

Introduction to Pandas: Data Analysis Made Easy

a powerful Python library for data manipulation and analysis.

Why Use Pandas?

Pandas is well suited for:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

Key features:

- Easy handling of **missing data**
- **Size mutability**: columns can be inserted and deleted from DataFrame and higher dimensional objects
- Automatic and explicit **data alignment**: objects can be explicitly aligned to a set of labels, or the data can be aligned automatically
- Powerful, flexible **group by functionality** to perform split-apply-combine operations on data sets
- Intelligent label-based **slicing, fancy indexing, and subsetting** of large data sets
- Intuitive **merging and joining** data sets
- Flexible **reshaping and pivoting** of data sets
- **Hierarchical labeling** of axes
- Robust **IO tools** for loading data from flat files, Excel files, databases, and HDF5
- **Time series functionality**: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

Series: One-Dimensional Labeled Data

- Definition of Series: a one-dimensional labeled array capable of holding any data type.

- Show syntax:

```
import pandas as pd
```

```
data = pd.Series([data], index=[])
```

```
bacteria = pd.Series([632, 1638, 569, 115], index=['Firmicutes', 'Proteobacteria', 'Actinobacteria', 'Bacteroidetes'])
```

```
bacteria
```

```
Firmicutes      632
Proteobacteria  1638
Actinobacteria   569
Bacteroidetes   115
dtype: int64
```

DataFrame: Two-Dimensional Tabular Data

- Definition of DataFrame: a 2D labeled data structure with columns that can hold different data types.
- Show syntax:

```
import pandas as pd  
data = pd.DataFrame(data, columns=columns)
```

A DataFrame has a second index, representing the columns:

```
data = pd.DataFrame({'value':[632, 1638, 569, 115, 433, 1130, 754, 555],  
                     'patient':[1, 1, 1, 1, 2, 2, 2, 2],  
                     'classification':['Firmicutes', 'Proteobacteria', 'Actinobacteria',  
                                     'Bacteroidetes', 'Firmicutes', 'Proteobacteria', 'Actinobacteria', 'Bacteroidetes']})  
data
```

	value	patient	classification
0	632	1	Firmicutes
1	1638	1	Proteobacteria
2	569	1	Actinobacteria
3	115	1	Bacteroidetes
4	433	2	Firmicutes
5	1130	2	Proteobacteria
6	754	2	Actinobacteria
7	555	2	Bacteroidetes

- Operations on DataFrames
 - selecting columns
 - filtering rows
 - basic statistics (transpose a matrix)
 - creates a separate copy of Dataframe
 - Add new columns
 - Remove columns

Loading Data into Pandas

- from CSV
 - Select rows
 - check NA

```
mb = pd.read_csv("data/microbiome.csv")  
mb.head()
```

	Taxon	Patient	Tissue	Stool
0	Firmicutes	1	632	305
1	Firmicutes	2	136	4182
2	Firmicutes	3	1174	703
3	Firmicutes	4	408	3946
4	Firmicutes	5	831	8605

Loading Data into Pandas

- from Excel
 - Choose sheet
- `pd.ExcelFile(...)`
- `pd.read_excel(...)`

```
mb_file = pd.ExcelFile('data/microbiome/MID1.xls')
mb_file
```

```
<pandas.io.excel._base.ExcelFile at 0x7fa83800e460>
```

```
mb1 = mb_file.parse("Sheet 1", header=None)
mb1.columns = ["Taxon", "Count"]
mb1.head()
```

	Taxon	Count
0	Archaea "Crenarchaeota" Thermoprotei Desulfuro...	7
1	Archaea "Crenarchaeota" Thermoprotei Desulfuro...	2
2	Archaea "Crenarchaeota" Thermoprotei Sulfoloba...	3
3	Archaea "Crenarchaeota" Thermoprotei Thermopro...	3
4	Archaea "Euryarchaeota" "Methanomicrobia" Meth...	7

Manipulating indices

- It refers to the process of altering the index (or row labels) of a data structure, such as a Series or DataFrame, to match a new set of labels or indices. This operation is useful when you need to realign data or fill in missing values based on a new index.

