Module 4: pandas and wrangling

PUBHLT 0411 · Python for Public Health Data Analysis

Soumik Purkayastha

Dept. of Biostatistics and Health Data Science, University of Pittsburgh

October 29, 2026

Twenty-three blanks.

Sometimes official statistics aren't reported because they're not trustworthy!

Sixty-seven counties, twenty-three blanks

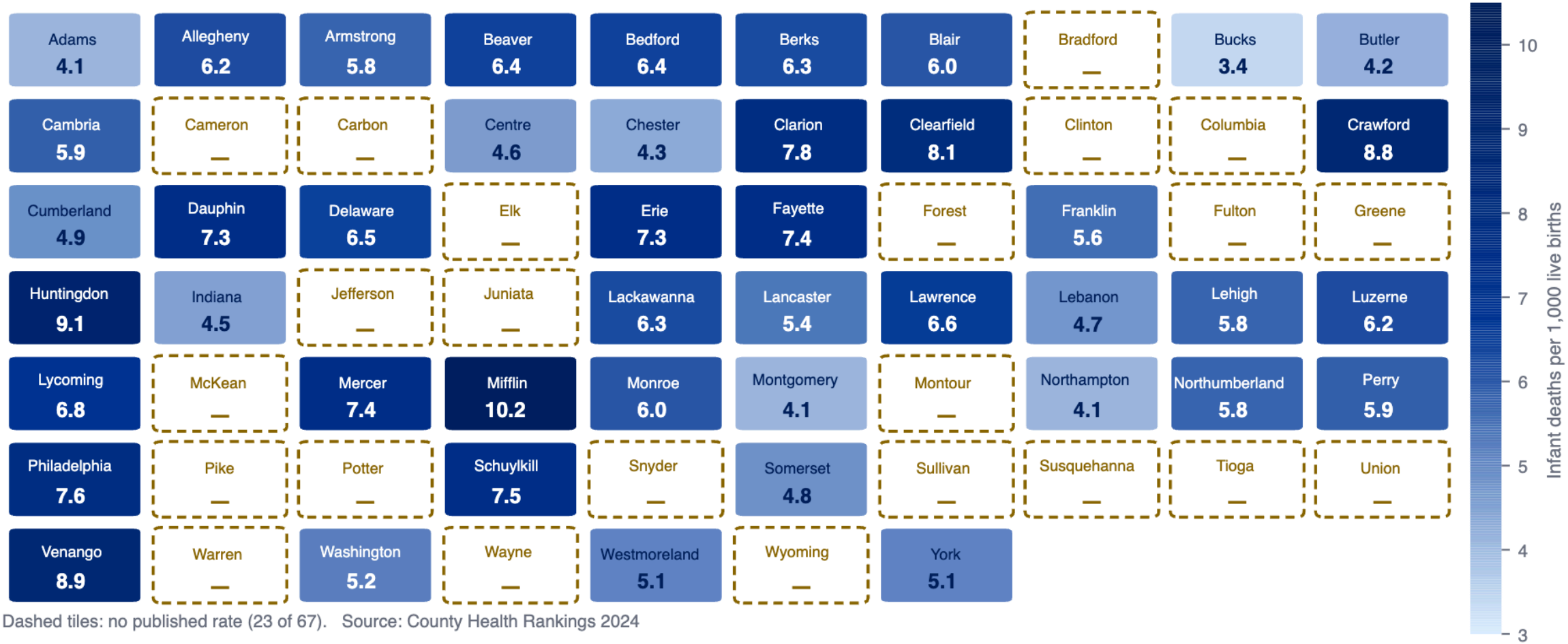


Figure 1. All 67 Pennsylvania counties in alphabetical order, one tile each. Tiles with a published infant mortality rate are shaded blue, darker for higher rates, from Bucks at 3.4 to Mifflin at 10.2 per 1,000. Twenty-three tiles are white with a dashed gold border and a dash in place of a number: Bradford, Cameron, Carbon, Clinton, Columbia, Elk, Forest, Fulton, Greene, Jefferson, Juniata, McKean, Montour, Pike, Potter, Snyder, Sullivan, Susquehanna, Tioga, Union, Warren, Wayne, and Wyoming.

