Python Tutorial – Part II: Data

Mark A. Austin

University of Maryland

austin@umd.edu
ENCE 201, Fall Semester 2025

September 2, 2025

Overview

Tabular and Non-Tabular Data Models

Tabular and Non-Tabular Data Models

Tabular Data (Structured Data)

Tabular data refers to data presented in rows and columns. Rows represent separate observations; columns represent measured (observation) variables.

Spreadsheet formalisms allow large amounts of raw data to be sorted and reorganized in an aggregated format.

Sources: forms, online transaction processing systems.

Formats: CSV, Excel, Google Sheets.

Tabular and Non-Tabular Data Models

Non-Tabular Data (Unstructured Data)

Non-tabular data refers to any data that does not fit into a traditional tabular structure (rows and columns).

Applications: Text, images, audio, video, and graph data.

Sources: E-mails, multimedia files, observational data.

Formats: Audio files (mp3); documents (pdf); images (jpeg, png), web pages (xml,html), geospatial/openstreetmap (osm).

Tabular and Non-Tabular Data Models

Examples:

Tabular Data

homogeneous
(contains a single type of data)

columns

rows	0	1	2
0	1.5	21.0	76.4
1	4.0	35.0	99.7
2	3.0	17.0	85.3
3	4.0	53.0	90.7

All numerical data

	0	1	2
0	'a'	'Mia'	'1'
1	'c'	'Lucas'	'x-1'
2	'e'	'Ang'	'zz'
3	'b'	'Jia'	'0.3'

All string data

heterogeneous
(contains multiple types of data)

	0	1	2
0	1.5	21.0	'Mia'
1	4.0	35.0	'Lucas'
2	3.0	17.0	'Ang'
3	4.0	53.0	'Jia'

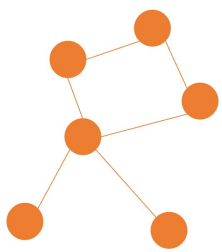
Numerical and string datatypes

Non-tabular Data Examples

Unstructured Text

```
'Twas brillig, and
the slithy toves
Did gyre and gimble
in the wabe:
All mimsy were the
borogoves,
And the mome raths
outgrabe.
```

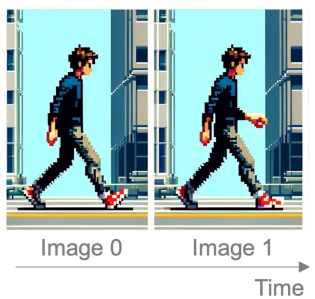
Network Data



Geospatial Data

Image Data

Video Data



Homogeneous and Heterogeneous Data

Homogeneous Data

Data that is uniform (or consistent) within a set. Data elements share a common format and kind (e.g., an array of integers).

Heterogeneous Data

Data that is not uniform (or consistent) in its structure, format or type (e.g., data from text files, image files, audio data streams).

Integration of Data Sources

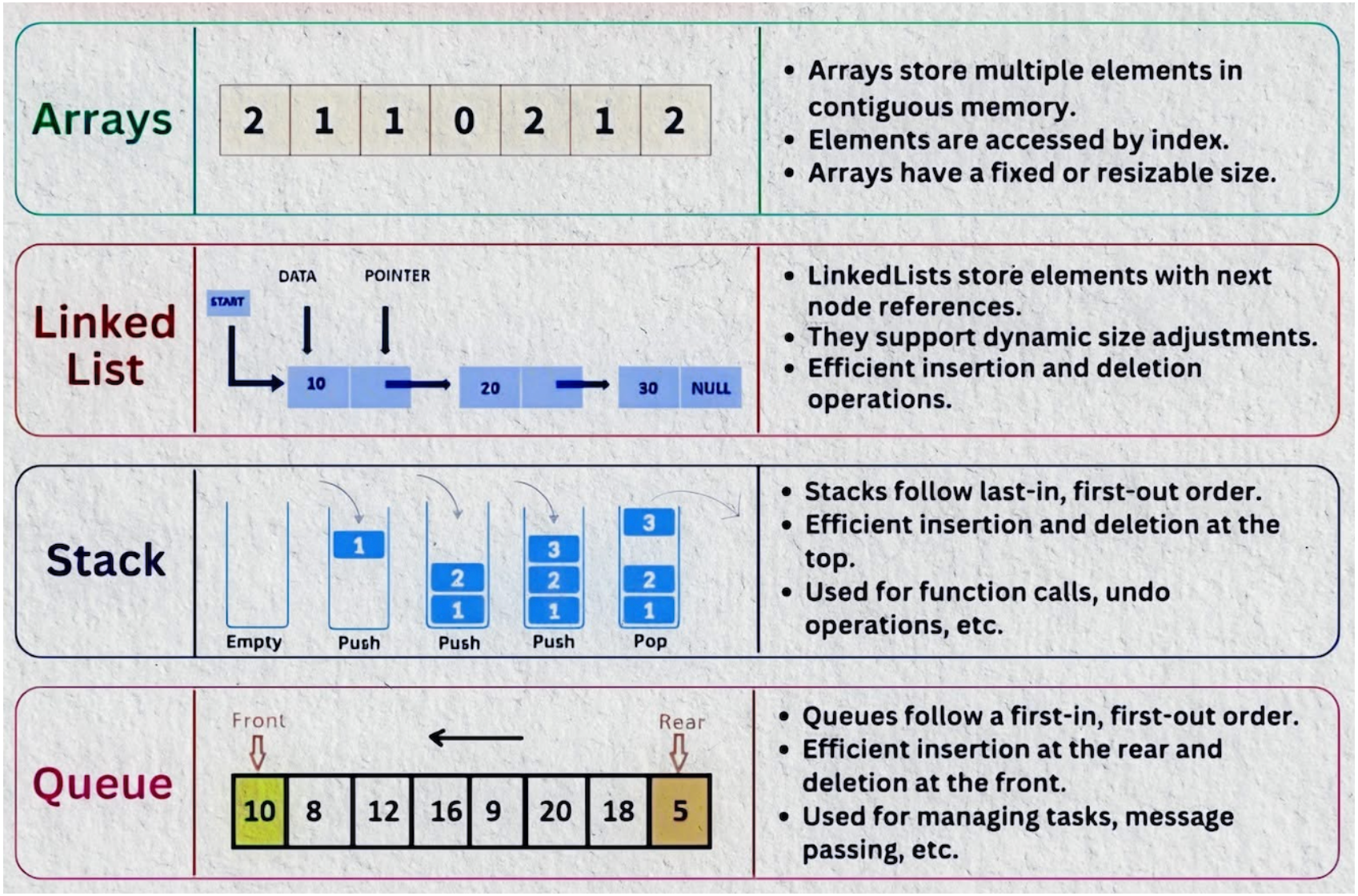
Combinations of homogeneous/heterogeneous data sources and their integration arise from a growing need to analyze diverse data from multiple perspectives (e.g., think eyes and ears).

Homogeneous and Heterogeneous Data

Example 1: Homogeneous/Heterogeneous Data in Python ...

