

# Pandas

Kittikun Jitpaired

## Introduction to Pandas

Pandas is a powerful Python library used for data manipulation and analysis. It provides data structures for efficiently storing large datasets and tools for working with them. The two primary data structures in pandas are:

**Series:** A one-dimensional labeled array

**DataFrame:** A two-dimensional labeled data structure with columns of potentially different types

Before we begin, make sure Pandas is installed:

```
pip install pandas
```

For Excel support, we also need:

```
pip install openpyxl
```

Let's start by importing Pandas:

```
import pandas as pd
```

## Creating DataFrames

### From a dictionary

When creating a DataFrame from a dictionary, the keys become column names, and the values become the data in those columns.

```
data = {
    'Name': ['John', 'Anna', 'Peter', 'Linda'],
    'Age': [28, 34, 29, 32],
    'City': ['New York', 'Paris', 'Berlin', 'London']
}

df = pd.DataFrame(data)

print(df)
display(df)
```

	Name	Age	City
0	John	28	New York
1	Anna	34	Paris
2	Peter	29	Berlin
3	Linda	32	London

	Name	Age	City
0	John	28	New York
1	Anna	34	Paris
2	Peter	29	Berlin
3	Linda	32	London

## From a list of dictionaries

Sometimes, your data might be in the form of a list of dictionaries, where each dictionary represents a row of data. This method allows for more flexibility in the data structure, as each dictionary (row) doesn't necessarily need to have the same keys (columns).

```
data = [
    {'Name': 'John', 'Age': 28, 'City': 'New York'},
    {'Name': 'Anna', 'Age': 34, 'City': 'Paris'},
    {'Name': 'Peter', 'Age': 29, 'City': 'Berlin'},
    {'Name': 'Linda', 'Age': 32, 'City': 'London'}
]

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

	Name	Age	City
0	John	28	New York
1	Anna	34	Paris
2	Peter	29	Berlin
3	Linda	32	London

## From a NumPy array

NumPy arrays are efficient for numerical computations. When you have data in a NumPy array, you can easily convert it to a DataFrame for further analysis.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

df = pd.DataFrame(arr, columns=['A', 'B', 'C'])

print(df)
```

	A	B	C
0	1	2	3
1	4	5	6
2	7	8	9

Note that when creating a DataFrame from a NumPy array, **you need to specify column names separately**, as arrays don't have built-in column labels.

## From a CSV file

CSV (Comma-Separated Values) files are one of the most common formats for storing tabular data. Pandas makes it easy to read these files into DataFrames.

```
df = pd.read_csv('ex_csv.csv')
print(df.head())
```

	Series	Value 1	Value 2	Value 3
0	A	9	96	2
1	B	61	71	28
2	C	36	87	38
3	D	54	77	60
4	E	18	32	99

## From an Excel Files

Pandas uses the `read_excel()` function for reading Excel files.

You may need to install additional library.

```
pip install openpyxl
```

```
# Basic reading
df = pd.read_excel('ex_excel.xlsx')
print('Basic reading')
print(df.head())

# Reading a specific sheet
df = pd.read_excel('ex_excel.xlsx', sheet_name='Sheet2')
print('\nSheet2')
print(df.head())

# Reading multiple sheets
xlsx = pd.ExcelFile('ex_excel.xlsx')
df1 = pd.read_excel(xlsx, 'Sheet1')
df2 = pd.read_excel(xlsx, 'Sheet2')
```

Basic reading

	Serie	Value 1	Value 2	Value 3
0	A	9	96	2
1	B	61	71	28
2	C	36	87	38
3	D	54	77	60
4	E	18	32	99

Sheet2

	Serie	Value 4	Value 5	Value 6
0	K	46	98	99
1	L	71	68	13
2	M	55	45	77
3	N	59	0	15
4	O	5	32	96

## Export DataFrame

### Writing DataFrames to CSV files

Writing DataFrames to CSV files is just as straightforward with the `to_csv()` method.

```
# Basic writing
df.to_csv('output.csv')

# Writing with specific options
df.to_csv('output.csv',
```

```

        index=False, # Don't write index
        columns=['Name', 'Age'], # Only write specific columns
        sep=';', # Use semicolon as separator
        encoding='utf-8') # Specify encoding

# Writing without header
df.to_csv('output.csv', header=False)

