imputar.")
    else:
        nulos_antes_rpm = df_cleaning[col_rpm].isnull().sum()
```

```
        logging.info(f"Nulos en '{col_rpm}' ANTES de la imputación:␣
↪{nulos_antes_rpm}")
        print(f"Nulos en '{col_rpm}' ANTES de la imputación: {nulos_antes_rpm}")

        if nulos_antes_rpm == 0:
            logging.info(f"No hay nulos que imputar en '{col_rpm}'.")
            print(f"No hay nulos que imputar en '{col_rpm}'.")
        else:
            # 1. Calcular estadísticas de los datos existentes no nulos
            existing_values_rpm = df_cleaning[col_rpm].dropna()
            if existing_values_rpm.empty:
                logging.warning(f"No hay valores no nulos en '{col_rpm}' para␣
↪calcular estadísticas. No se puede imputar.")
                print(f"Advertencia: No hay valores no nulos en '{col_rpm}'␣
↪para calcular estadísticas. No se puede imputar.")
            else:
                mean_rpm = existing_values_rpm.mean()
                std_rpm = existing_values_rpm.std()
                min_observed_rpm = existing_values_rpm.min() # Mínimo observado␣
↪no nulo
                max_observed_rpm = existing_values_rpm.max() # Máximo observado␣
↪no nulo

                logging.info(f"Estadísticas de '{col_rpm}' (existentes):␣
↪Media={mean_rpm:.2f}, StdDev={std_rpm:.2f}, MinObs={min_observed_rpm:.2f},␣
↪MaxObs={max_observed_rpm:.2f}")
                print(f"Estadísticas de '{col_rpm}' (existentes):␣
↪Media={mean_rpm:.2f}, StdDev={std_rpm:.2f}, MinObs={min_observed_rpm:.2f},␣
↪MaxObs={max_observed_rpm:.2f}")

                # 2. Definir parámetros para la imputación
                # Centro de la distribución para la imputación
                imputation_mean = mean_rpm
                # Dispersión: reducir la desviación estándar (si varianza/2 ->␣
↪std/sqrt(2))
                # Si la desviación estándar es 0 (todos los valores existentes␣
↪son iguales), usar un valor pequeño para evitar error en np.random.normal
                imputation_std = (std_rpm / np.sqrt(2)) if std_rpm > 0 else 0.1

                # Límites para los valores imputados
                imputation_lower_limit = 1.0 # Como solicitaste, mínimo 1
                # Límite superior: Podríamos usar el máximo observado o algo␣
↪basado en la media y std
                # Ser conservador y usar el máximo observado para no introducir␣
↪valores demasiado extremos.
                imputation_upper_limit = max_observed_rpm
```

```
                # Asegurar que el límite superior no sea menor que el inferior␣
↪(caso extremo)
                if imputation_upper_limit < imputation_lower_limit:
                    imputation_upper_limit = imputation_lower_limit +␣
↪imputation_std # Un pequeño margen si max_observed es muy bajo

                logging.info(f"Parámetros de imputación:␣
↪MediaImp={imputation_mean:.2f}, StdImp={imputation_std:.2f},␣
↪LimInf={imputation_lower_limit:.2f}, LimSup={imputation_upper_limit:.2f}")
                print(f"Parámetros de imputación: MediaImp={imputation_mean:.
↪2f}, StdImp={imputation_std:.2f}, LimInf={imputation_lower_limit:.2f},␣
↪LimSup={imputation_upper_limit:.2f}")

                # 3. Generar valores aleatorios para los nulos
                num_nulos_to_impute = nulos_antes_rpm
                np.random.seed(42) # Establece la semilla para reproducibilidad

                random_imputed_values = np.random.normal(loc=imputation_mean,␣
↪scale=imputation_std, size=num_nulos_to_impute)

                # 4. Aplicar límites (clipping) a los valores generados
                clipped_imputed_values = np.clip(random_imputed_values,␣
↪imputation_lower_limit, imputation_upper_limit)

                # Imputar los nulos
                # Es importante asignar los valores generados solo a las filas␣
↪que tienen NaN
                nan_indices_rpm = df_cleaning[df_cleaning[col_rpm].isnull()].
↪index
                if len(nan_indices_rpm) == len(clipped_imputed_values): #␣
↪Asegurar que las longitudes coinciden
                    df_cleaning.loc[nan_indices_rpm, col_rpm] =␣
↪clipped_imputed_values
                    imputed_count_rpm = num_nulos_to_impute # Asumimos que␣
↪todos se imputaron
                else:
                    # Esto no debería ocurrir si num_nulos_to_impute se calculó␣
↪correctamente
                    logging.error("Error: Discrepancia en el número de nulos y␣
↪valores generados para imputar.")
                    print("Error: Discrepancia en el número de nulos y valores␣
↪generados.")
                    imputed_count_rpm = 0


                nulos_despues_rpm = df_cleaning[col_rpm].isnull().sum()
```

```
                logging.info(f"Nulos en '{col_rpm}' DESPUÉS de la imputación:␣
 ↪{nulos_despues_rpm}")
                print(f"Nulos en '{col_rpm}' DESPUÉS de la imputación:␣
 ↪{nulos_despues_rpm}")
                if imputed_count_rpm > 0: # O nulos_antes_rpm >␣
 ↪nulos_despues_rpm
                    print(f"Se imputaron {nulos_antes_rpm - nulos_despues_rpm}␣
 ↪valores en '{col_rpm}'.")

                # Verificar estadísticas después de la imputación
                print(f"\nEstadísticas descriptivas actualizadas para␣
 ↪'{col_rpm}':")
                print(df_cleaning[col_rpm].describe().to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
 ↪la imputación de '{col_rpm}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,776 - INFO - Celda 12.8: Iniciando imputación de nulos para
'reviews_per_month' con estrategia personalizada.
2025-05-17 02:01:18,778 - INFO - Nulos en 'reviews_per_month' ANTES de la
imputación: 15818
2025-05-17 02:01:18,780 - INFO - Estadísticas de 'reviews_per_month'
(existentes): Media=1.38, StdDev=1.75, MinObs=0.01, MaxObs=90.00
2025-05-17 02:01:18,781 - INFO - Parámetros de imputación: MediaImp=1.38,
StdImp=1.24, LimInf=1.00, LimSup=90.00
2025-05-17 02:01:18,785 - INFO - Nulos en 'reviews_per_month' DESPUÉS de la
imputación: 0


--- Celda 12.8: Imputación de Nulos en 'reviews_per_month' ---
Nulos en 'reviews_per_month' ANTES de la imputación: 15818
Estadísticas de 'reviews_per_month' (existentes): Media=1.38, StdDev=1.75,
MinObs=0.01, MaxObs=90.00
Parámetros de imputación: MediaImp=1.38, StdImp=1.24, LimInf=1.00, LimSup=90.00
Nulos en 'reviews_per_month' DESPUÉS de la imputación: 0
Se imputaron 15818 valores en 'reviews_per_month'.

Estadísticas descriptivas actualizadas para 'reviews_per_month':
|       |   reviews_per_month |
|:------|--------------------:|
| count |              102058 |
| mean  |             1.42627 |
| std   |             1.64542 |
| min   |             0.01    |
| 25%   |             0.28    |
| 50%   |             1       |
```

```
| 75%   |             2.06    |
| max   |            90       |
```

[1017]:
```
# Celda 12.9: Imputación de Nulos en 'availability_365' con la Mediana
logging.info("Celda 12.9: Iniciando imputación de nulos para 'availability_365' con la mediana.")
print("\n--- Celda 12.9: Imputación de Nulos en 'availability_365' con Mediana ---")

if not df_cleaning.empty:
    col_avail = 'availability_365' # Nombre ya normalizado

    if col_avail not in df_cleaning.columns:
        logging.warning(f"Columna '{col_avail}' no encontrada. Omitiendo imputación.")
        print(f"Advertencia: Columna '{col_avail}' no encontrada. Se omite su imputación.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_avail]): # Ya debería ser Int64
        logging.warning(f"Columna '{col_avail}' no es de tipo numérico (Tipo: {df_cleaning[col_avail].dtype}). No se puede imputar con la mediana.")
        print(f"Advertencia: Columna '{col_avail}' no es de tipo numérico. No se puede imputar con la mediana.")
    else:
        nulos_antes_avail = df_cleaning[col_avail].isnull().sum()
        logging.info(f"Nulos en '{col_avail}' ANTES de la imputación con mediana: {nulos_antes_avail}")
        print(f"Nulos en '{col_avail}' ANTES de la imputación con mediana: {nulos_antes_avail}")

        if nulos_antes_avail == 0:
            logging.info(f"No hay nulos que imputar en '{col_avail}'.")
            print(f"No hay nulos que imputar en '{col_avail}'.")
        else:
            # Calcular la mediana de los valores existentes (no nulos)
            # La columna ya fue clipeada entre 0 y 365.
            median_avail = df_cleaning[col_avail].dropna().median()

            if pd.notna(median_avail):
                # La mediana de una columna Int64 puede ser float si hay un número par de elementos.
                # Como la columna es Int64, redondeamos la mediana y la convertimos a entero
                # para mantener la consistencia del tipo de dato.
                median_avail_int = int(round(median_avail))
```

```
                logging.info(f"La mediana calculada para '{col_avail}' es:␣
↪{median_avail} (se usará como entero: {median_avail_int})")
                print(f"La mediana calculada para '{col_avail}' es:␣
↪{median_avail:.2f} (se usará como entero: {median_avail_int})")

                # Imputar los valores nulos (pd.NA) con la mediana entera
                df_cleaning[col_avail].fillna(median_avail_int, inplace=True)
                imputed_count_avail = nulos_antes_avail -␣
↪df_cleaning[col_avail].isnull().sum()

                nulos_despues_avail = df_cleaning[col_avail].isnull().sum()
                logging.info(f"Nulos en '{col_avail}' DESPUÉS de la imputación:␣
↪{nulos_despues_avail}")
                print(f"Nulos en '{col_avail}' DESPUÉS de la imputación:␣
↪{nulos_despues_avail}")
                if imputed_count_avail > 0:
                    print(f"Se imputaron {imputed_count_avail} valores nulos en␣
↪'{col_avail}' con la mediana ({median_avail_int}).")
            else:
                logging.warning(f"No se pudo calcular la mediana para␣
↪'{col_avail}' (quizás todos los valores son nulos). No se imputaron valores.
↪")
                print(f"Advertencia: No se pudo calcular la mediana para␣
↪'{col_avail}'. No se imputaron valores.")

            # Verificar estadísticas después de la imputación
            print(f"\nEstadísticas descriptivas actualizadas para '{col_avail}':
↪")
            print(df_cleaning[col_avail].describe().to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
↪la imputación de '{col_avail}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,798 - INFO - Celda 12.9: Iniciando imputación de nulos para
'availability_365' con la mediana.
2025-05-17 02:01:18,800 - INFO - Nulos en 'availability_365' ANTES de la
imputación con mediana: 448
2025-05-17 02:01:18,802 - INFO - La mediana calculada para 'availability_365'
es: 96.0 (se usará como entero: 96)
2025-05-17 02:01:18,804 - INFO - Nulos en 'availability_365' DESPUÉS de la
imputación: 0


--- Celda 12.9: Imputación de Nulos en 'availability_365' con Mediana ---
Nulos en 'availability_365' ANTES de la imputación con mediana: 448
La mediana calculada para 'availability_365' es: 96.00 (se usará como entero:
```

```
96)
Nulos en 'availability_365' DESPUÉS de la imputación: 0
Se imputaron 448 valores nulos en 'availability_365' con la mediana (96).