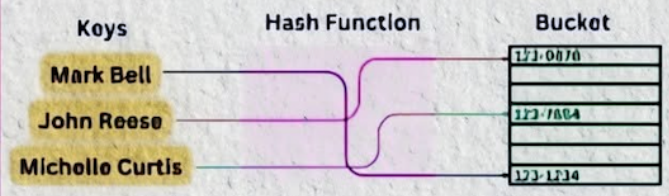
```
# Part 1: Array of Integers (homogeneous data) ...

  array01 = np.array( [ 0, 2, 4, 6, 8 ] )

# Part 2: Matrix of Integers (homogeneous data) ...

  array02 = np.array( ( [ 0, 1, 2, 3,  4 ],
                        [ 2, 4, 6, 8, 10 ] ) );

# Part 3: List of floating-point numbers (homogeneous data) ...

  list01 = [ 0.0, 1.0, 2.0, 4.0, 6.0 ];

# Part 4: List of ints, floats, booleans, and
#         character strings (heterogeneous data) ...

  list02 = [ 0, 'date', True, 4.0, 6.0 ];
```

Aside: From Data Models to Data Structures



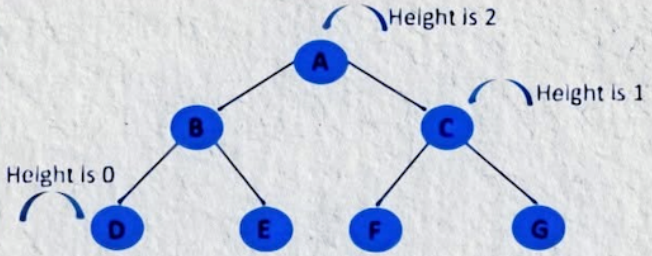
Aside: From Data Models to Data Structures

Hash Table



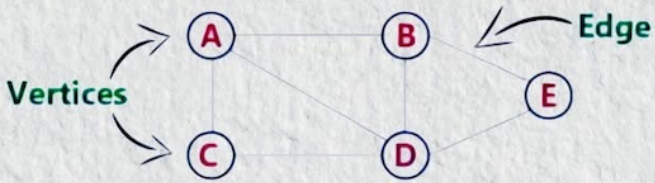
- Hash tables store key-value pairs for quick lookup.
- They use a hash function for indexing.
- Efficient retrieval, insertion, and deletion operations.

Tree



- Trees organize elements in a hierarchical structure.
- Elements have parent and child relationships.
- Used for hierarchy, searching, sorting, etc.

Graph



- Graphs represent relationships between entities.
- They consist of vertices and edges.
- Used for modeling networks, social connections, etc.

Tabular Data and Dataset Transformation

(Working with Pandas)

Working with Pandas

Introduction to Pandas

Pandas (derived from "panel data") is an open source Python Library that supports working and analysis of tabular data sets.

Benefits:

- Pandas can clean messy data sets, and make them readable and relevant.
- Pandas allows us to analyze large data sets and make conclusions based on statistical theories.
- Three data structures: Series, DataFrame and Panel.

Installation:

```
prompt >> pip3 install pandas
```

What can Pandas do?

Basic Operations:

- Create series and dataframes.
- Read CSV and JSON files.
- Plot data.

Clean Data:

- Clean empty cells.
- Fix wrong format.
- Remove duplicates.

Advanced Operations:

- Combine (concatenate, join, merge) Panda objects.
- Compute correlations.

Basic Operations

(Introduction to DataSeries and DataFrames)

Panda Series and DataFrames

Panda Series

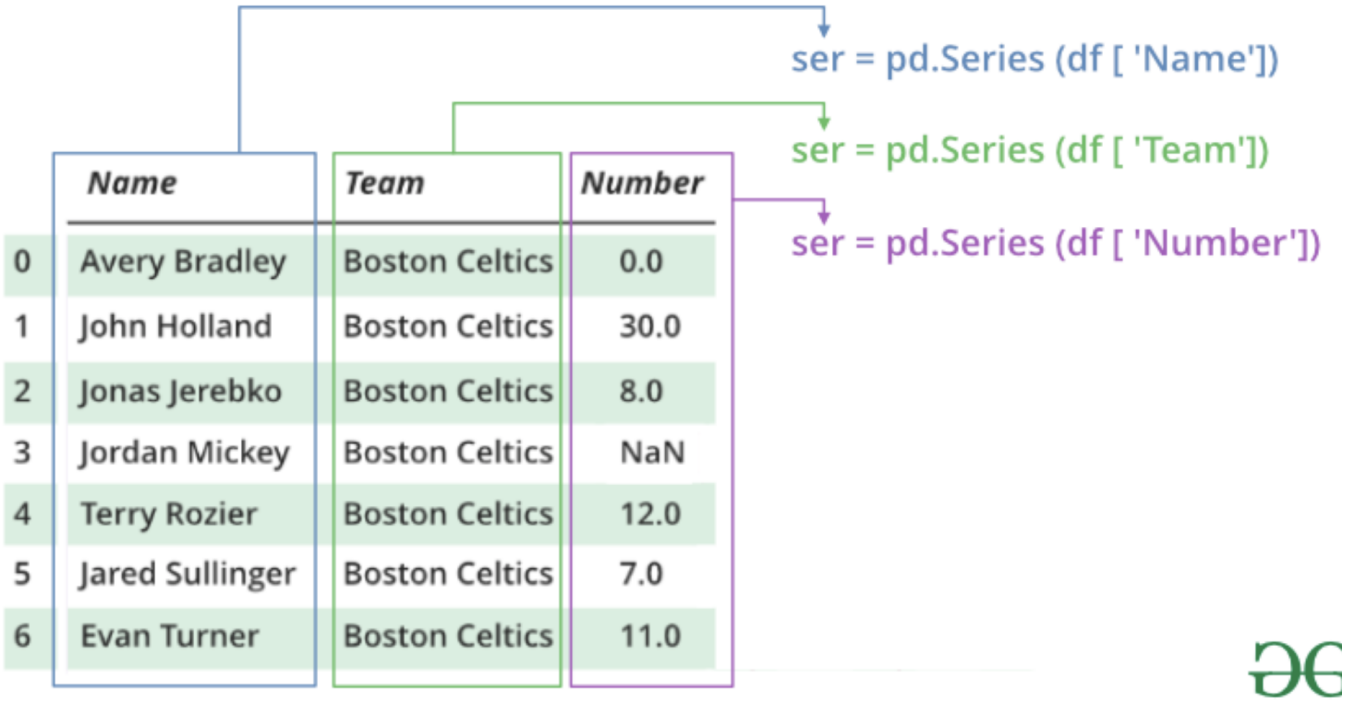
A Panda Series is a one-dimensional ... labeled array capable of holding data of any type (integer, string, float, python objects, etc.).

Panda DataFrame

A Panda DataFrame is a two-dimensional (potentially heterogeneous) tabular data structure with labeled axes for the rows and columns.

Panda Series

Panda Series Elements: columns, data ...



	Name	Team	Number
0	Avery Bradley	Boston Celtics	0.0
1	John Holland	Boston Celtics	30.0
2	Jonas Jerebko	Boston Celtics	8.0
3	Jordan Mickey	Boston Celtics	NaN
4	Terry Rozier	Boston Celtics	12.0
5	Jared Sullinger	Boston Celtics	7.0
6	Evan Turner	Boston Celtics	11.0

Basic Operations:

- Create a series; access elements; index and select data; binary operations; conversion operations.

