Data Types and Missing Values

Data Type

Pandas is a powerful Python library for data manipulation and analysis, providing flexible data structures designed to make working with structured (tabular, multidimensional, potentially heterogeneous) and time series data both easy and intuitive. One common task in exploratory data analysis (EDA) is changing the data type of columns in a DataFrame. Thispart of manual covers why and how to change data types in Pandas, including practical examples

1- Understanding Data Types in Pandas

Understanding data types in Pandas is crucial for effective data manipulation and analysis, as the type of data dictates what kind of operations can be performed on a column. Pandas is designed to work with diverse data types, including integers, floats, strings (objects in Pandas), booleans, datetime objects, and more. Here's a breakdown of the primary data types in Pandas and how they're used:

Numeric Types

Integers (int64, int32, int16, etc.): Used for representing whole numbers. The number following int indicates the bits allocated, affecting the range of values that can be stored.
Floats (float64, float32, etc.): Used for representing real numbers (i.e., numbers with decimals). Like integers, the number following float indicates the precision.

Object Type

Object (object): The default data type for textual data. A column with mixed types is also stored as an object type.

Datetime Types

Datetime (datetime64): Used for representing dates and times. Very useful for time series analysis.

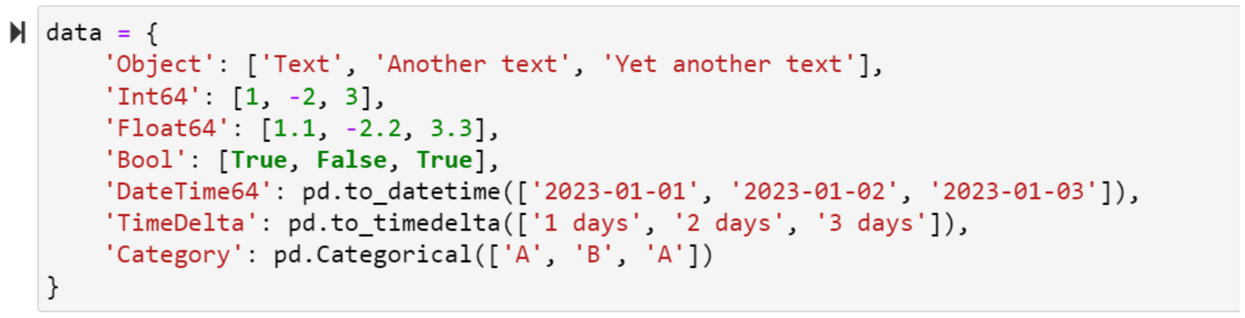
Boolean Type

Boolean (bool): Represents True or False values.

Categorical Data

Categorical (category): Used for data that can take on a limited, fixed number of categories. This type can be very memory efficient.

Lets make a data consist of all data types:



```python
data = {
    'Object': ['Text', 'Another text', 'Yet another text'],
    'Int64': [1, -2, 3],
    'Float64': [1.1, -2.2, 3.3],
    'Bool': [True, False, True],
    'DateTime64': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03']),
    'TimeDelta': pd.to_timedelta(['1 days', '2 days', '3 days']),
    'Category': pd.Categorical(['A', 'B', 'A'])
}
```

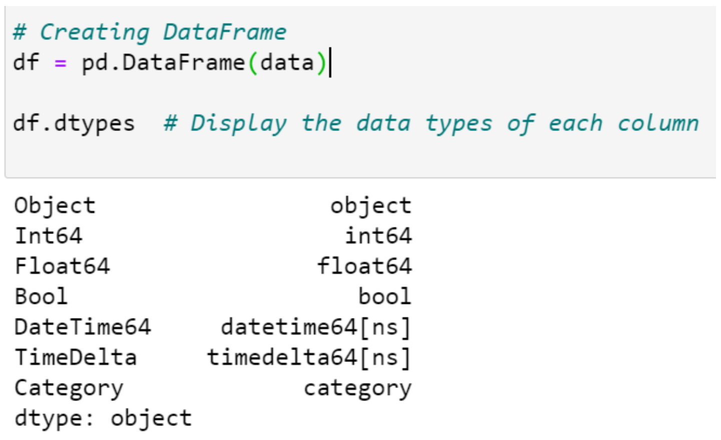
2- Understanding Each Type's Use

Numeric Types: Ideal for mathematical calculations and statistical analysis.
Object Type: While versatile, operations on object columns are usually slower and more memory-intensive. It's often beneficial to convert these columns to more specific types.
Datetime Types: Essential for time series data, allowing for easy manipulation and formatting of dates and times.
Boolean Type: Useful for filtering operations and conditions.
Categorical Data: Enhances performance and memory usage for columns with a small number of distinct values.

