Data Visualization BTech Computer Science Stream , January 2025 Week 5 - Data Cleaning and Preparation - Demonstration notebook Archana Praveek Kumar, Reg Number , Date: 06/01/2025

- During data analysis and modeling, a significant amount of time is spent on data preparation.
- Tasks include loading, cleaning, transforming, and rearranging data.
- Such tasks are often reported to take up 80% or more of an analyst's time.
- Pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

In [1]:
```
import numpy as np
import pandas as pd
```

In [4]:
```
# For float64 dtype, pandas uses NaN (Not a Number) to represent missing data
# Call this a sentinel value: when present, it indicates a missing (or null) value:
float_data = pd.Series([1.2, -3.5, np.nan, 0]) # to represent NA (Not available data)
float_data
#float_data.isna() # to check whether any value is NA or not
# float_data.notna()
```

Out[4]:
```
0     True
1     True
2    False
3     True
dtype: bool
```

In [3]:
```
# The built-in Python None value is also treated as NA:
string_data = pd.Series(["aardvark", np.nan, None, "avocado"])
string_data
#string_data.isna()
```

Out[3]:
```
0    aardvark
1         NaN
2        None
3     avocado
dtype: object
```

- NA data may either be data that does not exist or that exists but was not observed.
- When cleaning up data for analysis, it is important to analyse missing data
- to identify data collection problems or potential biases in the data caused by missing data.

In [2]:
```
# Strategies with missing data include 1. Dropping missing values
data1 = pd.Series([1, np.nan, 3.5, np.nan, 7])
data1
# by default drops any row containing a missing value
data1.dropna()
```

Out[2]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [9]:
```
data2 = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],
                      [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])
data2
#data2.dropna(how="all") # drops rows that are all NA
data2.dropna(axis="columns", how="all")# filter to removes columns which have all valu
```

Out[9]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

In [15]:
```
# Strategies with missing data include 2. Filling missing values
df3 = pd.DataFrame(np.random.standard_normal((7, 3)))
df3.iloc[:4, 1] = np.nan
df3.iloc[:2, 2] = np.nan
df3.fillna(0)
#df3
```

Out[15]:

	0	1	2
0	-0.409814	0.000000	0.000000
1	1.278383	0.000000	0.000000
2	-1.377074	0.000000	-0.073055
3	0.217108	0.000000	0.912957
4	0.301200	-0.475921	-0.078225
5	0.958026	-0.829628	0.194278
6	-1.102895	-1.917630	0.388435

In [17]:
```
# A dictionary where keys are column labels, and values are the fill values for those
df4 = pd.DataFrame(np.random.standard_normal((7, 3)))
df4.iloc[:4, 1] = np.nan
df4.iloc[:2, 2] = np.nan
#df4.fillna({1: 0.5, 2: 0})
#df4.fillna(df4.mean())
```

Out[17]:

	0	1	2
0	-0.105383	-0.266055	-0.106236
1	-0.799486	-0.266055	-0.106236
2	-0.397134	-0.266055	-0.784092
3	-0.392168	-0.266055	-1.299590
4	-0.356177	-1.420842	0.678405
5	-0.425674	1.203101	-1.024148
6	-0.068057	-0.580424	1.898248

In [7]:

```
# Can also use the replace method to replace 1 or more values.
data = pd.DataFrame({
    'A': [1, -999, 3, -1000],
    'B': [-1000, 5, -999, 7]
})
print(data)
# valid way to replace multiple values (-999 and -1000) with a single value (np.nan),
#If you want to modify the DataFrame or Series in place, use this argument. Otherwise,
data.replace([-999.0,-1000], np.nan, inplace=True)
print(data)
```

```
      A     B
0     1 -1000
1  -999     5
2     3  -999
3 -1000     7
     A    B
0  1.0  NaN
1  NaN  5.0
2  3.0  NaN
3  NaN  7.0
```

In [32]:

```
# Data Transformations include 1. removing duplicates
data5 = pd.DataFrame({"k1": ["one", "two"] * 3 + ["two"],
                      "k2": [1, 1, 2, 3, 3, 4, 4]})

#data5.duplicated()
data5.drop_duplicates()
#data5.drop_duplicates(subset=["k1"])
```

Out[32]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

In [2]:
```
# Data Transformations include 2. creating new columns
# DataFrame named data with two columns: "food" and "ounces"

data = pd.DataFrame({"food": ["salt", "bread", "butter",
                              "rice", "chips", "biscuits",
                              "chocolate", "honey", "sugar"],
                     "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

Out[2]:

	food	ounces
0	salt	4.0
1	bread	3.0
2	butter	12.0
3	rice	6.0
4	chips	7.5
5	biscuits	8.0
6	chocolate	3.0
7	honey	5.0
8	sugar	6.0

In [5]:
```
# The dictionary food_to_type maps different types of food to their food type.

food_to_type = {
  "salt": "condiment",
  "bread": "breakfast",
  "butter": "condiment",
  "rice": "grocery",
  "chips": "snack",
  "honey": "condiment",
  "sugar": "condiment",
  "chocolate": "snack"
}
```

In [6]:
```
#used to create a new column "ftype" in the data DataFrame, based on the values in the
#The map() function is applied to the "food" column, using the food_to_type dictionary

data["ftype"] = data["food"].map(food_to_type)
data
```

Out[6]:

	food	ounces	ftype
0	salt	4.0	condiment
1	bread	3.0	breakfast
2	butter	12.0	condiment
3	rice	6.0	grocery
4	chips	7.5	snack
5	biscuits	8.0	NaN
6	chocolate	3.0	snack
7	honey	5.0	condiment
8	sugar	6.0	condiment

In [8]:
```
# Data Transformations include 3. Discretization and Binning
# Discretization and Binning
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