Panda Series

Example 1: Manually create series from list:

```
# Part 1: Manually create series ...

a = [1, 2, 3, 4, 3, 2, 1 ]
myvar = pd.Series(a)
print(myvar)

# Part 2: Create series from a list with labels ...

myvar = pd.Series(a, index = ["a", "b", "c", "d", "c", "b", "a" ])
print(myvar)
```

Abbreviated Output: Parts 1 and 2 ...

```
Part 01
0    1
1    2
.....
5    2
6    1
dtype: int64
```

```
Part 02
a    1
b    2
.....
b    2
a    1
dtype: int64
```

Panda Series

Example 2: Manually create series from dictionary:

```
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

Output:

```
day1    420
day2    380
day3    390
dtype: int64
```

Panda Series

Example 3: Create series from NumPy functions

```
# series01 = pd.Series(np.arange(2,8)) ... ");

series01 = pd.Series(np.arange(2,8))
print(series01)
```

Output:

```
0    2
1    3
2    4
3    5
4    6
5    7
dtype: int64
```

Panda Series

Example 4: Create series from NumPy functions

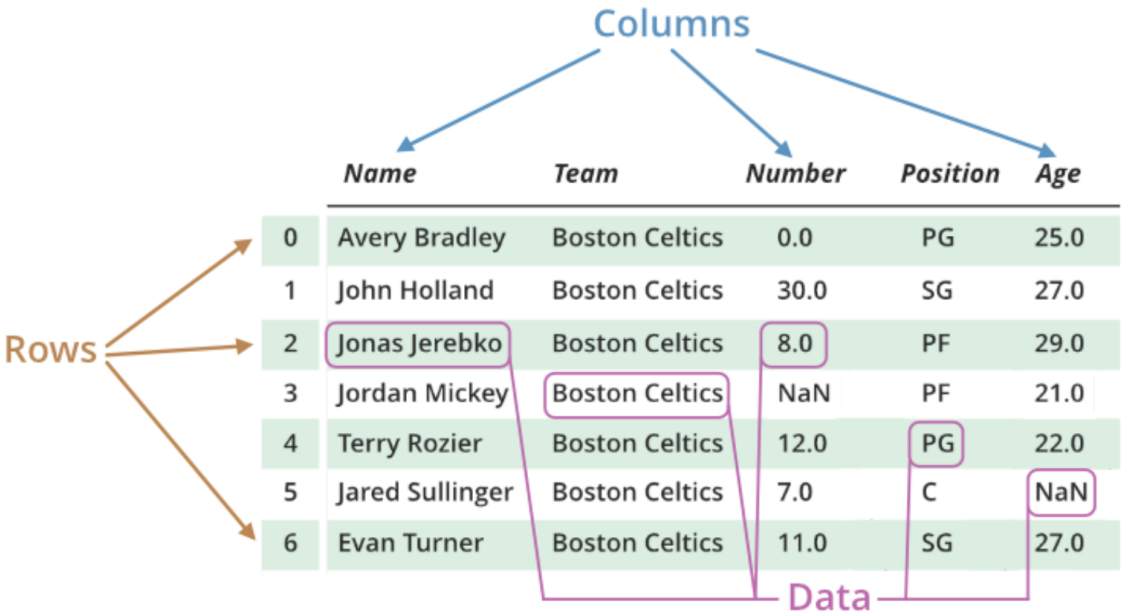
```
series02 = pd.Series( np.linspace(0,10,5) )
print(series02)

print( series02.size)
print( len(series02) )
print( series02.values )
```

Output:

```
0     0.0
1     2.5
2     5.0
3     7.5
4    10.0
dtype: float64

5                           # <-- series02.size ...
5                           # <-- series02 length ...
[ 0.   2.5  5.   7.5 10. ] # <-- series02 values ...
```

Panda DataFrames

Panda DataFrame Elements: rows, columns, data ...



	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Basic Operations:

- Create dataframe; deal with rows and columns; index and select data; iterate over rows and columns.

Working with Panda DataFrames

Data Loading:

```
import pandas as pd
df = pd.read_csv("datafile.csv")
```

Data Exploration and Inspection:

- `df.head()`: — inspect the first few rows of the dataframe.
- `df.describe()`: — generate descriptive statistics.
- `df.isnull().sum()`: — identify missing values.

Working with Panda DataFrames

Example 1: Manually create small dataset ...

```
mydataset = {
   'cars': [ "BMW", "Honda", "Acura"],
   'year': [ 2013,     2017,    2022]
}

myvar = pd.DataFrame(mydataset)
print(myvar)
```

Output:

```
     cars  year
0     BMW  2013
1   Honda  2017
2   Acura  2022
```

Working with Panda DataFrames

Example 2: Create dataframes from 1-d and 2-d arrays ...

```
myvar = pd.DataFrame( np.arange(1,8) )       # <-- dataframe from 1-d array
print(myvar)

df = pd.DataFrame( [ [1,2], [3,4], [5,6] ] ) # <-- dataframe from 2-d array
print(df)
```

Abbreviated Output:

```
Dataframe from 1-d np array
---------------------------
   0
0  1
1  2
2  3
....
5  6
6  7
```

```
Dataframe from 2-d np array
---------------------------
      0  1
   0  1  2
   1  3  4
   2  5  6
```

Working with Panda DataFrames

Example 3: Create simple dataframe from multiple series ...

```
data = {                                # <-- Create dataframe from
   "calories": [520, 480, 400],         #     multiple series.
   "duration": [ 50,  48,  40]
}

myvar = pd.DataFrame(data)
print(myvar)

index = ["day1", "day2", "day3"]  # <-- give each ros a new name.
myvar = pd.DataFrame(data, index)
print(myvar)
```

Output:

```
Part 1: dataframe from series
   calories  duration
0       520        50
1       480        48
2       400        40
```

```
Part 2: rename rows
      calories  duration
day1       520        50
day2       480        48
day3       400        40
```

Working with Panda DataFrames

Example 4: Create dataframe from JSON object ...

