DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

Itay Hen

Information Sciences Institute

USC Viterbi
School of Engineering

A Python Library to Analyze and Manipulate Data

PANDAS

What is Pandas

- **Pandas** = **Python Data Analysis Library**.
- Built on top of **NumPy** — optimized for working with **tabular or labeled data**.
- Provides powerful data structures:
  - **Series** → 1D labeled array
  - **DataFrame** → 2D labeled table (rows × columns)
- Widely used in **data science**, **machine learning**, **finance**, and **scientific research**.

Why use Pandas

- Makes data manipulation **simple, fast, and intuitive**.
- Handles **CSV, Excel, SQL, JSON**, and more with ease.
- Provides built-in **indexing**, **filtering**, and **grouping**.
- Integrates tightly with **NumPy**, **Matplotlib**
- Great for **cleaning, exploring, transforming**, and **analyzing** data.

Pandas Core data structures

- Series

```
import pandas as pd
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
```

- Acts like a labeled 1D NumPy array
- Attributes: `s.index`, `s.values`, `s.dtype`
- DataFrame

```
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
```

- Think of it like a **spreadsheet**
- Attributes: `df.columns`, `df.index`, `df.shape`, `df.dtypes`

Series vs DataFrames

A series can be seen as a one-dimensional array with index values whereas a DataFrame is a two-dimensional array having rows and columns

Series

	apples
0	3
1	2
2	0
3	1

Series

	oranges
0	0
1	3
2	7
3	2

DataFrame

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

Series: an indexed 1D array

```
>>> data = pd.Series([0.25, 0.5, -.75, 1.0])
>>> data
0    0.25
1    0.50
2   -0.75
3    1.00
dtype: float64
>>> data[3]
1.0
```

data access by int indices

```
>>> indices = ['a' ,'b', 'c', 'd']
>>> values = [0.25, 0.5, -.75, 1.0]
>>> data = pd.Series(values, index=indices)
>>> data
a    0.25
b    0.50
c   -0.75
d    1.00
dtype: float64
>>> data['b']
0.5
```

data access by named indices

Series: Selecting Data by index label: .loc[]

```
import pandas as pd

s = pd.Series(
    [10, 20, 30, 40],
    index=["a", "b", "c", "d"]
)

print(s)
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

```
s.loc["b"]
```

```
20
```

```
s.loc[["a", "d"]]
```

```
a    10
d    40
dtype: int64
```

```
s.loc["b":"d"]
```

```
b    20
c    30
d    40
dtype: int64
```

Series: Selecting Data by index label: .iloc[]

```
import pandas as pd

s = pd.Series(
    [10, 20, 30, 40],
    index=["a", "b", "c", "d"]
)

print(s)
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

```
s.iloc[1]
```

```
20
```

```
s.iloc[[0, 3]]
```

```
a    10
d    40
dtype: int64
```

```
s.iloc[1:3]
```

```
b    20
c    30
dtype: int64
```

Series methods

Some useful series functions or methods are: head(), tail() and count()

head(): Returns the first n members of the series. Default value is 5

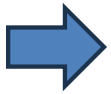
tail(): Returns the last n members of the series. Default value is 5

count(): Returns the total number of non NaN members of the series.

```
import pandas as pd

s = pd.Series([10, 20, 30, 40, 50, 60],
              index=["a", "b", "c", "d", "e", "f"])

print(s.head())        # default: first 5 elements
```



```
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

Or

```
print(s.tail(3))    # last 3 elements
```

Series: Hybrid between Dictionary and List

```
population_dict = {
    'California': 38332521,
    'Texas': 26448193,
    'New York': 19651127,
    'Florida': 19552860,
    'Illinois': 12882135,
}

population = pd.Series(population_dict)
```

```
print(population)
```

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

```
print(population['California'])
```

```
38332521
```

Series

- ordered sequence, homogenous type
  - usually numbers (but doesn't have to be)
- accessible with
  - numeric index
  - labeled index
- Metadata
  - name
  - datatype

```
s.name = "my_series"
print(s.name)

print(s.dtype)
```

```
print(population['California':'New York'])
```

```
California    38332521
Texas         26448193
New York      19651127
dtype: int64
```

inclusive

Series: Conditional Selection based on Boolean Array

