Computing for Economists

Data Wrangling

Toren Fronsdal and Ruby Zhang

Harvard University

Introduction

Introduction

The goal of this module is two-fold:

- Explore high-level concepts related to database and data visualization best practices
- Dive into specific tools for handling data in Python

This will be a more practical lecture compared to the rest of the course.

Table of Contents

Data Organization

“Tidy” Data

The principles of tidy (or normal) data provide a standard way to organize data values within a dataset

1. Each variable must have its own column
2. Each observation must have its own row
3. Each value must have its own cell

“Tidy” Data

	treatmenta	treatmentb
John Smith	-	2
Jane Doe	16	11
Mary Johnson	3	1

Table 1: not tidy

name	trt	result
John Smith	a	-
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

Table 2: tidy

“Tidy” Data

country	year	m014	m1524	m2534	m3544	m4554	m5564	m65	mu	f014
AD	2000	0	0	1	0	0	0	0	-	-
AE	2000	2	4	4	6	5	12	10	-	3
AF	2000	52	228	183	149	129	94	80	-	93
AG	2000	0	0	0	0	0	0	1	-	1
AL	2000	2	19	21	14	24	19	16	-	3
AM	2000	2	152	130	131	63	26	21	-	1
AN	2000	0	0	1	2	0	0	0	-	0
AO	2000	186	999	1003	912	482	312	194	-	247
AR	2000	97	278	594	402	419	368	330	-	121
AS	2000	-	-	-	-	1	1	-	-	-

Table 3: not tidy

country	year	sex	age	cases
AD	2000	m	0 — 14	0
AD	2000	m	15 — 24	0
AD	2000	m	25 — 34	1
AD	2000	m	35 — 44	0
AD	2000	m	45 — 54	0
AD	2000	m	55 — 64	0
AD	2000	m	65+	0
AE	2000	m	0 — 14	2
AE	2000	m	15 — 24	4
AE	2000	m	25 — 34	4
AE	2000	m	35 — 44	6
AE	2000	m	45 — 54	5
AE	2000	m	55 — 64	12
AE	2000	m	65+	10
AE	2000	f	0 — 14	3

Table 4: tidy

Problems with “Flat” Data Structure

product	price	market	market_share	country	firm	firm_profit	country_pop	market_size
Discraft Ultra-Star Sport Disc	$13	1	0.12	US	Discraft	12	332	23
Aerobie Pro Ring	$12	1	0.03	US	Aerobie	8	332	23
Discraft Sky Styler Sport Disc	$8	1	0.06	US	Discraft	12	332	23
Discraft Ultra-Star Sport Disc	$13	2	0.18	US	Discraft	12	332	17
Aerobie Pro Ring	$12	2	0.01	US	Aerobie	8	-	-
Discraft Sky Styler Sport Disc	$8	2	0.07	US	Discraft	12	332	17

Table 5: Frisbee Data

- Hard to understand / ambiguous (e.g., is *`firm_profit`* in the aggregate, within each market, or within each country?)
- Repeated information
- Missing data (how can *`market_size`* be missing for one product but not for other products in the same market?)
- Hard to manipulate when data is large

Relational Databases

A relational database organizes data into one or more tables (or “relations”), with a unique key identifying each row

Primary Key: a column (or set of columns) that uniquely identifies each row in a table

- Enforces data integrity and ensures that there are no duplicate records.

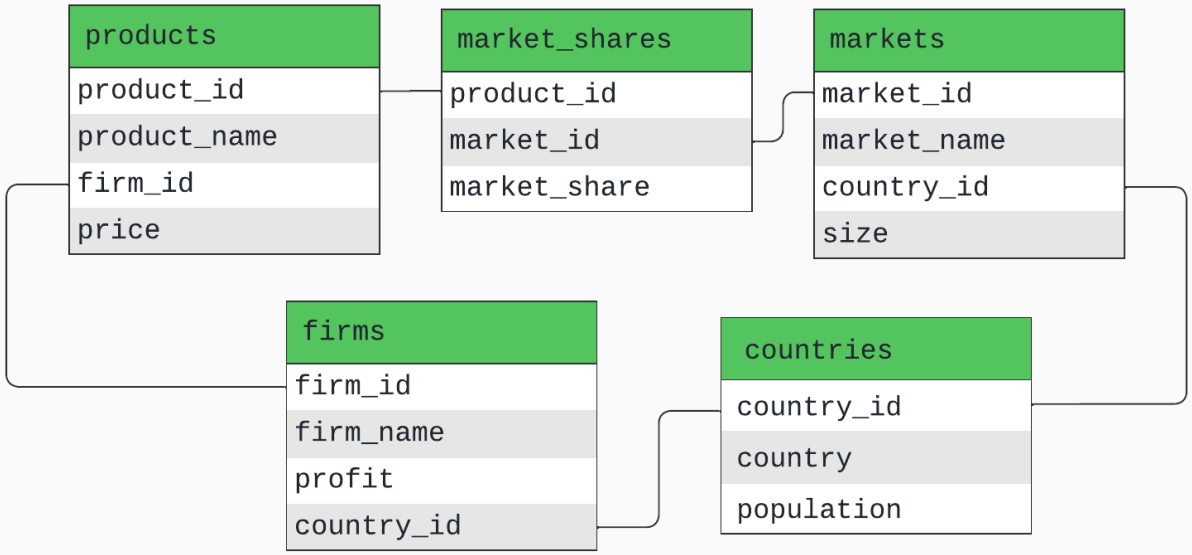
Foreign Key: a column that establishes a link between two tables by referencing the primary key of another table

- Used to create relationships between tables

Relational Databases

Primary Key: a column (or set of columns) that uniquely identifies each row in a table

Foreign Key: a column that establishes a link between two tables by referencing the primary key of another table



Relational Databases

Benefits of Relational Database Structure:

- Most storage-efficient
- Database is *self-documenting*
- Particularly important when data is large

Guiding Principles:

- Store cleaned data in tables with unique, non-missing keys
- Keep data normalized as far into your code pipeline as you can

SQL

Structured Query Language (SQL) is a programming language for storing and processing information in a relational database.

It is **50 years old** and there is a reason it has remained so dominant.

SQL is a **declarative** language: you specify the data, conditions, and what should be returned and SQL figures out the best way to execute

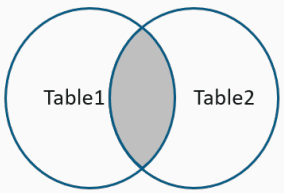
For more involved tasks, it's not as easy to manipulate data as in higher-level languages like R or Python, but it is especially powerful when working with large data.

When working with large data you should:

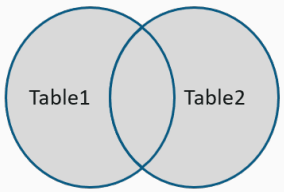
- Store data in a relational database
- Do as much data manipulation in SQL as possible before moving to another language

SQL Joins

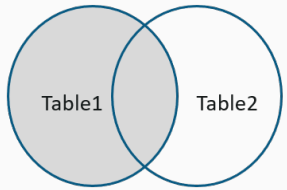
```sql
-- inner join
SELECT *  -- selects all columns
FROM products p
JOIN firms f ON p.firm_id = f.firm_id

-- full join
SELECT *
FROM firms f
FULL JOIN products p ON f.firm_id = p.firm_id

-- left join
SELECT *
FROM products p
LEFT JOIN firms f ON p.firm_id = f.firm_id

-- right join
SELECT *
FROM firms f
RIGHT JOIN products p ON f.firm_id = p.firm_id
```



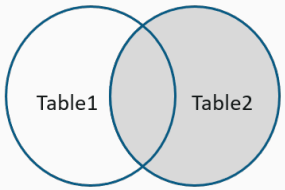
INNER JOIN



FULL JOIN



LEFT JOIN



RIGHT JOIN

Basic SQL Queries

```sql
-- filtering data
SELECT * -- selects all columns
FROM products
WHERE price > 10

-- sorting data
SELECT market_id, product_id, price
FROM market_shares
ORDER BY market_id, price DESC

-- aggregating data
SELECT market_id, AVG(price) AS avg_price, COUNT(*) AS n_products, COUNT(DISTINCT firm_id) n_firms
FROM products p
LEFT JOIN market_shares m ON p.product_id = m.product_id
LEFT JOIN firms ON p.firm_id = f.firm_id
GROUP BY market_id
```

Data Wrangling

Data Wrangling

Data wrangling is the process of cleaning and structuring raw data into the desired format for analysis. We will focus on *pandas*, which uses DataFrame structures.