Estadísticas descriptivas actualizadas para 'availability_365':
|       |   availability_365 |
|:------|-------------------:|
| count |             102058 |
| mean  |            140.244 |
| std   |            132.931 |
| min   |              1     |
| 25%   |              3     |
| 50%   |             96     |
| 75%   |            268     |
| max   |            365     |
```

[1018]:
```python
# Celda 12.10: Imputación Probabilística de Nulos en 'minimum_nights'
logging.info("Celda 12.10: Iniciando imputación probabilística de nulos para␣
 ↪'minimum_nights'.")
print("\n--- Celda 12.10: Imputación Probabilística de Nulos en␣
 ↪'minimum_nights' ---")

if not df_cleaning.empty:
    col_mn = 'minimum_nights' # Nombre ya normalizado

    if col_mn not in df_cleaning.columns:
        logging.warning(f"Columna '{col_mn}' no encontrada. Omitiendo␣
 ↪imputación.")
        print(f"Advertencia: Columna '{col_mn}' no encontrada. Se omite su␣
 ↪imputación.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_mn]): # Ya debería␣
 ↪ser Int64
        logging.warning(f"Columna '{col_mn}' no es de tipo numérico (Tipo:␣
 ↪{df_cleaning[col_mn].dtype}). No se puede imputar.")
        print(f"Advertencia: Columna '{col_mn}' no es de tipo numérico. No se␣
 ↪puede imputar.")
    else:
        nulos_antes_mn = df_cleaning[col_mn].isnull().sum()
        logging.info(f"Nulos en '{col_mn}' ANTES de la imputación:␣
 ↪{nulos_antes_mn}")
        print(f"Nulos en '{col_mn}' ANTES de la imputación: {nulos_antes_mn}")

        if nulos_antes_mn == 0:
            logging.info(f"No hay nulos que imputar en '{col_mn}'.")
            print(f"No hay nulos que imputar en '{col_mn}'.")
        else:
            # 1. Definir los posibles valores y sus pesos/probabilidades
```

```
            valores_imputacion_mn = np.arange(1, 9) # [1, 2, 3, 4, 5, 6, 7, 8]

            # Pesos para dar mayor probabilidad al 1 y distribuir el resto
            # Ejemplo: P(1) alta, el resto distribuidas.
            # Suma de pesos = 10 + 7*2 = 24
            # P(1) = 10/24 ~= 0.416
            # P(2..8) = 2/24 ~= 0.083 cada uno
            # Puedes ajustar estos pesos como desees.
            pesos_mn = [7, 3, 3, 3, 2, 2, 2, 2] # Pesos para [1, 2, 3, 4, 5, 6,␣
↪7, 8]

            # Normalizar los pesos para que sumen 1 (np.random.choice lo hace␣
↪internamente si se le pasan pesos)
            probabilidades_mn = np.array(pesos_mn) / np.sum(pesos_mn)

            logging.info(f"Valores para imputación en '{col_mn}':␣
↪{valores_imputacion_mn.tolist()}")
            logging.info(f"Probabilidades asociadas: {probabilidades_mn.
↪tolist()}")
            print(f"Valores para imputación en '{col_mn}':␣
↪{valores_imputacion_mn.tolist()}")
            print(f"Probabilidades asociadas: {[float(f'{p:.3f}') for p in␣
↪probabilidades_mn]}")


            # 2. Generar valores aleatorios para los nulos
            num_nulos_to_impute_mn = nulos_antes_mn
            random_imputed_values_mn = np.random.choice(
                valores_imputacion_mn,
                size=num_nulos_to_impute_mn,
                p=probabilidades_mn
            )

            # 3. Imputar los nulos
            nan_indices_mn = df_cleaning[df_cleaning[col_mn].isnull()].index
            if len(nan_indices_mn) == len(random_imputed_values_mn):
                df_cleaning.loc[nan_indices_mn, col_mn] =␣
↪random_imputed_values_mn
                imputed_count_mn = num_nulos_to_impute_mn
            else:
                logging.error("Error: Discrepancia en el número de nulos y␣
↪valores generados para imputar en minimum_nights.")
                print("Error: Discrepancia en el número de nulos y valores␣
↪generados en minimum_nights.")
                imputed_count_mn = 0
```

```
            nulos_despues_mn = df_cleaning[col_mn].isnull().sum()
            logging.info(f"Nulos en '{col_mn}' DESPUÉS de la imputación:␣
↪{nulos_despues_mn}")
            print(f"Nulos en '{col_mn}' DESPUÉS de la imputación:␣
↪{nulos_despues_mn}")
            if imputed_count_mn > 0:
                print(f"Se imputaron {nulos_antes_mn - nulos_despues_mn}␣
↪valores en '{col_mn}'.")

            # Verificar estadísticas y distribución después de la imputación
            print(f"\nEstadísticas descriptivas actualizadas para '{col_mn}':")
            print(df_cleaning[col_mn].describe().to_markdown())
            print(f"\nDistribución de valores imputados en '{col_mn}' (para los␣
↪que eran nulos):")
            # Para ver la distribución de los valores recién imputados:
            if imputed_count_mn > 0 :
                # Esto puede ser un poco más complejo de aislar si los índices␣
↪se mezclan.
                # Una forma es ver la distribución general ahora:
                print(df_cleaning[col_mn].value_counts(normalize=True).
↪sort_index().head(10).to_markdown()) # Mostrar los 10 primeros

else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
↪la imputación de '{col_mn}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,818 - INFO - Celda 12.10: Iniciando imputación
probabilística de nulos para 'minimum_nights'.
2025-05-17 02:01:18,820 - INFO - Nulos en 'minimum_nights' ANTES de la
imputación: 400
2025-05-17 02:01:18,821 - INFO - Valores para imputación en 'minimum_nights':
[1, 2, 3, 4, 5, 6, 7, 8]
2025-05-17 02:01:18,822 - INFO - Probabilidades asociadas: [0.2916666666666667,
0.125, 0.125, 0.125, 0.08333333333333333, 0.08333333333333333,
0.08333333333333333, 0.08333333333333333]
2025-05-17 02:01:18,826 - INFO - Nulos en 'minimum_nights' DESPUÉS de la
imputación: 0


--- Celda 12.10: Imputación Probabilística de Nulos en 'minimum_nights' ---
Nulos en 'minimum_nights' ANTES de la imputación: 400
Valores para imputación en 'minimum_nights': [1, 2, 3, 4, 5, 6, 7, 8]
Probabilidades asociadas: [0.292, 0.125, 0.125, 0.125, 0.083, 0.083, 0.083,
0.083]
Nulos en 'minimum_nights' DESPUÉS de la imputación: 0
Se imputaron 400 valores en 'minimum_nights'.
```

```
Estadísticas descriptivas actualizadas para 'minimum_nights':
```

	minimum_nights
count	102058
mean	7.95094
std	18.2695
min	1
25%	2
50%	3
75%	5
max	365

```
Distribución de valores imputados en 'minimum_nights' (para los que eran nulos):
```

minimum_nights	proportion
1	0.248996
2	0.230732
3	0.157655
4	0.065257
5	0.0593486
6	0.0153834
7	0.0396441
8	0.00272394
9	0.00153834
10	0.00909287

[1019]:
```
# Celda 12.11: Imputación de Nulos en 'calculated_host_listings_count' con la Media
logging.info("Celda 12.11: Iniciando imputación de nulos para 'calculated_host_listings_count' con la media.")
print("\n--- Celda 12.11: Imputación de Nulos en 'calculated_host_listings_count' con Media ---")

if not df_cleaning.empty:
    col_chlc = 'calculated_host_listings_count' # Nombre ya normalizado

    if col_chlc not in df_cleaning.columns:
        logging.warning(f"Columna '{col_chlc}' no encontrada. Omitiendo imputación.")
        print(f"Advertencia: Columna '{col_chlc}' no encontrada. Se omite su imputación.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_chlc]): # Ya debería ser Int64
        logging.warning(f"Columna '{col_chlc}' no es de tipo numérico (Tipo: {df_cleaning[col_chlc].dtype}). No se puede imputar con la media.")
```

```
        print(f"Advertencia: Columna '{col_chlc}' no es de tipo numérico. No se␣
↪puede imputar con la media.")
    else:
        nulos_antes_chlc = df_cleaning[col_chlc].isnull().sum()
        logging.info(f"Nulos en '{col_chlc}' ANTES de la imputación con media:␣
↪{nulos_antes_chlc}")
        print(f"Nulos en '{col_chlc}' ANTES de la imputación con media:␣
↪{nulos_antes_chlc}")

        if nulos_antes_chlc == 0:
            logging.info(f"No hay nulos que imputar en '{col_chlc}'.")
            print(f"No hay nulos que imputar en '{col_chlc}'.")
        else:
            # Calcular la media de los valores existentes (no nulos)
            mean_chlc = df_cleaning[col_chlc].dropna().mean()

            if pd.notna(mean_chlc):
                # Redondear la media y convertir a entero para mantener la␣
↪consistencia del tipo Int64
                mean_chlc_int = int(round(mean_chlc))

                logging.info(f"La media calculada para '{col_chlc}' es:␣
↪{mean_chlc} (se usará como entero: {mean_chlc_int})")
                print(f"La media calculada para '{col_chlc}' es: {mean_chlc:.
↪2f} (se usará como entero: {mean_chlc_int})")

                # Imputar los valores nulos (pd.NA) con la media entera
                df_cleaning[col_chlc].fillna(mean_chlc_int, inplace=True)
                imputed_count_chlc = nulos_antes_chlc - df_cleaning[col_chlc].
↪isnull().sum()

                nulos_despues_chlc = df_cleaning[col_chlc].isnull().sum()
                logging.info(f"Nulos en '{col_chlc}' DESPUÉS de la imputación:␣
↪{nulos_despues_chlc}")
                print(f"Nulos en '{col_chlc}' DESPUÉS de la imputación:␣
↪{nulos_despues_chlc}")
                if imputed_count_chlc > 0:
                    print(f"Se imputaron {imputed_count_chlc} valores nulos en␣
↪'{col_chlc}' con la media ({mean_chlc_int}).")
            else:
                logging.warning(f"No se pudo calcular la media para␣
↪'{col_chlc}' (quizás todos los valores son nulos). No se imputaron valores.")
                print(f"Advertencia: No se pudo calcular la media para␣
↪'{col_chlc}'. No se imputaron valores.")

            # Verificar estadísticas después de la imputación
```

```
            print(f"\nEstadísticas descriptivas actualizadas para '{col_chlc}':
 ↪")
            print(df_cleaning[col_chlc].describe().to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
 ↪la imputación de '{col_chlc}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,841 - INFO - Celda 12.11: Iniciando imputación de nulos para
'calculated_host_listings_count' con la media.
2025-05-17 02:01:18,843 - INFO - Nulos en 'calculated_host_listings_count' ANTES
de la imputación con media: 319
2025-05-17 02:01:18,845 - INFO - La media calculada para
'calculated_host_listings_count' es: 7.936936671286331 (se usará como entero: 8)
2025-05-17 02:01:18,846 - INFO - Nulos en 'calculated_host_listings_count'
DESPUÉS de la imputación: 0


--- Celda 12.11: Imputación de Nulos en 'calculated_host_listings_count' con
Media ---
Nulos en 'calculated_host_listings_count' ANTES de la imputación con media: 319
La media calculada para 'calculated_host_listings_count' es: 7.94 (se usará como
entero: 8)
Nulos en 'calculated_host_listings_count' DESPUÉS de la imputación: 0
Se imputaron 319 valores nulos en 'calculated_host_listings_count' con la media
(8).

Estadísticas descriptivas actualizadas para 'calculated_host_listings_count':
|       |   calculated_host_listings_count |
|:------|---------------------------------:|
| count |                           102058 |
| mean  |                          7.93713 |
| std   |                         32.2159  |
| min   |                          1       |
| 25%   |                          1       |
| 50%   |                          1       |
| 75%   |                          2       |
| max   |                        332       |
```

[1020]:
```
# Celda 12.12: Imputación de Nulos en 'review_rate_number' con Moda Condicionada
logging.info("Celda 12.12: Iniciando imputación de nulos para␣
 ↪'review_rate_number' con moda condicionada.")
print("\n--- Celda 12.12: Imputación de Nulos en 'review_rate_number' ---")

if not df_cleaning.empty:
    col_rrn = 'review_rate_number' # Nombre ya normalizado
    col_group = 'neighbourhood_group'
```

```
    col_nh = 'neighbourhood'

