# CSCI2202: Lecture 8 Dataframes and Visualisation

Finlay Maguire (finlay.maguire@dal.ca)
TA: Ehsan Baratnezhad (ethan.b@dal.ca)
TA: Precious Osadebamwen (precious.osadebamwen@dal.ca)

# Overview

- Tidy Data
- Data formats
- Pandas
    - Series
        - Indices & Accessing Items
        - Applying functions
        - Boolean Masks
        - Missing values
    - Combining Series in DataFrames
        - Selecting rows and columns
        - Filtering values
        - Grouping
        - Merge/concatenation
    - Plotting in pandas

# How to format data!

# Lots of data is messy and hard to parse programmatically

Australian Bureau of Statistics

Australian Bureau of Statistics

Table junk

**1800.0 Australian Marriage Law Postal Survey, 2017**

Released on 15 November 2017

**Table 5 Participation by Federal Electoral Division(a), Males and Age** Gender apartheid

| Yeah NA | | 18-19 years | 20-24 years | 25-29 years | 30-34 years | 35-39 years | 40-44 years | 45-49 years | 50-54 years | 55-59 years | 60-64 years |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total participants | 292 | 1,058 | 1,465 | 1,653 | 1,515 | 1,516 | 1,710 | 1,730 | 1,753 | 1,574 |
| Lingiari(c) | Eligible participants | 572 | 2,910 | 3,789 | 3,996 | 3,607 | 3,506 | 3,645 | 3,331 | 2,960 | 2,456 |
| Primary keynotes | Participation rate (%) | 51.0 | 36.4 | 38.7 | 41.4 | 42.0 | 43.2 | 46.9 | 51.9 | 59.2 | 64.1 |
| | | | | Comma on | | | | | | | |
| Merged cells | Total participants | 442 | 1,461 | 2,066 | 2,357 | 2,188 | 2,057 | 2,224 | 2,108 | 2,134 | 1,772 |
| Solomon | Eligible participants | 750 | 2,991 | 3,994 | 4,155 | 3,634 | 3,398 | 3,427 | 3,066 | 2,931 | 2,355 |
| | Participation rate (%) | 58.9 | 48.8 | 51.7 | 56.7 | 60.2 | 60.5 | 64.9 | 68.8 | 72.8 | 75.2 |
| | | | | | | | | | | | |
| | Total participants | 734 | 2,519 | 3,531 | 4,010 | 3,703 | 3,573 | 3,934 | 3,838 | 3,887 | 3,346 |
| *Northern Territory (Total)* | Eligible participants | 1,322 | 5,901 | 7,783 | 8,151 | 7,241 | 6,904 | 7,072 | 6,397 | 5,891 | 4,811 |
| | Participation rate (%) | 55.5 | 42.7 | 45.4 | 49.2 | 51.1 | 51.8 | 55.6 | 60.0 | 66.0 | 69.5 |
| | | | | Summary of data inside data | | | | | | | |
| **Australian Capital Territory Divisions** | Covariate as Subheading | | | | | | | | | | |
| | Total participants | 1,764 | 4,789 | 4,817 | 4,973 | 4,626 | 4,453 | 5,074 | 4,826 | 5,169 | 4,394 |
| Canberra(d) | Eligible participants | 2,260 | 6,471 | 6,448 | 6,509 | 5,983 | 5,805 | 6,302 | 5,902 | 6,044 | 5,057 |
| | Participation rate (%) | 78.1 | 74.0 | 74.7 | 76.4 | 77.3 | 76.7 | 80.5 | 81.8 | 85.5 | 86.9 |
| | | | | | | | | | | | |
| | Total participants | 1,477 | 4,687 | 5,178 | 5,786 | 6,025 | 5,463 | 5,191 | 4,208 | 3,948 | 3,465 |
| Fenner(e) | Eligible participants | 1,904 | 6,354 | 7,121 | 7,822 | 7,960 | 7,155 | 6,480 | 5,206 | 4,692 | 3,945 |
| | Participation rate (%) | 77.6 | 73.8 | 72.7 | 74.0 | 75.7 | 76.4 | 80.1 | 80.8 | 84.1 | 87.8 |
| | | | | NA Yeah | | | | | | | |
| | Total participants | 3,241 | 9,476 | 9,995 | 10,759 | 10,651 | 9,916 | 10,265 | 9,034 | 9,117 | 7,659 |
| *Australian Capital Territory (Total)* | Eligible participants | 4,164 | 12,825 | 13,569 | 14,331 | 13,943 | 12,960 | 12,782 | 11,108 | 10,736 | 9,002 |
| | Participation rate (%) | 77.8 | 73.9 | 73.7 | 75.1 | 76.4 | 76.5 | 80.3 | 81.3 | 84.9 | 87.3 |
| | | | | | | | | | | | |
| **Australia** | | | | | | | | | | | |
| | Total participants | 151,297 | 438,166 | 441,658 | 460,548 | 462,206 | 479,360 | 524,620 | 517,693 | 543,449 | 506,799 |
| *Total* | Eligible participants | 201,439 | 635,909 | 646,916 | 665,250 | 656,446 | 660,841 | 693,850 | 659,150 | 664,720 | 597,386 |
| | Participation rate (%) | 75.1 | 68.9 | 68.3 | 69.2 | 70.4 | 72.5 | 75.6 | 78.5 | 81.8 | 84.8 |