Key Steps:

1. Collect data: import raw datasets (CSV, JSON, Excel, etc.) as DataFrames
2. Clean data: correcting formats and identifying missing values
3. Transform data: reshaping, merging, normalizing, and standardizing the data
4. Explore data: descriptive statistics and visualization

Data Cleaning

Missing Data

```
df.dropna(inplace=True) # Drop NaNs
df['column_name'].fillna(value, inplace=True) # Replace missing value
df['column_name'].replace(old_value, new_value, inplace=True) # Replace specific value
```

Filtering and Removing

```
df = df[df['column_name'] > 2] # Filtering data
df.drop_duplicates(inplace=True) # Remove duplicates
```

Categorical Data

```
df['column_name'].astype('category', inplace=True) # Convert to category type
pd.get_dummies(df, columns=['column_name']) # Get dummy variables
```

Datetime

```
df['date_column'] = pd.to_datetime(df['date_column'])
```

Data Cleaning: Example

product	price	market	market_share	country	firm	firm_profit	country_pop	market_size
Discraft Ultra-Star Sport Disc	$13	1	0.12	US	Discraft	12	332	23
Aerobie Pro Ring	$12	1	0.03	US	Aerobie	8	332	23
Discraft Sky Styler Sport Disc	$8	1	0.06	US	Discraft	12	332	23
Discraft Ultra-Star Sport Disc	$13	2	0.18	US	Discraft	12	332	17
Aerobie Pro Ring	$12	2	0.01	US	Aerobie	8	-	-
Discraft Sky Styler Sport Disc	$8	2	0.07	US	Discraft	12	332	17

Table 6: Frisbee Data

Let's go back to our frisbee database. We would like to replace the dollar sign and "-" characters to convert them to numeric types.

```
df = df.replace({'\$': '', '-': np.nan}, regex=True) # regex looks for any instance
df = df.replace('\$', '', regex=True).replace('-', np.nan) # This is what we want! Why?
```

Note that ***df*** is the object, and the method ***replace*** is acting on the self!

In the second command, we are **chaining** commands. You can chain many commands in *pandas*.

```
# Convert to numeric types
cols = ['price', 'country_pop', 'market_size']
df[cols].astype(float, inplace=True)
```

Data Transformation

We will cover the following data transformations:

- Reshaping
- Merging
- Aggregating

Data Transformation: Reshaping and Pivot Tables

pivot: Long to Wide

```
df.pivot(index=['product'], columns=['market'], values=['price'])
# Similar except can aggregate over duplicates
df.pivot_table(index=['product'], columns=['market'], values=['price'], aggfunc='mean')
```

	price	
market	1	2
product		
Aerobie Pro Ring	12.0	12.0
Discraft Sky Styler Sport Disc	8.0	8.0
Discraft Ultra-Star Sport Disc	13.0	13.0

stack/unstack

```
    df.stack() # Decrease width of dataframe and produce MultiIndex
    df.unstack() # Reshape from stack to wide format
```

Data Transformation: Reshaping and Pivot Tables

wide_to_long: Reshapes wide to long using stubnames (easier syntax than *melt*)

Let's unpivot the pivot table from above! **Note the MultiIndexing.**

```
# Convert MultiIndex columns to regular columns with stubnames
dfw.columns = ['_'.join(col) for col in dfw.columns]

# Reset index to prepare for unpivoting
dfw = dfw.reset_index()
```

	product	price_1	price_2
0	Aerobie Pro Ring	12.0	12.0
1	Discraft Sky Styler Sport Disc	8.0	8.0
2	Discraft Ultra-Star Sport Disc	13.0	13.0

```
pd.wide_to_long(dfw, stubnames='price', i='product', j='market', sep='_')
```

		price
product	market	
Aerobie Pro Ring	1	12.0
Discraft Sky Styler Sport Disc	1	8.0
Discraft Ultra-Star Sport Disc	1	13.0
Aerobie Pro Ring	2	12.0
Discraft Sky Styler Sport Disc	2	8.0
Discraft Ultra-Star Sport Disc	2	13.0

Data Transformation: Merges and Joins

There are four ways to merge on a key variable:

```
pd.merge(data1, data2, how='left', on='key_var') # Keep if in data 1
pd.merge(data1, data2, how='right', on='key_var') # Keep if in data 2
pd.merge(data1, data2, how='inner', on='key_var') # Keep if in both
pd.merge(data1, data2, how='outer', on='key_var') # Keep all data rows
```

`join` is a more index-centric way of combining:

```
data1.join(data2, how='left') # Right, inner, and outer joins also viable
```

With both functions, you can join on multiple keys or indices:

```
pd.merge(data1, data2, on=['key1', 'key2'], how='inner')
pd.merge(data1, data2, left_on='column_name', right_index=True, how='inner')
```

Data Transformation: Aggregation

The *groupby* object returns an aggregated object that one can then apply methods to. The most useful ones are:

agg(): compute multiple aggregates at once (e.g., min, sum, count)

```
df.groupby('market').agg(['min', np.median, max])
```

	price			market_share			firm_profit			country_pop			market_size		
	min	median	max	min	median	max	min	median	max	min	median	max	min	median	max
market															
1	8.0	12.0	13.0	0.03	0.06	0.12	8	12.0	12	332.0	332.0	332.0	23.0	23.0	23.0
2	8.0	12.0	13.0	0.01	0.07	0.18	8	12.0	12	332.0	332.0	332.0	17.0	17.0	17.0

Note that aggregate functions *min, max* can be written with or without quotations. *median* is a Numpy function.

apply(): apply any function to group results, can return anything

```
# Calculate HHI for each market (incomplete data)
df.groupby('market').apply(lambda x: ((x['market_share']*100)**2).sum())
```

```
market
1    189.0
2    374.0
dtype: float64
```

Data Transformation: Aggregation

transform(): return transformed version of full data

```
# standardize
df.groupby('market').transform(lambda x: (x - x.mean())/x.std())
```

	price	market_share	firm_profit	country_pop	market_size
0	0.755929	1.091089	0.577350	NaN	NaN
1	0.377964	-0.872872	-1.154701	NaN	NaN
2	-1.133893	-0.218218	0.577350	NaN	NaN
3	0.755929	1.082543	0.577350	NaN	NaN
4	0.377964	-0.889231	-1.154701	NaN	NaN
5	-1.133893	-0.193311	0.577350	NaN	NaN

filter(): drop original data based on group properties

```
# Keep markets where the total market share is more than 25%
df.groupby('market').filter(lambda x: x['market_share'].sum() > 0.25)
```

	product	price	market	market_share	country	firm	firm_profit	country_pop	market_size
3	Discraft Ultra-Star Sport Disc	13.0	2	0.18	US	Discraft	12	332.0	17.0
4	Aerobie Pro Ring	12.0	2	0.01	US	Aerobie	8	NaN	NaN
5	Discraft Sky Styler Sport Disc	8.0	2	0.07	US	Discraft	12	332.0	17.0

An Aside on Vectorization

Vectorization

Vectorization refers to the process of applying operations to entire arrays or sequences of data at once, rather than iterating over individual elements one by one.

```
numbers = [1.0, 2.0, 3.0, 4.0]
result = []
for num in numbers:
    result.append(math.exp(num))
```

Looping

```
numbers = np.array([1.0, 2.0, 3.0, 4.0])
result = np.exp(numbers)
```

Vectorized with NumPy

In Python, vectorization is typically faster than looping since it can leverage more efficient low-level optimizations and parallelization. NumPy has a suite of built-in functions that perform vectorized, element-wise array calculations.

(In Julia, vectorization is not necessarily faster than looping.)

Broadcasting

Python automatically deals with different-size arrays by *`broadcasting`* the smaller array over the larger so that they are compatible, by aligning the right-most dimension. Two dimensions are compatible if:

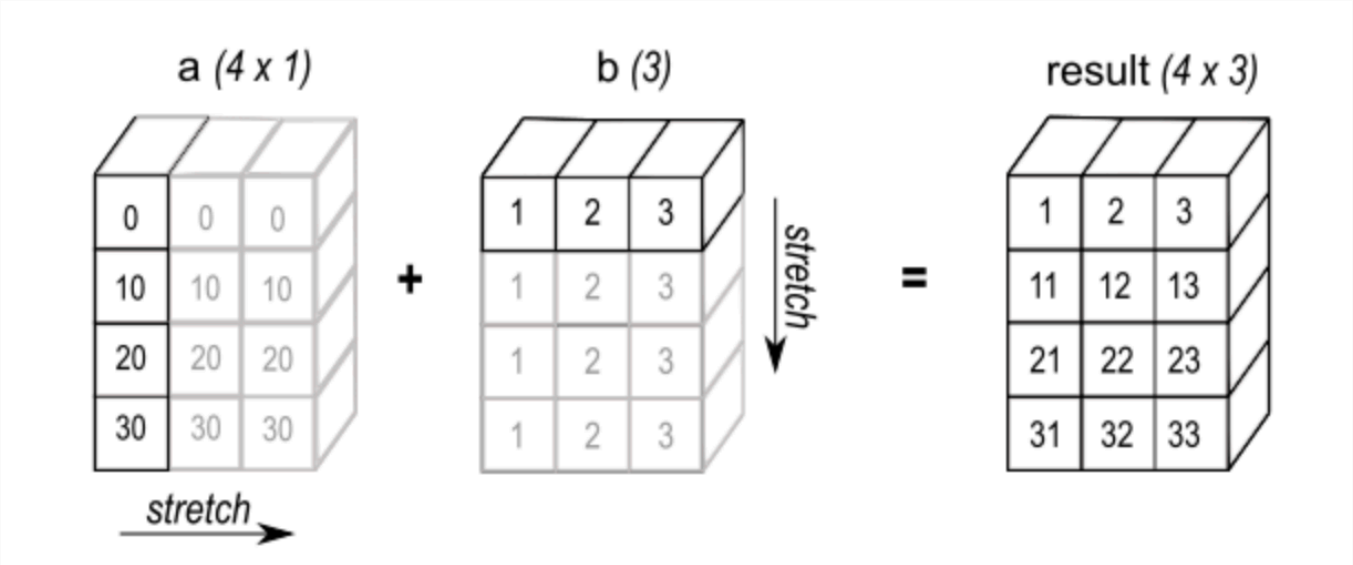
1. They are the same
2. One of them is 1

Adding an empty new axis with *`np.newaxis`* makes some functions such as taking outer products, adding a different vector to a 3D array of 2D arrays, etc. much easier. **However, code readability can be compromised!**

Broadcasting Example

Simple example:

```
a = np.array([0.0, 10.0, 20.0, 30.0])
b = np.array([1.0, 2.0, 3.0])
a[:, np.newaxis] + b
```



Data Visualization

Why Data Visualization?