```
g7_pop = pd.Series({
    'Canada': 35.467,
    'France': 63.951,
    'Germany': 80.94,
    'Italy': 60.665,
    'Japan': 127.061,
    'United Kingdom': 64.511,
    'United States': 318.523,
}, name='G7 population in millions')
```

```
g7_pop > 70
```

```
Canada            False
France            False
Germany            True
Italy             False
Japan              True
United Kingdom    False
United States      True
Name: G7 population in millions, dtype: bool
```

```
g7_pop[g7_pop > 70]
```

```
Germany           80.940
Japan            127.061
United States    318.523
Name: G7 population in millions, dtype: float64
```

```
g7_pop[(g7_pop > 80) | (g7_pop < 40)]
```

```
Canada            35.467
Germany           80.940
Japan            127.061
United States    318.523
Name: G7 population in millions, dtype: float64
```

```
g7_pop.mean()
```

```
107.30257142857144
```

```
g7_pop[g7_pop > g7_pop.mean()]
```

```
Japan            127.061
United States    318.523
Name: G7 population in millions, dtype: float64
```

What is (g7_pop>70).dtype?

`dtype('bool')`

Pandas DataFrames

- Data Frames can be created from any of the basic Data Structures like, lists, dictionaries, dictionaries of lists, arrays lists of dictionaries and also from series.

An empty DataFrame can be created as follows:

```
import pandas as pd
dFrameEmt = pd.DataFrame()
print(dFrameEmt)
```

Output

```
Empty DataFrame
Columns: []
Index: []
```

Creating a DataFrame

```
import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Charlie", "David"],
    "Age": [24, 30, 18, 35],
    "Score": [88, 92, 95, 70],
    "UT": [1, 2, 2, 1]
}

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

Output:

	Name	Age	Score	UT
0	Alice	24	88	1
1	Bob	30	92	2
2	Charlie	18	95	2
3	David	35	70	1

Pandas: DataFrame

```
d = {
    'col1': [1,2],
    'col2': [3,4]
}

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

```
df
```

	col1	col2
0	1	3
1	2	4

```
df2 = pd.DataFrame(np.random.randint(low=0, high=10, size=(5, 5)),
                   columns=['a', 'b', 'c', 'd', 'e'])
df2
```

	a	b	c	d	e
0	2	2	9	3	1
1	6	0	6	4	2
2	1	4	4	3	9
3	7	0	9	6	0
4	1	4	5	8	5

- Generalized 2-dimensional array
  - Flexible row and column indices
  - Each column is a Series: should have values of the same type
  - Different Columns in a DataFrame can have different types

np.random.randint: Return random integers from the “discrete uniform” distribution from low (inclusive) to high (exclusive), i.e., “half-open” interval [low, high).

Creating a DataFrame from a dictionary of Series

```
import pandas as pd
ResultSheet={'Arnab': pd.Series([90, 91, 97],
index=['Maths','Science','Hindi']),
'Ramit': pd.Series([92, 81, 96], index=['Maths','Science','Hindi']),
'Samridhi': pd.Series([89, 91, 88], index=['Maths','Science','Hindi']),
'Riya': pd.Series([81, 71, 67],   index=['Maths','Science','Hindi']),
'Mallika': pd.Series([94, 95, 99], index=['Maths','Science','Hindi'])}
ResultDF = pd.DataFrame(ResultSheet)
print(ResultDF)
```

Creating a DataFrame from a dictionary of Series

Output

	Arnab	Ramit	Samridhi	Riya	Mallika
Maths	90	92	89	81	94
Science	91	81	91	71	95
Hindi	97	96	88	67	99

Constructing a DataFrame from Series

```
population_dict = {
    'California': 38332521,
    'Texas': 26448193,
    'New York': 19651127,
    'Florida': 19552860,
    'Illinois': 12882135,
}
population = pd.Series(population_dict)
print(population)
```

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

```
area_dict = {
    'California': 423967,
    'Texas': 695662,
    'New York': 141297,
    'Florida': 170312,
    'Illinois': 149995
}

area = pd.Series(area_dict)
print(area)
```

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