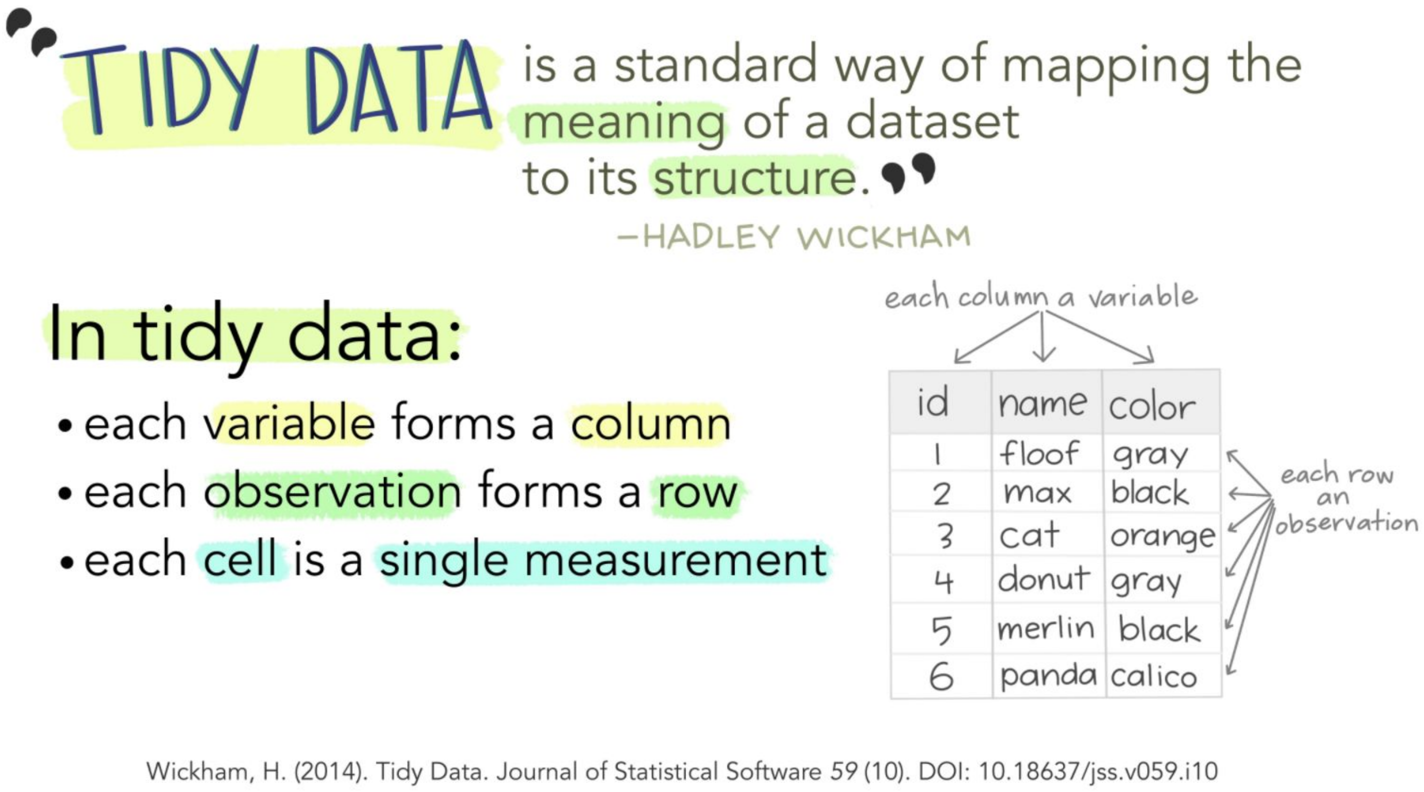
(a) The Federal Electoral Divisions are current as at 24 August 2017
(b) Includes those whose age is unknown
(c) Includes Christmas Island and the Cocos (Keeling) Islands
(d) Includes Norfolk Island
(e) Includes Jervis Bay

Return of the table junk

MS Excel or Die

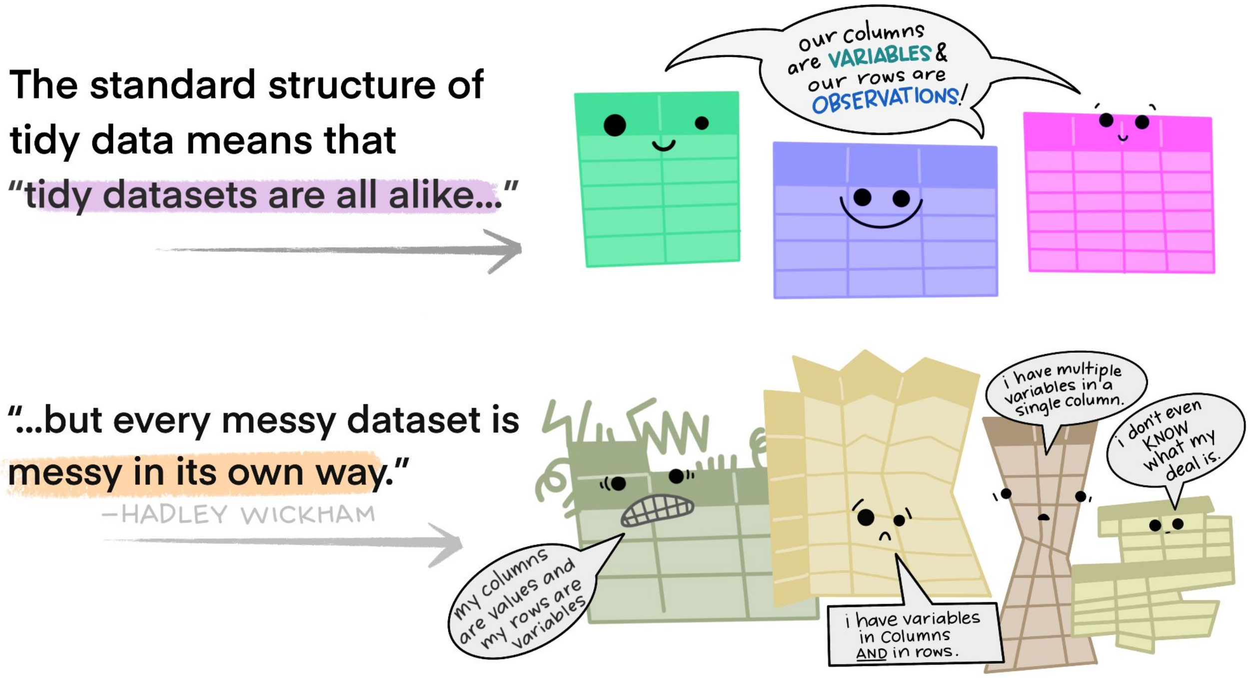
*https://medium.com/@miles.mcbain/tidying-the-australian-same-sex-marriage-postal-survey-data-with-r-5d35cea07962*