A simple use of `reindex` is to alter the order of the rows:

Slide Type Fragment ▾

```
baseball.reindex(baseball.index[::-1]).head() # reverse the order of the rows in the baseball DataFrame
```

	player	year	stint	team	lg	g	ab	r	h	X2b	...	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gdp
id																					
89534	alomasa02	2007	1	NYN	NL	8	22	1	3	1	...	0.0	0.0	0.0	0	3.0	0.0	0.0	0.0	0.0	0.0
89533	aloumo01	2007	1	NYN	NL	87	328	51	112	19	...	49.0	3.0	0.0	27	30.0	5.0	2.0	0.0	3.0	13.0
89530	ausmubr01	2007	1	HOU	NL	117	349	38	82	16	...	25.0	6.0	1.0	37	74.0	3.0	6.0	4.0	1.0	11.0
89526	benitar01	2007	1	SFN	NL	19	0	0	0	0	...	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0
89525	benitar01	2007	2	FLO	NL	34	0	0	0	0	...	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 22 columns

- Reindex in Series

```
import pandas as pd

# Create a Series with initial index
data = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# Reindexing with a new index
new_index = ['a', 'b', 'c', 'd']
reindexed_series = data.reindex(new_index)

print(reindexed_series)
```

In this example, the reindex operation creates a new Series with the provided new_index. Since 'd' is not present in the original index, Pandas fills it with a NaN (missing value).

- Reindex in DataFrame

```
import pandas as pd

# Create a DataFrame with initial index and columns
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data, index=['a', 'b', 'c'])

# Reindexing with new index and columns
new_index = ['a', 'b', 'c', 'd']
new_columns = ['A', 'B', 'C']
reindexed_df = df.reindex(index=new_index, columns=new_columns)

print(reindexed_df)
```

In this DataFrame example, reindex is used to change both the index and the columns. As with Series, missing elements are filled with NaN.

Indexing and Selection

- **Indexing** involves accessing a specific element or subset of elements within a data structure using its index or label. In Pandas, indexing can be performed using various methods:

Position-based Indexing:

```
# Numpy-style indexing  
hits[:3]
```

```
womacto01CHN2006    14  
schilcu01BOS2006     1  
myersmi01NYA2006     0  
Name: h, dtype: int64
```

Label-based Indexing:

```
# Indexing by label  
hits[['womacto01CHN2006', 'schilcu01BOS2006']]
```

```
womacto01CHN2006    14  
schilcu01BOS2006     1  
Name: h, dtype: int64
```

- **Selection** involves extracting specific portions of data from a data structure. It's closely related to indexing but often involves retrieving more than just a single element. Selection can be performed using methods like slicing and boolean indexing:
 - Slicing extracts a subset of elements using a range of indices. It works for both Series and DataFrames. For example:

```
subset = series[start_index:end_index]  
subset = dataframe[start_row:end_row, start_column:end_column]
```

In a `DataFrame` we can slice along either or both axes:

```
baseball_newind[['h', 'ab']] # select two specific columns, 'h' and 'ab'
```

	h	ab
womacto01CHN2006	5	50
schilcu01BOS2006	5	2
myersmi01NYA2006	5	0
helliri01MIL2006	5	3
johnsra05NYA2006	5	6
...
benitar01FLO2007	0	0
benitar01SFN2007	0	0
ausmubr01HOU2007	82	349
aloumo01NYN2007	112	328
alomasa02NYN2007	3	22

100 rows × 2 columns

- **Boolean Indexing:** involves using conditional statements to filter and extract elements that satisfy specific conditions. For example:

```
subset = series[series > threshold]  
subset = dataframe[dataframe['column'] > threshold]
```

```
baseball_newind[baseball_newind.ab>500]
```

	player	year	stint	team	lg	g	ab	r	h	X2b	...	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
gonzalu01ARI2006	gonzalu01	2006	1	ARI	NL	153	586	93	5	52	...	73.0	0.0	1.0	69	58.0	10.0	7.0	0.0	6.0	14.0
vizquom01SFN2007	vizquom01	2007	1	SFN	NL	145	513	54	126	18	...	51.0	14.0	6.0	44	48.0	6.0	1.0	14.0	3.0	14.0
thomafr04TOR2007	thomafr04	2007	1	TOR	AL	155	531	63	147	30	...	95.0	0.0	0.0	81	94.0	3.0	7.0	0.0	5.0	14.0
rodriiv01DET2007	rodriiv01	2007	1	DET	AL	129	502	50	141	31	...	63.0	2.0	2.0	9	96.0	1.0	1.0	1.0	2.0	16.0
griffke02CIN2007	griffke02	2007	1	CIN	NL	144	528	78	146	24	...	93.0	6.0	1.0	85	99.0	14.0	1.0	0.0	9.0	14.0
delgaca01NYN2007	delgaca01	2007	1	NYN	NL	139	538	71	139	30	...	87.0	4.0	0.0	52	118.0	8.0	11.0	0.0	6.0	12.0
biggicr01HOU2007	biggicr01	2007	1	HOU	NL	141	517	68	130	31	...	50.0	4.0	3.0	23	112.0	0.0	3.0	7.0	5.0	5.0