A DataFrame is a table with named columns

Python Code | Start Over | Run Code

```
1 import pandas as pd
2
3 births = pd.read_csv("data/pa_maternal_infant.csv")
4
5 print(births.shape)
6 print(births.head(3))
7 print()
8 print(births.dtypes)
```

`pd.read_csv()` reads an entire table in **one call**. `.shape` gives rows and columns, `.head(n)` the first rows, and `.dtypes` the type of each column.

Inspect before computing

A column read as text when it should be numeric produces no error at read time. The failure appears later, when arithmetic on it returns something wrong or raises `TypeError`.

`.info()` and `.describe()` report what is missing

Python Code | Start Over | Run Code

```
1 import pandas as pd
2
3 births = pd.read_csv("data/pa_maternal_infant.csv")
4
5 births.info()
6 print()
7 print(births[["infant_mortality_per_1000", "children_in_poverty_pct"]].describe().round(2))
```

The table has 67 rows. **`infant_mortality_per_1000` has 44 non-null values.**

> ⚠ **The gap is the finding, not an obstacle**
>
> Twenty-three counties are missing a rate — the answer to the question this session opened, found in the second minute of inspection. In `.describe()`, `count` is the number of **present** values, and every other statistic is computed over it.

A column is a Series; `.loc` and `.iloc` select rows

Python Code | Start Over | Run Code

```
1  import pandas as pd
2
3  births = pd.read_csv("data/pa_maternal_infant.csv")
4
5  rates = births["infant_mortality_per_1000"]
6  print(type(rates).__name__)
7  print(f"Mean: {rates.mean():.2f}   Reporting: {rates.count()} of {rates.size}")
8
9  by_county = births.set_index("county")
10 print(by_county.loc["Allegheny", "infant_mortality_per_1000"])
11 print(by_county.iloc[0, 0])
```

A **Series** is one column: values plus an **index** that labels them. `.count()` counts present values; `.size` counts rows. `.loc[row, column]` selects by **label**; `.iloc[row, column]` by **position**.

⚠ **`KeyError` names a label that is not there**

`births["infant_mortality"]` raises `KeyError: 'infant_mortality'` — the column is `infant_mortality_per_1000`. **Resolution:** print `df.columns.tolist()` and copy the name.

A boolean mask filters rows

Python Code | Start Over | Run Code

```
1 import pandas as pd
2
3 births = pd.read_csv("data/pa_maternal_infant.csv")
4
5 high_poverty = births[births["children_in_poverty_pct"] > 20]
6
7 print(high_poverty.shape)
8 print(high_poverty[["county", "children_in_poverty_pct"]].head())
```

The mask is Module 3's boolean array. Passing it to `[]` keeps the rows where it is `True`, and **every column comes along**.

This is what the DataFrame buys

Module 3's mask filtered one array and left the names behind. Here the county names, the rates, and the incomes all move together, because they are rows of one object.

Combining conditions requires `&`, `|`, and parentheses

Python Code | Start Over | Run Code

```python
import pandas as pd

births = pd.read_csv("data/pa_maternal_infant.csv")

both = births[
    (births["children_in_poverty_pct"] > 20) & (births["infant_mortality_per_1000"] > 7)
]

print(both[["county", "children_in_poverty_pct", "infant_mortality_per_1000"]])
```

⚠ The parentheses are mandatory

`&` binds more tightly than `>`. Without parentheses, Python evaluates `20 & births[...]` first and raises `TypeError`. This is the most common pandas error after `KeyError`.

Resolution: parenthesise every comparison. `(a > 1) & (b < 2)`, never `a > 1 & b < 2`.