# Tidy data is a standardised way of formatting data



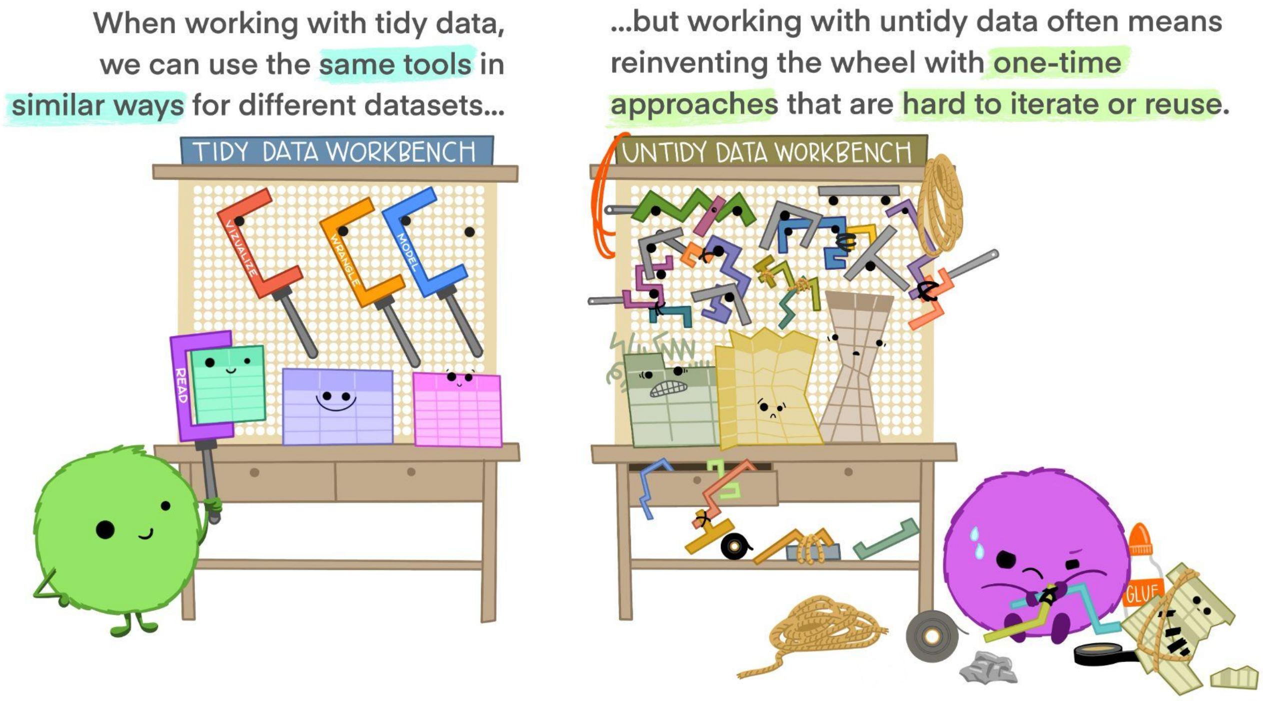

*Illustrations from the [Openscapes]() blog [Tidy Data for reproducibility, efficiency, and collaboration]() by Julia Lowndes and Allison Horst"*

# Standardised data enables standardised code



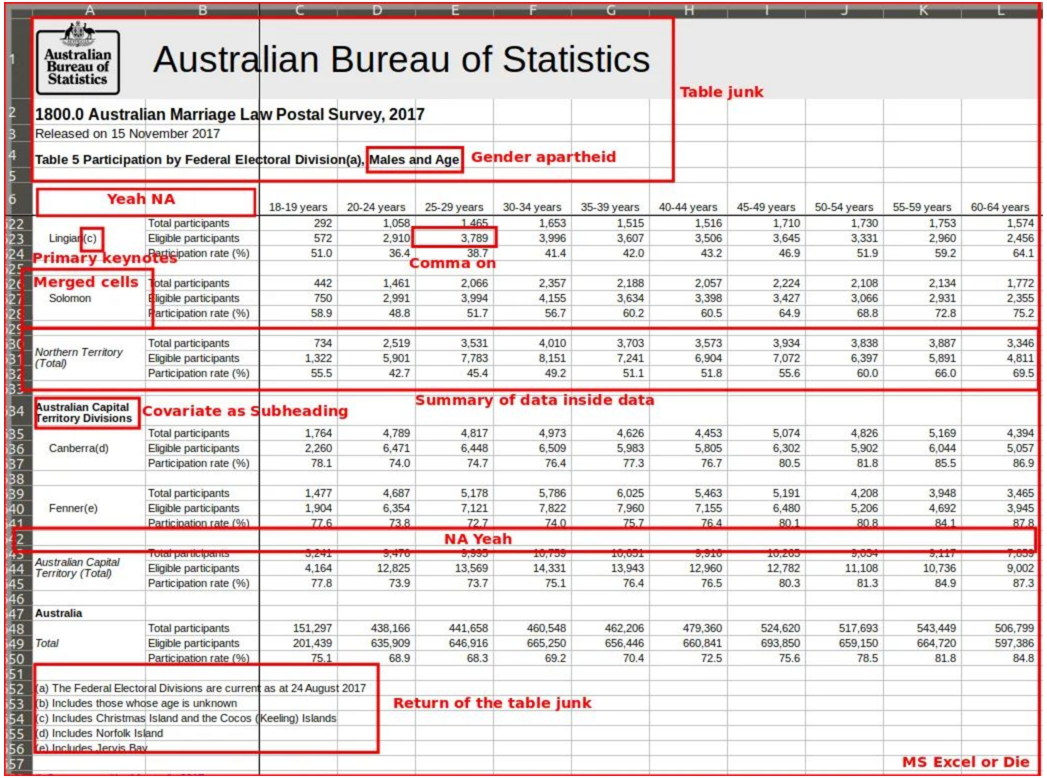

*Illustrations from the [Openscapes](#) blog [Tidy Data for reproducibility, efficiency, and collaboration](#) by Julia Lowndes and Allison Horst"*

# Standardised data enables standardised code




*Illustrations from the [Openscapes](#) blog [Tidy Data for reproducibility, efficiency, and collaboration](#) by Julia Lowndes and Allison Horst"*

# Example of untidy -> tidy data




Australian Bureau of Statistics

Australian Bureau of Statistics

1800.0 Australian Marriage Law Postal Survey, 2017

Released on 15 November 2017

**Table 5 Participation by Federal Electoral Division(a), Males and Age**

| | | | 18-19 years | 20-24 years | 25-29 years | 30-34 years | 35-39 years | 40-44 years | 45-49 years | 50-54 years | 55-59 years | 60-64 years |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 522 | | Total participants | 292 | 1,058 | 1,465 | 1,653 | 1,515 | 1,516 | 1,710 | 1,730 | 1,753 | 1,574 |
| 523 | Lingiari(c) | Eligible participants | 572 | 2,910 | 3,789 | 3,996 | 3,607 | 3,506 | 3,645 | 3,331 | 2,960 | 2,456 |
| 524 | | Participation rate (%) | 51.0 | 36.4 | 38.7 | 41.4 | 42.0 | 43.2 | 46.9 | 51.9 | 59.2 | 64.1 |
| 525 | | | | | | | | | | | | |
| 526 | | Total participants | 442 | 1,461 | 2,066 | 2,357 | 2,188 | 2,057 | 2,224 | 2,108 | 2,134 | 1,772 |
| 527 | Solomon | Eligible participants | 750 | 2,991 | 3,994 | 4,155 | 3,634 | 3,398 | 3,427 | 3,066 | 2,931 | 2,355 |
| 528 | | Participation rate (%) | 58.9 | 48.8 | 51.7 | 56.7 | 60.2 | 60.5 | 64.9 | 68.8 | 72.8 | 75.2 |
| 529 | | | | | | | | | | | | |
| 530 | *Northern Territory (Total)* | Total participants | 734 | 2,519 | 3,531 | 4,010 | 3,703 | 3,573 | 3,934 | 3,838 | 3,887 | 3,346 |
| 531 | | Eligible participants | 1,322 | 5,901 | 7,783 | 8,151 | 7,241 | 6,904 | 7,072 | 6,397 | 5,891 | 4,811 |
| 532 | | Participation rate (%) | 55.5 | 42.7 | 45.4 | 49.2 | 51.1 | 51.8 | 55.6 | 60.0 | 66.0 | 69.5 |
| 533 | | | | | | | | | | | | |
| 534 | **Australian Capital Territory Divisions** | | | | | | | | | | | |
| 535 | | Total participants | 1,764 | 4,789 | 4,817 | 4,973 | 4,626 | 4,453 | 5,074 | 4,826 | 5,169 | 4,394 |
| 536 | Canberra(d) | Eligible participants | 2,260 | 6,471 | 6,448 | 6,509 | 5,983 | 5,805 | 6,302 | 5,902 | 6,044 | 5,057 |
| 537 | | Participation rate (%) | 78.1 | 74.0 | 74.7 | 76.4 | 77.3 | 76.7 | 80.5 | 81.8 | 85.5 | 86.9 |
| 538 | | | | | | | | | | | | |
| 539 | | Total participants | 1,477 | 4,687 | 5,178 | 5,786 | 6,025 | 5,463 | 5,191 | 4,208 | 3,948 | 3,465 |
| 540 | Fenner(e) | Eligible participants | 1,904 | 6,354 | 7,121 | 7,822 | 7,960 | 7,155 | 6,480 | 5,206 | 4,692 | 3,945 |
| 541 | | Participation rate (%) | 77.6 | 73.8 | 72.7 | 74.0 | 75.7 | 76.4 | 80.1 | 80.8 | 84.1 | 87.8 |
| 542 | | | | | | | | | | | | |
| 543 | | Total participants | 3,241 | 9,476 | 9,995 | 10,759 | 10,651 | 9,916 | 10,265 | 9,034 | 9,117 | 7,859 |
| 544 | *Australian Capital Territory (Total)* | Eligible participants | 4,164 | 12,825 | 13,569 | 14,331 | 13,943 | 12,960 | 12,782 | 11,108 | 10,736 | 9,002 |
| 545 | | Participation rate (%) | 77.8 | 73.9 | 73.7 | 75.1 | 76.4 | 76.5 | 80.3 | 81.3 | 84.9 | 87.3 |
| 546 | | | | | | | | | | | | |
| 547 | **Australia** | | | | | | | | | | | |
| 548 | | Total participants | 151,297 | 438,166 | 441,658 | 460,548 | 462,206 | 479,360 | 524,620 | 517,693 | 543,449 | 506,799 |
| 549 | *Total* | Eligible participants | 201,439 | 635,909 | 646,916 | 665,250 | 656,446 | 660,841 | 693,850 | 659,150 | 664,720 | 597,386 |
| 550 | | Participation rate (%) | 75.1 | 68.9 | 68.3 | 69.2 | 70.4 | 72.5 | 75.6 | 78.5 | 81.8 | 84.8 |