7 rows × 22 columns

Indexing and Selection

- **Selection:** The indexing field `loc` allows us to select subsets of rows and columns in an intuitive way:

```
import pandas as pd

# Create a sample Series
data = pd.Series([10, 20, 30, 40], index=['A', 'B', 'C', 'D'])

# Using .loc to select a single element
value = data.loc['B']
print(value)  # Output: 20

# Using .loc to select a subset of elements
subset = data.loc[['A', 'C', 'D']]
print(subset)
```

```
import pandas as pd

# Create a sample DataFrame
data = {
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
}
df = pd.DataFrame(data, index=['X', 'Y', 'Z'])

# Using .loc to select a single element
value = df.loc['Y', 'B']
print(value)  # Output: 5

# Using .loc to select a row or rows
row_subset = df.loc['Y']
print(row_subset)

# Using .loc to select specific rows and columns
subset = df.loc[['X', 'Z'], ['A', 'C']]
print(subset)
```

When using `.loc` with a DataFrame, you provide two arguments separated by a comma:

`.loc[row_label(s), column_label(s)]`

Similarly, the cross-section method `xs` (not a field) extracts a single column or row *by label* and returns it as a `Series` :

```
baseball_newind.xs('myersmi01NYA2006')
```

```
player    myersmi01
year      2006
stint     1
team      NYA
lg        AL
g         62
ab        0
r         0
h         5
X2b       0
X3b       0
hr        0
rbi       0.0
sb        0.0
cs        0.0
bb        0
so        0.0
ibb       0.0
hbp       0.0
sh        0.0
sf        0.0
gidp      0.0
Name: myersmi01NYA2006, dtype: object
```

.iloc[] VS .loc[]

Position-based VS label-based indexing

- .iloc[] for position-based indexing
 - Selecting row: `df.iloc[n]` --- selects the n-th row
 - Multiple rows: `df.iloc[[n,m,...]]`
 - Slicing: `df.iloc[start: stop]`
- Selecting column: `df.iloc[:, n]` --- select the n-th column.
- Multiple columns: `df.iloc[:, [n, m, ...]]`
- Slicing: `df.iloc[:, start: stop]`
- Select the elements at the n-th row and m-th column: `df.iloc[n, m]`
- Select rows n, m,... and columns i,j,...: `df.iloc[[n, m,...], [i,j,...]]`

Sorting and Ranking

- `.sort_index()`:

This method sorts the DataFrame's index in ascending order by default. If the index consists of labels, they will be sorted accordingly.

```
baseball_newind.sort_index().head()
```

	player	year	stint	team	lg	g	ab	r	h	X2b	...	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
alomasa02NYN2007	alomasa02	2007	1	NYN	NL	8	22	1	3	1	...	0.0	0.0	0.0	0	3.0	0.0	0.0	0.0	0.0	0.0
aloumo01NYN2007	aloumo01	2007	1	NYN	NL	87	328	51	112	19	...	49.0	3.0	0.0	27	30.0	5.0	2.0	0.0	3.0	13.0
ausmubr01HOU2007	ausmubr01	2007	1	HOU	NL	117	349	38	82	16	...	25.0	6.0	1.0	37	74.0	3.0	6.0	4.0	1.0	11.0
benitar01FLO2007	benitar01	2007	2	FLO	NL	34	0	0	0	0	...	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0
benitar01SFN2007	benitar01	2007	1	SFN	NL	19	0	0	0	0	...	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0

- `.sort_index(ascending=False)`:

This method sorts the DataFrame's index in descending order. The `ascending=False` argument specifies that you want to sort the index in reverse order, meaning that higher index values will appear first.

```
baseball_newind.sort_index(ascending=False).head()
```

	player	year	stint	team	lg	g	ab	r	h	X2b	...	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
zaungr01TOR2007	zaungr01	2007	1	TOR	AL	110	331	43	80	24	...	52.0	0.0	0.0	51	55.0	8.0	2.0	1.0	6.0	9.0
womacto01CHN2006	womacto01	2006	2	CHN	NL	19	50	6	5	1	...	2.0	1.0	1.0	4	4.0	0.0	0.0	3.0	0.0	0.0
witasja01TBA2007	witasja01	2007	1	TBA	AL	3	0	0	0	0	...	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0
williwo02HOU2007	williwo02	2007	1	HOU	NL	33	59	3	6	0	...	2.0	0.0	0.0	0	25.0	0.0	0.0	5.0	0.0	1.0
wickmbo01ATL2007	wickmbo01	2007	1	ATL	NL	47	0	0	0	0	...	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0

- We can also sort by value, rather than label.
 - See more examples in github code.
- Ranking: does not re-arrange data, but instead returns an index that ranks each value relative to others in the Series.
 - .rank()
 - See more examples in github code

Missing Values/Data

- NA
- NaN
- None
-
- ?
- -
- -99999

Detecting Missing Values:

- `isna()` / `isnull()`:

Returns a DataFrame of the same shape as the original, with True where missing values are present.

- `notna()` / `notnull()`:

Returns a DataFrame with True where values are not missing.

Missing Values/Data

- **Removing Missing Values:**

```
df.dropna() # Drops rows with any missing values  
df.dropna(axis=1) # Drops columns with any missing values
```

- **Filling Missing Values:**

```
df.fillna(value) # Fills missing values with a specific value
```

Missing Values/Data

- `interpolate()`: Performs linear interpolation to fill missing values.

```
df.interpolate() # Performs linear interpolation along columns
```

- **Replacing Missing Values:**

```
df.replace(to_replace=np.nan, value=0) # Replaces NaN with 0
```

- **Imputation** involves estimating missing values based on existing data. Methods include mean, median, or machine learning techniques.

Data Summation

- `.sum()`
calculates the sum of values along a specified axis in a DataFrame or Series.
- `.mean()`
is valuable for understanding the average value of data and its distribution.
- `.describe()`
 - provides a quick statistical summary of a DataFrame, including various descriptive statistics for each column.
 - is applicable only to numeric columns. If you want to include non-numeric columns in the summary, you can use the `include` parameter like this:
`df.describe(include='all')`.

Writing data to files

- **Writing to CSV:**

```
# Write DataFrame to a CSV file  
df.to_csv('data.csv', index=False)
```

- **Writing to Excel:**

```
# Write DataFrame to an Excel file  
df.to_excel('data.xlsx', index=False)
```

index=False prevents writing row numbers as an extra column

In-class exercise

- Exercise 1: Understanding `.iloc[]` and `.loc[]`
- Objective: Familiarize students with basic `.iloc[]` and `.loc[]` operations.

```
import pandas as pd
```

```
data = {  
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],  
    'Age': [23, 35, 45, 36, 28],  
    'City': ['New York', 'Los Angeles', 'Chicago',  
            'Houston', 'Phoenix']  
}  
df = pd.DataFrame(data)  
print(df)
```

Tasks:

1. Select the row with index 2 using `.iloc[]`.
2. Select the 'Age' column using `.iloc[]`.
3. Select the first three rows using `.iloc[]` slicing.
4. Select the 'Name' and 'City' columns for the first three rows using `.iloc[]`.
5. Select the row for 'Charlie' using `.loc[]`.
6. Select the 'Age' column using `.loc[]`.
7. Select the 'City' column for 'Bob' and 'Eva' using `.loc[]`.

In-class exercise

- Exercise 2: Advanced Indexing
- Objective: Practice more complex `.iloc[]` and `.loc[]` operations including conditional selection.

```
import pandas as pd
```

```
data = {  
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],  
    'Age': [23, 35, 45, 36, 28],  
    'City': ['New York', 'Los Angeles', 'Chicago',  
            'Houston', 'Phoenix']  
}  
df = pd.DataFrame(data)  
print(df)
```

Setup:

1. Use the DataFrame from Exercise 1.

Tasks:

1. Select the rows where 'Age' is greater than 30 using `.loc[]`.
2. Select the rows where 'City' is either 'Chicago' or 'Houston' using `.loc[]`.
3. Change the index of the DataFrame to be the 'Name' column and then use `.loc[]` to select 'David'.
4. Select the 'Age' and 'City' columns for 'Bob' and 'Eva' using `.loc[]` after setting the index to 'Name'.