    # Verificar existencia de columnas
    if col_rrn not in df_cleaning.columns:
        logging.warning(f"Columna '{col_rrn}' no encontrada. Omitiendo␣
↪imputación.")
        print(f"Advertencia: Columna '{col_rrn}' no encontrada.")
    elif col_group not in df_cleaning.columns:
        logging.warning(f"Columna de referencia '{col_group}' no encontrada.␣
↪Estrategia principal no aplicable.")
        print(f"Advertencia: Columna '{col_group}' no encontrada.")
    elif col_nh not in df_cleaning.columns:
        logging.warning(f"Columna de referencia '{col_nh}' no encontrada.␣
↪Estrategia secundaria no aplicable.")
        print(f"Advertencia: Columna '{col_nh}' no encontrada.")
    # Verificar que review_rate_number es categórica o se puede tratar como tal
    elif not (pd.api.types.is_categorical_dtype(df_cleaning[col_rrn]) or pd.api.
↪types.is_numeric_dtype(df_cleaning[col_rrn])):
        logging.warning(f"Columna '{col_rrn}' no es categórica ni numérica␣
↪(Tipo: {df_cleaning[col_rrn].dtype}). Estrategia de moda no aplicable␣
↪directamente.")
        print(f"Advertencia: Columna '{col_rrn}' no es categórica ni numérica.␣
↪No se puede imputar con moda fácilmente.")
    else:
        nulos_antes_rrn = df_cleaning[col_rrn].isnull().sum()
        logging.info(f"Nulos en '{col_rrn}' ANTES de la imputación:␣
↪{nulos_antes_rrn}")
        print(f"Nulos en '{col_rrn}' ANTES de la imputación: {nulos_antes_rrn}")

        if nulos_antes_rrn == 0:
            logging.info(f"No hay nulos que imputar en '{col_rrn}'.")
            print(f"No hay nulos que imputar en '{col_rrn}'.")
        else:
            imputed_count_rrn = 0
            # Iterar sobre las filas donde 'review_rate_number' es nulo
            for index in df_cleaning[df_cleaning[col_rrn].isnull()].index:
                current_group_val = df_cleaning.loc[index, col_group]
                current_nh_val = df_cleaning.loc[index, col_nh]

                imputed_rrn_value = pd.NA

                # Estrategia 1: Moda por 'neighbourhood_group'
                if pd.notna(current_group_val):
                    moda_by_group = df_cleaning[
                        (df_cleaning[col_group] == current_group_val) &
                        (df_cleaning[col_rrn].notna())
```

```
                    ][col_rrn].mode()
                    if not moda_by_group.empty:
                        imputed_rrn_value = moda_by_group.iloc[0]
                        logging.debug(f"Imputando '{col_rrn}' para índice␣
↪{index} con moda de '{col_group}' ({current_group_val}):␣
↪{imputed_rrn_value}")

                # Estrategia 2: Moda por 'neighbourhood' (si la anterior falló␣
↪o no aplicó)
                if pd.isna(imputed_rrn_value) and pd.notna(current_nh_val):
                    moda_by_nh = df_cleaning[
                        (df_cleaning[col_nh] == current_nh_val) &
                        (df_cleaning[col_rrn].notna())
                    ][col_rrn].mode()
                    if not moda_by_nh.empty:
                        imputed_rrn_value = moda_by_nh.iloc[0]
                        logging.debug(f"Imputando '{col_rrn}' para índice␣
↪{index} con moda de '{col_nh}' ({current_nh_val}): {imputed_rrn_value}")

                # Estrategia 3: Moda Global (Fallback si las anteriores␣
↪fallaron)
                if pd.isna(imputed_rrn_value):
                    moda_global_rrn = df_cleaning[col_rrn].dropna().mode()
                    if not moda_global_rrn.empty:
                        imputed_rrn_value = moda_global_rrn.iloc[0]
                        logging.debug(f"Imputando '{col_rrn}' para índice␣
↪{index} con MODA GLOBAL: {imputed_rrn_value}")
                    else:
                        logging.warning(f"No se pudo calcular la moda global␣
↪para '{col_rrn}'. El valor para el índice {index} permanece nulo.")

                # Aplicar el valor imputado
                if pd.notna(imputed_rrn_value):
                    # Si la columna es categórica, el valor imputado debe ser␣
↪una categoría existente o se añadirá.
                    # Si el valor imputado no es del tipo correcto (ej. int␣
↪cuando la categoría espera strings), convertir.
                    # Como review_rate_number es 'category' de números, la moda␣
↪será uno de esos números.
                    df_cleaning.loc[index, col_rrn] = imputed_rrn_value
                    imputed_count_rrn +=1

            nulos_despues_rrn = df_cleaning[col_rrn].isnull().sum()
            logging.info(f"Nulos en '{col_rrn}' DESPUÉS de la imputación:␣
↪{nulos_despues_rrn}")
```

```
            print(f"Nulos en '{col_rrn}' DESPUÉS de la imputación:␣
 ↪{nulos_despues_rrn}")
            if imputed_count_rrn > 0:
                print(f"Se intentó imputar {imputed_count_rrn} valores en␣
 ↪'{col_rrn}'.")

            # Verificar estadísticas y distribución
            print(f"\nDistribución de valores para '{col_rrn}' después de la␣
 ↪imputación:")
            print(df_cleaning[col_rrn].value_counts(dropna=False,␣
 ↪normalize=True).sort_index().to_markdown())
            # Asegurar que el tipo sigue siendo category
            if not pd.api.types.is_categorical_dtype(df_cleaning[col_rrn]) and␣
 ↪df_cleaning[col_rrn].notna().any():
                logging.info(f"Re-convirtiendo '{col_rrn}' a tipo 'category'␣
 ↪después de la imputación.")
                df_cleaning[col_rrn] = df_cleaning[col_rrn].astype('category')
                print(f"Columna '{col_rrn}' re-convertida a tipo 'category'.␣
 ↪Nuevo tipo: {df_cleaning[col_rrn].dtype}")

else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
 ↪la imputación de '{col_rrn}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:18,862 - INFO - Celda 12.12: Iniciando imputación de nulos para
'review_rate_number' con moda condicionada.
/tmp/ipykernel_1305523/2430475895.py:21: DeprecationWarning:
is_categorical_dtype is deprecated and will be removed in a future version. Use
isinstance(dtype, pd.CategoricalDtype) instead
  elif not (pd.api.types.is_categorical_dtype(df_cleaning[col_rrn]) or
pd.api.types.is_numeric_dtype(df_cleaning[col_rrn])):
2025-05-17 02:01:18,865 - INFO - Nulos en 'review_rate_number' ANTES de la
imputación: 319


--- Celda 12.12: Imputación de Nulos en 'review_rate_number' ---
Nulos en 'review_rate_number' ANTES de la imputación: 319

2025-05-17 02:01:20,728 - INFO - Nulos en 'review_rate_number' DESPUÉS de la
imputación: 0

Nulos en 'review_rate_number' DESPUÉS de la imputación: 0
Se intentó imputar 319 valores en 'review_rate_number'.

Distribución de valores para 'review_rate_number' después de la imputación:
|   review_rate_number |   proportion |
|---------------------:|-------------:|
```

```
|                    1 |    0.0900076 |
|                    2 |    0.225088  |
|                    3 |    0.227175  |
|                    4 |    0.228458  |
|                    5 |    0.229272  |
```

```
/tmp/ipykernel_1305523/2430475895.py:88: DeprecationWarning:
is_categorical_dtype is deprecated and will be removed in a future version. Use
isinstance(dtype, pd.CategoricalDtype) instead
  if not pd.api.types.is_categorical_dtype(df_cleaning[col_rrn]) and
df_cleaning[col_rrn].notna().any():
```

[1021]:
```
# Celda 12.13: Imputación de Nulos en 'service_fee' con la Mediana
logging.info("Celda 12.13: Iniciando imputación de nulos para 'service_fee' con␣
 ↪la mediana.")
print("\n--- Celda 12.13: Imputación de Nulos en 'service_fee' con Mediana ---")

if not df_cleaning.empty:
    col_sf = 'service_fee' # Nombre ya normalizado

    if col_sf not in df_cleaning.columns:
        logging.warning(f"Columna '{col_sf}' no encontrada. Omitiendo␣
 ↪imputación.")
        print(f"Advertencia: Columna '{col_sf}' no encontrada. Se omite su␣
 ↪imputación.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_sf]): # Ya debería␣
 ↪ser float64
        logging.warning(f"Columna '{col_sf}' no es de tipo numérico (Tipo:␣
 ↪{df_cleaning[col_sf].dtype}). No se puede imputar con la mediana.")
        print(f"Advertencia: Columna '{col_sf}' no es de tipo numérico. No se␣
 ↪puede imputar con la mediana.")
    else:
        nulos_antes_sf = df_cleaning[col_sf].isnull().sum()
        logging.info(f"Nulos en '{col_sf}' ANTES de la imputación con mediana:␣
 ↪{nulos_antes_sf}")
        print(f"Nulos en '{col_sf}' ANTES de la imputación con mediana:␣
 ↪{nulos_antes_sf}")

        if nulos_antes_sf == 0:
            logging.info(f"No hay nulos que imputar en '{col_sf}'.")
            print(f"No hay nulos que imputar en '{col_sf}'.")
        else:
            # Calcular la mediana de los valores existentes (no nulos)
            median_sf = df_cleaning[col_sf].dropna().median()

            if pd.notna(median_sf):
```

```
                logging.info(f"La mediana calculada para '{col_sf}' es:␣
↪{median_sf:.2f}")
                print(f"La mediana calculada para '{col_sf}' es: {median_sf:.
↪2f}")

                # Imputar los valores nulos (NaN) con la mediana
                df_cleaning[col_sf].fillna(median_sf, inplace=True)
                imputed_count_sf = nulos_antes_sf - df_cleaning[col_sf].
↪isnull().sum()

                nulos_despues_sf = df_cleaning[col_sf].isnull().sum()
                logging.info(f"Nulos en '{col_sf}' DESPUÉS de la imputación:␣
↪{nulos_despues_sf}")
                print(f"Nulos en '{col_sf}' DESPUÉS de la imputación:␣
↪{nulos_despues_sf}")
                if imputed_count_sf > 0:
                    print(f"Se imputaron {imputed_count_sf} valores nulos en␣
↪'{col_sf}' con la mediana ({median_sf:.2f}).")
            else:
                logging.warning(f"No se pudo calcular la mediana para␣
↪'{col_sf}' (quizás todos los valores son nulos). No se imputaron valores.")
                print(f"Advertencia: No se pudo calcular la mediana para␣
↪'{col_sf}'. No se imputaron valores.")

            # Verificar estadísticas después de la imputación
            print(f"\nEstadísticas descriptivas actualizadas para '{col_sf}':")
            print(df_cleaning[col_sf].describe().to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
↪la imputación de '{col_sf}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:20,738 - INFO - Celda 12.13: Iniciando imputación de nulos para
'service_fee' con la mediana.
2025-05-17 02:01:20,740 - INFO - Nulos en 'service_fee' ANTES de la imputación
con mediana: 273
2025-05-17 02:01:20,744 - INFO - La mediana calculada para 'service_fee' es:
125.00
2025-05-17 02:01:20,746 - INFO - Nulos en 'service_fee' DESPUÉS de la
imputación: 0


--- Celda 12.13: Imputación de Nulos en 'service_fee' con Mediana ---
Nulos en 'service_fee' ANTES de la imputación con mediana: 273
La mediana calculada para 'service_fee' es: 125.00
Nulos en 'service_fee' DESPUÉS de la imputación: 0
Se imputaron 273 valores nulos en 'service_fee' con la mediana (125.00).
```

```
Estadísticas descriptivas actualizadas para 'service_fee':
|       |   service_fee |
|:------|--------------:|
| count |   102058      |
| mean  |      125.039  |
| std   |       66.2371 |
| min   |       10      |
| 25%   |       68      |
| 50%   |      125      |
| 75%   |      182      |
| max   |      240      |
```

[1022]:
```python
# Celda 12.14: Imputación de Nulos en 'name' con "Desconocido"
logging.info("Celda 12.14: Iniciando imputación de nulos para 'name' con␣
 ↪'Desconocido'.")
print("\n--- Celda 12.14: Imputación de Nulos en 'name' ---")

if not df_cleaning.empty:
    col_name_listing = 'name' # Nombre ya normalizado
    valor_imputacion_name = "Desconocido" # O "No especificado", "Unknown", etc.

    if col_name_listing not in df_cleaning.columns:
        logging.warning(f"Columna '{col_name_listing}' no encontrada. Omitiendo␣
 ↪imputación.")
        print(f"Advertencia: Columna '{col_name_listing}' no encontrada. Se␣
 ↪omite su imputación.")
    # No se necesita verificar el tipo estrictamente como numérico, ya que␣
 ↪fillna con string funciona en object/string
    elif not (pd.api.types.is_object_dtype(df_cleaning[col_name_listing]) or pd.
 ↪api.types.is_string_dtype(df_cleaning[col_name_listing]) or pd.api.types.
 ↪is_categorical_dtype(df_cleaning[col_name_listing])):
        logging.warning(f"Columna '{col_name_listing}' no es de tipo object,␣
 ↪string o category (Tipo: {df_cleaning[col_name_listing].dtype}). La␣
 ↪imputación con string podría ser inesperada.")
        print(f"Advertencia: Columna '{col_name_listing}' no es de un tipo de␣
 ↪texto esperado. Revisar antes de imputar con '{valor_imputacion_name}'.")
    else:
        nulos_antes_name = df_cleaning[col_name_listing].isnull().sum()
        logging.info(f"Nulos en '{col_name_listing}' ANTES de la imputación:␣
 ↪{nulos_antes_name}")
        print(f"Nulos en '{col_name_listing}' ANTES de la imputación:␣
 ↪{nulos_antes_name}")

        if nulos_antes_name == 0:
            logging.info(f"No hay nulos que imputar en '{col_name_listing}'.")
            print(f"No hay nulos que imputar en '{col_name_listing}'.")
```

```
        else:
            # Imputar los valores nulos (pd.NA o None) con el string␣
↪especificado
            df_cleaning[col_name_listing].fillna(valor_imputacion_name,␣
↪inplace=True)
            imputed_count_name = nulos_antes_name -␣
↪df_cleaning[col_name_listing].isnull().sum()

            nulos_despues_name = df_cleaning[col_name_listing].isnull().sum()
            logging.info(f"Nulos en '{col_name_listing}' DESPUÉS de la␣
↪imputación: {nulos_despues_name}")
            print(f"Nulos en '{col_name_listing}' DESPUÉS de la imputación:␣
↪{nulos_despues_name}")
            if imputed_count_name > 0:
                print(f"Se imputaron {imputed_count_name} valores nulos en␣
↪'{col_name_listing}' con '{valor_imputacion_name}'.")