"A picture is worth a thousand words."

- Many key empirical results can be distilled into key graphs or tables
- **What is the best way to clearly tell the story of your research results?**
- Visualization also helps identify patterns in the data

Principles of Data Visualization

Edward Tufte's principles from "The Visual Display of Quantitative Information" (1983) geared at:

Clarity

1. Show the data and maximize data-ink ratio
2. Remove "chartjunk" and distractions
3. Self-containing visual: integrate text and graphics

Integrity

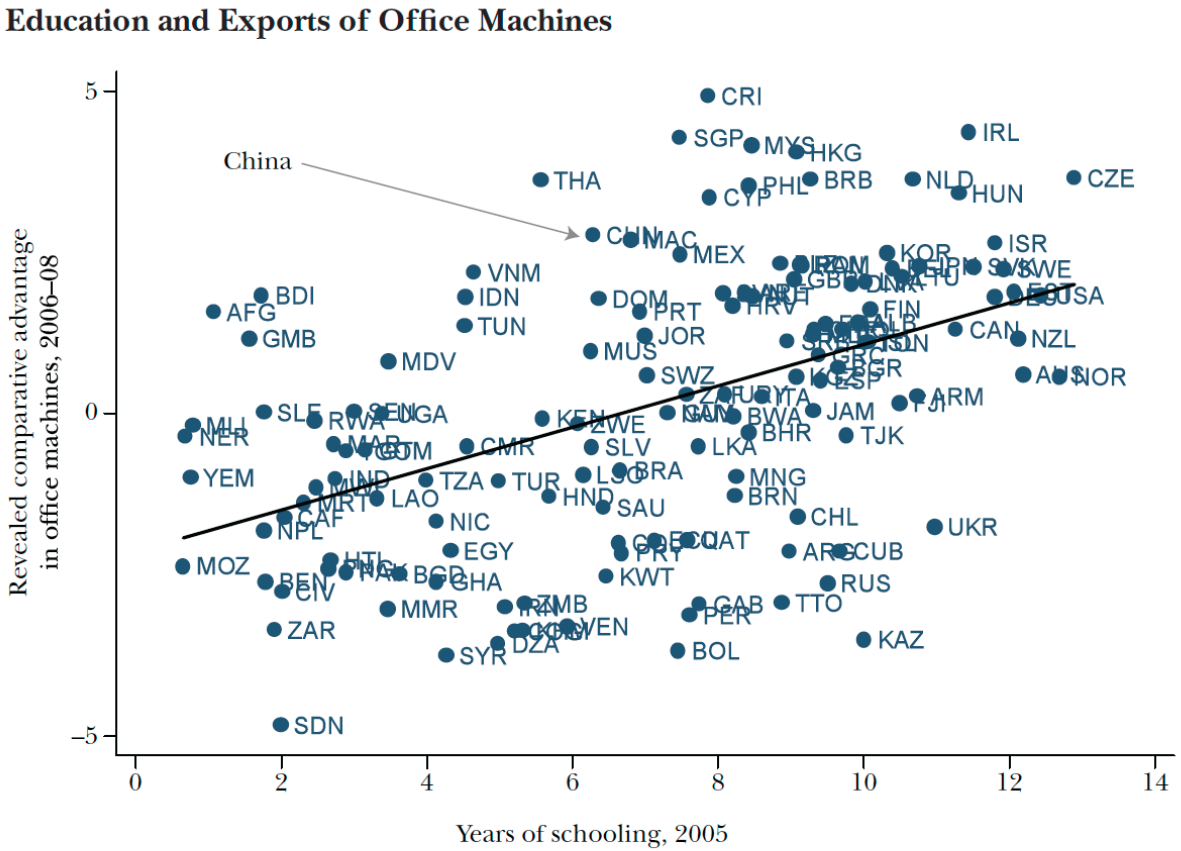
1. Respect proportions and do not distort axes
2. Use "small multiples" with similar datasets
3. Encourage comparisons
4. Reveal data at several levels

Examples from "An Economist's Guide to Visualizing Data" (Schwabish, 2014).

Example: Clutterplot

Figure 2A

A Clutterplot Example



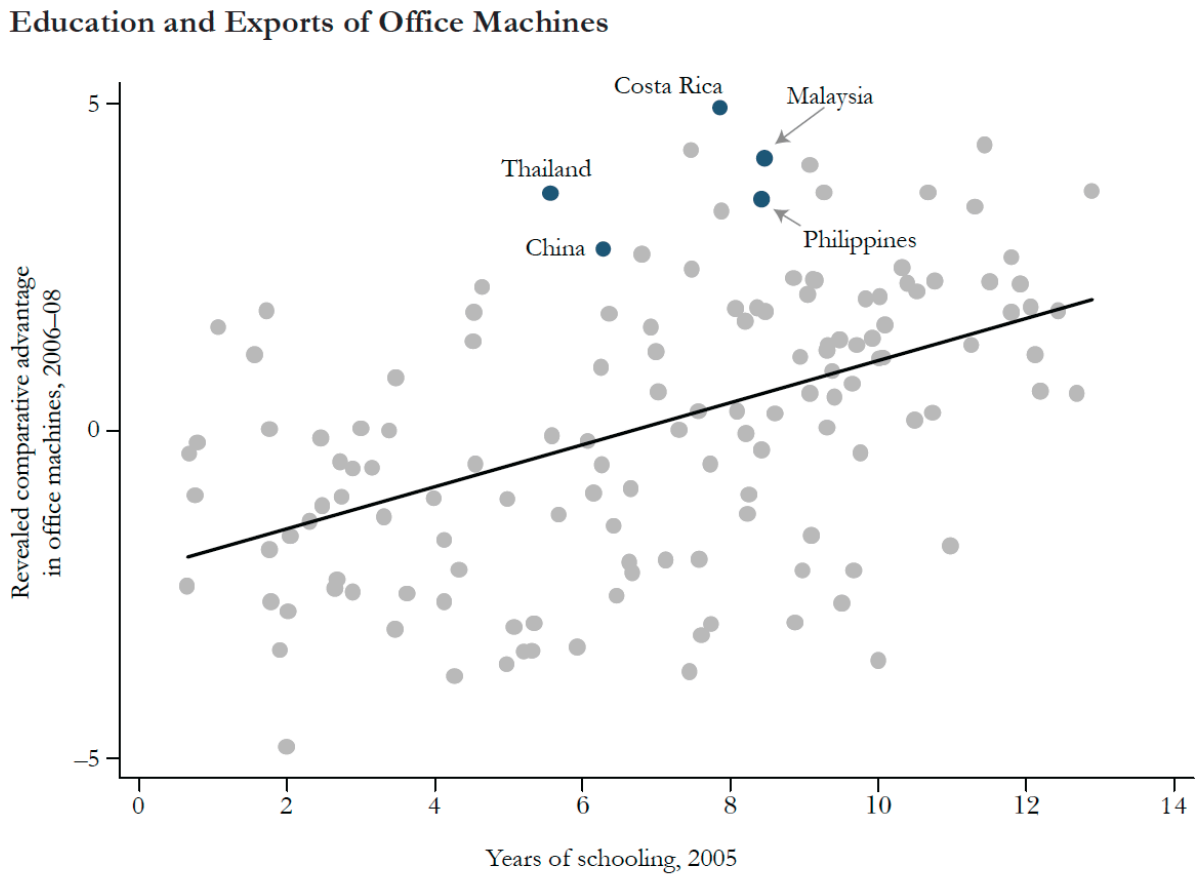
Source: Hanson (2012).