```
states = pd.DataFrame({'population': population,
                       'area': area})
```

```
states
```

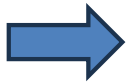
	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

Constructing a DataFrame from Series

```python
import pandas as pd

s1 = pd.Series([1, 2], index=["a", "b"])
s2 = pd.Series([10, 20, 30], index=["b", "c", "d"])

df = pd.DataFrame({"col1": s1, "col2": s2})
print(df)
```



	col1	col2
a	1.0	NaN
b	2.0	10.0
c	NaN	20.0
d	NaN	30.0

Selecting Data by integer index: .iloc[]

```
df.iloc[row_selection, column_selection]
```

```
df.iloc[3]
```

```
A   -1.142745
B   -1.058796
C   -1.220837
D   -0.674843
E   -1.321597
Name: 2023-11-03 00:00:00, dtype: float64
```

```
df.iloc[3:5, 0:2]
```

	A	B
2023-11-03	-1.142745	-1.058796
2023-11-04	0.312989	2.249870

```
df
```

	A	B	C	D	E
2023-10-31	-1.493580	-1.503900	0.021580	1.033983	-1.717857
2023-11-01	-0.822564	-1.627493	0.981578	0.667338	-0.097755
2023-11-02	-0.919850	0.074092	1.694413	-0.167714	-0.774181
2023-11-03	-1.142745	-1.058796	-1.220837	-0.674843	-1.321597
2023-11-04	0.312989	2.249870	-0.378607	-0.793130	-0.257939
2023-11-05	-1.626214	0.676031	-1.550592	-0.453813	-0.603013

```
df
```

	A	B	C	D	E
2023-10-31	-1.493580	-1.503900	0.021580	1.033983	-1.717857
2023-11-01	-0.822564	-1.627493	0.981578	0.667338	-0.097755
2023-11-02	-0.919850	0.074092	1.694413	-0.167714	-0.774181
2023-11-03	-1.142745	-1.058796	-1.220837	-0.674843	-1.321597
2023-11-04	0.312989	2.249870	-0.378607	-0.793130	-0.257939
2023-11-05	-1.626214	0.676031	-1.550592	-0.453813	-0.603013

Indexing with .iloc

.iloc — Integer-position Indexing
Use .iloc to select rows and columns **by numerical position**, not by label.

Key Features
Uses **0-based integer indices**
Supports row, column, and combined selection
End-exclusive slicing (like standard Python)
Ideal when labels are unknown, not sequential, or irrelevant

```
df.iloc[0]                  # First row
df.iloc[[0, 3, 5]]          # Specific rows by position
df.iloc[0:5]                # Row slice (0 to 4)
df.iloc[0, 2]               # Single cell (row 0, column 2)
df.iloc[:, 1:4]             # Columns 1–3
df.iloc[df['A'].to_numpy().argsort()]  # Positional boolean-like indexing
df.iloc[::2]                # Every second row
```

Selecting Data by index label: .loc[]

```
df.loc['2023-11-03']
```

```
A   -1.142745
B   -1.058796
C   -1.220837
D   -0.674843
E   -1.321597
Name: 2023-11-03 00:00:00, dtype: float64
```

```
df.loc[:, ['A', 'B']]
```

	A	B
2023-10-31	-1.493580	-1.503900
2023-11-01	-0.822564	-1.627493
2023-11-02	-0.919850	0.074092
2023-11-03	-1.142745	-1.058796
2023-11-04	0.312989	2.249870
2023-11-05	-1.626214	0.676031

multi-axis by label

```
df.loc['2023-11-01': '2023-11-04', ['A', 'B']]
```

	A	B
2023-11-01	-0.822564	-1.627493
2023-11-02	-0.919850	0.074092
2023-11-03	-1.142745	-1.058796
2023-11-04	0.312989	2.249870

> **Warning**
>
> Note that contrary to usual python slices, **both** the start and the stop are included

```
df.loc[row_selection, column_selection]
```

Indexing with .loc

.loc — Label-based Indexing

Use .loc to select **rows and columns by label**, not by position.

Key Features

- Selects by **index labels** (strings, dates, names, etc.)
- Supports **row**, **column**, and **row+column** selection
- **Inclusive** of the end label when slicing
- Fully supports **boolean masks**