In [9]:
```
bins = [18, 25, 35, 60, 100] #Assign custom labels to each bin:
age_categories = pd.cut(ages, bins)#Assigns each age to one of these intervals. If an
age_categories
```

Out[9]:
```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 6
0], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

In [10]:
```
age_categories.codes# Returns the integer codes representing the bin each age belongs
print(age_categories.categories)#Returns the IntervalIndex of the categories (the bins
print(age_categories.categories[0])#Accesses the first category (interval
print(pd.value_counts(age_categories))
```

```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval[int64, righ
t]')
(18, 25]
(18, 25]     5
(25, 35]     3
(35, 60]     3
(60, 100]    1
dtype: int64
```

In [11]:
```
#By default, intervals are right-closed (e.g., (18, 25]), meaning the right endpoint i
#the intervals become left-closed (e.g., [18, 25)), meaning the left endpoint is inclu
pd.cut(ages, bins, right=False)
```

Out[11]:
```
[[18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ..., [25, 35), [60, 100), [35, 6
0), [35, 60), [25, 35)]
Length: 12
Categories (4, interval[int64, left]): [[18, 25) < [25, 35) < [35, 60) < [60, 100)]
```

In [12]:
```
#Assigns custom labels to the bins:
#(18, 25] → "Youth"
#(25, 35] → "YoungAdult"
#(35, 60] → "MiddleAged"
#(60, 100] → "Senior"
```

```
#Each age is categorized into one of the groups based on the bin it falls into.
#For example:
#20 → "Youth"
#27 → "YoungAdult"
#61 → "Senior"
```

In [13]:
```
group_names = ["Youth", "YoungAdult", "MiddleAged", "Senior"]
pd.cut(ages, bins, labels=group_names)
```

Out[13]:
```
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Senior', 'Midd
leAged', 'MiddleAged', 'YoungAdult']
Length: 12
Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senior']
```

In [14]:
```
print(pd.value_counts(age_categories))
```

```
(18, 25]     5
(25, 35]     3
(35, 60]     3
(60, 100]    1
dtype: int64
```

In [49]:
```
# Data Transformations includes 4. Detecting and Filtering Outliers:
data = pd.DataFrame(np.random.standard_normal((1000, 4)))
data.describe()
```

Out[49]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.035320	-0.047482	-0.040091	0.006336
std	0.969095	0.962987	0.994311	1.019735
min	-3.333767	-3.194414	-3.108915	-3.645860
25%	-0.619875	-0.735556	-0.734509	-0.680576
50%	0.034764	-0.048538	-0.049977	0.040150
75%	0.688640	0.647824	0.621226	0.713369
max	3.525865	2.611678	3.366626	2.763474

In [51]:
```
col = data[2] #Extracts the third column (index 2) of the DataFrame data, Computes the
col[col.abs() > 3] #Filters the col Series, keeping only the rows where the absolute v
```

Out[51]:
```
134    3.366626
630   -3.108915
Name: 2, dtype: float64
```

In [53]:
```
#data.abs(): Computes the absolute value of all elements in the DataFrame.
# returns a DataFrame of the same shape with True for values where the condition is me
#.any(axis="columns"): Checks each row across all columns to see if any value in that
data[(data.abs() > 3).any(axis="columns")]
```

Out[53]:

	0	1	2	3
42	0.208011	-0.150923	-0.362528	-3.548824
134	0.193299	1.397822	3.366626	-2.372214
281	3.525865	0.283070	0.544635	0.462204
301	-0.450721	-0.080332	0.599947	-3.645860
524	-3.333767	-1.240685	-0.650855	0.076254
605	0.344072	0.581893	-1.116332	-3.018842
630	-0.555434	-0.048478	-3.108915	1.117755
760	-0.217146	-0.274138	1.188742	-3.183867
807	0.744019	1.741426	-2.214074	-3.140963
973	-0.848098	-3.194414	0.077839	-1.733549

In [54]:

```
data[data.abs() > 3] = np.sign(data) * 3 # Cap values at ±3
#np.sign(data) * 3: Multiplies the sign of each value by 3: Values greater than 3 are
data.describe()
```

Out[54]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.035128	-0.047288	-0.040348	0.007874
std	0.966234	0.962371	0.992790	1.014803
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.619875	-0.735556	-0.734509	-0.680576
50%	0.034764	-0.048538	-0.049977	0.040150
75%	0.688640	0.647824	0.621226	0.713369
max	3.000000	2.611678	3.000000	2.763474

In [63]:

```
# Data Transformation include 5. Computing indicator/dummy variables.
#key: Categorical column with values a, b, and c.
#data1: Numerical column with values from 0 to 5.
df = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
                   "data1": range(6)})
print(df)
pd.get_dummies(df["key"], dtype=float)
```

```
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   b      5
```

Out[63]:

	a	b	c
0	0.0	1.0	0.0
1	0.0	1.0	0.0
2	1.0	0.0	0.0
3	0.0	0.0	1.0
4	1.0	0.0	0.0
5	0.0	1.0	0.0