Example: Clutterplot Revised

Show the data that is needed in a clear way, remove unnecessary clutter

Figure 2B

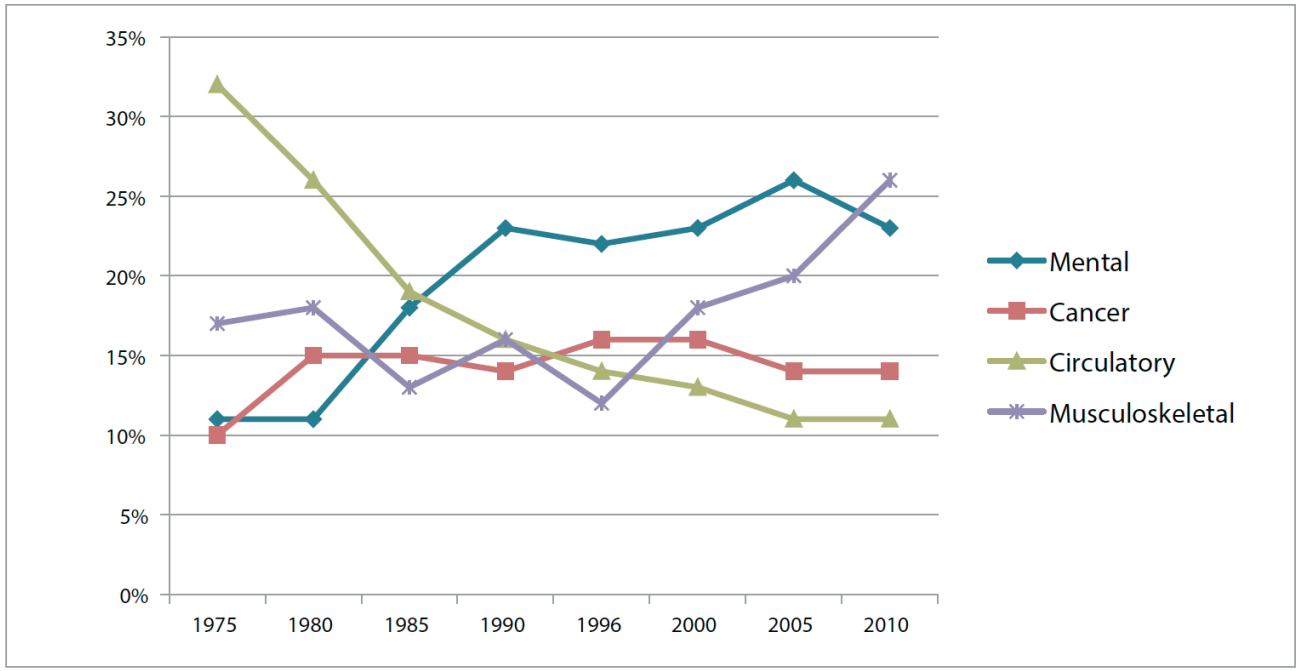
Revising the Clutterplot Example



Example: Spaghetti Chart

Figure 6A
A Spaghetti Chart

27. Initial DI Worker Awards by Major Cause of Disability—Calendar Years 1975-2010



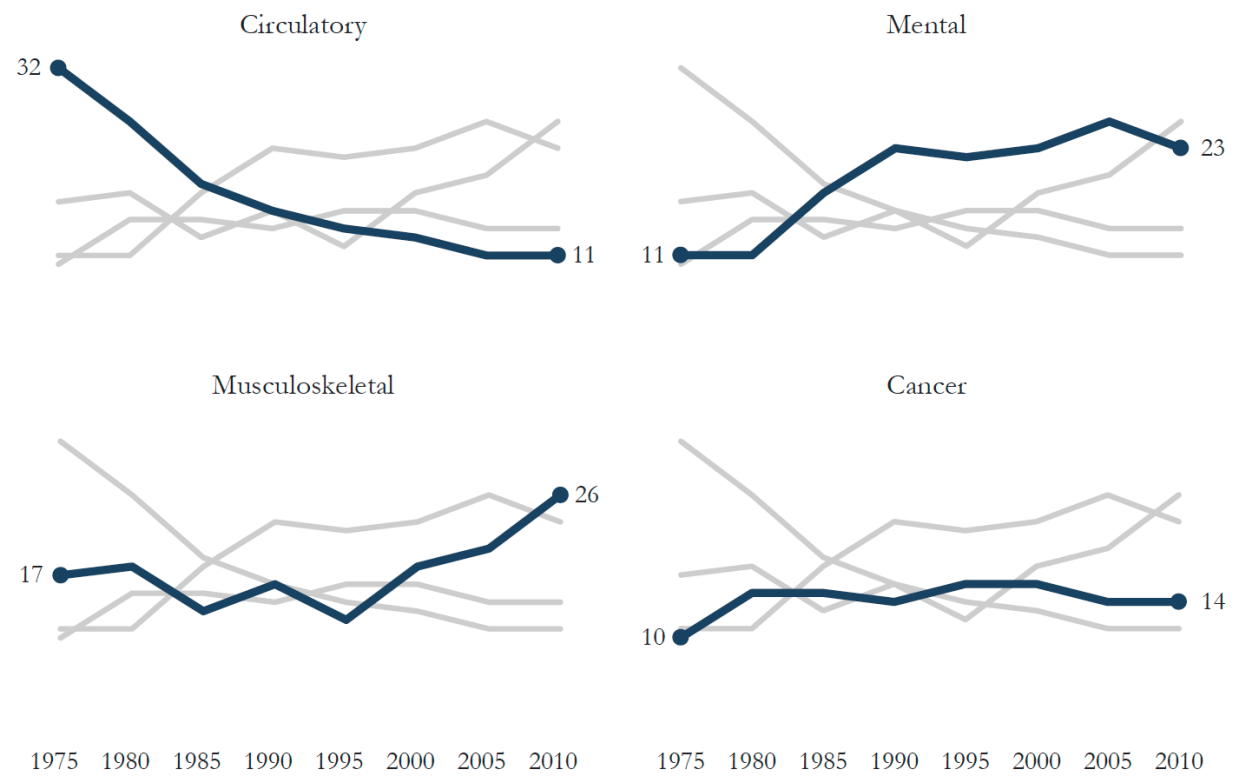
Source: Social Security Advisory Board (2012).

Example: Spaghetti Chart Revised

Use "small multiples" to clearly show each plot

Figure 6B

Revising the Spaghetti Chart



Example: Unbalanced Chart

Figure 5A

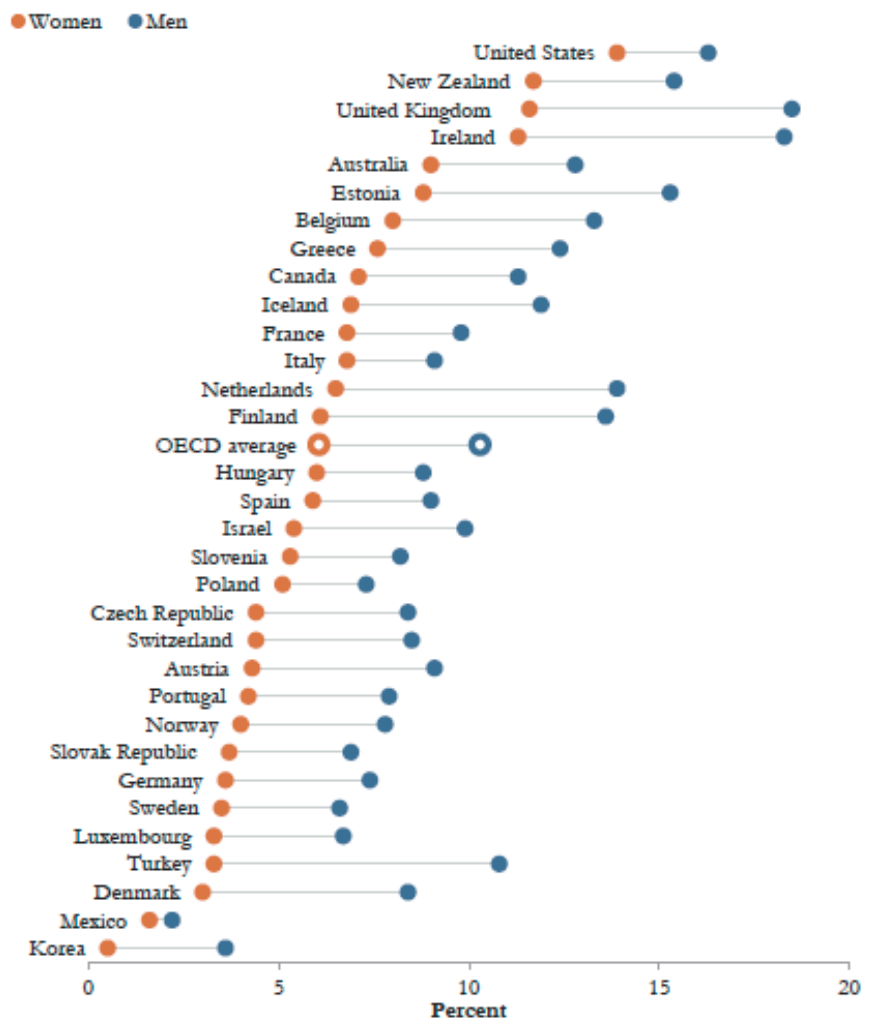
An Unbalanced Chart

Source: Author, based on OECD (no date) and Rampell (2013).