```
df.loc['row_label']                 # Single row
df.loc[['row1','row2']]             # Multiple rows
df.loc['row1':'row5']               # Slice by labels (inclusive)
df.loc['row1', 'colA']              # Single cell
df.loc[:, ['colA','colB']]          # Select columns
df.loc[df['A'] > 0]                 # Boolean mask
df.loc['row1', 'A':'C']             # Label-based column slice
```

DataFrame Attributes: index, columns, values

```
import pandas as pd

df = pd.DataFrame({
    "time": [0, 1, 2],
    "energy": [10.0, 9.5, 9.1]
})
```

	time	energy
0	0	10.0
1	1	9.5
2	2	9.1

```
df.index → [0, 1, 2]
```

```
df.columns → ['time', 'energy']
```

```
df.values →
[[ 0.  10.0],
 [ 1.   9.5],
 [ 2.   9.1]]
```

Row labels	`pd.Index`
Column labels	`pd.Index`
Raw data	`numpy.ndarray`

View the First (head) or Last (tail) rows

```
df.head()
```

	A	B	C	D	E
2023-10-31	-1.493580	-1.503900	0.021580	1.033983	-1.717857
2023-11-01	-0.822564	-1.627493	0.981578	0.667338	-0.097755
2023-11-02	-0.919850	0.074092	1.694413	-0.167714	-0.774181
2023-11-03	-1.142745	-1.058796	-1.220837	-0.674843	-1.321597
2023-11-04	0.312989	2.249870	-0.378607	-0.793130	-0.257939

```
df.tail(3)
```

	A	B	C	D	E
2023-11-03	-1.142745	-1.058796	-1.220837	-0.674843	-1.321597
2023-11-04	0.312989	2.249870	-0.378607	-0.793130	-0.257939
2023-11-05	-1.626214	0.676031	-1.550592	-0.453813	-0.603013

Statistical Summary

```
df.describe()
```

	A	B	C	D	E
count	6.000000	6.000000	6.000000	6.000000	6.000000
mean	-0.948660	-0.198366	-0.075411	-0.064697	-0.795390
std	0.693134	1.504648	1.252118	0.749385	0.623032
min	-1.626214	-1.627493	-1.550592	-0.793130	-1.717857
25%	-1.405871	-1.392624	-1.010280	-0.619586	-1.184743
50%	-1.031297	-0.492352	-0.178514	-0.310764	-0.688597
75%	-0.846885	0.525547	0.741579	0.458575	-0.344207
max	0.312989	2.249870	1.694413	1.033983	-0.097755

median

Descriptive Statistics & Aggregation (Common to Series & DataFrame)

- `sum()` — Sum of values
- `mean()` — Arithmetic mean
- `median()` — Median value
- `std()` — Standard deviation
- `var()` — Variance
- `min()` — Minimum value
- `max()` — Maximum value
- `count()` — Number of non-NA values
- `prod()` — Product of values
- `nunique()` — Count of unique values

Notes:

- On a *Series*, each method returns a **scalar**.
- On a *DataFrame*, each method returns **one value per column** (by default).

Selection

Selecting Columns

a. `df["Age"]`

```yaml
0    24
1    30
2    18
3    35
Name: Age, dtype: int64
```

b. `df[["Name", "Score"]]`

```markdown
      Name  Score
0    Alice     88
1      Bob     92
2  Charlie     95
3    David     70
```

Selecting Rows

`df.iloc[1]` **(2nd row)**

```pgsql
Name      Bob
Age        30
Score      92
UT          2
Name: 1, dtype: object
```

`df[df["Score"] > 90]`

```markdown
      Name  Age  Score  UT
1      Bob   30     92   2
2  Charlie   18     95   2
```

Filtering

```
df[(df.Age > 20) & (df.UT == 1)]
```

Rows with **Age > 20** and **UT = 1**:

	Name	Age	Score	UT
0	Alice	24	88	1
3	David	35	70	1

Grouping and Aggregation