            # Verificar la frecuencia del valor imputado
            print(f"\nFrecuencia de valores en '{col_name_listing}' (mostrando␣
↪'Desconocido' si existe):")
            if valor_imputacion_name in df_cleaning[col_name_listing].unique():
                print(df_cleaning[col_name_listing].value_counts().head().
↪to_markdown()) # Mostrar los más comunes
                print(f"  Conteo de '{valor_imputacion_name}':␣
↪{df_cleaning[col_name_listing].value_counts().get(valor_imputacion_name,␣
↪0)}")
            else:
                print(f"El valor '{valor_imputacion_name}' no se encontró␣
↪después de la imputación (esto sería inesperado si hubo nulos).")

            # Asegurar que el tipo de dato sigue siendo apropiado (object o␣
↪string)
            # Si era category, fillna podría añadir la nueva categoría si no␣
↪existía.
            if not (pd.api.types.is_object_dtype(df_cleaning[col_name_listing])␣
↪or pd.api.types.is_string_dtype(df_cleaning[col_name_listing])):
                if pd.api.types.
↪is_categorical_dtype(df_cleaning[col_name_listing]):
                    # Si era category y 'Desconocido' no era una categoría, se␣
↪habrá añadido
                    if valor_imputacion_name not in␣
↪df_cleaning[col_name_listing].cat.categories:
                        # Esto es informativo, fillna en category maneja esto.
                        logging.info(f"'{valor_imputacion_name}' fue añadido␣
↪como nueva categoría a '{col_name_listing}'.")
                else: # Si cambió a otro tipo inesperado
```

```
                    logging.warning(f"El tipo de '{col_name_listing}' cambió a␣
 ↪{df_cleaning[col_name_listing].dtype}. Convirtiendo a string.")
                    df_cleaning[col_name_listing] =␣
 ↪df_cleaning[col_name_listing].astype(str)
            print(f"Tipo de dato final para '{col_name_listing}':␣
 ↪{df_cleaning[col_name_listing].dtype}")

else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
 ↪la imputación de '{col_name_listing}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:20,761 - INFO - Celda 12.14: Iniciando imputación de nulos para
'name' con 'Desconocido'.
2025-05-17 02:01:20,769 - INFO - Nulos en 'name' ANTES de la imputación: 250
2025-05-17 02:01:20,794 - INFO - Nulos en 'name' DESPUÉS de la imputación: 0


--- Celda 12.14: Imputación de Nulos en 'name' ---
Nulos en 'name' ANTES de la imputación: 250
Nulos en 'name' DESPUÉS de la imputación: 0
Se imputaron 250 valores nulos en 'name' con 'Desconocido'.

Frecuencia de valores en 'name' (mostrando 'Desconocido' si existe):
| name                          |   count |
|:------------------------------|--------:|
| Desconocido                   |     250 |
| Home away from home           |      33 |
| Hillside Hotel                |      30 |
| Water View King Bed Hotel Room |      30 |
| Brooklyn Apartment            |      27 |
  Conteo de 'Desconocido': 250
Tipo de dato final para 'name': object
```

[1023]:
```
# Celda 12.15: Imputación de Nulos en 'price' con Media Condicionada por␣
 ↪'neighbourhood'
logging.info("Celda 12.15: Iniciando imputación de nulos para 'price' con media␣
 ↪condicionada por 'neighbourhood'.")
print("\n--- Celda 12.15: Imputación de Nulos en 'price' ---")

if not df_cleaning.empty:
    col_price = 'price' # Nombre ya normalizado
    col_nh_ref = 'neighbourhood' # Columna de referencia, ya normalizada

    if col_price not in df_cleaning.columns:
        logging.warning(f"Columna '{col_price}' no encontrada. Omitiendo␣
 ↪imputación.")
```

```
        print(f"Advertencia: Columna '{col_price}' no encontrada. Se omite su imputación.")
    elif col_nh_ref not in df_cleaning.columns:
        logging.warning(f"Columna de referencia '{col_nh_ref}' no encontrada. No se puede imputar '{col_price}' con esta estrategia principal.")
        print(f"Advertencia: Columna '{col_nh_ref}' no encontrada.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_price]): # Ya debería ser float64
        logging.warning(f"Columna '{col_price}' no es de tipo numérico (Tipo: {df_cleaning[col_price].dtype}). No se puede imputar con la media.")
        print(f"Advertencia: Columna '{col_price}' no es de tipo numérico. No se puede imputar con la media.")
    else:
        nulos_antes_price = df_cleaning[col_price].isnull().sum()
        logging.info(f"Nulos en '{col_price}' ANTES de la imputación: {nulos_antes_price}")
        print(f"Nulos en '{col_price}' ANTES de la imputación: {nulos_antes_price}")

        if nulos_antes_price == 0:
            logging.info(f"No hay nulos que imputar en '{col_price}'.")
            print(f"No hay nulos que imputar en '{col_price}'.")
        else:
            # Calcular la media global como fallback una sola vez
            media_global_price = df_cleaning[col_price].dropna().mean()
            if pd.notna(media_global_price):
                logging.info(f"Media global de '{col_price}' (para fallback): {media_global_price:.2f}")
                print(f"Media global de '{col_price}' (para fallback): {media_global_price:.2f}")
            else:
                logging.warning(f"No se pudo calcular la media global de '{col_price}'. El fallback podría no funcionar.")
                print(f"Advertencia: No se pudo calcular la media global de '{col_price}'.")


            imputed_count_price = 0
            # Iterar sobre las filas donde 'price' es nulo
            for index in df_cleaning[df_cleaning[col_price].isnull()].index:
                current_neighbourhood_val = df_cleaning.loc[index, col_nh_ref]

                imputed_price_value = pd.NA # Valor por defecto

                # Estrategia 1: Media por 'neighbourhood'
                if pd.notna(current_neighbourhood_val):
```

```
                    mean_price_neighbourhood = df_cleaning[
                        (df_cleaning[col_nh_ref] == current_neighbourhood_val) &
                        (df_cleaning[col_price].notna()) # Usar solo precios no␣
↪nulos para la media
                    ][col_price].mean()

                    if pd.notna(mean_price_neighbourhood):
                        imputed_price_value = mean_price_neighbourhood
                        logging.debug(f"Imputando '{col_price}' para índice␣
↪{index} con media de '{col_nh_ref}' ({current_neighbourhood_val}):␣
↪{imputed_price_value:.2f}")
                    else:
                        logging.warning(f"No se pudo calcular la media de␣
↪'{col_price}' para '{col_nh_ref}' '{current_neighbourhood_val}' (índice␣
↪{index}). Usando fallback si es posible.")

                # Estrategia 2: Media Global (Fallback)
                if pd.isna(imputed_price_value) and pd.
↪notna(media_global_price):
                    imputed_price_value = media_global_price
                    logging.debug(f"Imputando '{col_price}' para índice {index}␣
↪con MEDIA GLOBAL: {imputed_price_value:.2f}")
                elif pd.isna(imputed_price_value) and pd.
↪isna(media_global_price):
                     logging.error(f"No se pudo imputar '{col_price}' para el␣
↪índice {index} ni con media de neighbourhood ni con media global (no␣
↪calculable).")


                # Aplicar el valor imputado
                if pd.notna(imputed_price_value):
                    df_cleaning.loc[index, col_price] = imputed_price_value
                    imputed_count_price +=1

            nulos_despues_price = df_cleaning[col_price].isnull().sum()
            logging.info(f"Nulos en '{col_price}' DESPUÉS de la imputación:␣
↪{nulos_despues_price}")
            print(f"Nulos en '{col_price}' DESPUÉS de la imputación:␣
↪{nulos_despues_price}")
            if imputed_count_price > 0: # O nulos_antes_price >␣
↪nulos_despues_price
                print(f"Se imputaron {nulos_antes_price - nulos_despues_price}␣
↪valores en '{col_price}'.")
            elif nulos_antes_price == nulos_despues_price and nulos_antes_price␣
↪> 0:
```

```
                print(f"No se pudieron imputar los nulos restantes en␣
 ↪'{col_price}' con la estrategia actual (posiblemente media global no␣
 ↪calculable y neighbourhoods sin datos).")

            # Verificar estadísticas después de la imputación
            print(f"\nEstadísticas descriptivas actualizadas para '{col_price}':
 ↪")
            print(df_cleaning[col_price].describe().to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
 ↪la imputación de '{col_price}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:20,911 - INFO - Celda 12.15: Iniciando imputación de nulos para
'price' con media condicionada por 'neighbourhood'.
2025-05-17 02:01:20,913 - INFO - Nulos en 'price' ANTES de la imputación: 247
2025-05-17 02:01:20,914 - INFO - Media global de 'price' (para fallback): 625.36


--- Celda 12.15: Imputación de Nulos en 'price' ---
Nulos en 'price' ANTES de la imputación: 247
Media global de 'price' (para fallback): 625.36

2025-05-17 02:01:21,367 - INFO - Nulos en 'price' DESPUÉS de la imputación: 0

Nulos en 'price' DESPUÉS de la imputación: 0
Se imputaron 247 valores en 'price'.

Estadísticas descriptivas actualizadas para 'price':
|       |      price |
|:------|-----------:|
| count | 102058     |
| mean  |    625.357 |
| std   |    331.272 |
| min   |     50     |
| 25%   |    341     |
| 50%   |    625     |
| 75%   |    912     |
| max   |   1200     |
```

[1024]:
```
# Celda 12.16: Imputación de Nulos en 'construction_year' con un valor␣
 ↪placeholder (e.g., 0)
logging.info("Celda 12.16: Iniciando imputación de nulos para␣
 ↪'construction_year' con un valor placeholder.")
print("\n--- Celda 12.16: Imputación de Nulos en 'construction_year' ---")

if not df_cleaning.empty:
    col_cy = 'construction_year' # Nombre ya normalizado
```

```
    valor_placeholder_cy = 0  # Valor numérico para representar "Desconocido"
                              # Podría ser -1, 999, o un año muy antiguo como␣
↪1000 si 0 es problemático.
                              # Si 0 ya existe y tiene significado, elige otro.

    if col_cy not in df_cleaning.columns:
        logging.warning(f"Columna '{col_cy}' no encontrada. Omitiendo␣
↪imputación.")
        print(f"Advertencia: Columna '{col_cy}' no encontrada. Se omite su␣
↪imputación.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_cy]): # Ya debería␣
↪ser Int64
        logging.warning(f"Columna '{col_cy}' no es de tipo numérico (Tipo:␣
↪{df_cleaning[col_cy].dtype}). No se puede imputar con {valor_placeholder_cy}.
↪")
        print(f"Advertencia: Columna '{col_cy}' no es de tipo numérico. No se␣
↪puede imputar.")
    else:
        nulos_antes_cy = df_cleaning[col_cy].isnull().sum()
        logging.info(f"Nulos (pd.NA) en '{col_cy}' ANTES de la imputación:␣
↪{nulos_antes_cy}")
        print(f"Nulos (pd.NA) en '{col_cy}' ANTES de la imputación:␣
↪{nulos_antes_cy}")

        if nulos_antes_cy == 0:
            logging.info(f"No hay nulos que imputar en '{col_cy}'.")
            print(f"No hay nulos que imputar en '{col_cy}'.")
        else:
            # Verificar si el valor placeholder ya existe y cuántas veces
            if valor_placeholder_cy in df_cleaning[col_cy].unique():
                count_placeholder_antes = df_cleaning[df_cleaning[col_cy] ==␣
↪valor_placeholder_cy].shape[0]
                logging.info(f"El valor placeholder '{valor_placeholder_cy}' ya␣
↪existe {count_placeholder_antes} veces en '{col_cy}'.")
                print(f"Advertencia: El valor placeholder␣
↪'{valor_placeholder_cy}' ya existe {count_placeholder_antes} veces en␣
↪'{col_cy}'.")