Example: Unbalanced Chart Revised

Figure 5B

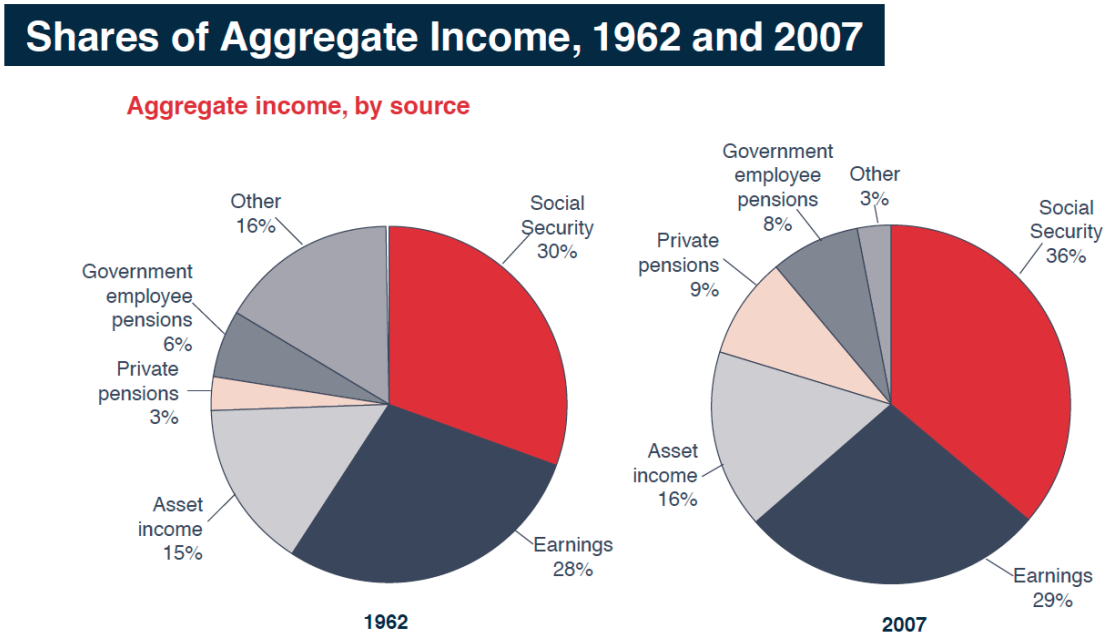
Revising the Unbalanced Chart



Example: Comparison Pie Chart

Figure 9A

Two Pie Charts for Comparison



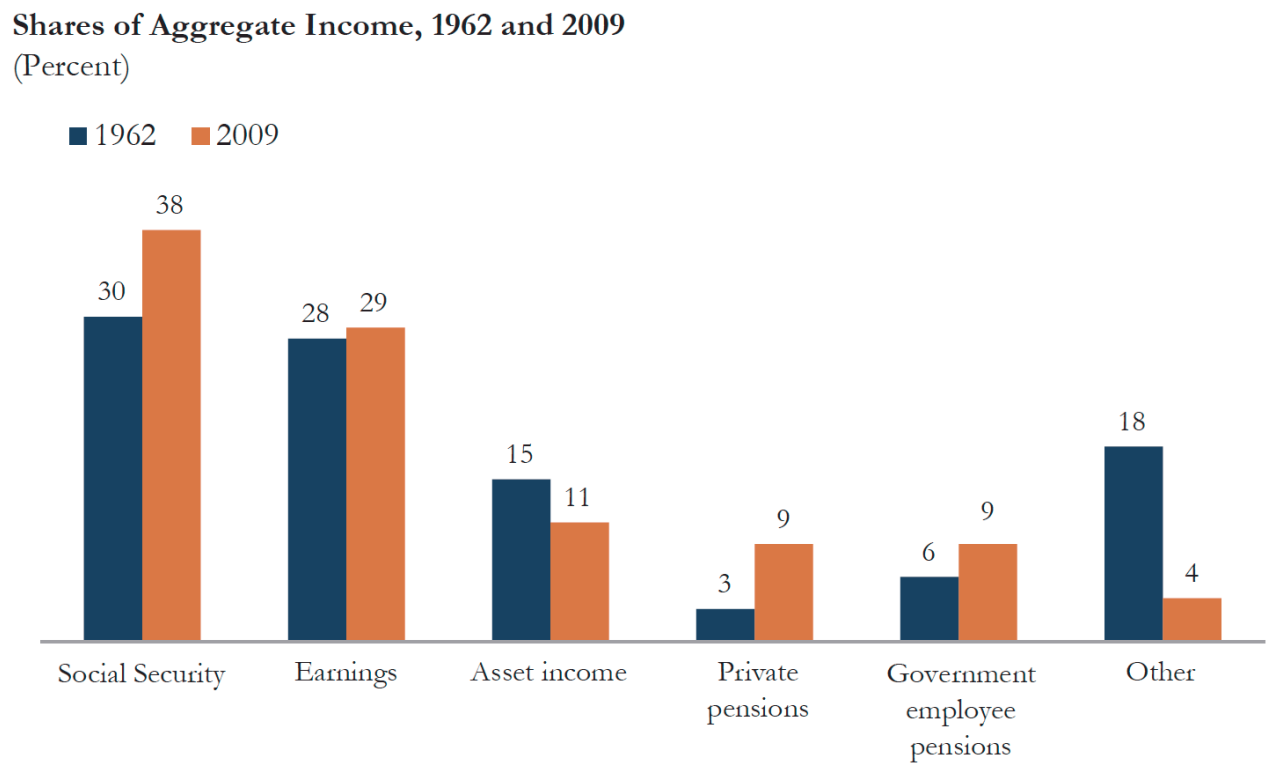
Source: Social Security Administration (2009).

Example: Comparison Pie Chart Revised

What is the goal of the graph? Comparison between years or distribution within year?

Figure 9B

Alternative to a Pie Chart: A Paired Column Chart



Considerations of Specific Plots

(1) What kind of data is represented?

- Continuous vs. discrete data
- Ordered vs. unordered
- Number of variables and groups

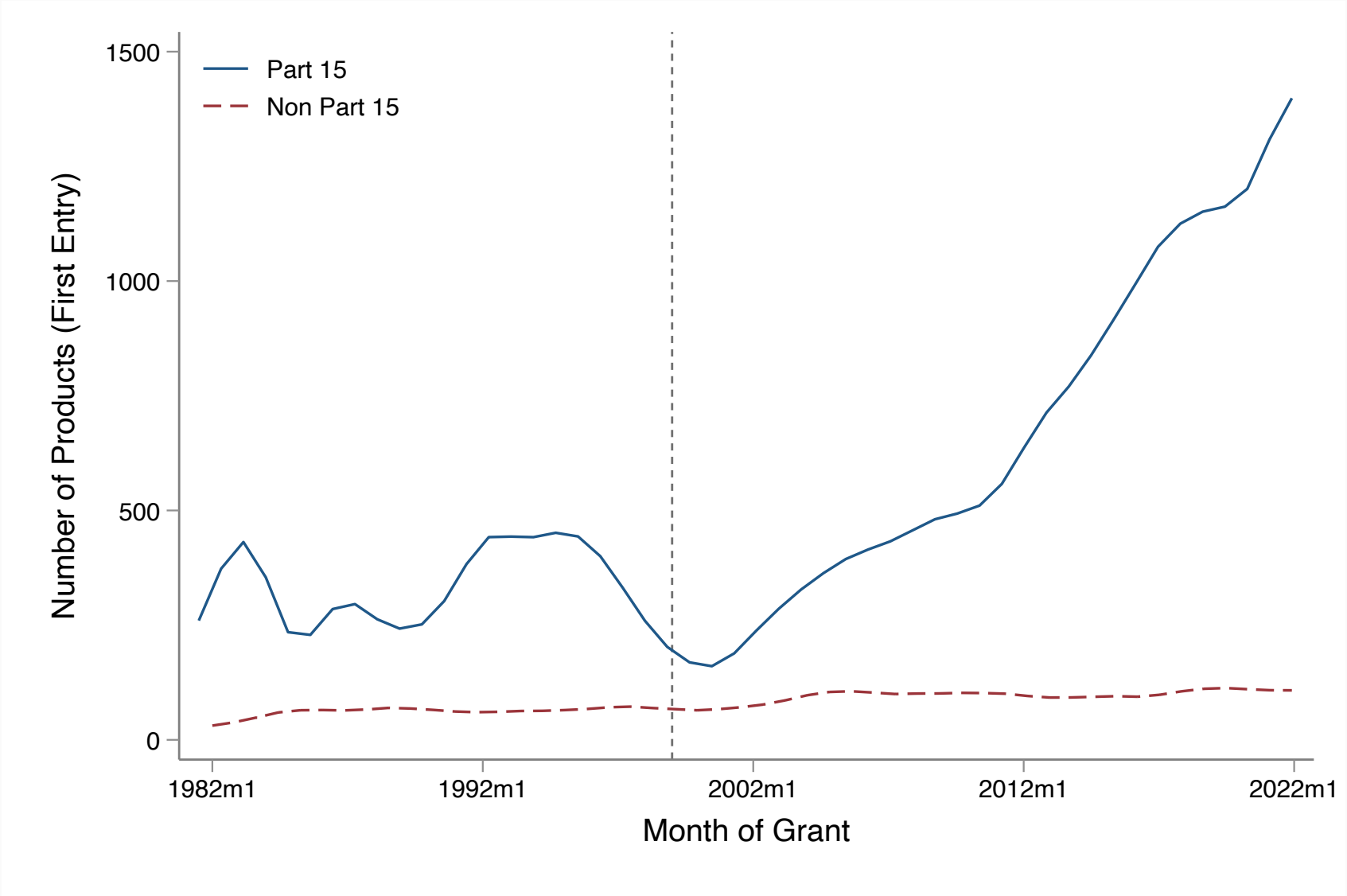
(2) What would you like to say about the data?