`.assign()`, `.rename()`, and `.drop()` change the columns

Python Code | Start Over | Run Code

```python
import pandas as pd

overdose = pd.read_csv("data/pa_overdose_county_year.csv")
o2023 = overdose[overdose["year"] == 2023]

checked = o2023.assign(rate_check=lambda d: d["overdose_deaths"] / d["population"] * 100_000)
tidy = checked.rename(columns={"rate_per_100k": "rate_published"}).drop(columns=["year"])

print(tidy.head().round(2))
```

`.assign()` adds a derived column. `.rename(columns={old: new})` renames. `.drop(columns=[...])` removes. **All three return a new DataFrame** and leave the original unchanged.

> ⚠ **`.drop()` needs `columns=`**
>
> `.drop("year")` alone raises `KeyError`, because `.drop()` removes a *row* by default.

`.groupby()` splits, applies, and combines

Python Code | Start Over | Run Code

```python
import pandas as pd

overdose = pd.read_csv("data/pa_overdose_county_year.csv")

by_year = overdose.groupby("year")["rate_per_100k"].agg(["count", "mean", "median"])

print(by_year.round(2))
print()
print(by_year.sort_values("mean", ascending=False).round(2).head(2))
```

`.groupby(column)` splits the rows into groups, applies the aggregation to each, and combines the results into a table indexed by the grouping value. `.sort_values()` orders rows, and **every column travels with its row**.

> **`count` is the denominator**
>
> `.agg()` takes several functions at once. `count` reports how many rows carried a value in each group — the number the mean and median were computed over.

`.merge()` attaches a second table's columns

Python Code | Start Over | Run Code

```
1  import pandas as pd
2
3  births = pd.read_csv("data/pa_maternal_infant.csv")
4  counties = pd.read_csv("data/pa_counties.csv")
5
6  print(counties["metro"].value_counts())
7
8  joined = births.merge(counties, on="county", how="left")
9
10 print(f"births: {births.shape}   counties: {counties.shape}   joined: {joined.shape}")
11 print(joined[["county", "infant_mortality_per_1000", "region", "metro"]].head(3))
```

`on=` names the **key** column. The result carries the columns of both tables.

The grouping variables live in a different file

Neither `region` nor `metro` appears in any outcome file. **Every grouped comparison by place is possible only after a join.** Compare the row count before and after every join.

Missing is not zero

Python Code | Start Over | Run Code

```python
import pandas as pd

births = pd.read_csv("data/pa_maternal_infant.csv")

print(births.isna().sum())
print()

reported = births["infant_mortality_per_1000"]
print(f"Mean, skipping missing:  {reported.mean():.2f}  over {reported.count()} counties")
print(f"Mean, filling with zero: {reported.fillna(0).mean():.2f}  over {reported.size} counties")
```

`.isna().sum()` counts the gaps in every column. Filling with zero asserts that **no infant died** in those twenty-three counties. The state withheld the figure precisely because it is not zero.

> **`.fillna()` writes a claim into the data**
>
> Every fill value is an assertion about the world. Zero asserts absence. The mean asserts the county is typical. Neither is neutral, and both belong in the writeup.

Grouped rates are biased downward by suppression

Python Code | Start Over | Run Code

```python
import pandas as pd

overdose = pd.read_csv("data/pa_overdose_county_year.csv")
counties = pd.read_csv("data/pa_counties.csv")

o2023 = overdose[overdose["year"] == 2023].merge(counties, on="county", how="left")

def rate(g):
    return g["overdose_deaths"].sum() / g["population_2023"].sum() * 100_000

naive = o2023.groupby("region").apply(rate, include_groups=False)
correct = o2023.dropna(subset=["overdose_deaths"]).groupby("region").apply(rate, include_groups=False)

print(pd.DataFrame({"naive": naive.round(1), "correct": correct.round(1)}))
```

⚠ **`.sum()` skips `NaN`, but the denominator counts every county**

A suppressed county contributes its **population** and none of its **deaths**. The rate is therefore too low, and no error is raised.