            # Imputar los valores nulos (pd.NA) con el valor placeholder
            df_cleaning[col_cy].fillna(valor_placeholder_cy, inplace=True)
            imputed_count_cy = nulos_antes_cy - df_cleaning[col_cy].isnull().
↪sum()

            nulos_despues_cy = df_cleaning[col_cy].isnull().sum()
            logging.info(f"Nulos en '{col_cy}' DESPUÉS de la imputación:␣
↪{nulos_despues_cy}")
```

```
            print(f"Nulos en '{col_cy}' DESPUÉS de la imputación:␣
 ↪{nulos_despues_cy}") # Debería ser 0
            if imputed_count_cy > 0:
                print(f"Se imputaron {imputed_count_cy} valores nulos en␣
 ↪'{col_cy}' con '{valor_placeholder_cy}'.")

            # Verificar la frecuencia del valor imputado
            count_placeholder_despues = df_cleaning[df_cleaning[col_cy] ==␣
 ↪valor_placeholder_cy].shape[0]
            print(f"\nFrecuencia del valor placeholder '{valor_placeholder_cy}'␣
 ↪en '{col_cy}' después de la imputación: {count_placeholder_despues}")

            print(f"\nEstadísticas descriptivas actualizadas para '{col_cy}':")
            print(df_cleaning[col_cy].describe().to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
 ↪la imputación de '{col_cy}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:21,381 - INFO - Celda 12.16: Iniciando imputación de nulos para
'construction_year' con un valor placeholder.
2025-05-17 02:01:21,382 - INFO - Nulos (pd.NA) en 'construction_year' ANTES de
la imputación: 214
2025-05-17 02:01:21,384 - INFO - Nulos en 'construction_year' DESPUÉS de la
imputación: 0


--- Celda 12.16: Imputación de Nulos en 'construction_year' ---
Nulos (pd.NA) en 'construction_year' ANTES de la imputación: 214
Nulos en 'construction_year' DESPUÉS de la imputación: 0
Se imputaron 214 valores nulos en 'construction_year' con '0'.

Frecuencia del valor placeholder '0' en 'construction_year' después de la
imputación: 214

Estadísticas descriptivas actualizadas para 'construction_year':
|       |   construction_year |
|:------|--------------------:|
| count |              102058 |
| mean  |             2008.27 |
| std   |               92.2383 |
| min   |                0    |
| 25%   |             2007    |
| 50%   |             2012    |
| 75%   |             2017    |
| max   |             2022    |
```

[1025]:
```
# Celda 12.17: Imputación de Nulos en 'number_of_reviews' con la Media
logging.info("Celda 12.17: Iniciando imputación de nulos para␣
 ↪'number_of_reviews' con la media.")
print("\n--- Celda 12.17: Imputación de Nulos en 'number_of_reviews' con Media␣
 ↪---")

if not df_cleaning.empty:
    col_nor = 'number_of_reviews' # Nombre ya normalizado

    if col_nor not in df_cleaning.columns:
        logging.warning(f"Columna '{col_nor}' no encontrada. Omitiendo␣
 ↪imputación.")
        print(f"Advertencia: Columna '{col_nor}' no encontrada. Se omite su␣
 ↪imputación.")
    elif not pd.api.types.is_numeric_dtype(df_cleaning[col_nor]): # Ya debería␣
 ↪ser Int64
        logging.warning(f"Columna '{col_nor}' no es de tipo numérico (Tipo:␣
 ↪{df_cleaning[col_nor].dtype}). No se puede imputar con la media.")
        print(f"Advertencia: Columna '{col_nor}' no es de tipo numérico. No se␣
 ↪puede imputar con la media.")
    else:
        nulos_antes_nor = df_cleaning[col_nor].isnull().sum()
        logging.info(f"Nulos en '{col_nor}' ANTES de la imputación con media:␣
 ↪{nulos_antes_nor}")
        print(f"Nulos en '{col_nor}' ANTES de la imputación con media:␣
 ↪{nulos_antes_nor}")

        if nulos_antes_nor == 0:
            logging.info(f"No hay nulos que imputar en '{col_nor}'.")
            print(f"No hay nulos que imputar en '{col_nor}'.")
        else:
            # Calcular la media de los valores existentes (no nulos)
            mean_nor = df_cleaning[col_nor].dropna().mean()

            if pd.notna(mean_nor):
                # Redondear la media y convertir a entero para mantener la␣
 ↪consistencia del tipo Int64
                mean_nor_int = int(round(mean_nor))

                logging.info(f"La media calculada para '{col_nor}' es:␣
 ↪{mean_nor} (se usará como entero: {mean_nor_int})")
                print(f"La media calculada para '{col_nor}' es: {mean_nor:.2f}␣
 ↪(se usará como entero: {mean_nor_int})")

                # Imputar los valores nulos (pd.NA) con la media entera
                df_cleaning[col_nor].fillna(mean_nor_int, inplace=True)
```

```
                imputed_count_nor = nulos_antes_nor - df_cleaning[col_nor].
↪isnull().sum()

                nulos_despues_nor = df_cleaning[col_nor].isnull().sum()
                logging.info(f"Nulos en '{col_nor}' DESPUÉS de la imputación:␣
↪{nulos_despues_nor}")
                print(f"Nulos en '{col_nor}' DESPUÉS de la imputación:␣
↪{nulos_despues_nor}")
                if imputed_count_nor > 0:
                    print(f"Se imputaron {imputed_count_nor} valores nulos en␣
↪'{col_nor}' con la media ({mean_nor_int}).")
            else:
                logging.warning(f"No se pudo calcular la media para '{col_nor}'␣
↪(quizás todos los valores son nulos). No se imputaron valores.")
                print(f"Advertencia: No se pudo calcular la media para␣
↪'{col_nor}'. No se imputaron valores.")

            # Verificar estadísticas después de la imputación
            print(f"\nEstadísticas descriptivas actualizadas para '{col_nor}':")
            print(df_cleaning[col_nor].describe().to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
↪la imputación de '{col_nor}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:21,400 - INFO - Celda 12.17: Iniciando imputación de nulos para
'number_of_reviews' con la media.
2025-05-17 02:01:21,402 - INFO - Nulos en 'number_of_reviews' ANTES de la
imputación con media: 183
2025-05-17 02:01:21,404 - INFO - La media calculada para 'number_of_reviews' es:
27.517948466257668 (se usará como entero: 28)
2025-05-17 02:01:21,405 - INFO - Nulos en 'number_of_reviews' DESPUÉS de la
imputación: 0


--- Celda 12.17: Imputación de Nulos en 'number_of_reviews' con Media ---
Nulos en 'number_of_reviews' ANTES de la imputación con media: 183
La media calculada para 'number_of_reviews' es: 27.52 (se usará como entero: 28)
Nulos en 'number_of_reviews' DESPUÉS de la imputación: 0
Se imputaron 183 valores nulos en 'number_of_reviews' con la media (28).

Estadísticas descriptivas actualizadas para 'number_of_reviews':
|       |   number_of_reviews |
|:------|--------------------:|
| count |              102058 |
| mean  |             27.5188 |
| std   |             49.5273 |
```

```
| min   |              0      |
| 25%   |              1      |
| 50%   |              7      |
| 75%   |             30      |
| max   |           1024      |
```

[1026]:
```
# Celda 12.18: Corrección de Errores Tipográficos en 'neighbourhood_group'
logging.info("Celda 12.18: Corrigiendo errores tipográficos en␣
 ↪'neighbourhood_group'.")
print("\n--- Celda 12.18: Corrección de Errores Tipográficos en␣
 ↪'neighbourhood_group' ---")

if not df_cleaning.empty:
    col_ng = 'neighbourhood_group' # Nombre ya normalizado

    if col_ng not in df_cleaning.columns:
        logging.warning(f"Columna '{col_ng}' no encontrada. Omitiendo␣
 ↪corrección de errores tipográficos.")
        print(f"Advertencia: Columna '{col_ng}' no encontrada. Se omite la␣
 ↪corrección.")
    # Asegurarse de que es un tipo donde .replace con dict y .str (si se usa)␣
 ↪tiene sentido.
    # Ya debería ser 'category' o 'object'/'string'.
    elif not (pd.api.types.is_categorical_dtype(df_cleaning[col_ng]) or \
              pd.api.types.is_object_dtype(df_cleaning[col_ng]) or \
              pd.api.types.is_string_dtype(df_cleaning[col_ng])):
        logging.warning(f"Columna '{col_ng}' no es de un tipo adecuado para␣
 ↪reemplazo de strings (Tipo: {df_cleaning[col_ng].dtype}).")
        print(f"Advertencia: Columna '{col_ng}' no es de un tipo adecuado para␣
 ↪corrección de errores tipográficos.")
    else:
        # Primero, veamos los valores únicos ANTES de la corrección para␣
 ↪confirmar
        if pd.api.types.is_categorical_dtype(df_cleaning[col_ng]):
            unique_values_before_ng = df_cleaning[col_ng].cat.categories.
 ↪tolist()
            if df_cleaning[col_ng].isnull().any(): # Si hay NaNs en la categoría
                unique_values_before_ng.append(pd.NA) # o np.nan si prefieres␣
 ↪consistencia en la lista
        else: # object o string
            unique_values_before_ng = df_cleaning[col_ng].unique().tolist()

        logging.info(f"Valores únicos en '{col_ng}' ANTES de la corrección:␣
 ↪{unique_values_before_ng}")
        print(f"Valores únicos en '{col_ng}' ANTES de la corrección:␣
 ↪{unique_values_before_ng}")
```

```
        # Definir el mapeo de correcciones
        # Estos son ejemplos basados en el PDF. Debes verificar tus propios␣
↪datos para identificar errores.
        corrections_map_ng = {
            'brookln': 'Brooklyn',
            'manhatan': 'Manhattan',
            # Añade más correcciones aquí si las identificas, por ejemplo:
            # 'Manhatann': 'Manhattan',
            # 'queenz': 'Queens'
        }

        # Aplicar las correcciones
        # Si la columna es de tipo 'category', reemplazar las categorías y␣
↪luego los valores.
        # Si es object/string, reemplazar directamente.

        original_dtype_ng = df_cleaning[col_ng].dtype
        num_changes_made = 0

        # Haremos el replace directamente. Si es categoría, los valores no␣
↪mapeados se mantienen.
        # Si una categoría es reemplazada por otra existente, se fusionan.
        # Si una categoría es reemplazada por una nueva, la nueva categoría se␣
↪añade (si la columna es category).

        for incorrect_val, correct_val in corrections_map_ng.items():
            # Contar cuántas filas se verán afectadas por este reemplazo␣
↪específico
            # Necesitamos manejar el caso si la columna es string/object o␣
↪category
            if pd.api.types.is_categorical_dtype(original_dtype_ng):
                if incorrect_val in df_cleaning[col_ng].cat.categories:
                    count_before_replace = (df_cleaning[col_ng] ==␣
↪incorrect_val).sum()
                else:
                    count_before_replace = 0 # El valor incorrecto no es una␣
↪categoría, no se reemplazará
            else: # object/string
                count_before_replace = (df_cleaning[col_ng] == incorrect_val).
↪sum()

            if count_before_replace > 0:
                df_cleaning[col_ng] = df_cleaning[col_ng].
↪replace({incorrect_val: correct_val})
```

```
                num_changes_made += count_before_replace # Asumimos que todos␣
↪los reemplazos ocurrieron
                logging.info(f"Reemplazado '{incorrect_val}' por␣
↪'{correct_val}' en '{col_ng}' ({count_before_replace} ocurrencias).")
                print(f"Reemplazado '{incorrect_val}' por '{correct_val}' en␣
↪'{col_ng}' ({count_before_replace} ocurrencias).")

        if num_changes_made > 0:
            logging.info("Correcciones tipográficas aplicadas.")
            # Si la columna era categórica, es bueno recategorizar para␣
↪eliminar las categorías antiguas si ya no se usan
            # y asegurar que el tipo de dato es el deseado.
            if pd.api.types.is_categorical_dtype(original_dtype_ng):
                # Obtener las categorías que realmente existen en los datos␣
↪después del replace
                current_values_in_col = df_cleaning[col_ng].dropna().unique()
                df_cleaning[col_ng] = pd.Categorical(df_cleaning[col_ng],␣
↪categories=sorted(current_values_in_col), ordered=False)
                logging.info(f"'{col_ng}' re-categorizada para optimizar␣
↪categorías.")
                print(f"'{col_ng}' re-categorizada.")
            elif pd.api.types.is_object_dtype(original_dtype_ng) or pd.api.
↪types.is_string_dtype(original_dtype_ng):
                # Si era object/string y queremos convertirla a category por␣
↪primera vez o de nuevo
                if df_cleaning[col_ng].nunique(dropna=False) <= 20: # Umbral de␣
↪ejemplo
                    df_cleaning[col_ng] = df_cleaning[col_ng].astype('category')
                    logging.info(f"'{col_ng}' convertida a category después de␣
↪las correcciones.")
                    print(f"'{col_ng}' convertida a category después de las␣
↪correcciones.")
        else:
            logging.info(f"No se encontraron valores que coincidan con el mapa␣
↪de correcciones en '{col_ng}'.")
            print(f"No se aplicaron correcciones tipográficas en '{col_ng}' (no␣
↪se encontraron los valores incorrectos especificados).")