- Finding trends
- Comparisons between different groups
- Distribution of a measure
- Relationship between variables
- Breakdown of a whole

Source

Line Plot

1D or 2D ordered arrays

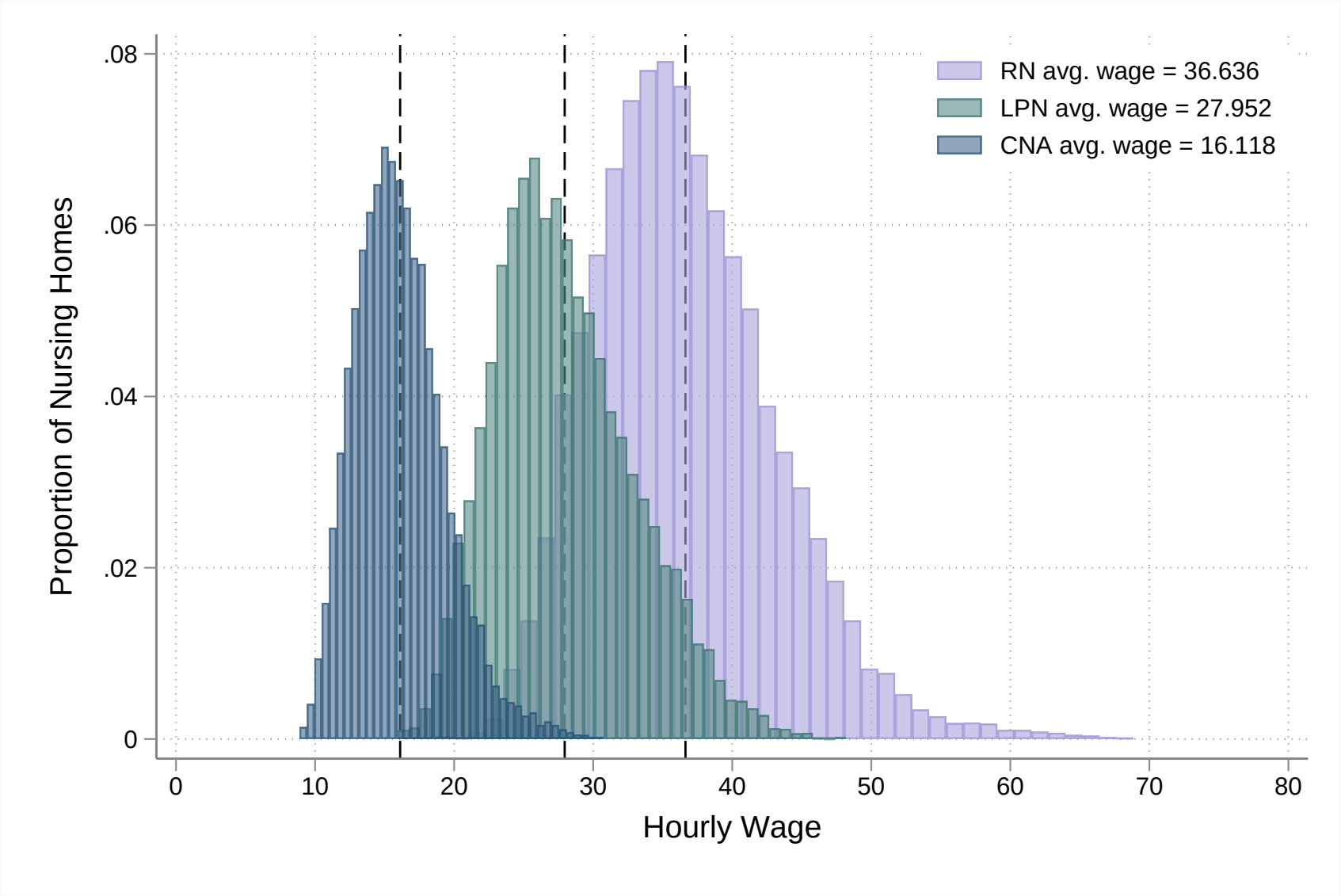


Scatter Plot

Data points are unordered, individual instances with a possible relationship between two variables

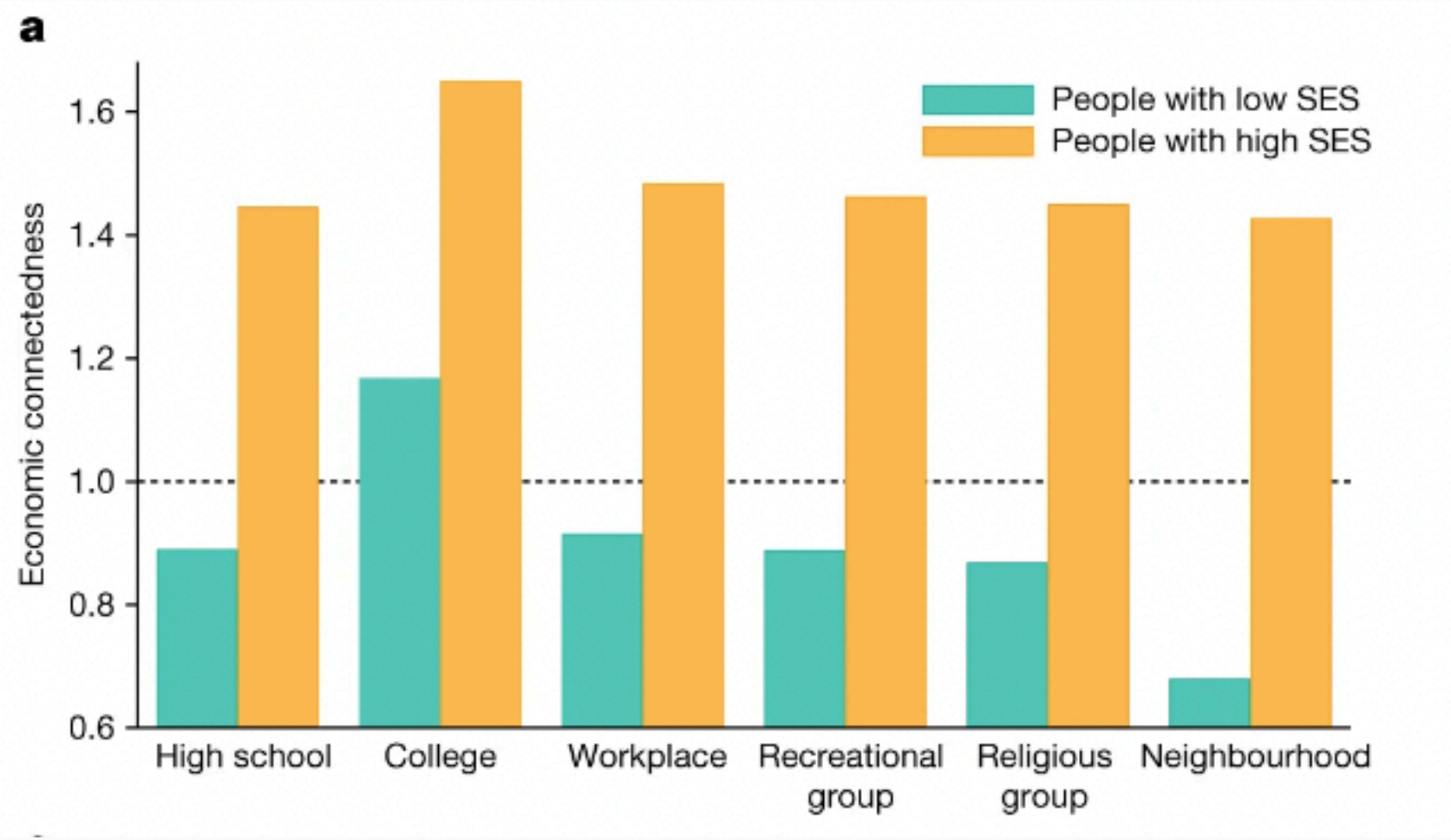
Histogram/Kernel Density Plot

Distribution of 1D unordered array (emphasis on shape)