`.melt()` and `.pivot()` reshape a table

Python Code | Start Over | Run Code

```python
import pandas as pd

wide = pd.DataFrame({
    "county": ["Allegheny", "Philadelphia", "Centre"],
    "2021": [58.5, 82.4, 7.6],
    "2022": [56.6, 91.4, 8.9],
    "2023": [54.7, 84.8, 4.4],
})

long = wide.melt(id_vars="county", var_name="year", value_name="rate_per_100k")
long = long.astype({"year": int})
print(long.head(4))

back = long.pivot(index="county", columns="year", values="rate_per_100k")
print(back)
```

`.melt()` : wide → long. One row per county-year, with the year as data. **`.pivot()` : long → wide**, the reverse.

Long form is what `.groupby()` and plotting expect

Published tables arrive wide because wide is readable; analysis wants long because long is groupable.

`pd.to_datetime()` makes a date column behave like a date

Python Code | Start Over | Run Code

```
1  import pandas as pd
2
3  covid = pd.read_csv("data/pa_covid_monthly.csv")
4  covid["month"] = pd.to_datetime(covid["month"])
5
6  by_month = covid.groupby("month")["covid_deaths"].sum()
7
8  print(f"Months covered: {by_month.size}")
9  print(f"Peak month:     {by_month.idxmax().date()}  ({by_month.max()} deaths)")
10 print()
11 print(covid.groupby(covid["month"].dt.year)["covid_deaths"].sum())
```

A text column sorts alphabetically. A datetime column sorts chronologically and exposes `.dt.year`, `.dt.month`, and `.dt.day_name()`.

The twenty-three.

Which counties, and what they share.

Exercise: who is missing, and what do they share?

Exercise | Start Over | Show Hint | Run Code

```
1  import pandas as pd
2
3  births = pd.read_csv("data/pa_maternal_infant.csv")
4  counties = pd.read_csv("data/pa_counties.csv")
5
6  joined = births.merge(counties, on="county", how="______")
7
8  joined["suppressed"] = joined["infant_mortality_per_1000"].isna()
9
10 print(f"Suppressed counties: {joined['suppressed'].sum()} of {len(joined)}")
11 print()
12 print(joined.groupby("suppressed")["population_2023"].median())
13 print()
14 print(pd.crosstab(joined["metro"], joined["suppressed"]))
```

`pd.crosstab(a, b)` counts one categorical against another.