3- Checking Data Types in a DataFrame

To understand what data types you're working with in a Pandas DataFrame, you can use the **df.dtypes** attribute. This will list each column's name along with its data type, helping you identify which columns might need type conversion for more efficient analysis or to enable certain types of operations.

Lets convert our data to datafram and check data type of each column:



```
# Creating DataFrame
df = pd.DataFrame(data)

df.dtypes  # Display the data types of each column
```

```
Object                  object
Int64                    int64
Float64                float64
Bool                      bool
DateTime64      datetime64[ns]
TimeDelta      timedelta64[ns]
Category              category
dtype: object
```

Why It Matters

Knowing the data types is the first step in data preprocessing. It helps in identifying potential issues with data quality, such as numeric values being read as objects due to formatting issues or missing values. Furthermore, converting data to the appropriate type can lead to significant improvements in memory usage and computational efficiency, especially when dealing with large datasets.

Next, we'll look into how to change these data types to suit our analysis needs, which is a critical skill in ensuring data integrity and optimizing the performance of your data analysis tasks.

4- Changing Data Types in Pandas

Changing the data type of a column in a DataFrame is a common operation in data preprocessing. It involves converting a column from one type to another, such as converting a string to a datetime object or an integer to a float. This operation is crucial for ensuring data consistency,enabling accurate analysis, and optimizing memory usage.

Common Reasons for Changing Data Types:

1. **Ensuring data consistency:** Data imported from various sources may not always align with the desired or expected types for analysis.

2. **Optimizing memory usage:** Converting data types can significantly reduce memory usage,particularly when working with large datasets.
3. **Enabling specific data operations**: Certain operations require specific data types. For example, arithmetic operations require numeric types, while string operations require objects of type string.

How to Change Data Types in Pandas:

Pandas provides several methods for changing data types, including

- astype()
- to_numeric(),
- to_datetime(),
- to_timedelta().

1. Using astype():

The astype() method is used to cast a pandas object to a specified dtype. It can convert one or more columns to the desired data types.
Example:
df['column_name'] = df['column_name'].astype('desired_type')

2. Using to_numeric():

The to_numeric() function is used to convert a column to a numeric type. It is useful for converting columns that pandas has incorrectly identified as object types due to mixed data types or the presence of non-numeric values.
Example:
df['column_name'] = pd.to_numeric(df['column_name'], errors='coerce')

3. Using to_datetime() and to_timedelta():

These functions are used to convert columns to datetime and timedelta objects, respectively. They are particularly useful for time series analysis.
Example:
df['date_column'] = pd.to_datetime(df['date_column'])

```
df['duration_column'] = pd.to_timedelta(df['duration_column'])
```

Practical Examples

Let's consider a DataFrame named df with columns 'A', 'B', and 'C', where 'A' contains dates in string format, 'B' contains numeric values in string format, and 'C' contains integer values.