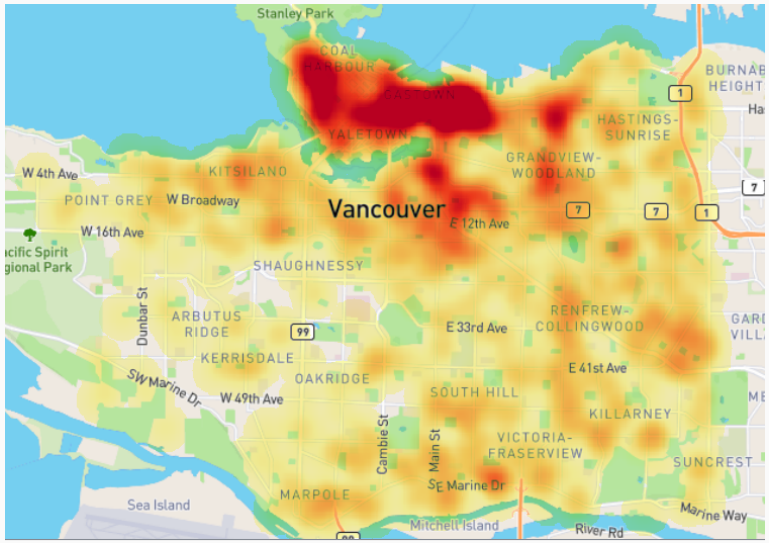
Bar Chart

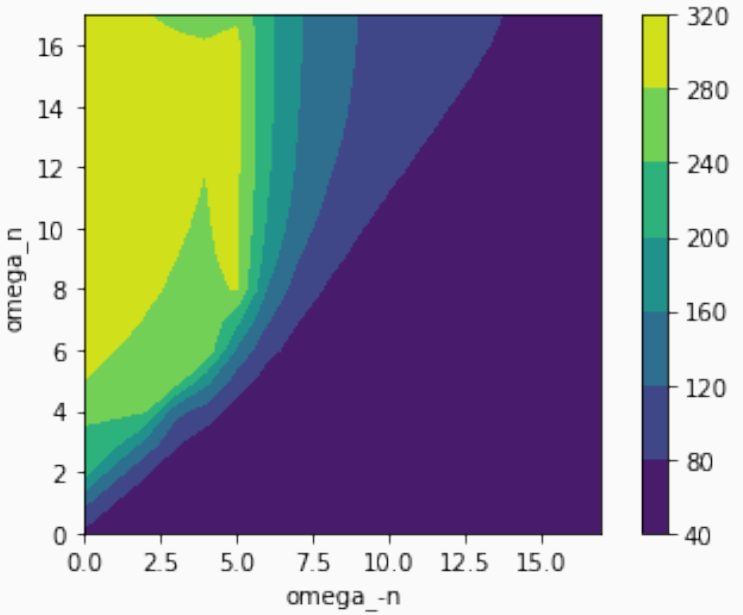
Discrete, 1D data sets or comparisons thereof



Heat Maps

Plot $f : \mathbb{R}^2 \to \mathbb{R}$ with sufficiently refined domain. Note that maps are two dimensional (longitude and latitude).





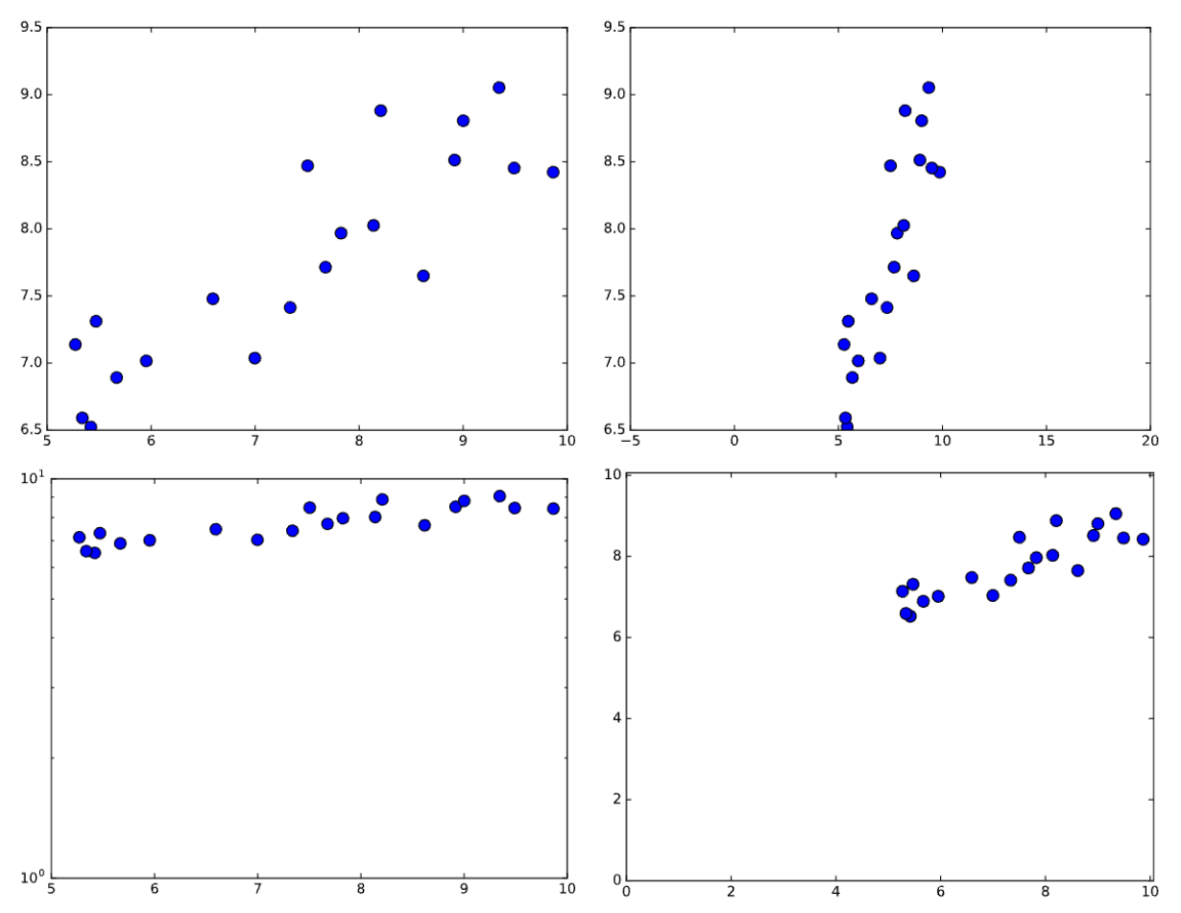
General Visualization Questions

- What are the appropriate axes and window limits? Scaling?
- How should colors and size be used to contrast or emphasize the key points?
- What are the relevant and necessary tick marks or labels?
- Is it clear to have multiple datasets on the same plot or multiple?
- If necessary, how should the data be sorted to be displayed logically?

Data Misrepresentation

Show only what you need but also accurately represent the data!

- Choose axes scales and window limits wisely
- Clearly state any transformations (e.g. log-scale) or subsetting (e.g. winsorizing) of the data



Data Analysis with Visualization

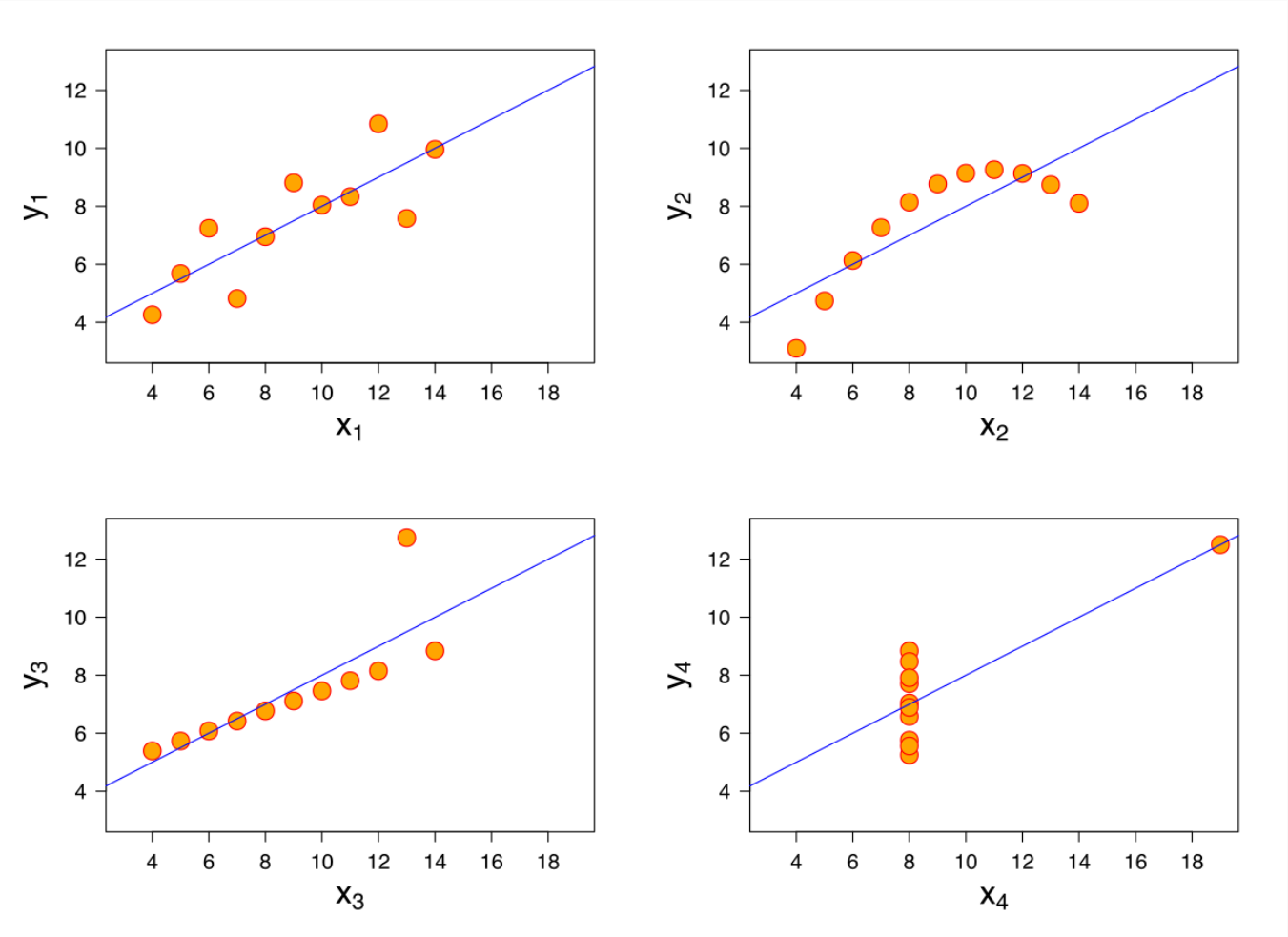
Apart from final results, researchers use data viz as a way to explore and check the data.