```
# Create JSON object (same format as Python dictionary) ...

data = {
  "Duration":{ "0":60,  "1":60, "2":60,   "3":45,  "4":45,  "5":60 },
  "Pulse":{    "0":110, "1":117, "2":103, "3":109, "4":117, "5":102 },
  "Maxpulse":{ "0":130, "1":145, "2":135, "3":175, "4":148, "5":127 },
  "Calories":{ "0":409, "1":479, "2":340, "3":282, "4":406, "5":300 }
}

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

Output:

```
   Duration  Pulse  Maxpulse  Calories
0        60    110       130       409
1        60    117       145       479
2        60    103       135       340
3        45    109       175       282
4        45    117       148       406
5        60    102       127       300
```

Working with Panda DataFrames

Example 5: Select rows and columns from dataframe ...

```
# Select columns of a dataframe ...

print( df[ [ 'Duration','Calories'] ].head() )

# Selecting rows of a dataframe ...

print( df.loc['1'].head() )     # <-- extract and print row 1
print( df.loc['2'].head() )     # <-- extract and print row 2
```

Output:

```
Columns of dataframe
--------------------

   Duration  Calories
0        60       409
1        60       479
2        60       340
3        45       282
4        45       406

Row 1
-----------------

Duration     60
Pulse       117
Maxpulse    145
Calories    479
Name: 1, dtype: int64

Row 2
-----------------

Duration     60
Pulse       103
Maxpulse    135
Calories    340
Name: 2, dtype: int64
```

Working with Pandas

Example 6: Read and plot CSV data file.

```
df = pd.read_csv('../data/AirPassengers.csv')
print(df.head())

print(df.info())  # <-- print dataframe info and shape ...
print(df.shape)
```

Output:

```
     Month  #Passengers
0  1949-01          112
1  1949-02          118
2  1949-03          132
3  1949-04          129
4  1949-05          121
```

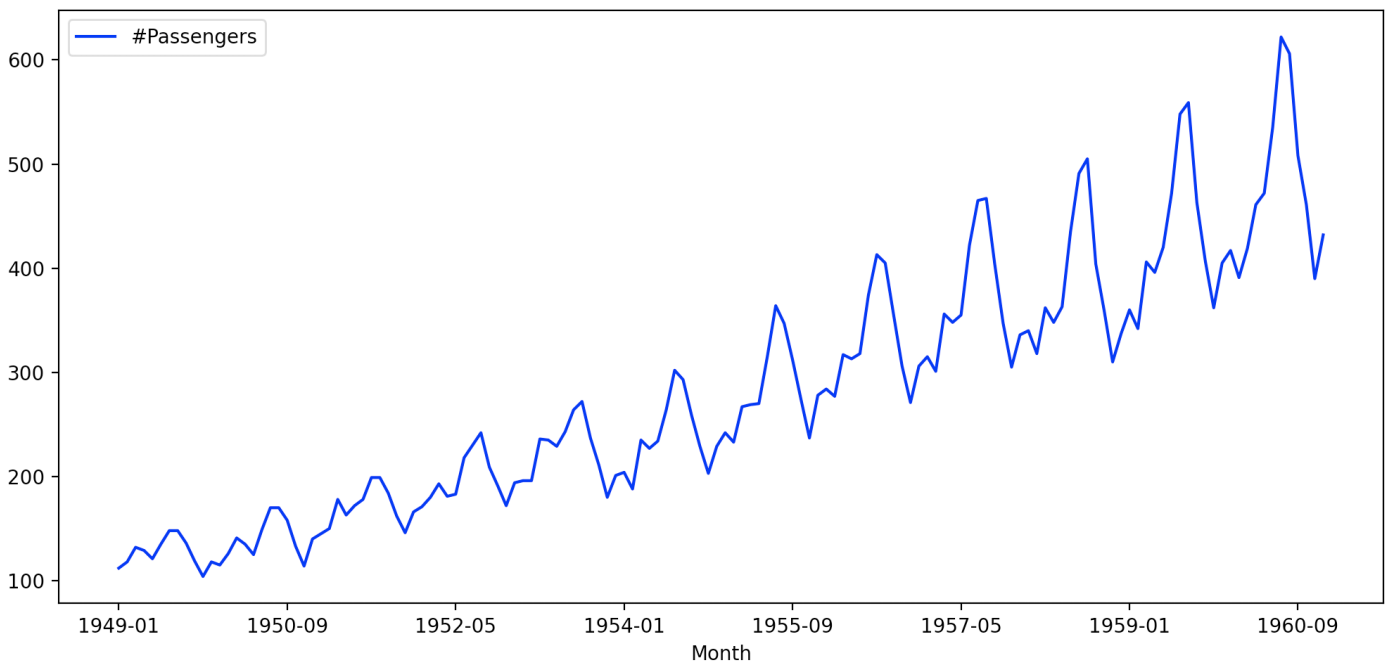
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 2 columns):
#   Column       Non-Null Count  Dtype
---  ------       --------------  -----
0   Month        144 non-null    object
1   #Passengers  144 non-null    int64
dtypes: int64(1), object(1)
memory usage: 2.4+ KB
None
(144, 2)
```

Working with Pandas

Example 6: (continued)

```
import matplotlib.pyplot as plt

ax = plt.gca()
df.plot(kind='line',x='Month',y='#Passengers',color='blue',ax=ax)
plt.show()
```

Output:



Intermediate

Operations

(Data Cleaning, Data Transformation, etc ...)

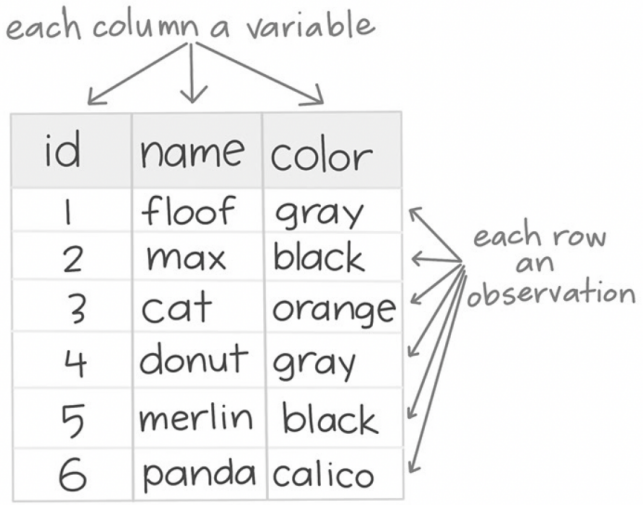
Tidy and Messy Data

Data Cleaning

Data cleaning involves identification and correction of errors, inconsistencies, and missing values in a dataset.

Tidy Data

- Each variable forms a column.
- Each observation forms a row.
- Each cell is a single measurement.



Source: Wickham H. (2014), Tidy Data, Journal Statistical Software, Vol. 59. No 10.

Tidy and Messy Data

Messy Tabular Data

- Incomplete/missing data.
- Data with spelling mistakes.
- Data with ambiguous wording.
- Data misclassified (wrong column).
- Inconsistent formatting.
- Data with anomalies.
- Data is too noisy.
- Data duplication.
- Data is completely outdated.

Data Cleaning Functions

Handling Missing Values:

- `dropna()`: Remove rows or columns with missing values.
- `fillna()`: Fill missing values with a specified value or moethod (e.g., median, mean).
- `isna()` or `isnull()`: Detect missing values and return a boolean mask.
- `replace()`: Replace missing values with a specific value.

Removing Duplicates:

- `duplicated()`: Detects duplicate rows, returning a boolean mask.
- `drop_duplicates()`: Removes duplicate rows.

Data Cleaning Tasks

Data Type Conversions:

- `astype()`: Converts columns to a different datatype.
- `to_numeric()`: Converts columns to numeric types.
- `to_datetime()`: Converts columns to datetime objects.

Removing Irrelevant Columns:

- `drop()`: Remove specified columns.

Renaming Columns:

- `rename()`: Renames columns to meaningful names.

Data Reformatting:

- `apply()`: Ensure consistent formatting for dates, numbers and strings, and standardized categorical variables.

Example 1: Basic Data Cleaning

Create dictionary of data:

```
Python: d01 = { 'ID': [123, 251, 301, 444, 649, 256, 649],
-------       'Name': ['Andrea','Mark','Angela','Nina','Eve','Joe','Eve'],
               'Age': [ 45, 45, None, 29, 52, 60, 52 ],
              'City': ['Vancouver','DC','Sydney','NYC', None,'Denver', None],
           'Country': ['Canada','USA','Australia','USA', None,'USA', None],
            'Salary': [ 60000, 80000, None, 45000, np.nan, 180000, np.nan] }

        df01 = pd.DataFrame(d01)
```

Output:

	ID	Name	Age	City	Country	Salary
0	123	Andrea	45.0	Vancouver	Canada	60000.0
1	251	Mark	45.0	DC	USA	80000.0
2	301	Angela	NaN	Sydney	Australia	NaN
3	444	Nina	29.0	NYC	USA	45000.0
4	649	Eve	52.0	None	None	NaN
5	256	Joe	60.0	Denver	USA	180000.0
6	649	Eve	52.0	None	None	NaN

Example 1: Basic Data Cleaning

Fill missing cities with 'Unknown':

```
Python:     df01['City'].fillna('Unknown', inplace=True)
-------     print(df01)
```

Output:

	ID	Name	Age	City	Country	Salary
0	123	Andrea	45.0	Vancouver	Canada	60000.0
1	251	Mark	45.0	DC	USA	80000.0
2	301	Angela	NaN	Sydney	Australia	NaN
3	444	Nina	29.0	NYC	USA	45000.0
4	649	Eve	52.0	Unknown	None	NaN
5	256	Joe	60.0	Denver	USA	180000.0
6	649	Eve	52.0	Unknown	None	NaN

Example 1: Data Cleaning

Fill missing ages with mean value:

```
Python:    df01['Age'].fillna( df01['Age'].mean(), inplace=True)
-------
```

```
Output:
-------
```

	ID	Name	Age	City	Country	Salary
0	123	Andrea	45.000000	Vancouver	Canada	60000.0
1	251	Mark	45.000000	DC	USA	80000.0
2	301	Angela	47.166667	Sydney	Australia	NaN
3	444	Nina	29.000000	NYC	USA	45000.0
4	649	Eve	52.000000	Unknown	None	NaN
5	256	Joe	60.000000	Denver	USA	180000.0
6	649	Eve	52.000000	Unknown	None	NaN

Example 1: Basic Data Cleaning

Convert age data type to int:

```
Python:    df01['Age'] = df01['Age'].astype(int)
-------
```

```
Output:
-------
```

	ID	Name	Age	City	Country	Salary
0	123	Andrea	45	Vancouver	Canada	60000.0
1	251	Mark	45	DC	USA	80000.0
2	301	Angela	47	Sydney	Australia	NaN
3	444	Nina	29	NYC	USA	45000.0
4	649	Eve	52	Unknown	None	NaN
5	256	Joe	60	Denver	USA	180000.0
6	649	Eve	52	Unknown	None	NaN

Example 1: Basic Data Cleaning

Remove duplicate rows:

```
Python:    df01.drop_duplicates(inplace=True)
-------
```

Output:

	ID	Name	Age	City	Country	Salary
0	123	Andrea	45	Vancouver	Canada	60000.0
1	251	Mark	45	DC	USA	80000.0
2	301	Angela	47	Sydney	Australia	NaN
3	444	Nina	29	NYC	USA	45000.0
4	649	Eve	52	Unknown	None	NaN
5	256	Joe	60	Denver	USA	180000.0

Remove rows with missing salaries:

```
Python:    df01.dropna(subset=['Salary'], inplace=True)
-------
```

Output:

	ID	Name	Age	City	Country	Salary
0	123	Andrea	45	Vancouver	Canada	60000.0
1	251	Mark	45	DC	USA	80000.0
3	444	Nina	29	NYC	USA	45000.0
5	256	Joe	60	Denver	USA	180000.0

Example 2: Data Transformation with apply()

Reformat: Sample dataset and dataframe ...

```
# Sample dataframe ...

data = { 'A': [  0,  1,  2,  3 ],
         'B': [  4,  5,  6,  7 ],
         'C': [  8,  9, 10, 11 ],
         'D': [ 12, 13, 14, 15 ] }
df = pd.DataFrame(data)
print(df)
```

```
# Output ...

    A   B   C   D
0   0   4   8  12
1   1   5   9  13
2   2   6  10  14
3   3   7  11  15
```

Apply square function:

```
# Square function ...

def square(x):
    return x * x

# Apply square function ...

  df_squared = df.apply(square)
  print(df_squared)
```

```
# Output ...

    A   B   C   D
0   0  16  64  144
1   1  25  81  169
2   4  36  100 196
3   9  49  121 225
```

Example 2: Data Transformation with apply()

Reformat: Sum row and column elements ...

```
df1 = df.apply(np.sum, axis=0)  # <--- sum column elements ...
print(df1)

df1 = df.apply(np.sum, axis=1)  # <--- sum row elements ...
print(df1)
```

Output:

```
A     6
B    22
C    38
D    54

dtype: int64
0    24
1    28
2    32
3    36
dtype: int64
```

Example 2: Data Transformation with apply()

Define custom function:

```
def custom_function( x, low, high ):
    return low <= x <= high;
```

Apply custom function:

```
    data   = pd.Series( [ 1, 3, 5, 7, 9, 11, 13 ] )
    result = data.apply( custom_function, low=5, high=9 )
    print( result )
```

Output:

```
0    False
1    False
2     True
3     True
4     True
5    False
6    False
dtype: bool
```

Advanced Operations

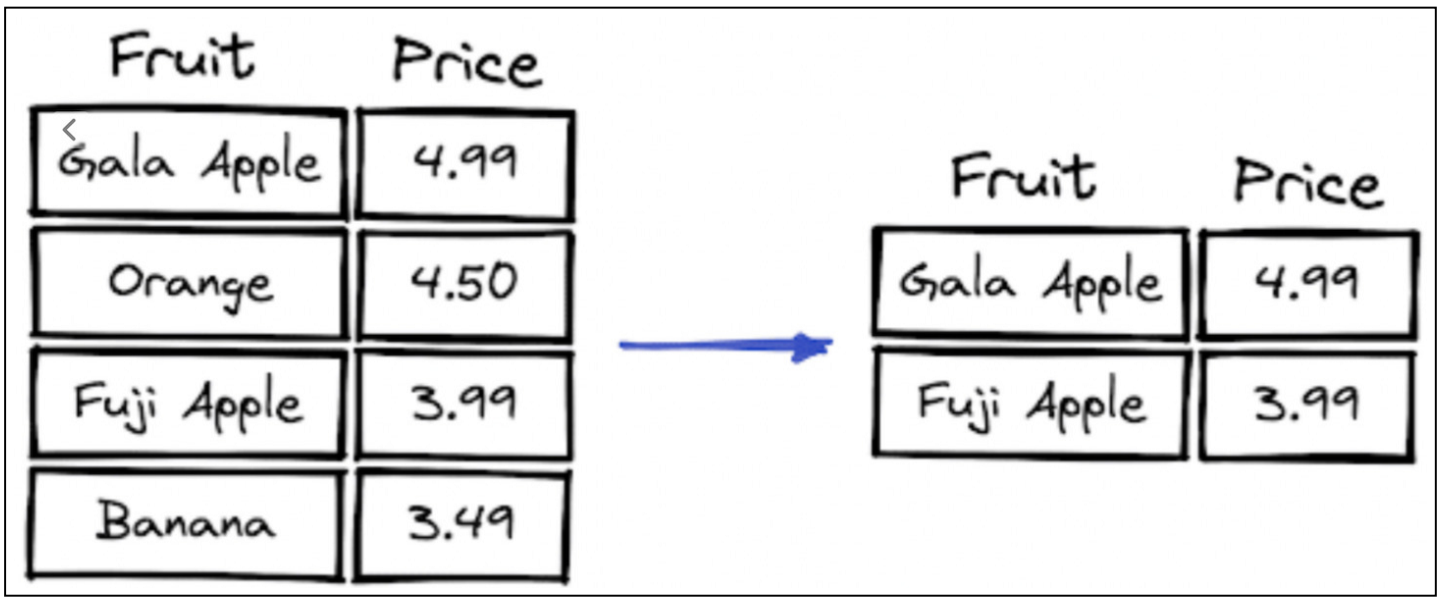
(Data Filtering, Data Merge)

Data Filtering

Data Filtering Data

Data filtering in Python involves selection of subsets (i.e., rows and columns) of data based upon certain conditions.

Example: Just get me the apples ...



Four Strategies for Basic Filtering

Filter on a Single Column:

- Use operators – equality operator, range of values in a column – to filter contents of a single column.

Filtering with Multiple Columns:

- Use multiple values – for example a range – to filter contents of a single column.

Filtering based on a List of Values:

- Use list of values to filter on a single column.

String Filtering:

- Filter rows based on contents of string functions, e.g., `.str.startswith()` and `.str.contains()`.

Example 1: Basic Data Filtering

Sample Dataset: Cities, states, politics, population

Dataset:

```
data = {'City': ['Washington DC', 'Los Angeles', ... 'Austin', 'Baltimore'],
        'State': [ 'Null', 'CA', 'CO', 'WA', 'CA', 'TX', 'TX', 'MD'],
        'Politics': [ 'Blue', 'Blue', 'Red', 'Blue', ... 'Blue', 'Blue'],
        'Population': [ 10, 30, 20, 15, 20, 15, 10, 10 ] }

df = pd.DataFrame(data)
```

Output:

	City	State	Politics	Population
0	Washington DC	Null	Blue	10
1	Los Angeles	CA	Blue	30
2	Denver	CO	Red	20
3	Seattle	WA	Blue	15
4	San Francisco	CA	Blue	20
5	Dallas	TX	Red	15
6	Austin	TX	Blue	10
7	Baltimore	MD	Blue	10

Example 1: Basic Data Filtering

Filter rows based on numeric value (pop $\geq$ 20)

```
Filter:     filtered_pop = df[df['Population'] >= 20]
-------
```

```
Output:                City State Politics  Population
-------     1    Los Angeles    CA     Blue          30
            2         Denver    CO      Red          20
            4  San Francisco    CA     Blue          20
```

Filter rows multiple conditions (pop $>$ 25) and (state $==$ 'CA')

```
Filter:     filter_multi = df[(df['Population'] > 25) & (df['State'] == 'CA')]
-------
```

```
Output:              City State Politics  Population
-------     1  Los Angeles    CA     Blue          30
```

Example 1: Basic Data Filtering

Filter rows based on list of states (states = ['CA', 'TX'])

```
Filter:     states_to_include  = ['CA', 'TX']
-------     filtered_states = df[df['State'].isin( states_to_include)]

Output:                 City State Politics  Population
-------     1    Los Angeles    CA     Blue          30
            4  San Francisco    CA     Blue          20
            5         Dallas    TX      Red          15
            6         Austin    TX     Blue          10
```

Filter state names starts with a 'C' ...

```
Filter:     filter_by_name01 = df[ df['State'].str.startswith('C') ]
-------     print( filter_by_name01 )

Output:                 City State Politics  Population
-------     1    Los Angeles    CA     Blue          30
            2         Denver    CO      Red          20
            4  San Francisco    CA     Blue          20
```

Example 1: Basic Data Filtering

Filter booleans matching (politics == 'Red')

```
Filter:         filtered_politics = df['Politics'].eq('Red')
-------

Output:         0    False
-------         1    False
                2     True
                3    False
                4    False
                5     True
                6    False
                7    False
                Name: Politics, dtype: bool
```

Source Code: See: python-code.d/pandas/

Customized Filtering with filter()

Function filter():

- `filter()`: Filters subset of a dataframe based upon specified index labels.
- Filter rows and columns based upon the `axis` argument.

Parameters:

- **items:** list-like. Keep labels from axis which are in items.
- **like:** str. Keep labels from axis for which `"like in label == True"`.
- **regex:** Keep labels from axis for which `re.search(regex,label) == True`.
- **axis:** Indicates the axis to filter on – select rows with `axis=0`, select columns with `axis=1`.

Example 2: Customized Filtering with filter()

Create dictionary of data:

Python:

```
data01 = {'ID': [123, 251, 301, 444, 649, 256, 649],
          'Name': ['Andrea', 'Nicol', 'Angela', 'Nina', ... , 'Eve'],
          'Age': [45, 45, None, 29, 18, 32, 18],
          'City': [ 'Toronto', 'Auckland', 'Sydney', ... ,None],
          'Country': [ 'Canada', 'NZ', 'Australia', .... 'Australia'
          'Salary': [ 60000, ... 45000, np.nan, 180000, np.nan]}

index = [ "friend", "family", "family", "friend", "student", "work",
df01  = pd.DataFrame(data01, index )
print(df01)
```

Output:

	ID	Name	Age	City	Country	Salary
friend	123	Andrea	45.0	Toronto	Canada	60000.0
family	251	Nicol	45.0	Auckland	NZ	80000.0
family	301	Angela	NaN	Sydney	Australia	NaN
friend	444	Nina	29.0	NYC	USA	45000.0
student	649	Eve	18.0	None	Australia	NaN
work	256	Joe	32.0	Berkeley	USA	180000.0
student	649	Eve	18.0	None	Australia	NaN

Example 2: Customized Filtering with filter()

Select columns by Name and City:

```
Python:     filtered_columns = df01.filter( [ "Name", "City" ] );
-------     print( filtered_columns )
```

```
Output:                Name      City
-------     friend    Andrea   Toronto
            family     Nicol  Auckland
            family    Angela    Sydney
            friend      Nina       NYC
            student      Eve       None
            work         Joe  Berkeley
            student      Eve       None
```

Filter data by row (index equal to family):