```
import pandas as pd

df = pd.DataFrame({
    "team": ["A", "A", "B", "B", "B"],
    "score": [10, 15, 20, 25, 30],
    "time": [1.0, 2.0, 1.5, 2.5, 3.0]
})

print(df)
```

	team	score	time
0	A	10	1.0
1	A	15	2.0
2	B	20	1.5
3	B	25	2.5
4	B	30	3.0

```
df.groupby("team").mean()
```

team	score	time
A	12.5	1.5
B	25.0	2.333333

Grouping and Aggregation

```
import pandas as pd

df = pd.DataFrame({
    "team": ["A", "A", "B", "B", "B"],
    "score": [10, 15, 20, 25, 30],
    "time": [1.0, 2.0, 1.5, 2.5, 3.0]
})

print(df)
```

	team	score	time
0	A	10	1.0
1	A	15	2.0
2	B	20	1.5
3	B	25	2.5
4	B	30	3.0

```
df.groupby("team")["score"].mean()
```

```
team
A    12.5
B    25.0
Name: score, dtype: float64
```

This gives a **Series** instead of a DataFrame.

Grouping

```
df = pd.read_csv('enrollment_dsci.csv')
df
```

	Year	Semester	Instructor	Students
0	2020	Fall	Abramson	69
1	2021	Spring	Arens	62
2	2021	Fall	Farzindar	48
3	2021	Fall	Arens	50
4	2022	Spring	Farzindar	45
5	2022	Spring	Hermjakob	44
6	2023	Spring	Ambite	44
7	2023	Fall	Satyukov	59
8	2023	Fall	Singh	70

```
df.groupby('Year')['Semester'].count()
```

```
Year
2020    1
2021    3
2022    2
2023    3
Name: Semester, dtype: int64
```

```
df.groupby(['Year'])['Students'].sum()
```

```
Year
2020     69
2021    160
2022     89
2023    173
Name: Students, dtype: int64
```

```
df.groupby(['Year', 'Semester'])['Students'].sum()
```

```
Year  Semester
2020  Fall         69
2021  Fall         98
      Spring       62
2022  Spring       89
2023  Fall        129
      Spring       44
Name: Students, dtype: int64
```

Sorting

```
df.sort_values("Score")
```

Sort by score ascending:

	Name	Age	Score	UT	Age2	Pass
3	David	35	70	1	1225	False
0	Alice	24	88	1	576	True
1	Bob	30	92	2	900	True
2	Charlie	18	95	2	324	True

In pandas, **sorting does NOT happen in place unless you explicitly ask for it.**

Sorting

column

```
df.sort_index(axis=1, ascending=False)
```

	E	D	C	B	A
2023-10-31	-1.717857	1.033983	0.021580	-1.503900	-1.493580
2023-11-01	-0.097755	0.667338	0.981578	-1.627493	-0.822564
2023-11-02	-0.774181	-0.167714	1.694413	0.074092	-0.919850
2023-11-03	-1.321597	-0.674843	-1.220837	-1.058796	-1.142745
2023-11-04	-0.257939	-0.793130	-0.378607	2.249870	0.312989
2023-11-05	-0.603013	-0.453813	-1.550592	0.676031	-1.626214

```
df.sort_values(by=['B'])
```

	A	B	C	D	E
2023-11-01	-0.822564	-1.627493	0.981578	0.667338	-0.097755
2023-10-31	-1.493580	-1.503900	0.021580	1.033983	-1.717857
2023-11-03	-1.142745	-1.058796	-1.220837	-0.674843	-1.321597
2023-11-02	-0.919850	0.074092	1.694413	-0.167714	-0.774181
2023-11-05	-1.626214	0.676031	-1.550592	-0.453813	-0.603013
2023-11-04	0.312989	2.249870	-0.378607	-0.793130	-0.257939

Adding new columns

a. `df["Age2"] = df["Age"] ** 2`

markdown

	Name	Age	Score	UT	Age2
0	Alice	24	88	1	576
1	Bob	30	92	2	900
2	Charlie	18	95	2	324
3	David	35	70	1	1225

b. `df["Pass"] = df["Score"] >= 80`

yaml

	Name	Age	Score	UT	Age2	Pass
0	Alice	24	88	1	576	True
1	Bob	30	92	2	900	True
2	Charlie	18	95	2	324	True
3	David	35	70	1	1225	False

Common Operations