        # Mostrar valores únicos DESPUÉS de la corrección
        if pd.api.types.is_categorical_dtype(df_cleaning[col_ng]):
            unique_values_after_ng = df_cleaning[col_ng].cat.categories.tolist()
            if df_cleaning[col_ng].isnull().any():
                 unique_values_after_ng.append(pd.NA)
        else:
            unique_values_after_ng = df_cleaning[col_ng].unique().tolist()
```

```
        logging.info(f"Valores únicos en '{col_ng}' DESPUÉS de la corrección:␣
↪{unique_values_after_ng}")
        print(f"\nValores únicos en '{col_ng}' DESPUÉS de la corrección:␣
↪{unique_values_after_ng}")
        print(f"Tipo de dato final para '{col_ng}': {df_cleaning[col_ng].
↪dtype}")
        print(f"\nDistribución de valores para '{col_ng}' después de la␣
↪corrección:")
        print(df_cleaning[col_ng].value_counts(dropna=False).to_markdown())
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se puede realizar␣
↪la corrección en '{col_ng}'.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:21,423 - INFO - Celda 12.18: Corrigiendo errores tipográficos
en 'neighbourhood_group'.
/tmp/ipykernel_1305523/322709299.py:13: DeprecationWarning: is_categorical_dtype
is deprecated and will be removed in a future version. Use isinstance(dtype,
pd.CategoricalDtype) instead
  elif not (pd.api.types.is_categorical_dtype(df_cleaning[col_ng]) or \
/tmp/ipykernel_1305523/322709299.py:20: DeprecationWarning: is_categorical_dtype
is deprecated and will be removed in a future version. Use isinstance(dtype,
pd.CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(df_cleaning[col_ng]):
2025-05-17 02:01:21,427 - INFO - Valores únicos en 'neighbourhood_group' ANTES
de la corrección: ['Bronx', 'Brooklyn', 'Manhattan', 'None', 'Queens', 'Staten
Island', 'brookln', 'manhatan']
/tmp/ipykernel_1305523/322709299.py:54: DeprecationWarning: is_categorical_dtype
is deprecated and will be removed in a future version. Use isinstance(dtype,
pd.CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(original_dtype_ng):
2025-05-17 02:01:21,429 - INFO - Reemplazado 'brookln' por 'Brooklyn' en
'neighbourhood_group' (1 ocurrencias).
/tmp/ipykernel_1305523/322709299.py:54: DeprecationWarning: is_categorical_dtype
is deprecated and will be removed in a future version. Use isinstance(dtype,
pd.CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(original_dtype_ng):
2025-05-17 02:01:21,432 - INFO - Reemplazado 'manhatan' por 'Manhattan' en
'neighbourhood_group' (1 ocurrencias).
2025-05-17 02:01:21,433 - INFO - Correcciones tipográficas aplicadas.
/tmp/ipykernel_1305523/322709299.py:72: DeprecationWarning: is_categorical_dtype
is deprecated and will be removed in a future version. Use isinstance(dtype,
pd.CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(original_dtype_ng):
2025-05-17 02:01:21,435 - INFO - 'neighbourhood_group' re-categorizada para
optimizar categorías.
/tmp/ipykernel_1305523/322709299.py:89: DeprecationWarning: is_categorical_dtype
```

```
is deprecated and will be removed in a future version. Use isinstance(dtype,
pd.CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(df_cleaning[col_ng]):
2025-05-17 02:01:21,435 - INFO - Valores únicos en 'neighbourhood_group' DESPUÉS
de la corrección: ['Bronx', 'Brooklyn', 'Manhattan', 'None', 'Queens', 'Staten
Island']


--- Celda 12.18: Corrección de Errores Tipográficos en 'neighbourhood_group' ---
Valores únicos en 'neighbourhood_group' ANTES de la corrección: ['Bronx',
'Brooklyn', 'Manhattan', 'None', 'Queens', 'Staten Island', 'brookln',
'manhatan']
Reemplazado 'brookln' por 'Brooklyn' en 'neighbourhood_group' (1 ocurrencias).
Reemplazado 'manhatan' por 'Manhattan' en 'neighbourhood_group' (1 ocurrencias).
'neighbourhood_group' re-categorizada.

Valores únicos en 'neighbourhood_group' DESPUÉS de la corrección: ['Bronx',
'Brooklyn', 'Manhattan', 'None', 'Queens', 'Staten Island']
Tipo de dato final para 'neighbourhood_group': category

Distribución de valores para 'neighbourhood_group' después de la corrección:
| neighbourhood_group   |   count |
|:----------------------|--------:|
| Manhattan             |   43558 |
| Brooklyn              |   41631 |
| Queens                |   13197 |
| Bronx                 |    2694 |
| Staten Island         |     949 |
| None                  |      29 |
```

[1027]:
```python
# Celda 12.3: Verificación del Estado Actual de Nulos
logging.info("Celda 12.3: Verificando el estado actual de valores nulos por␣
 ↪columna en df_cleaning.")

if not df_cleaning.empty:
    nulos_actual_counts = df_cleaning.isnull().sum()
    nulos_actual_percentage = (nulos_actual_counts / len(df_cleaning)) * 100

    df_nulos_actual = pd.DataFrame({
        'Columna': df_cleaning.columns,
        'Nulos': nulos_actual_counts,
        'Porcentaje_Nulos': nulos_actual_percentage,
        'Tipo_Dato': df_cleaning.dtypes # Añadir tipo de dato para contexto
    }).reset_index(drop=True) # Asegurar que el índice sea simple

    # Mostrar solo columnas que AÚN tienen nulos, ordenadas de mayor a menor␣
 ↪porcentaje
```

```
    df_nulos_actual_sorted = df_nulos_actual[df_nulos_actual['Nulos'] > 0].
↪sort_values(
        by='Porcentaje_Nulos', ascending=False
    )

    if not df_nulos_actual_sorted.empty:
        print("Cantidad y porcentaje de valores nulos ACTUALES por columna␣
↪(solo columnas con nulos):")
        print(df_nulos_actual_sorted.to_markdown(index=False))
        logging.info("Tabla de estado actual de nulos por columna generada y␣
↪mostrada.")
    else:
        print("¡Excelente! No se encontraron valores nulos en el DataFrame␣
↪df_cleaning en este momento.")
        logging.info("No se encontraron valores nulos en df_cleaning␣
↪actualmente.")
else:
    logging.warning("El DataFrame df_cleaning está vacío. No se pueden calcular␣
↪los nulos.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:21,446 - INFO - Celda 12.3: Verificando el estado actual de
valores nulos por columna en df_cleaning.
2025-05-17 02:01:21,460 - INFO - No se encontraron valores nulos en df_cleaning
actualmente.
```

```
¡Excelente! No se encontraron valores nulos en el DataFrame df_cleaning en este
momento.
```

[1028]:
```
if 'country_code' in df_cleaning.columns:
    df_cleaning.drop(columns=['country_code'], inplace=True)
    print("Columna 'country_code' eliminada de df_cleaning.")
else:
    print("La columna 'country_code' ya no existe en df_cleaning.")
```

```
Columna 'country_code' eliminada de df_cleaning.
```

[1029]:
```
df_cleaning.info() # Mostrar información del DataFrame después de las␣
↪imputaciones
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 102058 entries, 0 to 102057
Data columns (total 24 columns):
 #   Column                          Non-Null Count   Dtype
---  ------                          --------------   -----
 0   id                              102058 non-null  int64
 1   name                            102058 non-null  object
 2   host_id                         102058 non-null  int64
```

```
 3   neighbourhood_group               102058 non-null  category
 4   neighbourhood                     102058 non-null  category
 5   lat                               102058 non-null  float64
 6   long                              102058 non-null  float64
 7   country                           102058 non-null  category
 8   cancellation_policy               102058 non-null  category
 9   room_type                         102058 non-null  category
 10  construction_year                 102058 non-null  Int64
 11  price                             102058 non-null  float64
 12  service_fee                       102058 non-null  float64
 13  minimum_nights                    102058 non-null  Int64
 14  number_of_reviews                 102058 non-null  Int64
 15  last_review                       102058 non-null  datetime64[ns]
 16  reviews_per_month                 102058 non-null  float64
 17  review_rate_number                102058 non-null  category
 18  calculated_host_listings_count    102058 non-null  Int64
 19  availability_365                  102058 non-null  Int64
 20  is_host_unconfirmed               102058 non-null  bool
 21  is_host_verified                  102058 non-null  bool
 22  is_instant_bookable_false_policy  102058 non-null  bool
 23  is_instant_bookable_true_policy   102058 non-null  bool
dtypes: Int64(5), bool(4), category(6), datetime64[ns](1), float64(5), int64(2),
object(1)
memory usage: 12.5+ MB
```

[1030]:
```
print(df_cleaning.columns.tolist())
```

```
['id', 'name', 'host_id', 'neighbourhood_group', 'neighbourhood', 'lat', 'long',
'country', 'cancellation_policy', 'room_type', 'construction_year', 'price',
'service_fee', 'minimum_nights', 'number_of_reviews', 'last_review',
'reviews_per_month', 'review_rate_number', 'calculated_host_listings_count',
'availability_365', 'is_host_unconfirmed', 'is_host_verified',
'is_instant_bookable_false_policy', 'is_instant_bookable_true_policy']
```

[1031]:
```
# Celda 13: Subida del DataFrame Limpio a PostgreSQL

# Variables para la nueva tabla
NEW_TABLE_NAME = 'cleaned_airbnb_listings'
logging.info(f"Celda 13: Preparando para subir df_cleaning a la nueva tabla␣
 ↪'{NEW_TABLE_NAME}' en PostgreSQL.")
```

```
2025-05-17 02:01:21,524 - INFO - Celda 13: Preparando para subir df_cleaning a
la nueva tabla 'cleaned_airbnb_listings' en PostgreSQL.
```

[1032]:
```
# Celda 13.1: Definir Esquema y Crear Tabla Vacía en PostgreSQL
logging.info(f"Celda 13.1: Definiendo esquema y creando la tabla vacía␣
 ↪'{NEW_TABLE_NAME}'.")
print(f"\n--- Celda 13.1: Creando Tabla Vacía '{NEW_TABLE_NAME}' ---")
```

```
# Asegúrate de que las credenciales de DB están cargadas (deberían estarlo␣
 ↪desde celdas anteriores)
# Re-establecer conexión si es necesario o si el engine se cerró.
engine_upload = None # Definir fuera del try para el finally

try:
    if not all([POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_HOST, POSTGRES_PORT,␣
 ↪POSTGRES_DATABASE]):
        raise ValueError("Credenciales de PostgreSQL no disponibles. Asegúrate␣
 ↪de que se cargaron.")