```
Python:       filter_family = df01.filter( like='fam', axis = 0 )
```

```
Output:                 ID    Name   Age      City    Country   Salary
-------     family  251   Nicol  45.0  Auckland         NZ  80000.0
            family  301  Angela   NaN    Sydney  Australia      NaN
```

Data Merge and Concatenation

Data Merge

Data merge operations combine data frames based on columns or indices. Similar to structured query language (SQL) operations.

Function: `merge()`

- Combines dataframes based on common columns or indices – by default, `merge()` keeps only rows with matching keys in both dataframes.

Four types of join:

- `inner` returns only matching rows.
- `left` returns all rows from the left dataframe and matching rows from the right dataframe.

Data Merge and Concatenation

- `right` returns all rows from the right dataframe and matching rows from the left dataframe.
- `outer` returns all rows from both dataframes.

Modifications:

- `on` parameter specifies the columns to join on.
- `left_on` and `right_on` can be used when column names are different in dataframes.

Function: concat()

- Concatenates dataframes either vertically (axis = 0) or horizontally (axis = 1).

Function: join()

- By default, joins dataframes based on their index.

Merging and Concatenating Data

Functions: concat(), append() and merge() ...

concat, append

Table A

User ID	Balance
100	3000
101	200

Table B

User ID	Balance
200	100
201	500

```
t1 = pandas.DataFrame([[100, 3000], [101, 200]])
t2 = pandas.DataFrame([[200, 100], [201, 500]])

pandas.concat([t1, t2])

t1.append(t2)
```

result

User ID	Balance
100	3000
101	200
200	100
201	500

merge

Table C

User ID	User Name
100	John
101	Annie

Table D

User ID	Balance
100	100
101	500

```
t3 = pandas.DataFrame({'key': [100, 101],
                       'name': ['John', 'Annie']})
t4 = pandas.DataFrame({'key': [100, 101],
                       'balance': [100, 500]})

t3.merge(t4)
```

result

User ID	User Name	Balance
100	John	100
101	Annie	500

Merging and Concatenating Data

Functions: join() and combine() ...

join

Table E

User ID	User Name
100	John
101	Annie
102	Joe

Table F

User ID	Balance
100	100
101	500

```
t5 = pandas.DataFrame({'key': [100, 101, 102],
                       'name': ['John', 'Annie', 'Joe']})
t6 = pandas.DataFrame({'key': [100, 101],
                       'balance': [100, 500]})

t5.set_index('key').join(t6.set_index('key'))
```

result

User ID	User Name	Balance
100	John	100
101	Annie	500
102	Joe	NaN

combine

Table G

User ID	Balance
100	3000
101	200

Table H

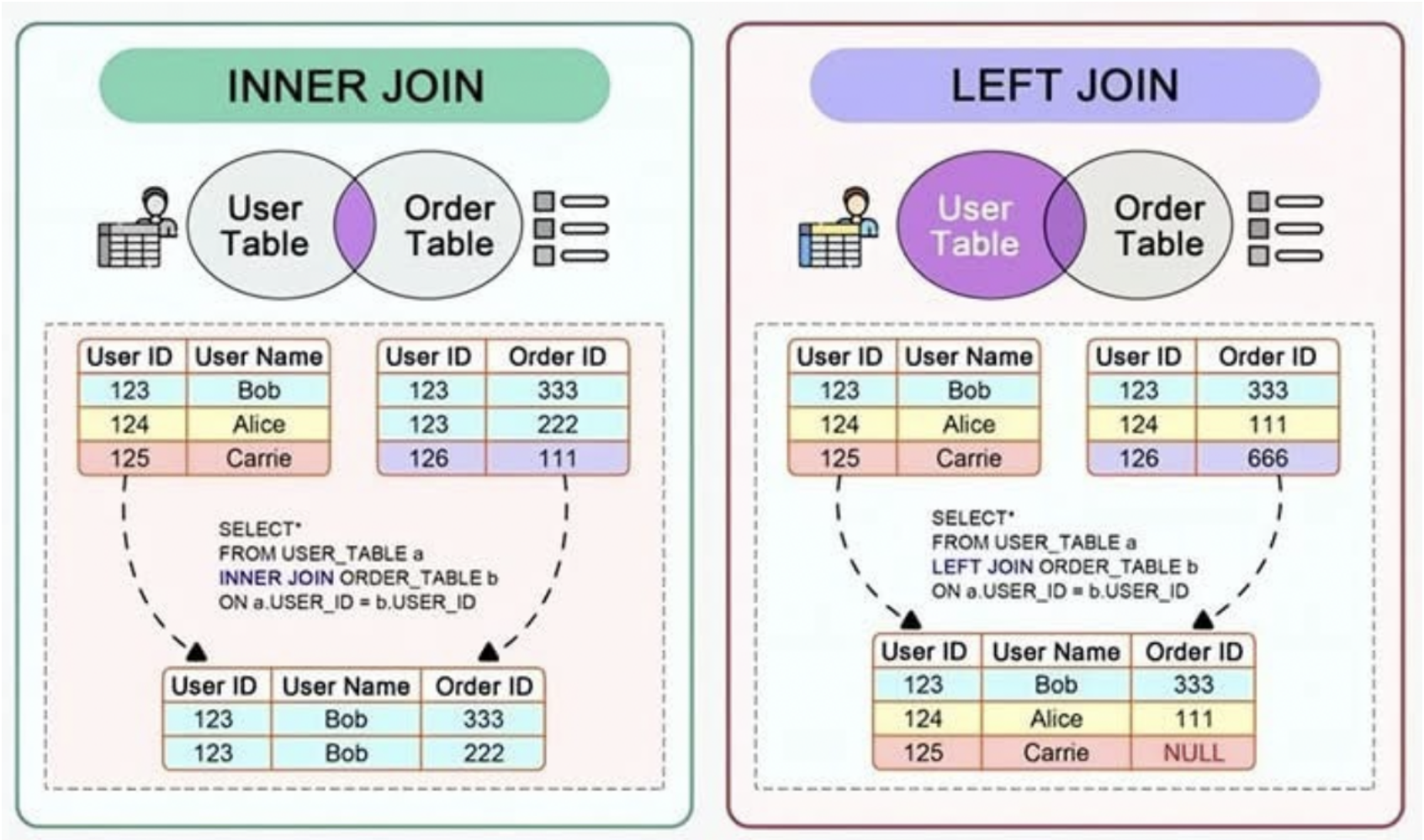
User ID	Balance
100	100
101	500

```
t7 = pandas.DataFrame({'key': [100, 101],
                       'balance': [3000, 200]})
t8 = pandas.DataFrame({'key': [100, 101],
                       'balance': [100, 500]})