1. Converting 'A' from string to datetime:

```
df['A'] = pd.to_datetime(df['A'])
```

2. Converting 'B' from string to numeric:

```
df['B'] = pd.to_numeric(df['B'], errors='coerce')
```

3. Converting 'C' from integer to float:

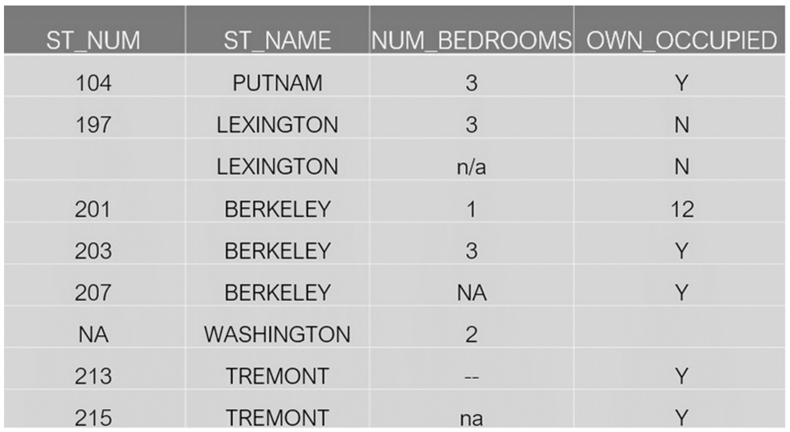
```
df['C'] = df['C'].astype(float)
```

—---

Missing Values

Handling missing values is a critical step in data preprocessing and analysis. Missing data can arise due to various reasons, such as errors in data collection, non-response in surveys, or deletions of existing data entries. Pandas provides a comprehensive set of tools to deal with missing values, enabling data scientists to clean, analyze, and impute missing data effectively. The part of this manual introduces these tools and techniques for managing missing values in Pandas DataFrames

1- What are Missing Values?



ST_NUM	ST_NAME	NUM_BEDROOMS	OWN_OCCUPIED
104	PUTNAM	3	Y
197	LEXINGTON	3	N
	LEXINGTON	n/a	N
201	BERKELEY	1	12
203	BERKELEY	3	Y
207	BERKELEY	NA	Y
NA	WASHINGTON	2	
213	TREMONT	--	Y
215	TREMONT	na	Y

.

1- Detecting Missing Values:

Detecting missing values is the first step in handling them. Pandas represents missing values as NaN (Not a Number) for numeric data types and None or NaN for object data types.
The primary functions used to detect missing values are:

- isna(): Returns a boolean same-sized object indicating if the elements are NA. NA values, such as None or numpy.NaN, gets mapped to True values.
- notna(): Returns a boolean same-sized object indicating if the elements are not NA. Non-missing values get mapped to True values

2- Handling Missing Values

Once missing values are detected, there are several strategies to handle them, Some are following:

- Drop Null or Missing Values Altogether
- Fill Missing Values using Imputation (Mean , Median , Mode)
- Predicting Missing Values with Machine Learning Algorithm

Removing Missing Values:

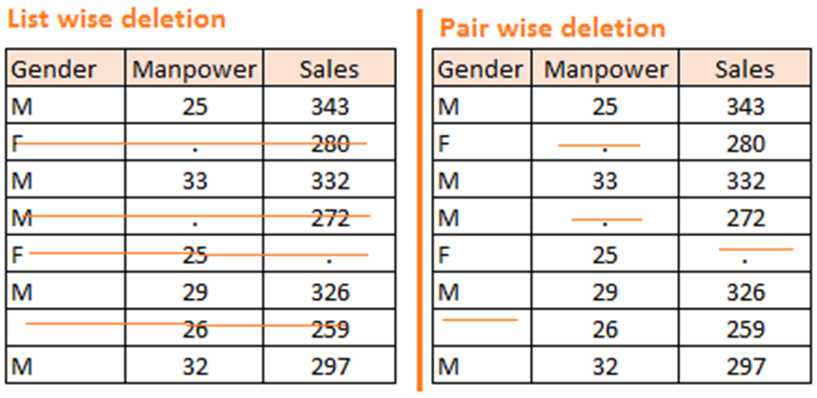
This is the fastest and easiest step to handle missing values.

This method reduces the quality of our model as it reduces sample size.