(a) The Federal Electoral Divisions are current as at 24 August 2017
(b) Includes those whose age is unknown
(c) Includes Christmas Island and the Cocos (Keeling) Islands
(d) Includes Norfolk Island
(e) Includes Jervis Bay

*https://medium.com/@miles.mcbain/tidying-the-australian-same-sex-marriage-postal-survey-data-with-r-5d35cea07962*

# Example of untidy -> tidy data

| | city | gender | age | state | area_sq_km |
|---|---|---|---|---|---|
| **0** | Adelaide | Female | 18-19 years | SA | 76 |
| **1** | Adelaide | Female | 20-24 years | SA | 76 |
| **2** | Adelaide | Female | 25-29 years | SA | 76 |
| **3** | Adelaide | Female | 30-34 years | SA | 76 |
| **4** | Adelaide | Female | 35-39 years | SA | 76 |
| **5** | Adelaide | Female | 40-44 years | SA | 76 |
| **6** | Adelaide | Female | 45-49 years | SA | 76 |
| **7** | Adelaide | Female | 50-54 years | SA | 76 |
| **8** | Adelaide | Female | 55-59 years | SA | 76 |
| **9** | Adelaide | Female | 60-64 years | SA | 76 |

*https://medium.com/@miles.mcbain/tidying-the-australian-same-sex-marriage-postal-survey-data-with-r-5d35cea07962*

# “Tidy data” is “long” instead of “wide”

| id | bp1 | bp2 |
|---|---|---|
| A | 100 | 120 |
| B | 140 | 115 |
| C | 120 | 125 |

Clean Wide Data

“Melting” →

← “Pivoting”

| id | measurement | value |
|---|---|---|
| A | bp1 | 100 |
| A | bp2 | 120 |
| B | bp1 | 140 |
| B | bp2 | 115 |
| C | bp1 | 120 |
| C | bp2 | 125 |

Tidy Long data

# Tidy Data


A data set is **tidy** iff:

1. Each **variable** is in its own **column**
2. Each **case** is in its own **row**
3. Each **value** is in its own **cell**

Data munging (and cleaning) is the hardest part of scientific data analysis

# How is data stored in files?

# Many different data file-formats

- Excel (.xls/.xlsx)
- Comma-separated values (.csv/.tsv)
- Javascript Object Notation (.json)
- Extensible Markup Language (.xml)
- Databases (SQL/noSQL)

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | city | gender | age | state | area_sq_km |
| 2 | Adelaide | Female | 18-19 years | SA | 76 |
| 3 | Adelaide | Female | 20-24 years | SA | 76 |
| 4 | Adelaide | Female | 25-29 years | SA | 76 |
| 5 | Adelaide | Female | 30-34 years | SA | 76 |
| 6 | Adelaide | Female | 35-39 years | SA | 76 |
| 7 | Adelaide | Female | 40-44 years | SA | 76 |
| 8 | Adelaide | Female | 45-49 years | SA | 76 |
| 9 | Adelaide | Female | 50-54 years | SA | 76 |
| 10 | Adelaide | Female | 55-59 years | SA | 76 |
| 11 | Adelaide | Female | 60-64 years | SA | 76 |


```
city,gender,age,state,area_sq_km
Adelaide,Female,18-19 years,SA,76
Adelaide,Female,20-24 years,SA,76
Adelaide,Female,25-29 years,SA,76
Adelaide,Female,30-34 years,SA,76
Adelaide,Female,35-39 years,SA,76
Adelaide,Female,40-44 years,SA,76
Adelaide,Female,45-49 years,SA,76
Adelaide,Female,50-54 years,SA,76
Adelaide,Female,55-59 years,SA,76
Adelaide,Female,60-64 years,SA,76
```

```
{
   "city": {
      "0": "Adelaide",
      "2": "Adelaide"
   },
   "gender": {
      "0": "Female",
      "2": "Female"
   },
   "age": {
      "0": "18-19 years",
      "2": "25-29 years"
   },
```

```
<data>
  <row>
    <index>0</index>
    <city>Adelaide</city>
    <gender>Female</gender>
    <age>18-19 years</age>
    <state>SA</state>
    <area_sq_km>76</area_sq_km>
  </row>
  <row>
    <index>2</index>
    <city>Adelaide</city>
```