    DATABASE_URL_UPLOAD = f"postgresql+psycopg2://{POSTGRES_USER}:
 ↪{POSTGRES_PASSWORD}@{POSTGRES_HOST}:{POSTGRES_PORT}/{POSTGRES_DATABASE}"
    engine_upload = create_engine(DATABASE_URL_UPLOAD)

    # Columnas especificadas y sus tipos SQL
    # Asegúrate que los nombres de columna coincidan con los de df_cleaning (ya␣
 ↪normalizados)
    schema_definition = {
        'id': 'BIGINT PRIMARY KEY', # id como clave primaria
        'name': 'TEXT',
        'host_id': 'BIGINT',
        'neighbourhood_group': 'TEXT', # o VARCHAR(255)
        'neighbourhood': 'TEXT',       # o VARCHAR(255)
        'lat': 'DOUBLE PRECISION',
        'long': 'DOUBLE PRECISION',
        'country': 'TEXT',             # o VARCHAR(255)
        'cancellation_policy': 'TEXT', # o VARCHAR(255)
        'room_type': 'TEXT',           # o VARCHAR(255)
        'construction_year': 'INTEGER',
        'price': 'DOUBLE PRECISION',   # o DECIMAL(10,2)
        'service_fee': 'DOUBLE PRECISION', # o DECIMAL(10,2)
        'minimum_nights': 'INTEGER',
        'number_of_reviews': 'INTEGER',
        'last_review': 'TIMESTAMP WITHOUT TIME ZONE',
        'reviews_per_month': 'DOUBLE PRECISION',
        'review_rate_number': 'INTEGER', # Asumiendo que la categoría␣
 ↪representa 1-5
        'calculated_host_listings_count': 'INTEGER',
        'availability_365': 'INTEGER',
        'is_host_unconfirmed': 'BOOLEAN',
        'is_host_verified': 'BOOLEAN',
        'is_instant_bookable_false_policy': 'BOOLEAN',
        'is_instant_bookable_true_policy': 'BOOLEAN'
    }
```

```
    # Las columnas que quieres en tu tabla
    target_columns_for_db = [
        'id', 'name', 'host_id', 'neighbourhood_group', 'neighbourhood', 'lat',␣
↪'long',
        'country', 'cancellation_policy', 'room_type', 'construction_year',
        'price', 'service_fee', 'minimum_nights', 'number_of_reviews',␣
↪'last_review',
        'reviews_per_month', 'review_rate_number',␣
↪'calculated_host_listings_count',
        'availability_365', 'is_host_unconfirmed', 'is_host_verified',
        'is_instant_bookable_false_policy', 'is_instant_bookable_true_policy'
    ]

    # Construir la sentencia CREATE TABLE
    # Usar comillas dobles para nombres de columna por si acaso, aunque los␣
↪normalizados no las necesitan
    columns_sql_definitions = []
    for col_name in target_columns_for_db:
        if col_name in schema_definition:
            columns_sql_definitions.append(f'"{col_name}"␣
↪{schema_definition[col_name]}')
        else:
            # Fallback si una columna no está en schema_definition (debería␣
↪estarlo)
            logging.warning(f"Tipo SQL no definido explícitamente para␣
↪'{col_name}'. Se usará TEXT por defecto si `to_sql` infiere.")
            # Sin embargo, la creación explícita de abajo fallará si no está.␣
↪Mejor asegurarse que todas están.
            raise KeyError(f"Tipo SQL no definido explícitamente para la␣
↪columna requerida: '{col_name}'")


    create_table_query = f"""
    CREATE TABLE IF NOT EXISTS "{NEW_TABLE_NAME}" (
        {', '.join(columns_sql_definitions)}
    );
    """
    # Usamos IF NOT EXISTS para evitar error si la tabla ya existe.
    # Si quieres reemplazarla siempre, usa DROP TABLE IF EXISTS␣
↪"{NEW_TABLE_NAME}"; antes.
    # Por ahora, vamos a hacer DROP y CREATE para asegurar una tabla limpia con␣
↪el esquema correcto.

    drop_table_query = f'DROP TABLE IF EXISTS "{NEW_TABLE_NAME}";'
```

```
    with engine_upload.connect() as connection:
        trans = connection.begin()
        try:
            logging.info(f"Ejecutando: {drop_table_query}")
            connection.execute(text(drop_table_query)) # Necesitas importar␣
 ↪text de sqlalchemy
            logging.info(f"Tabla '{NEW_TABLE_NAME}' eliminada si existía.")
            print(f"Tabla '{NEW_TABLE_NAME}' eliminada si existía.")

            logging.info(f"Ejecutando: {create_table_query}")
            connection.execute(text(create_table_query)) # Necesitas importar␣
 ↪text de sqlalchemy
            trans.commit()
            logging.info(f"Tabla vacía '{NEW_TABLE_NAME}' creada exitosamente␣
 ↪con el esquema definido.")
            print(f"Tabla vacía '{NEW_TABLE_NAME}' creada exitosamente.")
        except Exception as e_exec:
            trans.rollback()
            logging.error(f"Error al crear la tabla '{NEW_TABLE_NAME}':␣
 ↪{e_exec}")
            raise e_exec # Relanzar para detener la ejecución si la creación␣
 ↪falla

    # Importar text
    from sqlalchemy import text

except Exception as e:
    logging.error(f"Error en la Celda 13.1 (conexión o definición de tabla):␣
 ↪{e}")
    print(f"Error en la Celda 13.1: {e}")
# finally: # Mantener el engine abierto para la siguiente celda de inserción
#     if engine_upload:
#         engine_upload.dispose()
#         logging.info("Conexión del motor SQLAlchemy (engine_upload) dispuesta.
 ↪")
```

```
2025-05-17 02:01:21,554 - INFO - Celda 13.1: Definiendo esquema y creando la
tabla vacía 'cleaned_airbnb_listings'.


--- Celda 13.1: Creando Tabla Vacía 'cleaned_airbnb_listings' ---

WARNING:  la base de datos «airbnb» tiene una discordancia de versión de
ordenamiento ("collation")
DETAIL:  La base de datos fue creada usando la versión de ordenamiento 2.31,
pero el sistema operativo provee la versión 2.35.
HINT:  Reconstruya todos los objetos en esta base de datos que usen el
ordenamiento por omisión y ejecute ALTER DATABASE airbnb REFRESH COLLATION
```

```
VERSION, o construya PostgreSQL con la versión correcta de la biblioteca.
2025-05-17 02:01:21,575 - INFO - Ejecutando: DROP TABLE IF EXISTS
"cleaned_airbnb_listings";
2025-05-17 02:01:21,578 - INFO - Tabla 'cleaned_airbnb_listings' eliminada si
existía.
2025-05-17 02:01:21,580 - INFO - Ejecutando:
    CREATE TABLE IF NOT EXISTS "cleaned_airbnb_listings" (
        "id" BIGINT PRIMARY KEY, "name" TEXT, "host_id" BIGINT,
"neighbourhood_group" TEXT, "neighbourhood" TEXT, "lat" DOUBLE PRECISION, "long"
DOUBLE PRECISION, "country" TEXT, "cancellation_policy" TEXT, "room_type" TEXT,
"construction_year" INTEGER, "price" DOUBLE PRECISION, "service_fee" DOUBLE
PRECISION, "minimum_nights" INTEGER, "number_of_reviews" INTEGER, "last_review"
TIMESTAMP WITHOUT TIME ZONE, "reviews_per_month" DOUBLE PRECISION,
"review_rate_number" INTEGER, "calculated_host_listings_count" INTEGER,
"availability_365" INTEGER, "is_host_unconfirmed" BOOLEAN, "is_host_verified"
BOOLEAN, "is_instant_bookable_false_policy" BOOLEAN,
"is_instant_bookable_true_policy" BOOLEAN
    );


Tabla 'cleaned_airbnb_listings' eliminada si existía.

2025-05-17 02:01:21,595 - INFO - Tabla vacía 'cleaned_airbnb_listings' creada
exitosamente con el esquema definido.

Tabla vacía 'cleaned_airbnb_listings' creada exitosamente.
```

[1033]:
```python
# Celda 13.2: Preparar df_cleaning para la Carga
logging.info(f"Celda 13.2: Preparando df_cleaning para la carga a␣
 ↪'{NEW_TABLE_NAME}'.")
print(f"\n--- Celda 13.2: Preparando df_cleaning ---")

if not df_cleaning.empty:
    # Columnas especificadas para la base de datos (mismas que␣
 ↪target_columns_for_db de la celda anterior)
    final_columns_for_upload = [
        'id', 'name', 'host_id', 'neighbourhood_group', 'neighbourhood', 'lat',␣
 ↪'long',
        'country', 'cancellation_policy', 'room_type', 'construction_year',
        'price', 'service_fee', 'minimum_nights', 'number_of_reviews',␣
 ↪'last_review',
        'reviews_per_month', 'review_rate_number',␣
 ↪'calculated_host_listings_count',
        'availability_365', 'is_host_unconfirmed', 'is_host_verified',
        'is_instant_bookable_false_policy', 'is_instant_bookable_true_policy'
    ]

    # Verificar si todas las columnas necesarias están en df_cleaning
```

```
  missing_cols = [col for col in final_columns_for_upload if col not in␣
↪df_cleaning.columns]
  if missing_cols:
      logging.error(f"Faltan las siguientes columnas en df_cleaning para la␣
↪carga: {missing_cols}")
      print(f"ERROR: Faltan las siguientes columnas en df_cleaning:␣
↪{missing_cols}")
      # Detener o manejar el error apropiadamente
      raise ValueError(f"Columnas faltantes en df_cleaning: {missing_cols}")

  # Crear un nuevo DataFrame solo con las columnas deseadas y en el orden␣
↪correcto
  df_to_upload = df_cleaning[final_columns_for_upload].copy()
  logging.info(f"df_to_upload creado con {df_to_upload.shape[1]} columnas en␣
↪el orden especificado.")

  # Verificación de tipos y conversiones finales si son necesarias:
  # Pandas Int64 (nullable int) se mapea bien a BIGINT/INTEGER en SQL.
  # Pandas float64 se mapea bien a DOUBLE PRECISION/FLOAT.
  # Pandas datetime64[ns] se mapea bien a TIMESTAMP.
  # Pandas bool se mapea bien a BOOLEAN.
  # Pandas category/object/string se mapean a TEXT/VARCHAR.
  # La columna 'review_rate_number' es category, pero sus valores son␣
↪numéricos.
  # Si to_sql tiene problemas con category de números a INTEGER SQL, la␣
↪convertimos a Int64 aquí.
  if 'review_rate_number' in df_to_upload.columns and pd.api.types.
↪is_categorical_dtype(df_to_upload['review_rate_number']):
      try:
          # Intentar convertir las categorías a Int64 si representan números
          df_to_upload['review_rate_number'] =␣
↪df_to_upload['review_rate_number'].astype('Int64')
          logging.info("Columna 'review_rate_number' (category) convertida a␣
↪Int64 para la carga a SQL.")
      except Exception as e_astype:
          logging.warning(f"No se pudo convertir 'review_rate_number'␣
↪(category) a Int64: {e_astype}. Se cargará como texto si es posible.")
          # Si falla, se cargará como texto si el tipo SQL es TEXT/VARCHAR.␣
↪Si es INTEGER, podría fallar la carga.


  print(f"df_to_upload preparado con {df_to_upload.shape[0]} filas y␣
↪{df_to_upload.shape[1]} columnas.")
  print("Primeras filas de df_to_upload:")
  print(df_to_upload.head(2).to_markdown(index=False))
  print("\nInfo de df_to_upload:")
```

```
    df_to_upload.info()

else:
    logging.warning("El DataFrame df_cleaning está vacío. No hay datos para␣
 ↪preparar o subir.")
    print("El DataFrame df_cleaning está vacío.")
```

```
2025-05-17 02:01:21,619 - INFO - Celda 13.2: Preparando df_cleaning para la
carga a 'cleaned_airbnb_listings'.