If the missing data is less than 5 % for a large dataset , we can delete missing values :

Deleting 500 records from 10,000 records

- dropna(): Removes missing values from a DataFrame. You can specify how='any' to drop rows with any missing values or how='all' to drop rows where all values are missing.
  Example: df.dropna()
  # Removes rows with any missing values
  df.dropna(how=any)
  # Removes rows where all values are missing
  df.dropna(how=all)



List wise deletion

Gender	Manpower	Sales
M	25	343
~~F~~	~~.~~	~~280~~
M	33	332
~~M~~	~~.~~	~~272~~
~~F~~	~~25~~	~~.~~
M	29	326
	~~26~~	~~259~~
M	32	297

Pair wise deletion

Gender	Manpower	Sales
M	25	343
F	~~.~~	280
M	33	332
M	~~.~~	272
F	25	~~.~~
M	29	326
	26	259
M	32	297

Filling Missing Values:

- fillna(): Allows you to replace missing values with a specified value or method (e.g., 'ffill' for forward fill, 'backfill'/'bfill' for backward fill).
  Example:
  # Replaces missing values with 0
  df.fillna(0)
  # Forward fills missing values
  df.fillna(method='ffill')

3- Imputing Missing Values

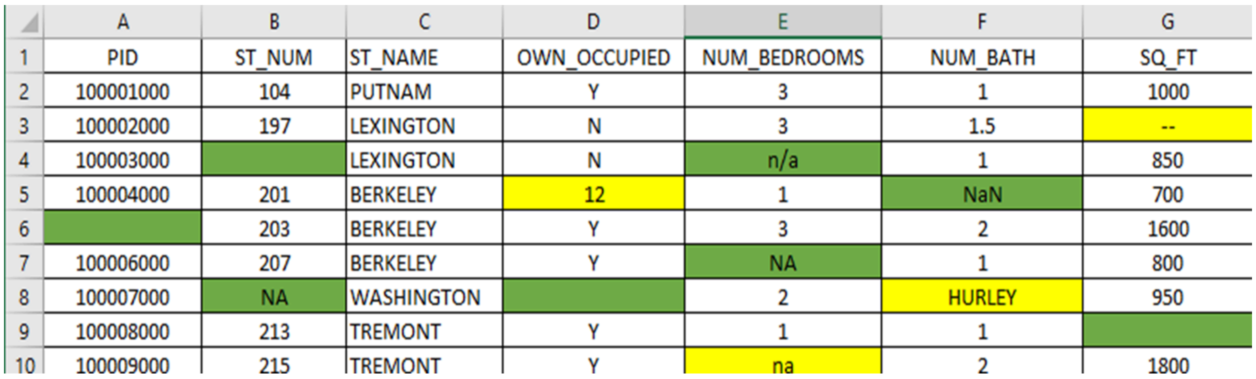
We can impute the missing values.

- Using mean and median for numerical attributes.
  - Mean is sensitive to outliers.
- Using mode for Categorical Attributes.
- Using a Machine Learning Algorithm like KNN.
  - It will give the maximum close estimation , but
  - It can be compute intensive

.Pandas provides basic imputation techniques, but for more advanced methods, libraries such as Scikit-learn can be used.

Example of mean imputation:
df['column_name'].fillna(df['column_name'].mean(), inplace=True)

An Example Use Case

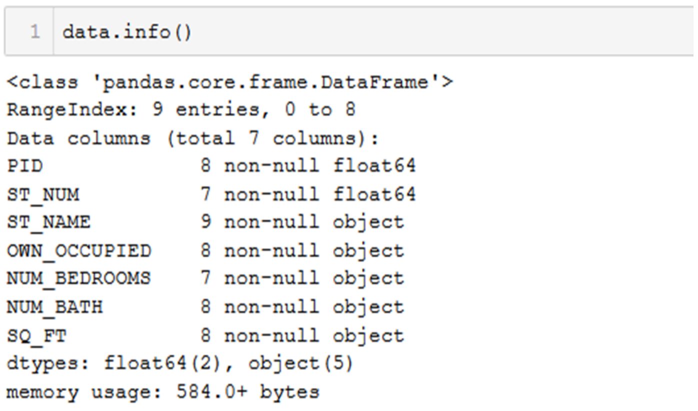


	A	B	C	D	E	F	G
1	PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
2	100001000	104	PUTNAM	Y	3	1	1000
3	100002000	197	LEXINGTON	N	3	1.5	--
4	100003000		LEXINGTON	N	n/a	1	850
5	100004000	201	BERKELEY	12	1	NaN	700
6		203	BERKELEY	Y	3	2	1600
7	100006000	207	BERKELEY	Y	NA	1	800
8	100007000	NA	WASHINGTON		2	HURLEY	950
9	100008000	213	TREMONT	Y	1	1	
10	100009000	215	TREMONT	Y	na	2	1800