In-class exercise

- Exercise 3: Practical Application
- Objective: Apply `.iloc[]` and `.loc[]` in a more realistic scenario.

```
data = {  
    'Product': ['Apple', 'Banana', 'Carrot', 'Doughnut', 'Egg'],  
    'Price': [0.5, 0.3, 0.4, 1.0, 0.2],  
    'In Stock': [True, True, False, True, False]  
}  
df = pd.DataFrame(data)  
print(df)
```

Tasks:

1. Select the products that are in stock using `.loc[]`.
2. Update the price of 'Doughnut' to 1.2 using `.loc[]`.
3. Select all products with a price less than 0.5 using `.loc[]`.
4. Select the 'Product' and 'Price' columns for items that are not in stock using `.loc[]`.

Data Wrangling with Pandas

What is data wrangling?

- Data wrangling, also known as data munging or data preprocessing, refers to **the process of cleaning, transforming, and organizing raw data into a more structured and usable format for analysis.**
- In the context of Pandas, data wrangling involves using various techniques and methods to handle and prepare data in a way that makes it suitable for analysis, modeling, visualization, and other data-related tasks.

Handling Date and Time Data:

- Parsing and formatting date and time data (e.g., using `.to_datetime()`).
- Extracting components of date and time (e.g., using `.dt` accessor).
- Resampling and time-based aggregation.

The `datetime` built-in library handles temporal information down to the nanosecond.

```
from datetime import datetime
```

```
now = datetime.now()  
now
```

```
datetime.datetime(2023, 8, 30, 22, 26, 47, 512750)
```

```
now.day
```

```
30
```

```
now.weekday()
```

```
2
```

Data Transformation:

- Changing data types (e.g., using `.astype()`).
- Applying functions or operations to columns (e.g., using `.apply()` or `.map()`).
- Aggregating and summarizing data (e.g., using `.groupby()` and aggregation functions).
- **Merging, joining, and concatenating** DataFrames (e.g., using `.merge()` or `.concat()`).

- Merging and Joining DataFrame objects

In Pandas, we can combine tables according to the value of one or more **keys** that are used to identify rows, much like an index. Using a trivial example:

```
df1 = pd.DataFrame(dict(id=range(4), age=np.random.randint(18, 31, size=4)))
df2 = pd.DataFrame(dict(id=list(range(3)) + list(range(3)), score=np.random.random(size=6)))

print (df1)
print ("\n")
print (df2)
```

	id	age
0	0	24
1	1	23
2	2	18
3	3	26

	id	score
0	0	0.789056
1	1	0.567427
2	2	0.193432
3	0	0.277328
4	1	0.317467
5	2	0.353191

- Inner join

Let's say you have two tables:

Table A with columns: ID, Age

Table B with columns: ID, Score

You want to combine data from both tables based on the common "ID" column.

```
pd.merge(df1, df2)
```

	id	age	score
0	0	24	0.789056
1	0	24	0.277328
2	1	23	0.567427
3	1	23	0.317467
4	2	18	0.193432
5	2	18	0.353191