*Many other markup languages (.html/.yaml/.toml/.hcl/.ini)*

# Tabular Data in Python: Pandas

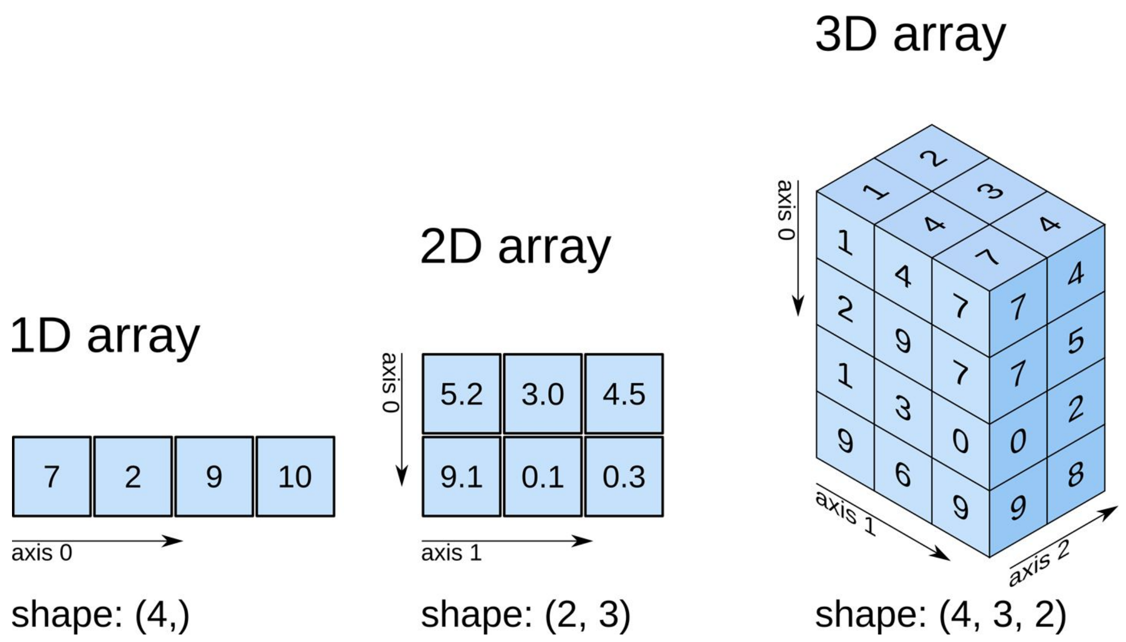
# Pandas

Comprehensive Python library for data manipulation and analysis, in particular tables and time series.

- Pandas data frames = tables
- Same concept as R data.frames/tibbles; STATA/SAS data sets; excel sheets
- Most widely used data library in python
- Supports interaction with CSV, SQL, JSON,  ...
- Integrates directly with rest of “PyData” ecosystem e.g., Jupyter, numpy, matplotlib

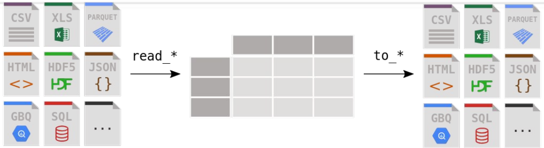
pandas.pydata.org

# Built on top of numpy: efficient arrays/matrices and maths!



# I/O of many formats built-in to pandas

| Format Type | Data Description | Reader | Writer |
|---|---|---|---|
| text | CSV | read_csv | to_csv |
| text | Fixed-Width Text File | read_fwf | NA |
| text | JSON | read_json | to_json |
| text | HTML | read_html | to_html |
| text | LaTeX | Styler.to_latex | NA |
| text | XML | read_xml | to_xml |
| text | Local clipboard | read_clipboard | to_clipboar |
| binary | MS Excel | read_excel | to_excel |




| Format Type | Data Description | Reader | Writer |
|---|---|---|---|
| binary | OpenDocument | read_excel | NA |
| binary | HDF5 Format | read_hdf | to_hdf |
| binary | Feather Format | read_feather | to_feather |
| binary | Parquet Format | read_parquet | to_parquet |
| binary | ORC Format | read_orc | to_orc |
| binary | Stata | read_stata | to_stata |
| binary | SAS | read_sas | NA |
| binary | SPSS | read_spss | NA |
| binary | Python Pickle Format | read_pickle | to_pickle |
| SQL | SQL | read_sql | to_sql |
| SQL | Google BigQuery;:ref:*read_gbq<io.bigquery>*;:ref:*to_gbq<io.bigquery>* | | |

*https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html*

# Other data manipulation libraries in Python

- Polars (pola.rs)
  - Recent multi-threaded rust-based python library
  - Mostly compatible with pandas syntax
  - Very fast for data that fits into your computer memory
- Dask (dask.org)
  - Multi-threaded package for distributed and out-of-core
  - Great for data bigger than memory and analyses across computers
  - Mostly compatible with pandas syntax
  - Dask scales from single machines to clusters
- PySpark (spark.apache.org)
  - Python interface to Apache Spark dataframes
  - Quite different interface
  - Built for massive datasets across large numbers of computers
- csv (docs.python.org/3/library/csv.html)
  - Standard library installed by default in python
  - Mostly just a parser for I/O of CSV files
  - Very slow

# How to use Pandas

# Series and DataFrames

Pandas provides two types of classes for handling data:

1. `Series`: a one-dimensional labeled array holding data of any type such as integers, strings, Python objects etc.

2. `DataFrame`: a two-dimensional data structure that holds data like a two-dimensional array or a table with rows and columns.

# Pandas Series: fast dictionary for storing 1-dimensional data

```
import pandas as pd

s_i = pd.Series([1, 3, 5])

s_i

      0    1

      1    3

      2    5

      dtype: int64
```

```
s_l = pd.Series([1, 'A', [1.0, 2.0]],

                index=['A', 'B', 'C']

                 name='text')

s_l

      A     1

      B     'A'

      C     [1.0, 2.0]

      Name: text dtype: object
```

```
print(s_l.at['C'])

[1.0, 2.0]

print(s_l.iat[2])

[1.0, 2.0]

print(s_l.loc[['A', 'B']])

      A     1

      B     A

      Name: text, dtype: object
```

- Basically a fancy dictionary that can be named and uses default positional index or a custom index
- Access values: `.at[index]`, `.iat[position]`, or `.loc[index or iterable of indices]`
- Other useful methods:
    - `.dtypes`: get the types of each item in Series
    - `.index`: get the index of the Series and `.values` get just the values in the Series
    - `.size`: get the number of values in the Series (i.e., `s_l.size` would return 3
    - `.shape`: get the shape of the Series (i.e., `s_l.shape` would return (3,)
    - `.name`: get the name (i.e., s_l.name would return "text")

# Series are mutable and have lots of useful methods

```
s_x = pd.Series({‘A’: 1, ‘B’: 2.0, ‘C’: 5},
                name=”nums”)
s_x
     A    1
     B    2.0
     C    5
     Name: nums dtype: object


print(s_x.sum(), s_x.mean(), s_x.median()
     s_x.max(), s_x.min(), s_x.std())
8.0 2.6665 2.0 5.0 1.0 2.0816
```

```
s_x.at[‘A’] = 10
s_x.at[‘A’]
10
s_x.loc[:] = [1.0, 1.0, 1.0]
s_x
     A    1.0
     B    1.0
     C    1.0
Name: nums, dtype: float64
```

Why float64 not just “float”? pandas uses more efficient types from numpy

# Applying functions to Series

```
s_x = pd.Series({'A': 1, 'B': 2.0, 'C': 5})

s_x = s_x.apply(lambda x: x+1)

s_x

A     2.0

B     3.0

C     6.0

dtype: float64
```

```
s_x / 2  # same as s_x.div(2)

A     0.5

B     1.0

C     2.5

Name: nums, dtype: float64

pd.Series({'A': 1, 'C': "text"}) * 2

A          2

C     texttext

dtype: object
```

- Series implement methods that allow you to perform a variety of data manipulation operations (add, div, mult, power etc) and special methods that let you use operators.