```
df['BMI'] = df['Weight'] / (df['Height']/100)**2  # new column
df.sort_values('Age', ascending=False)
df.groupby('Department')['Salary'].mean()
df.dropna()               # remove missing data
df.fillna(0)              # replace NaN with 0
```

- Vectorized operations like NumPy
- Powerful **groupby**, **merge**, and **pivot** functions

File I/O in Pandas

- Reading:

```
import pandas as pd

df = pd.read_csv("data.csv")          # CSV
df = pd.read_json("data.json")        # JSON
df = pd.read_xml("data.xml")          # XML
df = pd.read_html("page.html")[0]     # HTML tables (returns list)
df = pd.read_excel("file.xlsx")       # Excel (xls/xlsx)
```

Importing and exporting data

```
df = pd.read_csv('data.csv')
df.head()      # first 5 rows
df.info()      # column info
df.describe() # summary stats
```

- Other file types:
  - `pd.read_excel()`, `pd.read_json()`, `pd.read_sql()`
- Write back to disk:
  - `df.to_csv('output.csv', index=False)`

Example workflow

```
import pandas as pd


df = pd.read_csv('employees.csv')
df = df[df['Salary'] > 50000]
avg_age = df.groupby('Department')['Age'].mean()
print(avg_age)
```

Load → **Filter**→ **Analyze** → **Summarize** — all in a few lines.

File I/O: csv

```
df = pd.read_csv('gdp.csv', sep=',')
```

```
df.head()
```

	Country	Population	GDP	Area	HDI	Continent
0	Canada	35.467	1785387	9984670	0.913	America
1	France	63.951	2832687	640679	0.888	Europe
2	Germany	80.940	3874437	357114	0.916	Europe
3	Italy	60.665	2167744	301336	0.873	Europe
4	Japan	127.061	4602367	377930	0.891	Asia

- Using the parameter sep, open file in any format, csv, tsv, etc

File I/O: Excel

```
df = pd.read_excel('tac_baseline_nuggets_v2.xlsx', 'tac_baseline_output_2', index_col=None).fillna('MISSING')
```

```
df.head()
```

	Nugget ID	Nugget Complexity	Evaluator ID	System	Task ID	Start timestamp	End timestamp
0	GEO-R-2a.1	9	0-base	TA_C_baseline	GEO-R-2a	2023-07-24T22:53:13.124Z	2023-07-24T22:53:43.529Z
1	MT-R-2a.1	5	0-base	TA_C_baseline	MT-R-2a	2023-07-24T23:15:27.264Z	2023-07-24T23:15:46.757Z
2	ICT-task2.1	10	0-base	TA_C_baseline	ICT-task2	2023-07-24T23:20:29.591Z	2023-07-24T23:20:33.295Z
3	ICT-task1.1	7	0-base	TA_C_baseline	ICT-task1	2023-07-24T23:26:56.491Z	2023-07-24T23:27:30.328Z
4	GEO-S-3.1	6	1-base	TA_C_baseline	GEO-S-3	2023-07-28T16:36:49.321Z	2023-07-28T16:37:03.597Z

- Need to install openpyxl
  - pip install openpyxl
- Read any sheet in the excel
- Set Index columns
- Handle missing values

HTML Tables to DataFrame: read_html()

- **Extracts HTML tables** from a webpage or HTML string.
- Returns a **list of DataFrames**, one per `<table>` element found.
- Automatically **parses** `<table>`, `<tr>`, `<th>`, **and** `<td>` **tags**.
- Requires `lxml` or `html5lib` as an HTML parser.
- Performs **automatic type inference** (e.g., converts numbers and dates).
- Accepts either:
  - a **URL**,
  - a **file path**, or
  - a **string containing HTML**.
- Recognizes **only real** `<table>` **tags** (ignores div-based layouts).

HTML Tables to DataFrame: read_html()

Sovereign states and dependencies by population

Note: A numbered rank is assigned to the 193 member states of the United Nations, plus the two observer states to the United Nations General Ass
Dependent territories and constituent countries that are parts of sovereign states are not assigned a numbered rank. In addition, sovereign states w
limited recognition are included, but not assigned a number rank.

	Country / Dependency	Population	% of world	Date	Source (official or from the United Nations)	
–	**World**	**8,068,484,000**	**100%**	**29 Oct 2023**	**UN projection**[3]	
1	China	1,411,750,000	17.5%	31 Dec 2022	Official estimate[4]	[b]
2	India	1,392,329,000	17.3%	1 Mar 2023	Official projection[5]	[c]
3	United States	335,581,000	4.2%	29 Oct 2023	National population clock[7]	[d]
4	Indonesia	279,118,866	3.5%	1 Jul 2023	National annual projection[8]	
5	Pakistan	241,499,431	3.0%	1 Mar 2023	2023 census result[9]	[e]
6	Nigeria	216,783,400	2.7%	21 Mar 2022	Official projection[10]	
7	Brazil	203,062,512	2.5%	1 Aug 2022	2022 census result[11]	
8	Bangladesh	169,828,911	2.1%	14 Jun 2022	2022 census result[12]	
9	Russia	146,424,729	1.8%	1 Jan 2023	Official estimate[13]	[f]
10	Mexico	129,202,482	1.6%	30 Jun 2023	National quarterly estimate[14]	
11	Japan	124,340,000	1.5%	1 Oct 2023	Official estimate[15]	
12	Philippines	110,737,000	1.4%	29 Oct 2023	National population clock[16]	
13	Ethiopia	107,334,000	1.3%	1 Jul 2023	National annual projection[17]	
14	Egypt	105,486,000	1.3%	29 Oct 2023	National population clock[18]	
15	Vietnam	100,000,000	1.2%	Apr 2023	Official estimate[19]	