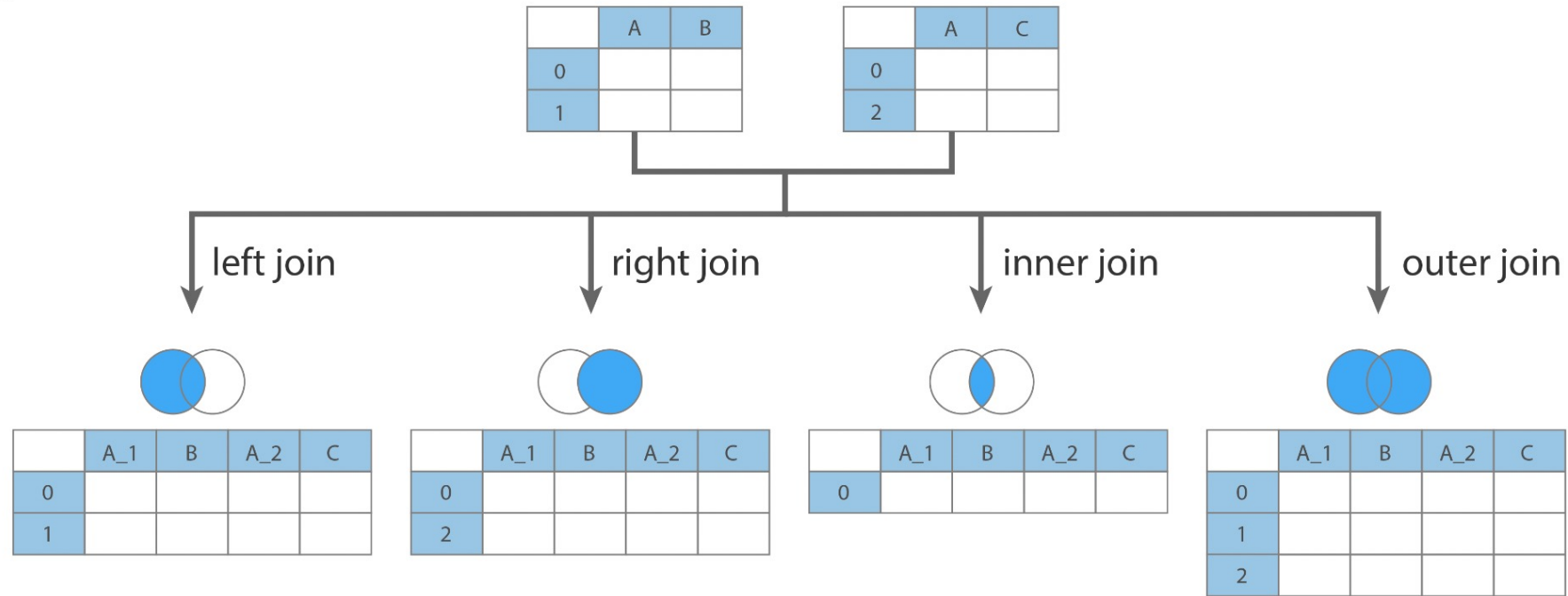
- Outer join

An outer join is a type of database operation used to combine data from two or more tables based on a common key or column. Unlike an inner join, which includes only the rows with matching values in the specified columns, an outer join **includes all rows from both tables, filling in missing values** with NaN (or other specified values) for **non-matching rows**.

```
pd.merge(df1, df2, how='outer')
```

	id	age	score
0	0	24	0.789056
1	0	24	0.277328
2	1	23	0.567427
3	1	23	0.317467
4	2	18	0.193432
5	2	18	0.353191
6	3	26	NaN

Joins



See more example codes in [github](#) for different ways of join.

• Concatenation

A common data manipulation is appending rows or columns to a dataset that already conform to the dimensions of the existing rows or columns, respectively. In NumPy, this is done either with `concatenate` or the convenience functions `c_` and `r_`:

```
np.concatenate([np.random.random(5), np.random.random(5)])
```

```
array([0.55627575, 0.33710572, 0.66255833, 0.71689758, 0.67439145,  
       0.83783671, 0.51672834, 0.36357064, 0.15608224, 0.53061771])
```

```
np.r_[np.random.random(5), np.random.random(5)]
```

```
array([0.7101207 , 0.95955382, 0.01601188, 0.35911948, 0.87334783,  
       0.13092789, 0.33191177, 0.35616117, 0.17734696, 0.26362765])
```

```
np.c_[np.random.random(5), np.random.random(5)]
```

```
array([[0.96997383, 0.54221631],  
       [0.66499924, 0.22282137],  
       [0.5768224 , 0.8373148 ],  
       [0.8814404 , 0.39929759],  
       [0.37281569, 0.72967276]])
```

`concatenate` arrays along the first axis, stacking arrays vertically to form rows in the resulting array.

`c_` concatenates arrays along the second axis, which is typically the column axis.

Data Transformation:

- Changing data types (e.g., using `.astype()`).
- Applying functions or operations to columns (e.g., using `.apply()` or `.map()`).
- Aggregating and summarizing data (e.g., using `.groupby()` and aggregation functions).
- Merging, joining, and concatenating DataFrames (e.g., using `.merge()` or `.concat()`).