# Many more functions associated with data

```
s_x = pd.Series({'A':5.0, 'B':1.0, 'C':3.0})

s_x = s_x.sort_values()

s_x

B     1.0

C     3.0

A     5.0

dtype: float64
```

```
s_y = pd.Series({'A':50, 'B':15, 'C':20})

s_y = s_y.rank()

A     3.0

B     1.0

C     2.0

dtype: float64
```

- Many other methods such as sort_values, rank, sample etc.

# A list of boolean values can be used to filter a Series

```
s_fil = pd.Series([1, 4, 10])
0     1
1     4
2     10
dtype: int64


s_fil[[True, False, False]]
0     1
dtype: int64
s_fil[~[True, True, False]]
2     10
dtype: int64
```

List of boolean values is sometimes called a “mask”

Commonly we apply a mask to filter values from a Series

~ works like `not` it flips the boolean values

```
bools = s_fil >= 4
bools
0     False
1     False
2     True
dtype: bool


s_fil[bools]
# alternatively: s_fil.loc[bools]
1     4
2     10
dtype: int64
```

# Missing values are important in real-world data

```
import numpy as np

s_nan = pd.Series([1, np.nan, 1e5])

0          1.0

1          NaN

2     100000.0

dtype: float64


s_nan.isna()

0     False

1     True

2     False

dtype: bool
```

With missing values we can:

- Drop them (`s_nan.dropna()` or `s_nan[~s_nan.isna()]`)
- Replace them (`s_nan.fillna(0)`)
- Live with them (make sure your code handles them appropriately though)

```
s_nan.dropna()

0          1.0

2     100000.0

dtype: float64
```

```
s_nan[~s_nan.isna()]

0          1.0

2     100000.0

dtype: float64
```

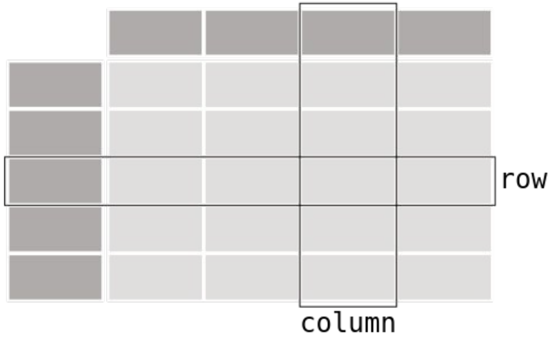
```
s_nan.fillna(0)

0          1.0

1          0.0

2     100000.0

dtype: float64
```

# Combining Series into a DataFrame

# A collection of Series (of equal length) form a DataFrame

```
 df = pd.DataFrame({'x': pd.Series([1,2]),
                   'y': pd.Series([3,4])})
df
     x     y
0    1     3
1    2     4


pd.DataFrame([[1, 2], [3, 4]],
             columns=['x', 'y'])
     x     y
0    1     3
1    2     4
```

```
pd.DataFrame({'x': [1,2],
              'y': [3,4]})
     x     y
0    1     3
1    2     4
```




Individual Columns **AND** Rows are Series

# pd.read_x for parsing data files to a DataFrame

- pd.read_csv(FILEPATH) - default delimiter/separator is a “,”
- Read TSVs with pd.read_csv(FILEPATH, sep=’\t’)
- Also pd.read_excel, pd.read_json, pd.read_xml etc

In [1]:
```python
import pandas as pd
students = pd.read_csv('students.csv')
students
```

Out[1]:

| | Name | City |
|---|---|---|
| 0 | Donald Duck | Copenhagen |
| 1 | Goofy | Aarhus |
| 2 | Mickey Mouse | Aarhus |

students.csv
```
Name,City
"Donald Duck","Copenhagen"
"Goofy","Aarhus"
"Mickey Mouse","Aarhus”
```

# Selecting columns and rows

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

countries['name'] # select column

countries.name # same as above

countries[['name', 'capital']] # select
multiple columns

countries.head(2) # first two rows

countries[1:3] # slicing rows, rows 1 and 2

countries[::2] # slicing rows,
```

| | Table: country | | | |
|---|---|---|---|---|
| | name | population | area | capital |
| 0 | 'Denmark' | 5748769 | 42931 | 'Copenhagen' |
| 1 | 'Germany' | 82800000 | 357168 | 'Berlin' |
| 2 | 'USA' | 325719178 | 9833520 | 'Washington, D.C.' |
| 3 | 'Iceland' | 334252 | 102775 | 'Reykjavik' |

# Selecting by labels

```
df = pd.read_csv('countries.csv')
countries.at[1] # first position row
pd.Series: Germany, 828000, 357165, ‘Berlin’
countries.at[0, area] # row, column
42931 # value
countries.loc[:, ‘capital’] # series capitals
countries.loc[[2,3], [‘name’, ‘capital’]]
     name           capital
2    USA            Washington, D.C.
3    Iceland        Reykjavik
```

Table: country

| | name | population | area | capital |
|---|---|---|---|---|
| 0 | 'Denmark' | 5748769 | 42931 | 'Copenhagen' |
| 1 | 'Germany' | 82800000 | 357168 | 'Berlin' |
| 2 | 'USA' | 325719178 | 9833520 | 'Washington, D.C.' |
| 3 | 'Iceland' | 334252 | 102775 | 'Reykjavik' |

Same as Series
Single value:
*df.at[row_index,*
*col_name(optional)]*
Multiple values:
*df.loc[>=1 row labels,*
*>=1 col names(optional)]*

# Creating new columns from existing columns

```
df['area_pp'] = df['area'] / df['population']

df.columns

['name', 'population', 'area', 'capital', area_pp]
```

| Table: country | | | |
|---|---|---|---|
| name | population | area | capital |
| 'Denmark' | 5748769 | 42931 | 'Copenhagen' |
| 'Germany' | 82800000 | 357168 | 'Berlin' |
| 'USA' | 325719178 | 9833520 | 'Washington, D.C.' |
| 'Iceland' | 334252 | 102775 | 'Reykjavik' |

# Filtering DataFrames: boolean masks

```
df = pd.read_csv('countries.csv')
df[df['area'] > 150000]
   name     population   area        capital
1 Germany   82800000     357168      Berlin
2 USA       325719178    9833520     Washington D.C.
df.loc[df['area'] > 15000, ['name', 'capital']]
     name        capital
1    Germany     Berlin
2    USA         Washington, D.C.,
```

| | Table: country | | | |
|---|---|---|---|---|
| | name | population | area | capital |
| 0 | 'Denmark' | 5748769 | 42931 | 'Copenhagen' |
| 1 | 'Germany' | 82800000 | 357168 | 'Berlin' |
| 2 | 'USA' | 325719178 | 9833520 | 'Washington, D.C.' |
| 3 | 'Iceland' | 334252 | 102775 | 'Reykjavik' |

# Applying functions to DataFrames (axis labels)