A Pandas dataframe can recognize the following values as missing:

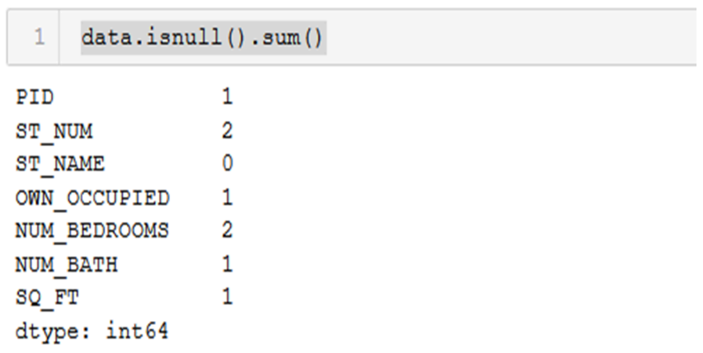
- NA
- NaN
- n/a
- N/A
- Blank Field

How many missing values are there?



```
data.info()
```

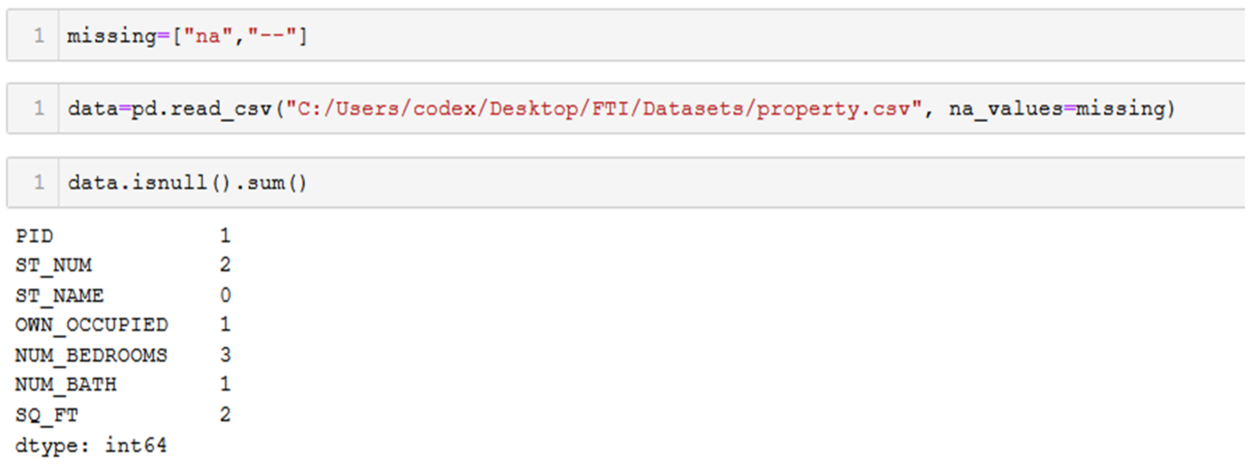
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9 entries, 0 to 8
Data columns (total 7 columns):
PID             8 non-null float64
ST_NUM          7 non-null float64
ST_NAME         9 non-null object
OWN_OCCUPIED    8 non-null object
NUM_BEDROOMS    7 non-null object
NUM_BATH        8 non-null object
SQ_FT           8 non-null object
dtypes: float64(2), object(5)
memory usage: 584.0+ bytes
```



```
data.isnull().sum()
```

```
PID             1
ST_NUM          2
ST_NAME         0
OWN_OCCUPIED    1
NUM_BEDROOMS    2
NUM_BATH        1
SQ_FT           1
dtype: int64
```

Defining the Missing values



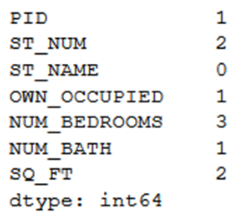
```
missing=["na","--"]
```