choose_smaller = lambda x,y: x if x.sum() < y.sum() else y

t7.combine(t8, choose_smaller)
```

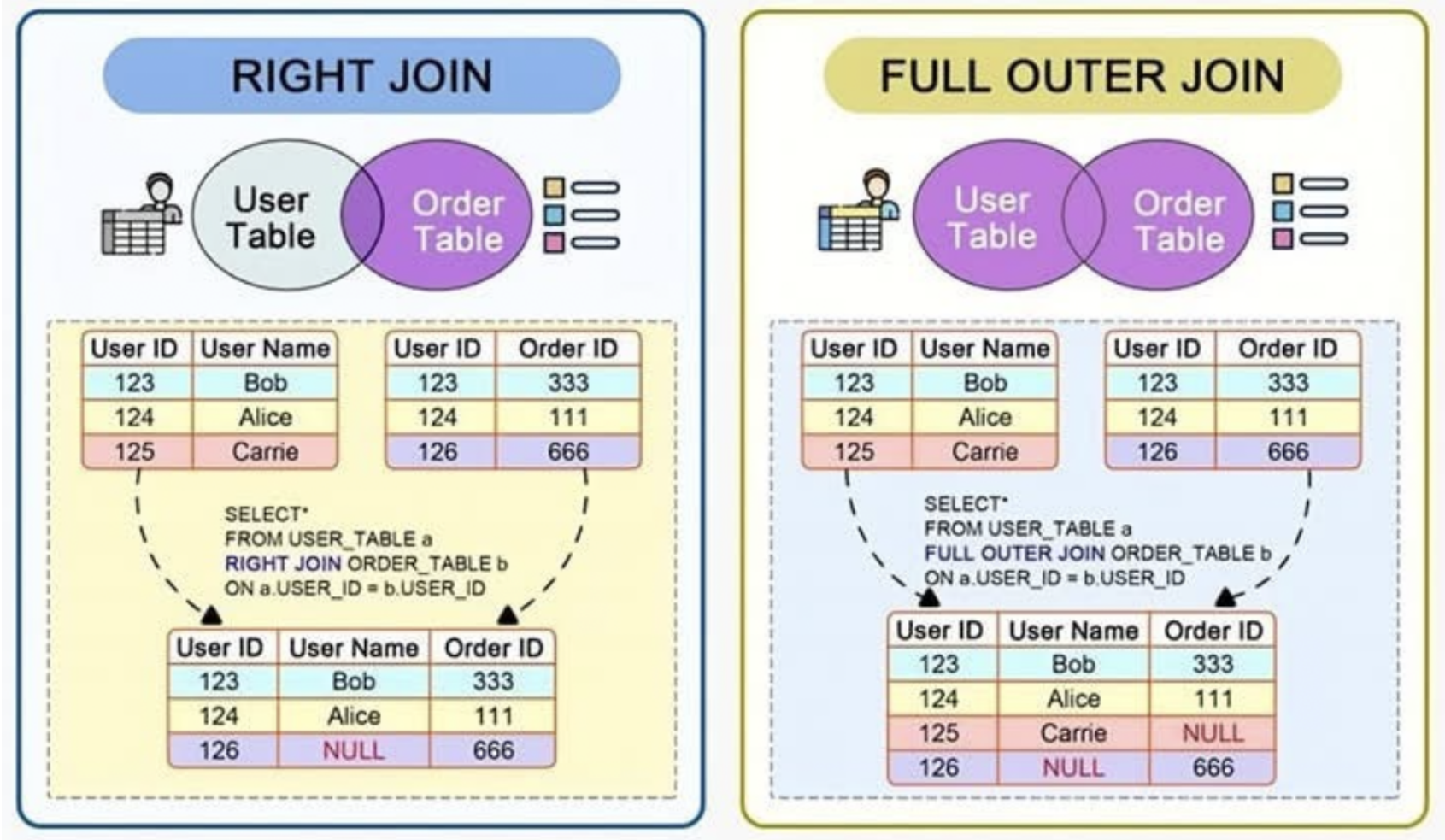
result

User ID	Balance
100	100
101	200

Structured Query Language (SQL) Joins



Structured Query Language (SQL) Joins



Example 1: Merging Dataframes

Sample Dataset: Students and Courses

Dataset 1: Students

	Student ID	Name	Gender	GPA	Institution
0	A1	Angela	F	4.0	MBHS
1	A2	Anne	F	3.6	RHHS
2	A3	Alex	M	3.0	UMCP
3	A4	Nina	F	3.2	CNHS
4	A5	Shirley	F	2.7	UMCP
5	A6	David	M	4.0	UMCP
6	A7	Alex	M	3.5	UMBC

Dataset 2: Courses

	Student ID	Name	School	Class	Grade
0	A1	Angela	UMCP	Math 201	A+
1	A3	Alex	UMCP	ENCE 201	B-
2	A7	Alex	UMCP	ENCE 353	B+
3	A2	Anne	UMCP	Math 201	A-
4	A2	Anne	UMCP	Math 215	B+

Example 1: Merging Dataframes

Merge Tables (inner merge, default)

```
Python:   inner_merge = pd.merge( students01, classes01,
-------                           on='Student ID')
          print( inner_merge)

Output:
-------
```

	Student ID	Name_x	Gender	GPA	Institution	Name_y	School	Class	Grade
0	A1	Angela	F	4.0	MBHS	Angela	UMCP	Math 201	A+
1	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 201	A-
2	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 215	B+
3	A3	Alex	M	3.0	UMCP	Alex	UMCP	ENCE 201	B-
4	A7	Alex	M	3.5	UMBC	Alex	UMCP	ENCE 353	B+

Example 1: Merging Dataframes

Merge Tables (left merge)

```
Python:  left_merge = pd.merge( students01, classes01,
-------                         on='Student ID', how='left' )
         print(left_merge)

Output:
-------
```

	Student ID	Name_x	Gender	GPA	Institution	Name_y	School	Class	Grade
0	A1	Angela	F	4.0	MBHS	Angela	UMCP	Math 201	A+
1	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 201	A-
2	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 215	B+
3	A3	Alex	M	3.0	UMCP	Alex	UMCP	ENCE 201	B-
4	A4	Nina	F	3.2	CNHS	NaN	NaN	NaN	NaN
5	A5	Shirley	F	2.7	UMCP	NaN	NaN	NaN	NaN
6	A6	David	M	4.0	UMCP	NaN	NaN	NaN	NaN
7	A7	Alex	M	3.5	UMBC	Alex	UMCP	ENCE 353	B+

Example 1: Merging Dataframes

Merge Tables (right merge)

```
Python:     right_merge = pd.merge( students01, classes01,
-------                             on='Student ID', how='right' )
            print( right_merge )

Output:
-------
```

	Student ID	Name_x	Gender	GPA	Institution	Name_y	School	Class	Grade
0	A1	Angela	F	4.0	MBHS	Angela	UMCP	Math 201	A+
1	A3	Alex	M	3.0	UMCP	Alex	UMCP	ENCE 201	B-
2	A7	Alex	M	3.5	UMBC	Alex	UMCP	ENCE 353	B+
3	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 201	A-
4	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 215	B+

Example 1: Merging Dataframes

Merge Tables (outer merge)

```
Python:     outer_merge = pd.merge( students01, classes01,
-------                             on='Student ID', how='outer' )
            print( outer_merge )
```

Output:

	Student ID	Name_x	Gender	GPA	Institution	Name_y	School	Class	Grade
0	A1	Angela	F	4.0	MBHS	Angela	UMCP	Math 201	A+
1	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 201	A-
2	A2	Anne	F	3.6	RHHS	Anne	UMCP	Math 215	B+
3	A3	Alex	M	3.0	UMCP	Alex	UMCP	ENCE 201	B-
4	A4	Nina	F	3.2	CNHS	NaN	NaN	NaN	NaN
5	A5	Shirley	F	2.7	UMCP	NaN	NaN	NaN	NaN
6	A6	David	M	4.0	UMCP	NaN	NaN	NaN	NaN
7	A7	Alex	M	3.5	UMBC	Alex	UMCP	ENCE 353	B+