Twenty-three rural counties

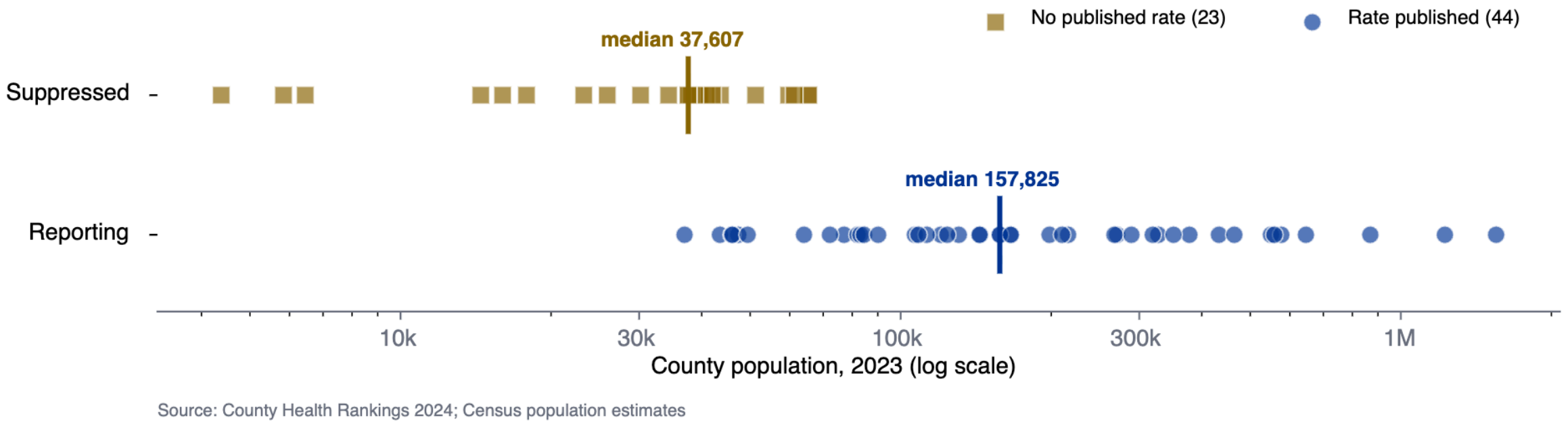


Figure 2. Each mark is one county, positioned by population on a logarithmic scale. The twenty-three counties with no published infant mortality rate, gold squares on the upper row, sit almost entirely below 70,000 residents with a median of 37,607. The forty-four counties that do publish a rate, blue circles on the lower row, run from about 45,000 to well over a million with a median of 157,825. The two groups barely overlap.

Twenty-three of Pennsylvania's sixty-seven counties have no published infant mortality rate. Eighteen of them are nonmetro. Median population: **37,607**, against **157,825** for the counties that report.

Challenge: what does the rural rate describe?

Group the joined table by metro status and report how many counties stand behind each mean.

Exercise | Start Over | Show Hint | Run Code

```
by_metro = joined.groupby("metro")["infant_mortality_per_1000"].agg(["size", "______", "mean"])

print(by_metro.round(2))
```

Your turn!

Nonmetro infant mortality is 7.38 per 1,000 against 5.68 for metro. How many nonmetro counties does 7.38 describe, and **which** ones are left out? Is 7.38 the rate of rural Pennsylvania?

Session outcomes

1. Reading a table with `pd.read_csv()`; **Series**, **DataFrame**, and **Index**
2. Inspecting with `.head()`, `.info()`, `.describe()`, `.shape`, and `.dtypes`
3. Selecting with `[]`, `.loc` by label, and `.iloc` by position
4. Filtering with boolean masks and **`&` and `|` with parentheses**
5. Changing columns with `.assign()`, `.rename()`, and `.drop()`
6. Summarising with **`.groupby()`** and `.agg()`; ordering with `.sort_values()`
7. Joining with **`.merge()`** on a shared key, and checking the row count
8. Handling missing data with `.isna()`, `.dropna()`, and `.fillna()` — **missing is not zero**
9. Reshaping with `.melt()` and `.pivot()`; dates with `pd.to_datetime()` and `.dt`

Preparation for the lab, Thursday 11/5

The lab is **six days after this lecture, across Election Day.** It restates each construct before exercising it, so returning to it cold is expected and planned for.

Module 5 turns these tables into figures.

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

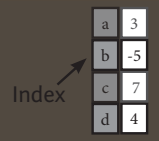
Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

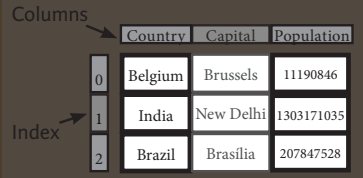
Series

A **one-dimensional** labeled array capable of holding any data type




```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



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

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasília'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
-5

>>> df[1:]
   Country     Capital  Population
1   India   New Delhi  1303171035
2  Brazil    Brasília   207847528
```

Get one element

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0],[0]]
'Belgium'
>>> df.iat([0],[0])
'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc[[0], ['Country']]
'Belgium'
>>> df.at([0], ['Country'])
'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
Country      Brazil
Capital    Brasília
Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:,'Capital']
0     Brussels
1    New Delhi
2     Brasília
```

Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
'New Delhi'
```