```
df = pd.DataFrame({'A': [1, 1],
                   'B': [1, 5], "C": [-1, -2]})

df

   A  B  C

0  1  1 -1

1  1  5 -2

df.sum()

A    2

B    6

C   -3

dtype: int64
```

```
df.sum(axis=1)

0    1

1    4

dtype: int64
```

Works same as with Series just does it automatically to all row or col Series

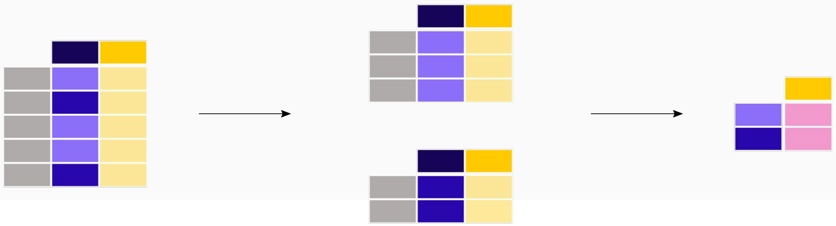
Most methods take an axis kwarg:

- Default is axis=0
- axis=0 means apply down cols
- axis=1 means apply across rows

# groupby - very powerful way to apply methods to groups

```
df = pd.DataFrame({
      "A": ["foo", "bar", "foo", "bar"],
       "B": [1, 5, 2, 5]})
df.groupby('A').sum()
A      B
bar   10
foo    3
df.groupby('A').nunique()
A     B
bar   1
foo   2
```

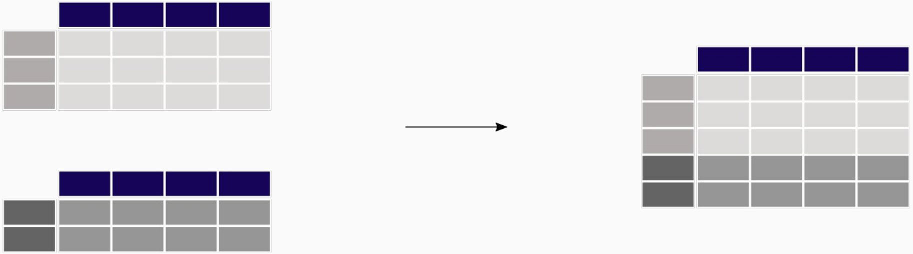
```
df = pd.DataFrame({"A": ["foo", "bar", "foo"],
                  "B": [1, 5, 2],
                 "C": [-1, -2, -5]})
df.groupby('A').mean()
A      B    C
bar   5.0 -2.0
foo   1.5 -3.0
```



# Combining DataFrames - concat

```
df1 = pd.DataFrame({'A': [1, 1], 'B': [1, 5],
                    "C": [-1, -2]})

df2 = df1 + 1

df2 = df2.drop('A', axis=1)

pd.concat([df1, df2])

   A   B  C

0   1   1 -1

1   1   5 -2

0  nan  2  0

1  nan  6 -1
```

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

   A  B  C  B  C

0  1  1 -1  2  0

1  1  5 -2  6 -1
```



- Note: concatenate will lead to repeated row/col labels unless they have different vals
- Missing values will be filled with np.nan unless you specify otherwise

# Fancier joins using labels intelligently: merge

```
left = pd.DataFrame({"key": ["foo", "bar"], "lval":
[1, 2]})

   key  lval

0  foo     1

1  foo     2

right = pd.DataFrame({"key": ["foo", "bar"],
"rval": [4, 5]})

   key  rval

0  foo     4

1  foo     5
```

```
pd.merge(left, right, on="key")

   key  lval  rval

0  foo     1     4

1  bar     2     5
```