```
url = 'https://en.wikipedia.org/wiki/List_of_countries_and_depende
tables = pd.read_html(url)
```

```
len(tables)
```

```
3
```

```
tables[0]
```

	Unnamed: 0	Country / Dependency	Population	% of world	Date	Source (official or from the United Nations)	Unnamed: 6
0	–	World	8068484000	100%	29 Oct 2023	UN projection[3]	NaN
1	1	China	1411750000	17.5%	31 Dec 2022	Official estimate[4]	[b]
2	2	India	1392329000	17.3%	1 Mar 2023	Official projection[5]	[c]
3	3	United States	335581000	4.2%	29 Oct 2023	National population clock[7]	[d]
4	4	Indonesia	279118866	3.5%	1 Jul 2023	National annual projection[8]	NaN
...	...	...	...	...	...	...	...
237	–	Tokelau (NZ)	1647	0%	1 Jan 2019	2019 Census [230]	NaN
238	–	Niue	1549	0%	1 Jul 2021	National annual projection[182]	NaN
239	195	Vatican City	764	0%	26 Jun 2023	Official figure[231]	[af]
240	–	Cocos (Keeling) Islands (Australia)	593	0%	30 Jun 2020	2021 Census[232]	NaN
241	–	Pitcairn Islands (UK)	47	0%	1 Jul 2021	Official estimate[233]	NaN

242 rows × 7 columns

File I/O in Pandas

- Writing:

```
df.to_csv("out.csv", index=False)       # CSV
df.to_json("out.json", orient="records")
df.to_xml("out.xml")                    # XML
df.to_html("table.html")                # HTML
df.to_excel("out.xlsx", index=False)    # Excel
```

Pandas: Additional Resources

- Official site: https://pandas.pydata.org
  - Documentation: https://pandas.pydata.org/docs/
  - Cheat Sheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf
- Tutorials:
  - https://pandas.pydata.org/docs/getting_started/index.html#getting-started
  - Slides based on: https://pandas.pydata.org/docs/user_guide/10min.html#min
  - Data Analysis with Python
    - https://www.freecodecamp.org/learn/data-analysis-with-python/
  - Data Analysis with Python: Zero to Pandas
    - https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas
  - Data Analysis with Python and Pandas Tutorial Introduction
    - https://pythonprogramming.net/data-analysis-python-pandas-tutorial-introduction/
- Book: Python for Data Analysis, Wes McKinney, 2018
    - https://learning.oreilly.com/library/view/python-for-data/9781491957653/?ar=
- You can read O'Reilly books for free through the USC Library!!!