--- Celda 13.2: Preparando df_cleaning ---

2025-05-17 02:01:21,642 - INFO - df_to_upload creado con 24 columnas en el orden
especificado.
/tmp/ipykernel_1305523/3137182911.py:36: DeprecationWarning:
is_categorical_dtype is deprecated and will be removed in a future version. Use
isinstance(dtype, pd.CategoricalDtype) instead
  if 'review_rate_number' in df_to_upload.columns and
pd.api.types.is_categorical_dtype(df_to_upload['review_rate_number']):
2025-05-17 02:01:21,653 - INFO - Columna 'review_rate_number' (category)
convertida a Int64 para la carga a SQL.

df_to_upload preparado con 102058 filas y 24 columnas.
Primeras filas de df_to_upload:
|      id | name                                |     host_id |
neighbourhood_group   | neighbourhood   |     lat |     long | country       |
cancellation_policy   | room_type       |   construction_year |   price |
service_fee |   minimum_nights |   number_of_reviews | last_review         |
reviews_per_month |   review_rate_number |   calculated_host_listings_count |
availability_365 | is_host_unconfirmed   | is_host_verified   |
is_instant_bookable_false_policy   | is_instant_bookable_true_policy   |
|--------:|:-----------------------------------|------------:|:-----------------
-----|:----------------|--------:|---------:|:-------------|:-----------------
----|:----------------|--------------------:|--------:|-------------:|---------
--------:|--------------------:|:--------------------|--------------------:|----
-----------------:|:-----------------------------------:|-------------------:|:----
-----------------|:-------------------|:-----------------------------------|:--
---------------------------------|
| 1001254 | Clean & quiet apt home by the park | 80014485718 | Brooklyn
| Kensington      | 40.6475 | -73.9724 | United States | strict               |
Private room    |                2020 |     966 |           193 |
10 |                   9 | 2021-10-19 00:00:00 |               0.21 |
4 |                                6 |                286 | True
| False              | True                               | False
|
| 1002102 | Skylit Midtown Castle               | 52335172823 | Manhattan
| Midtown         | 40.7536 | -73.9838 | United States | moderate             |
Entire home/apt |                2007 |     142 |            28 |
```

```
30 |                  45 | 2022-05-21 00:00:00 |                0.38 |
4 |                                2 |                228 | False
| True               | True                                | False
|

Info de df_to_upload:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 102058 entries, 0 to 102057
Data columns (total 24 columns):
 #   Column                            Non-Null Count   Dtype
---  ------                            --------------   -----
 0   id                                102058 non-null  int64
 1   name                              102058 non-null  object
 2   host_id                           102058 non-null  int64
 3   neighbourhood_group               102058 non-null  category
 4   neighbourhood                     102058 non-null  category
 5   lat                               102058 non-null  float64
 6   long                              102058 non-null  float64
 7   country                           102058 non-null  category
 8   cancellation_policy               102058 non-null  category
 9   room_type                         102058 non-null  category
 10  construction_year                 102058 non-null  Int64
 11  price                             102058 non-null  float64
 12  service_fee                       102058 non-null  float64
 13  minimum_nights                    102058 non-null  Int64
 14  number_of_reviews                 102058 non-null  Int64
 15  last_review                       102058 non-null  datetime64[ns]
 16  reviews_per_month                 102058 non-null  float64
 17  review_rate_number                102058 non-null  Int64
 18  calculated_host_listings_count    102058 non-null  Int64
 19  availability_365                  102058 non-null  Int64
 20  is_host_unconfirmed               102058 non-null  bool
 21  is_host_verified                  102058 non-null  bool
 22  is_instant_bookable_false_policy  102058 non-null  bool
 23  is_instant_bookable_true_policy   102058 non-null  bool
dtypes: Int64(6), bool(4), category(5), datetime64[ns](1), float64(5), int64(2),
object(1)
memory usage: 13.2+ MB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 102058 entries, 0 to 102057
Data columns (total 24 columns):
 #   Column                            Non-Null Count   Dtype
---  ------                            --------------   -----
 0   id                                102058 non-null  int64
 1   name                              102058 non-null  object
 2   host_id                           102058 non-null  int64
 3   neighbourhood_group               102058 non-null  category
 4   neighbourhood                     102058 non-null  category
```

```
 5   lat                                 102058 non-null  float64
 6   long                                102058 non-null  float64
 7   country                             102058 non-null  category
 8   cancellation_policy                 102058 non-null  category
 9   room_type                           102058 non-null  category
 10  construction_year                   102058 non-null  Int64
 11  price                               102058 non-null  float64
 12  service_fee                         102058 non-null  float64
 13  minimum_nights                      102058 non-null  Int64
 14  number_of_reviews                   102058 non-null  Int64
 15  last_review                         102058 non-null  datetime64[ns]
 16  reviews_per_month                   102058 non-null  float64
 17  review_rate_number                  102058 non-null  Int64
 18  calculated_host_listings_count      102058 non-null  Int64
 19  availability_365                    102058 non-null  Int64
 20  is_host_unconfirmed                 102058 non-null  bool
 21  is_host_verified                    102058 non-null  bool
 22  is_instant_bookable_false_policy    102058 non-null  bool
 23  is_instant_bookable_true_policy     102058 non-null  bool
dtypes: Int64(6), bool(4), category(5), datetime64[ns](1), float64(5), int64(2),
object(1)
memory usage: 13.2+ MB
```

[1034]:
```python
# Celda 13.3: Insertar Datos en la Nueva Tabla
logging.info(f"Celda 13.3: Insertando datos de df_to_upload en la tabla␣
 ↪'{NEW_TABLE_NAME}'.")
print(f"\n--- Celda 13.3: Insertando Datos en '{NEW_TABLE_NAME}' ---")

# Reutilizar engine_upload si sigue disponible y la conexión no se cerró␣
 ↪explícitamente
# O recrearlo si es necesario (como está arriba, se crea en Celda 13.1 y se␣
 ↪asume disponible)

if 'df_to_upload' in locals() and not df_to_upload.empty and 'engine_upload' in␣
 ↪locals() and engine_upload is not None:
    try:
        logging.info(f"Intentando insertar {len(df_to_upload)} filas en␣
 ↪'{NEW_TABLE_NAME}'.")
        # Usar if_exists='append' ya que la tabla fue creada vacía.
        # chunksize puede ayudar con la memoria para DataFrames grandes.
        df_to_upload.to_sql(
            name=NEW_TABLE_NAME,
            con=engine_upload,
            if_exists='append',
            index=False,
            chunksize=1000 # Ajusta según el tamaño de tu DF y memoria
        )
```

```
        logging.info(f"Datos insertados exitosamente en la tabla␣
 ↪'{NEW_TABLE_NAME}'.")
        print(f"Datos insertados exitosamente en la tabla '{NEW_TABLE_NAME}'.")

    except Exception as e:
        logging.error(f"Error al insertar datos en la tabla '{NEW_TABLE_NAME}':␣
 ↪{e}")
        print(f"Error al insertar datos en '{NEW_TABLE_NAME}': {e}")
    finally:
        if engine_upload: # Cerrar el engine después de la operación final
            engine_upload.dispose()
            logging.info("Conexión del motor SQLAlchemy (engine_upload)␣
 ↪dispuesta después de la carga.")
            print("Motor de DB (engine_upload) dispuesto.")

elif 'df_to_upload' not in locals() or df_to_upload.empty:
    logging.warning("df_to_upload no está definido o está vacío. No se␣
 ↪insertaron datos.")
    print("df_to_upload no está definido o está vacío. No se insertaron datos.")
elif 'engine_upload' not in locals() or engine_upload is None:
    logging.warning("El motor de base de datos (engine_upload) no está␣
 ↪inicializado. No se pueden insertar datos.")
    print("Motor de DB (engine_upload) no inicializado. No se pueden insertar␣
 ↪datos.")
```

```
2025-05-17 02:01:21,694 - INFO - Celda 13.3: Insertando datos de df_to_upload en
la tabla 'cleaned_airbnb_listings'.
2025-05-17 02:01:21,697 - INFO - Intentando insertar 102058 filas en
'cleaned_airbnb_listings'.


--- Celda 13.3: Insertando Datos en 'cleaned_airbnb_listings' ---

2025-05-17 02:01:27,016 - INFO - Datos insertados exitosamente en la tabla
'cleaned_airbnb_listings'.
2025-05-17 02:01:27,017 - INFO - Conexión del motor SQLAlchemy (engine_upload)
dispuesta después de la carga.

Datos insertados exitosamente en la tabla 'cleaned_airbnb_listings'.
Motor de DB (engine_upload) dispuesto.
```

[1035]:
```
# Celda 13.4: Verificar Datos en PostgreSQL (Opcional)
logging.info(f"Celda 13.4: Verificando las primeras filas de la tabla␣
 ↪'{NEW_TABLE_NAME}' desde PostgreSQL.")
print(f"\n--- Celda 13.4: Verificando Datos en '{NEW_TABLE_NAME}' ---")

engine_verify = None
try:
```

```
    # Se necesita un nuevo engine o reutilizar uno si no se cerró
    if not all([POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_HOST, POSTGRES_PORT,␣
↪POSTGRES_DATABASE]):
        raise ValueError("Credenciales de PostgreSQL no disponibles para␣
↪verificación.")

    DATABASE_URL_VERIFY = f"postgresql+psycopg2://{POSTGRES_USER}:
↪{POSTGRES_PASSWORD}@{POSTGRES_HOST}:{POSTGRES_PORT}/{POSTGRES_DATABASE}"
    engine_verify = create_engine(DATABASE_URL_VERIFY)

    query_verify = f'SELECT * FROM "{NEW_TABLE_NAME}" LIMIT 5;'
    df_from_db_cleaned = pd.read_sql_query(text(query_verify),␣
↪con=engine_verify) # Usar text()

    logging.info(f"Primeras 5 filas obtenidas de '{NEW_TABLE_NAME}' en␣
↪PostgreSQL:")
    print(f"Primeras 5 filas de la tabla '{NEW_TABLE_NAME}' consultadas desde␣
↪PostgreSQL:")
    if not df_from_db_cleaned.empty:
        print(df_from_db_cleaned.to_markdown(index=False))
    else:
        logging.info("La consulta no devolvió filas o la tabla está vacía␣
↪después de la carga.")
        print("La tabla parece estar vacía o la consulta no devolvió resultados.
↪")

except Exception as e:
    logging.error(f"Error al consultar datos desde la tabla '{NEW_TABLE_NAME}'␣
↪en PostgreSQL: {e}")
    print(f"Error al consultar datos desde '{NEW_TABLE_NAME}': {e}")
finally:
    if engine_verify:
        engine_verify.dispose()
        logging.info("Conexión del motor SQLAlchemy (engine_verify) dispuesta.")
        print("Motor de DB (engine_verify) dispuesto.")
```

```
2025-05-17 02:01:27,025 - INFO - Celda 13.4: Verificando las primeras filas de
la tabla 'cleaned_airbnb_listings' desde PostgreSQL.
WARNING:  la base de datos «airbnb» tiene una discordancia de versión de
ordenamiento ("collation")
DETAIL:  La base de datos fue creada usando la versión de ordenamiento 2.31,
pero el sistema operativo provee la versión 2.35.
HINT:  Reconstruya todos los objetos en esta base de datos que usen el
ordenamiento por omisión y ejecute ALTER DATABASE airbnb REFRESH COLLATION
VERSION, o construya PostgreSQL con la versión correcta de la biblioteca.
2025-05-17 02:01:27,037 - INFO - Primeras 5 filas obtenidas de
'cleaned_airbnb_listings' en PostgreSQL:
```

2025-05-17 02:01:27,040 - INFO - Conexión del motor SQLAlchemy (engine_verify) dispuesta.

--- Celda 13.4: Verificando Datos en 'cleaned_airbnb_listings' ---
Primeras 5 filas de la tabla 'cleaned_airbnb_listings' consultadas desde PostgreSQL:

id	name	host_id	neighbourhood_group	neighbourhood	lat	long	country	cancellation_policy	room_type	construction_year	price	service_fee	minimum_nights	number_of_reviews	last_review	reviews_per_month	review_rate_number	calculated_host_listings_count	availability_365	is_host_unconfirmed	is_host_verified	is_instant_bookable_false_policy	is_instant_bookable_true_policy
1001254	Clean & quiet apt home by the park	80014485718	Brooklyn	Kensington	40.6475	-73.9724	United States	strict	Private room	2020	966	193	10	9	2021-10-19 00:00:00	0.21	4	6	286	True	False	True	False
1002102	Skylit Midtown Castle	52335172823	Manhattan	Midtown	40.7536	-73.9838	United States	moderate	Entire home/apt	2007	142	28	30	45	2022-05-21 00:00:00	0.38	4	2	228	False	True	True	False
1002403	THE VILLAGE OF HARLEM…NEW YORK !	78829239556	Manhattan	Harlem	40.809	-73.9419	United States	flexible	Private room	2005	620	124	3	0	2019-06-10 18:47:53.419777	1.98936	5	1	352	False	False	False	True
1002755	Desconocido	85098326012	Brooklyn	Clinton Hill	40.6851	-73.9598	United States	moderate	Entire home/apt	2005	368	74	30	270	2019-07-05 00:00:00	4.64	4	1	322	True	False	False

```
| True                                |
| 1003689 | Entire Apt: Spacious Studio/Loft by central park | 92037596077 |
Manhattan              | East Harlem     | 40.7985 | -73.944  | United States |
moderate               | Entire home/apt |                2009 |     204 |
41 |               10 |                    9 | 2018-11-19 00:00:00        |
0.1     |                    3 |                                1 |
289 | False                 | True               | True
| False                               |
Motor de DB (engine_verify) dispuesto.
```

[1036]:
```
engine.dispose() # Cerrar todas las conexiones en el pool del engine
logging.info("Conexiones del motor de SQLAlchemy dispuestas (cerradas).")
```

```
2025-05-17 02:01:27,046 - INFO - Conexiones del motor de SQLAlchemy dispuestas
(cerradas).
```