**how kwarg:**

**left**: Use keys from left frame only

**right**:Use keys from right frame only

**outer**:Use union of keys from both

**inner**:Use intersection of keys from both

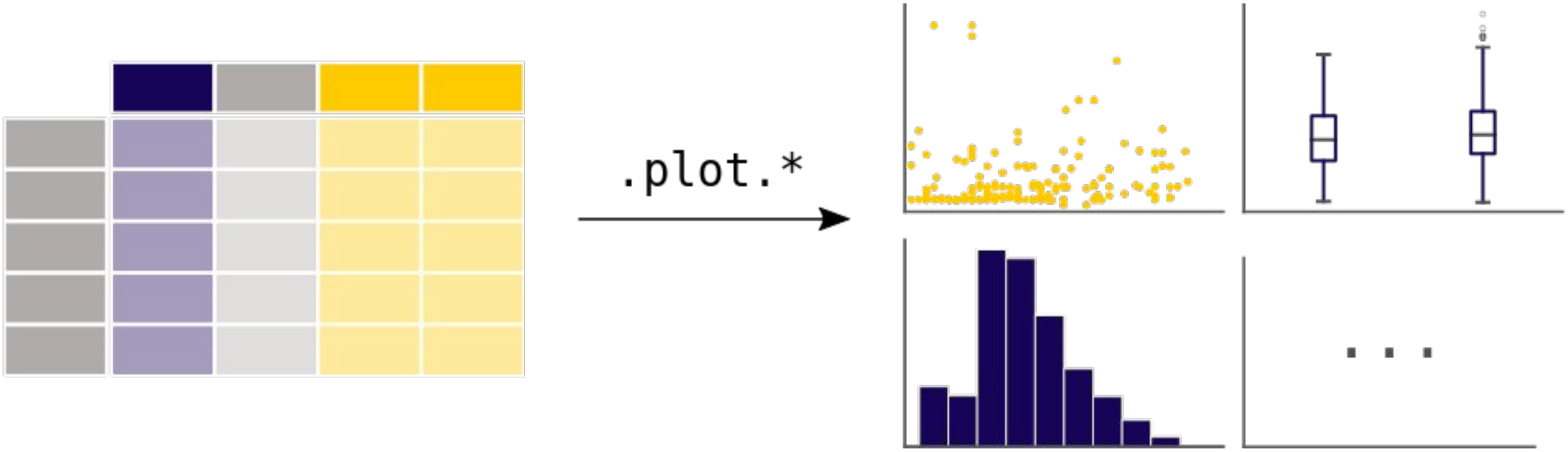
# Plotting data in python

*Data Visualisation can be and actually is an entire course (CSCI4166/6406)*

# Many plotting libraries

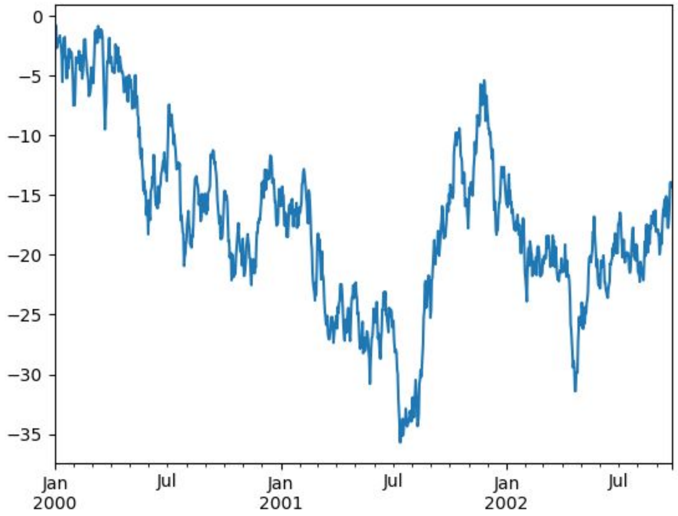
- Matplotlib
  - Most commonly used
  - Very customisable but quite manual (have to specify each element you want)
  - Object oriented and state based ways of using - confusing mix of methods in online documentation
- Seaborn
  - Higher-order plotting library (does a lot of things at once: like a grid of graphs)
  - Useful for scientific data - what I mainly use.
- Plotly
  - Great for interactive plots and dashboards
  - Contains a lot of javascript and a slightly different approach to data
- Plotnine
  - Copy of ggplot from R

# Pandas also has built-in plotting (using matplotlib)



# .plot will try and guess the appropriate plot for a Series (default lineplot)

```
ts = pd.Series(np.random.randn(1000),
                index=pd.date_range("1/1/2000",
                                    periods=1000))

ts = ts.cumsum()

ts.plot()
```

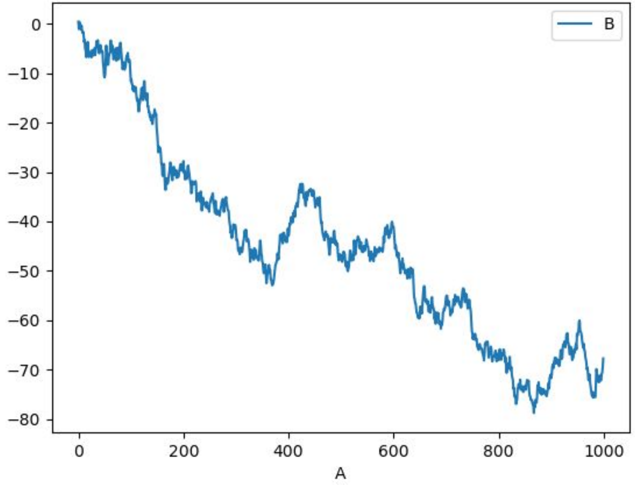


# .plot will guess a categorical variable to groupby

```
df = pd.DataFrame(np.random.randn(1000, 4),
                  index=ts.index,
                  columns=list("ABCD"))
df = df.cumsum()
df.plot()
```

# Pass in specific columns to plot against each other

```
df3 = pd.DataFrame(np.random.randn(1000, 2),
                   columns=["B", "C"]).cumsum()
df3["A"] = pd.Series(list(range(len(df))))
df3.plot(x="A", y="B");
```

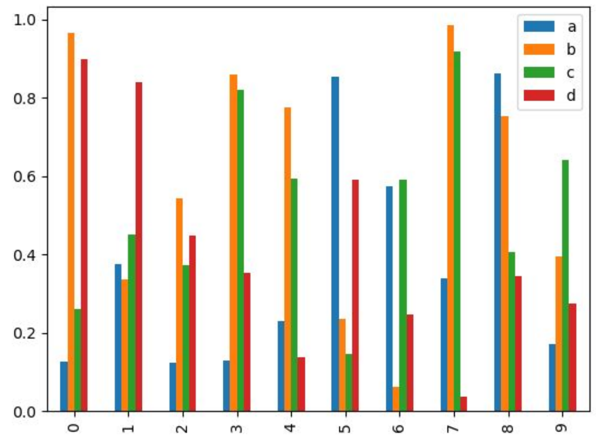


# Kind kwarg allows you to select plot type

Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the `kind` keyword argument to [`plot()`](), and include:

- 'bar' or 'barh' for bar plots
- 'hist' for histogram
- 'box' for boxplot
- 'kde' or 'density' for density plots
- 'area' for area plots
- 'scatter' for scatter plots
- 'hexbin' for hexagonal bin plots
- 'pie' for pie plots

```
df2 = pd.DataFrame(np.random.rand(10, 4),
        columns=["a", "b", "c", "d"])

df2.plot(kind='bar') # or df2.plot.bar()
```

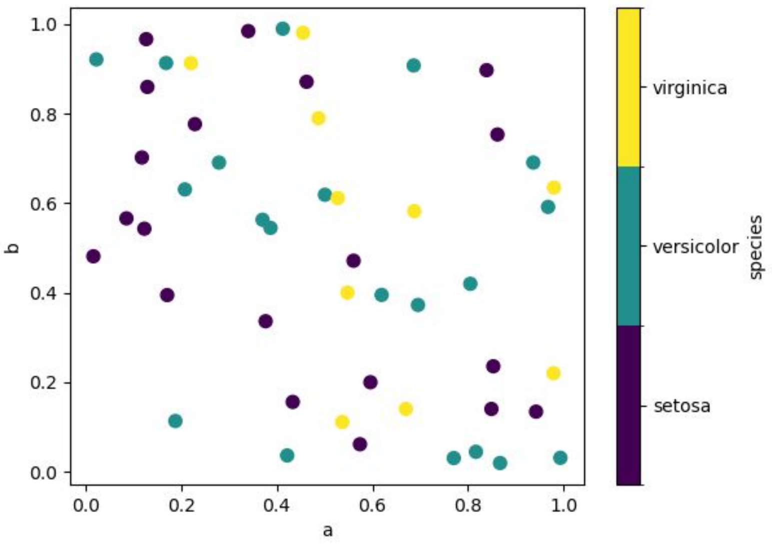


# Important plotting arguments

Most plot types will accept these 3 import kwags:

- x - which col to plot on x-axis
- y - which col to plot on the y-axis
- c - which col to colour the points/bars/box etc using (e.g., discrete categorical groups or a continuous spectrum)

df.plot(kind='scatter', x="a", y="b", c="species")



# Summary

- Consistent tidy data expected by most data function sin python
- Data can be stored in lots of formats - CSV is the simplest and most common in Science
- Pandas is one of many data handling libraries and the most common
- Pandas Series contain 1-dimensional indexed data and support fast access, handy functions, filtering values with boolean masks, and ways of dealing with missing values
- Pandas combines Series in 2-dimensional DataFrames where each row or column is actually a Series
- Pandas supports lots of ways of selecting rows/cols and filtering the values (the same methods as Series but applied to everything)
- Methods/functions can be applied across rows or down columns using axis
- Grouping by values in 1 column lets you apply functions to subsets of your DataFrame
- Merge and concatenation let you combine multiple DataFrames in a predictable way
- Python has a lot of plotting libraries, pandas built-in plotting library lets you quickly plot your data using `df.plot`