```
data=pd.read_csv("C:/Users/codex/Desktop/FTI/Datasets/property.csv", na_values=missing)
```

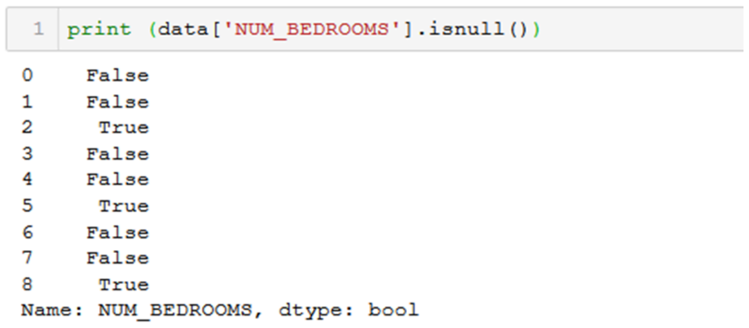
```
data.isnull().sum()
```

```
PID             1
ST_NUM          2
ST_NAME         0
OWN_OCCUPIED    1
NUM_BEDROOMS    3
NUM_BATH        1
SQ_FT           2
dtype: int64
```

Checking Individual Columns for Null Values



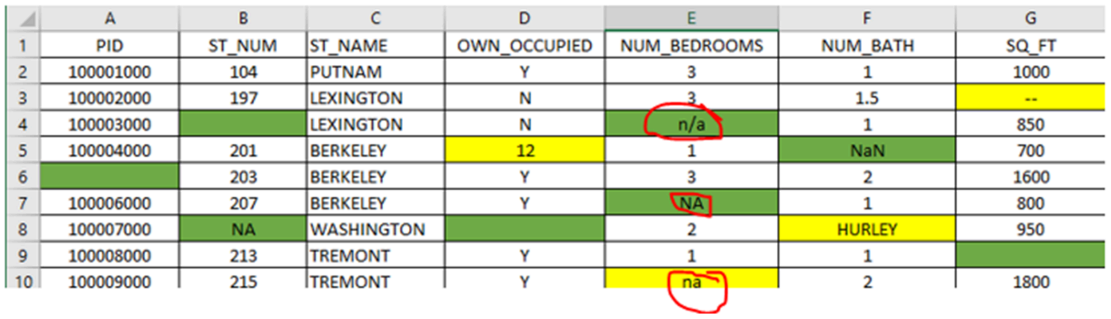
```
PID             1
ST_NUM          2
ST_NAME         0
OWN_OCCUPIED    1
NUM_BEDROOMS    3
NUM_BATH        1
SQ_FT           2
dtype: int64
```



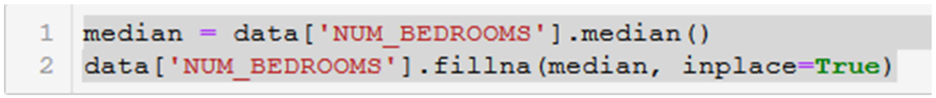
```
print (data['NUM_BEDROOMS'].isnull())
```

```
0    False
1    False
2     True
3    False
4    False
5     True
6    False
7    False
8     True
Name: NUM_BEDROOMS, dtype: bool
```

Replacing with Median - Num_Bedrooms



	A	B	C	D	E	F	G
1	PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
2	100001000	104	PUTNAM	Y	3	1	1000
3	100002000	197	LEXINGTON	N	3	1.5	--
4	100003000		LEXINGTON	N	n/a	1	850
5	100004000	201	BERKELEY	12	1	NaN	700
6		203	BERKELEY	Y	3	2	1600
7	100006000	207	BERKELEY	Y	NA	1	800
8	100007000	NA	WASHINGTON		2	HURLEY	950
9	100008000	213	TREMONT	Y	1	1	
10	100009000	215	TREMONT	Y	na	2	1800



```
median = data['NUM_BEDROOMS'].median()
data['NUM_BEDROOMS'].fillna(median, inplace=True)
```

Deleting Rows with Null Values

In Some Cases we can delete all the rows with Null Values

- data.dropna(inplace =True)