Select rows and columns

Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1)  |  (s > 2)]
>>> df[df['Population']>1200000000]
```

Series `s` where value is not >1
`s` where value is <-1 or >2
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```

Set index `a` of Series `s` to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)
Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis
Sort by the values along an axis
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows,columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idxmin()/df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cummulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
a    10.0
b     NaN
c     5.0
d     7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
a    10.0
b    -5.0
c    5.0
d    7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> pd.to_sql('myDf', engine)
```

DataCamp
Learn Python for Data Science Interactively

Python For Data Science *Cheat Sheet*

Pandas

Learn Python for Data Science **Interactively** at www.DataCamp.com

Reshaping Data

Pivot

```
>>> df3= df2.pivot(index='Date',
                   columns='Type',
                   values='Value')
```

Spread rows into columns

	Date	Type	Value
0	2016-03-01	a	11.432
1	2016-03-02	b	13.031
2	2016-03-01	c	20.784
3	2016-03-03	a	99.906
4	2016-03-02	a	1.303
5	2016-03-03	c	20.784

Type	a	b	c
Date			
2016-03-01	11.432	NaN	20.784
2016-03-02	1.303	13.031	NaN
2016-03-03	99.906	NaN	20.784

Pivot Table

```
>>> df4 = pd.pivot_table(df2,
                         values='Value',
                         index='Date',
                         columns='Type'])
```

Spread rows into columns

Stack / Unstack

```
>>> stacked = df5.stack()
>>> stacked.unstack()
```

Pivot a level of column labels
Pivot a level of index labels

		0	1
1	5	0.233482	0.390959
2	4	0.184713	0.237102
3	3	0.433522	0.429401

Unstacked

1	5	0	0.233482
		1	0.390959
2	4	0	0.184713
		1	0.237102
3	3	0	0.433522
		1	0.429401

Stacked

Melt

```
>>> pd.melt(df2,
            id_vars=["Date"],
            value_vars=["Type", "Value"],
            value_name="Observations")
```

Gather columns into rows

	Date	Type	Value
0	2016-03-01	a	11.432
1	2016-03-02	b	13.031
2	2016-03-01	c	20.784
3	2016-03-03	a	99.906
4	2016-03-02	a	1.303
5	2016-03-03	c	20.784

	Date	Variable	Observations
0	2016-03-01	Type	a
1	2016-03-02	Type	b
2	2016-03-01	Type	c
3	2016-03-03	Type	a
4	2016-03-02	Type	a
5	2016-03-03	Type	c
6	2016-03-01	Value	11.432
7	2016-03-02	Value	13.031
8	2016-03-01	Value	20.784
9	2016-03-03	Value	99.906
10	2016-03-02	Value	1.303
11	2016-03-03	Value	20.784

Iteration

```
>>> df.iteritems()
>>> df.iterrows()
```

(Column-index, Series) pairs
(Row-index, Series) pairs

Advanced Indexing

Also see NumPy Arrays

Selecting

```
>>> df3.loc[:,(df3>1).any()]
>>> df3.loc[:,(df3>1).all()]
>>> df3.loc[:,df3.isnull().any()]
>>> df3.loc[:,df3.notnull().all()]
```

Select cols with any vals >1
Select cols with vals > 1
Select cols with NaN
Select cols without NaN

Indexing With isin

```
>>> df[(df.Country.isin(df2.Type))]
>>> df3.filter(items="a","b"])
>>> df.select(lambda x: not x%5)
```

Find same elements
Filter on values
Select specific elements

Where

```
>>> s.where(s > 0)
```

Subset the data

Query

```
>>> df6.query('second > first')
```

Query DataFrame

Setting/Resetting Index

```
>>> df.set_index('Country')
>>> df4 = df.reset_index()
>>> df = df.rename(index=str,
                   columns={"Country":"cntry",
                            "Capital":"cptl",
                            "Population":"ppltn"})
```