```

## Writing Excel Files with Pandas

Writing to Excel files is done using the `to_excel()` method.

```

# Basic writing
df.to_excel('output.xlsx', sheet_name='Sheet1')

# Writing multiple DataFrames to different sheets
with pd.ExcelWriter('output.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet2')

```

## Basic DataFrame Operations

Once you have a DataFrame, there are numerous operations you can perform to explore and manipulate your data:

### Viewing Data

These methods allow you to quickly inspect your DataFrame:

```

data = {
    'Name': ['John', 'Anna', 'Peter', 'Linda'],
    'Age': [28, 34, 29, 32],
    'City': ['New York', 'Paris', 'Berlin', 'London']
}
df = pd.DataFrame(data)

```

```
print(df.head(3)) # First 3 rows
```

	Name	Age	City
0	John	28	New York
1	Anna	34	Paris
2	Peter	29	Berlin

```
print(df.tail(3)) # Last 3 rows
```

	Name	Age	City
1	Anna	34	Paris
2	Peter	29	Berlin
3	Linda	32	London

DataFrame info, including column types and non-null counts

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
#   Column  Non-Null Count  Dtype 
---  -
0   Name    4 non-null       object
1   Age     4 non-null       int64  
2   City    4 non-null       object
dtypes: int64(1), object(2)
memory usage: 228.0+ bytes
None
```

Statistical summary of numerical columns

```
print(df.describe())
```

	Age
count	4.000000
mean	30.750000
std	2.753785
min	28.000000
25%	28.750000
50%	30.500000
75%	32.500000
max	34.000000

These operations are crucial for getting a quick overview of your data, understanding its structure, and identifying potential issues or patterns.

## Accessing Data

Accessing a single column

```
print(df['Name'])
```

```
0    John
1    Anna
2    Peter
3    Linda
Name: Name, dtype: object
```

Accessing multiple columns

```
print(df[['Name', 'Age']])
```

```
   Name  Age
0  John   28
1  Anna   34
2 Peter   29
3 Linda   32
```

Accessing a row by label

```
print(df.loc[0])
```

```
Name    John
Age      28
City    New York
Name: 0, dtype: object
```

Accessing a row by integer index

```
print(df.iloc[0])
```

```
Name    John
Age      28
City    New York
Name: 0, dtype: object
```

Accessing a specific value

```
print(df.loc[0, 'Name'])
```

John

## Adding and Deleting Columns

DataFrames are mutable, allowing you to add or remove columns as needed:

Adding a new column

```
df['Salary'] = [50000, 60000, 55000, 65000]
print(df)
```

	Name	Age	City	Salary
0	John	28	New York	50000
1	Anna	34	Paris	60000
2	Peter	29	Berlin	55000
3	Linda	32	London	65000

Deleting a column

```
df = df.drop('Salary', axis=1)
print(df)
```

	Name	Age	City
0	John	28	New York
1	Anna	34	Paris
2	Peter	29	Berlin
3	Linda	32	London

## Filtering row

Filtering allows you to focus on specific subsets of your data:

```
print(df[df['Age'] > 30])
```

	Name	Age	City
1	Anna	34	Paris
3	Linda	32	London



```
print(df[df['Age'] > 30]['City'])
```

```
1    Paris
3    London
Name: City, dtype: object
```

```
print(df[(df['Age'] > 30) & (df['City'] == 'London')]) # AND condition
```

```
   Name  Age  City
3  Linda   32  London
```

```
print(df[(df['Age'] > 30) | (df['City'] == 'New York')]) # OR condition
```

```
   Name  Age  City
0  John   28  New York
1  Anna   34   Paris
3  Linda   32  London
```

### Filtering row (using .loc())

Filtering allows you to focus on specific subsets of your data:

```
print(df.loc[df['Age'] > 30])
```

```
   Name  Age  City
1  Anna   34  Paris
3  Linda   32  London
```

```
print(df.loc[df['Age'] > 30, 'City'])
```

```
1    Paris
3    London
Name: City, dtype: object
```

```
print(df.loc[(df['Age'] > 30) & (df['City'] == 'London')]) # AND condition
```

```
   Name  Age  City
3  Linda   32  London
```

```
print(df.loc[(df['Age'] > 30) | (df['City'] == 'New York')]) # OR condition
```

	Name	Age	City
0	John	28	New York
1	Anna	34	Paris
3	Linda	32	London

## Data Manipulation

Data manipulation is at the heart of data analysis. Pandas provides powerful tools for sorting, grouping, aggregating, and transforming data:

### Sorting

Sorting allows you to order your data based on one or more columns:

```
print(df.sort_values('Age', ascending=False)) # Sort descending
```

	Name	Age	City
1	Anna	34	Paris
3	Linda	32	London
2	Peter	29	Berlin
0	John	28	New York

```
print(df.sort_values(['City', 'Age'])) # Sort by multiple columns
```

	Name	Age	City
2	Peter	29	Berlin
3	Linda	32	London
0	John	28	New York
1	Anna	34	Paris

### Grouping and Aggregation

Grouping allows you to split your data into groups based on some criteria and then perform calculations on each group:

```
print(df.groupby('City')['Age'].mean()) # Mean age by city
```

```
City
Berlin      29.0
London      32.0
New York    28.0
Paris       34.0
Name: Age, dtype: float64
```

```
print(df.groupby('City').agg({'Age': 'mean', 'Name': 'count'})) # Multiple aggregations
```

```
      Age  Name
City
Berlin  29.0    1
London  32.0    1
New York 28.0    1
Paris   34.0    1
```

Pandas provides a wide range of aggregation functions. Here are some commonly used ones:

1. `count()`: Count of non-null values
2. `sum()`: Sum of values
3. `mean()`: Arithmetic mean of values
4. `median()`: Median of values
5. `min()`, `max()`: Minimum and maximum values
6. `std()`, `var()`: Standard deviation and variance
7. `first()`, `last()`: First and last non-null values
8. `nunique()`: Number of unique values
9. `quantile()`: Quantile of values
10. `agg()`: Allows applying multiple aggregation functions at once

## Applying Functions

You can apply custom functions to your data using the `apply` method:

```
df['Name_Length'] = df['Name'].apply(len) # Apply a function to a column
print(df)
```

```
   Name  Age  City  Name_Length
0  John   28 New York           4
1  Anna   34   Paris           4
2 Peter   29  Berlin           5
3 Linda   32  London           5
```

This allows for complex transformations and feature engineering.

## Merging

Combining data from different sources is a common task in data analysis:

```
df2 = pd.DataFrame({
    'City': ['New York', 'Paris', 'Berlin', 'London'],
    'Country': ['USA', 'France', 'Germany', 'UK']
})
merged_df = pd.merge(df, df2, on='City')
print(merged_df)
```

	Name	Age	City	Name_Length	Country
0	John	28	New York	4	USA
1	Anna	34	Paris	4	France
2	Peter	29	Berlin	5	Germany
3	Linda	32	London	5	UK

Merging allows you to combine data from different sources **based on common columns or indices**.

## Mapping DataFrames

Mapping allows you to replace values in a DataFrame based on a dictionary:

```
data = {
    'Name': ['John', 'Anna', 'Peter', 'Linda'],
    'Age': [28, 34, 29, 32],
    'City': ['New York', 'Paris', 'Berlin', 'London']
}
df = pd.DataFrame(data)

# Create a mapping dictionary
city_country = {
    'New York': 'USA',
    'Paris': 'France',
    'Berlin': 'Germany',
    'London': 'UK'
}

# Apply the mapping to create a new column
df['Country'] = df['City'].map(city_country)
print(df)
```

	Name	Age	City	Country
0	John	28	New York	USA
1	Anna	34	Paris	France
2	Peter	29	Berlin	Germany
3	Linda	32	London	UK

Mapping is useful for categorizing data, replacing codes with meaningful labels, or performing lookups based on a dictionary.

## Concatenating DataFrames

Concatenation allows you to combine DataFrames along a particular axis:

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2']},
                    index=['K0', 'K1', 'K2'])

df2 = pd.DataFrame({'C': ['C0', 'C1', 'C2'],
                    'D': ['D0', 'D1', 'D2']},
                    index=['K0', 'K1', 'K2'])
```

Concatenate along columns (axis=1)

```
result = pd.concat([df1, df2], axis=1)
print(result)
```

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	C1	D1
K2	A2	B2	C2	D2

Concatenate along rows (axis=0)

```
df3 = pd.DataFrame({'A': ['A3', 'A4'],
                    'B': ['B3', 'B4']},
                    index=['K3', 'K4'])

result = pd.concat([df1, df3])
print(result)
```