- The `.groupby()` function in Pandas is used to **group rows** of a DataFrame based on the values in one or more columns. It's a powerful tool for performing operations on groups of data and aggregating information based on specific criteria.

```
import pandas as pd

data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Value': [10, 20, 15, 25, 30]}

df = pd.DataFrame(data)

# Grouping by the 'Category' column
grouped = df.groupby('Category')

# Calculating the sum of 'Value' for each group
sum_by_category = grouped['Value'].sum()

print(sum_by_category)
```

```
Category
A      55
B      45
Name: Value, dtype: int64
```

- **Aggregation** in Pandas refers to the process of combining multiple data points into a single summary value. It involves performing a computation on a group of data elements and summarizing the results. Aggregation is often used in combination with the `.groupby()` function to analyze and summarize data based on specific categories or groups.
 - **Sum**: Calculates the sum of values in a group.
 - **Mean**: Computes the average of values in a group.
 - **Median**: Computes the middle value of values in a group.
 - **Max**: Finds the maximum value in a group.
 - **Min**: Finds the minimum value in a group.
 - **Count**: Counts the number of non-null values in a group.
 - **Size**: Counts the total number of values in a group (including null values).
 - **Std**: Computes the standard deviation of values in a group.
 - **Var**: Computes the variance of values in a group.
 - **Apply**: Applies a custom function to a group.

```

import pandas as pd

data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Value': [10, 20, 15, 25, 30]}

df = pd.DataFrame(data)

# Grouping by the 'Category' column and calculating aggregations
aggregated = df.groupby('Category').agg({
    'Value': ['sum', 'mean', 'max']
})

print(aggregated)

```

	Value		
Category	sum	mean	max
A	55	18.333333	30
B	45	22.500000	25

Data Reshaping:

- Pivoting data from long to wide format (e.g., using `.pivot()`).
- Melting data from wide to long format (e.g., using `.melt()`).
- Transposing data (e.g., using `.T`).

- The `.stack()` method in Pandas is used to transform or reshape a DataFrame from a wide format to a long format by "stacking" the columns into a single column, resulting in a MultiIndex Series or DataFrame.

To complement this, `unstack` pivots from rows back to columns.

```
stacked = cdystonia.stack()  
stacked
```

```
0    patient      1  
    obs          1  
    week         0  
    site         1  
    id           1  
    ...  
630  id           11  
    treat      5000U  
    age        57  
    sex         M  
    twstrs      51  
Length: 5679, dtype: object
```

```
stacked.unstack().head()
```

	patient	obs	week	site	id	treat	age	sex	twstrs
0	1	1	0	1	1	5000U	65	F	32
1	1	2	2	1	1	5000U	65	F	30
2	1	3	4	1	1	5000U	65	F	24
3	1	4	8	1	1	5000U	65	F	37
4	1	5	12	1	1	5000U	65	F	39

Data Cleaning:

- Handling missing values (e.g., using `.dropna()` or `.fillna()`).
- Removing duplicate rows (e.g., using `.drop_duplicates()`).
- Correcting inconsistent or erroneous data.

Creating New Features:

- Deriving new columns based on existing ones.
- Using conditional statements to create categorical features.

- Using Basic Arithmetic Operations:

```
import pandas as pd

data = {'A': [10, 20, 30],
        'B': [5, 15, 25]}

df = pd.DataFrame(data)

# Derive a new column 'C' as the sum of columns 'A' and 'B'
df['C'] = df['A'] + df['B']
```

- use built-in functions or user-defined functions to calculate new column values.

```
# Define a function to calculate the square of a number
def square(x):
    return x ** 2

# Derive a new column 'A_squared' using the 'square' function
df['A_squared'] = df['A'].apply(square)
```

- Using Conditional Statements:

```
# Derive a new column 'D' with values based on a condition  
df['D'] = df['A'].apply(lambda x: 'High' if x > 20 else 'Low')
```

- Combining Multiple Columns:

```
# Derive a new column 'F' using a combination of columns 'A' and 'B'  
df['F'] = df['A'].astype(str) + '_' + df['B'].astype(str)
```

Handling Outliers and Anomalies:

- Identifying and handling outliers using statistical techniques.

Scaling and Normalization:

- Standardizing or normalizing numeric data.