Set the index
Reset the index
Rename DataFrame

Reindexing

```
>>> s2 = s.reindex(['a','c','d','e','b'])
```

Forward Filling

```
>>> df.reindex(range(4),
               method='ffill')
   Country     Capital   Population
0  Belgium    Brussels    11190846
1    India   New Delhi  1303171035
2   Brazil    Brasília   207847528
3   Brazil    Brasília   207847528
```

Backward Filling

```
>>> s3 = s.reindex(range(5),
                   method='bfill')
0  3
1  3
2  3
3  3
4  3
```

MultiIndexing

```
>>> arrays = [np.array([1,2,3]),
              np.array([5,4,3])]
>>> df5 = pd.DataFrame(np.random.rand(3, 2), index=arrays)
>>> tuples = list(zip(*arrays))
>>> index = pd.MultiIndex.from_tuples(tuples,
                                      names=['first', 'second'])
>>> df6 = pd.DataFrame(np.random.rand(3, 2), index=index)
>>> df2.set_index(["Date", "Type"])
```

Duplicate Data

```
>>> s3.unique()
>>> df2.duplicated('Type')
>>> df2.drop_duplicates('Type', keep='last')
>>> df.index.duplicated()
```

Return unique values
Check duplicates
Drop duplicates
Check index duplicates

Grouping Data

Aggregation

```
>>> df2.groupby(by=['Date','Type']).mean()
>>> df4.groupby(level=0).sum()
>>> df4.groupby(level=0).agg({'a':lambda x:sum(x)/len(x),
                              'b': np.sum})
```

Transformation

```
>>> customSum = lambda x: (x+x%2)
>>> df4.groupby(level=0).transform(customSum)
```

Missing Data

```
>>> df.dropna()
>>> df3.fillna(df3.mean())
>>> df2.replace("a", "f")
```

Drop NaN values
Fill NaN values with a predetermined value
Replace values with others

Combining Data

data1

	X1	X2
a	11.432
b	1.303
c	99.906

data2

	X1	X3
a	20.784
b	NaN
d	20.784

Merge

```
>>> pd.merge(data1,
             data2,
             how='left',
             on='X1')
```

	X1	X2	X3
a	11.432	20.784
b	1.303	NaN
c	99.906	NaN

```
>>> pd.merge(data1,
             data2,
             how='right',
             on='X1')
```

	X1	X2	X3
a	11.432	20.784
b	1.303	NaN
d	NaN	20.784

```
>>> pd.merge(data1,
             data2,
             how='inner',
             on='X1')
```

	X1	X2	X3
a	11.432	20.784
b	1.303	NaN

```
>>> pd.merge(data1,
             data2,
             how='outer',
             on='X1')
```

	X1	X2	X3
a	11.432	20.784
b	1.303	NaN
c	99.906	NaN
d	NaN	20.784

Join

```
>>> data1.join(data2, how='right')
```

Concatenate

Vertical

```
>>> s.append(s2)
```

Horizontal/Vertical

```
>>> pd.concat([s,s2],axis=1, keys=['One','Two'])
>>> pd.concat([data1, data2], axis=1, join='inner')
```

Dates

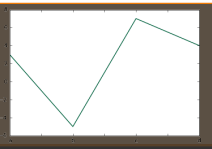
```
>>> df2['Date']= pd.to_datetime(df2['Date'])
>>> df2['Date']= pd.date_range('2000-1-1',
                               periods=6,
                               freq='M')
>>> dates = [datetime(2012,5,1), datetime(2012,5,2)]
>>> index = pd.DatetimeIndex(dates)
>>> index = pd.date_range(datetime(2012,2,1), end, freq='BM')
```

Visualization

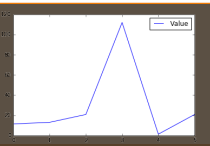
Also see Matplotlib

```
>>> import matplotlib.pyplot as plt
```

```
>>> s.plot()
>>> plt.show()
```



```
>>> df2.plot()
>>> plt.show()
```



DataCamp
Learn Python for Data Science Interactively

That’s all folks!

See you on Thursday — no class Tuesday, it’s Election Day.