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2
K3	A3	B3
K4	A4	B4

Concatenate with different indexes

```
df4 = pd.DataFrame({'A': ['A5', 'A6'],
                    'C': ['C5', 'C6']},
                    index=['K1', 'K6'])

result = pd.concat([df1, df4])
print(result)
```

	A	B	C
K0	A0	B0	NaN
K1	A1	B1	NaN
K2	A2	B2	NaN
K1	A5	NaN	C5
K6	A6	NaN	C6

```
result = pd.concat([df1, df4], ignore_index=True)
print(result)
```

	A	B	C
0	A0	B0	NaN
1	A1	B1	NaN
2	A2	B2	NaN
3	A5	NaN	C5
4	A6	NaN	C6

```
result = pd.concat([df1, df4]).reset_index() # reset index of DataFrame
print(result)
```

	index	A	B	C
0	K0	A0	B0	NaN
1	K1	A1	B1	NaN
2	K2	A2	B2	NaN
3	K1	A5	NaN	C5
4	K6	A6	NaN	C6

## Handling Missing Data

Missing data is common in real-world datasets. Let's see how to handle it:

```
df = pd.DataFrame({
    'A': [1, np.nan, 4],
    'B': [5, np.nan, np.nan],
    'C': [9, 10, 11]})
```

Check for missing values

```
print(df.isnull())
```

	A	B	C
0	False	False	False
1	True	True	False
2	False	True	False

Drop rows with missing values

```
print(df.dropna())
```

	A	B	C
0	1.0	5.0	9

Fill missing values with 0

```
print(df.fillna(0))
```

	A	B	C
0	1.0	5.0	9
1	0.0	0.0	10
2	4.0	0.0	11

Fill missing values with mean

```
df['B'] = df['B'].fillna(df['B'].mean())
print(df)
```

	A	B	C
0	1.0	5.0	9
1	NaN	5.0	10
2	4.0	5.0	11

## Reshaping Data

In data analysis, we often need to reshape our data. Two common operations for this are pivot and unpivot, which transform data between “wide” and “long” formats.

### Wide vs. Long Tables

Wide Format (or Wide Table):

- Each subject’s repeated responses are in a single row.
- Each response is in a separate column.
- Usually easier for humans to read.

Long Format (or Long Table):

- Each row is a single subject-response combination.
- Usually easier for machines to process and for certain types of analysis.

Let’s look at an example:

Wide format data

```
wide_data = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Math': [90, 70],
    'Science': [85, 80]
})

print("Wide Format:")
print(wide_data)
```

Wide Format:

	Name	Math	Science
0	Alice	90	85
1	Bob	70	80

Long format data

```
long_data = pd.DataFrame({
    'Name': ['Alice', 'Alice', 'Bob', 'Bob'],
    'Subject': ['Math', 'Science', 'Math', 'Science'],
    'Score': [90, 85, 70, 80]
})

print("Long Format:")
print(long_data)
```



Long Format:

	Name	Subject	Score
0	Alice	Math	90
1	Alice	Science	85
2	Bob	Math	70
3	Bob	Science	80

### Unpivot (Wide to Long)

Unpivot (also known as “melt” in Pandas) is the opposite operation, transforming data from wide format to long format.

- It turns columns into rows.
- In Pandas, we use the `melt()` function for this operation.

```
# Unpivot operation (Wide to Long)
melted = wide_data.melt(id_vars=['Name'], var_name='Subject', value_name='Score')

print("After Melt (Wide to Long):")
print(melted)
```

After Melt (Wide to Long):

	Name	Subject	Score
0	Alice	Math	90
1	Bob	Math	70
2	Alice	Science	85
3	Bob	Science	80

### Pivot (Long to Wide)

Pivot is an operation that transforms data from long format to wide format.

- It typically uses one column to create new columns.
- Values from another column fill these new columns.

```
# Pivot operation (Long to Wide)
pivoted = long_data.pivot(index='Name', columns='Subject', values='Score')

print("After Pivot (Long to Wide):")
print(pivoted)
```

After Pivot (Long to Wide):

Subject	Math	Science
Name		
Alice	90	85
Bob	70	80

When to Use Each Format

1. Use Long Format when:

- Performing statistical analyses that assume each observation is a row.
- Creating certain types of visualizations (e.g., with libraries like Seaborn).
- Working with time-series data.

2. Use Wide Format when:

- Creating summary tables for reports.
- Performing operations that require values to be in the same row.