- Does the data make sense?
  - Histogram of variables
  - Scatterplots of variables
- Does the analysis make sense?
  - Pre- and post-trends (event studies, DiDs)
  - Checking placebo tests

Summary statistics and regression parameters do not fully characterize the data!

Anscombe's Quartet (1973)

$$\mu_x = 9 \ (11) \qquad \mu_y = 7.5 \ (4.13)$$
$$y = 0.5x + 3 \qquad (R^2 = 0.67)$$



Data Visualization Tools

The libraries here are specific to Python for plotting (with hyperlinks):

- **Matplotlib**: Foundational plotting library in Python
- Pandas: convenient data visualization (built on matplotlib)
- Seaborn: advanced statistical visualization (built on matplotlib)
- Plotnine: for R-users, Python implementation of *`ggplot2`*

Interactive Visualization Libraries (with web-interfaces):

- Plotly: inherently interactive plots, broad language support
- Bokeh: more customizable and efficient with large data sets

We will mainly be focusing on *`matplotlib`*, which is the most customizable.

Matplotlib Basics

A *figure* is the final image that may contain many *axes* (plots).

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

State-Based Interface (i.e. MATLAB): state of current figure and plotting area stored in the background

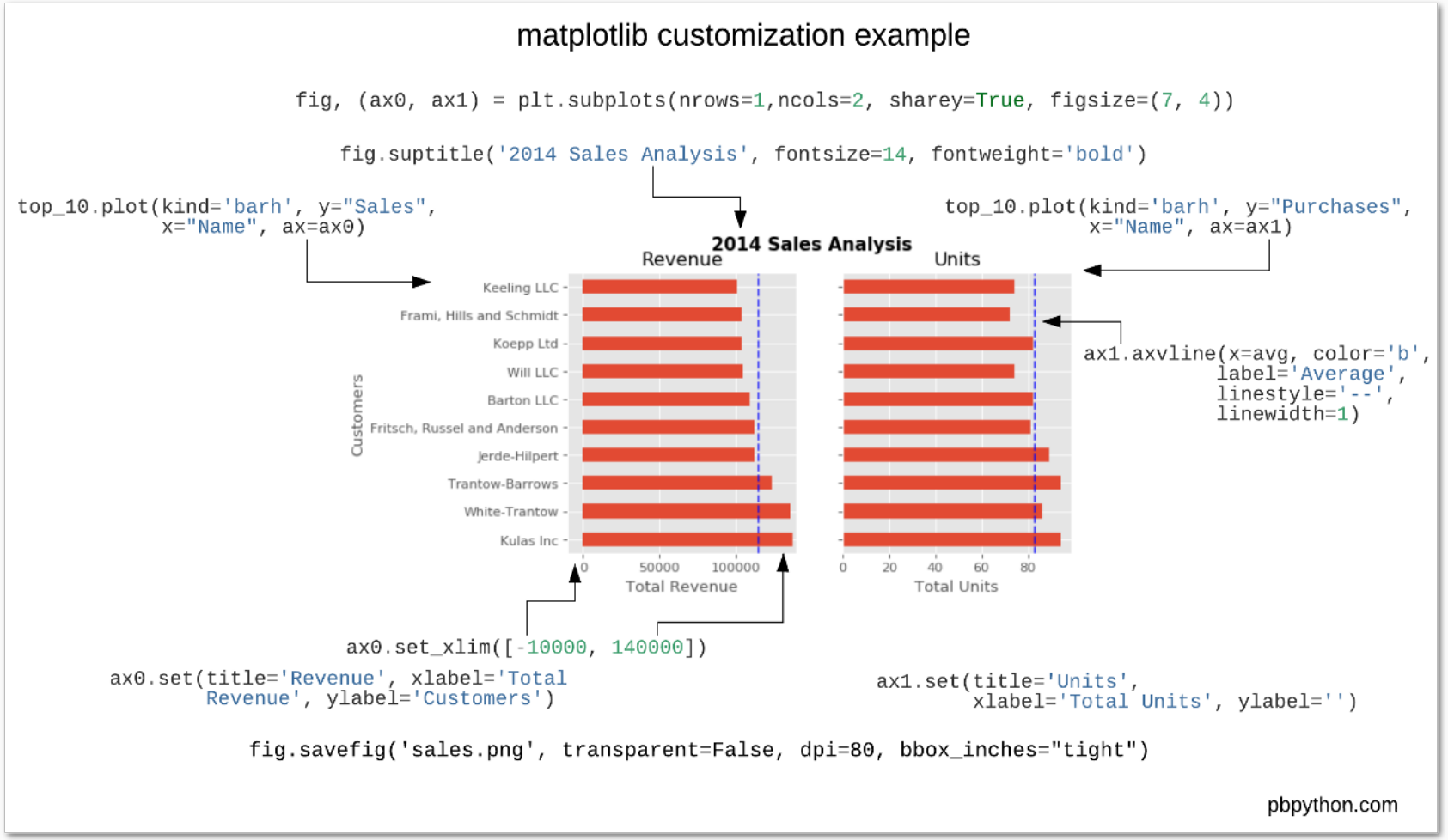
```
# Plot using the state-based interface
plt.plot(x, y)
plt.title('State-Based Plotting')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
```

Object-Oriented Interface: create figure and axes objects and call methods on them. Recommended for more complex plots.

```
# Create figure and axes objects
fig, ax = plt.subplots()

# Plot using the object-oriented interface
ax.plot(x, y)
ax.set_title('Object-Oriented Plotting')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
```

Customizing Matplotlib

Matplotlib is highly customizable, can be used on top of seaborn and pandas!



Seaborn

Seaborn is built on top of *matplotlib* with a more abstracted interface. One might use Seaborn first:

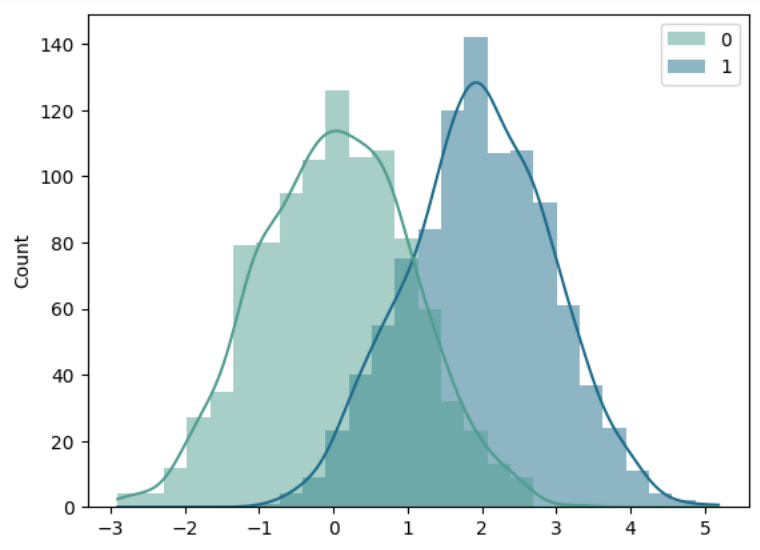
- Easy Interface: built-in functions for complex and statistical visualizations
- Aesthetics: the default themes and color palettes are pre-designed to be pleasing and informative
- DataFrame Integration: very easy to use directly with a pandas DataFrame
- Facetting: *FacetGrid* allows matrix of plots with common axes

Everything is then customizable with *matplotlib* functions

Seaborn and Matplotlib

Default Seaborn

```
# Generate some sample data
data_x = np.random.randn(1000)
data_y = np.random.randn(1000) + 2

# Use Seaborn's histplot
sns.histplot(data=(data_x, data_y), fill=True,
        kde="True", palette="crest", alpha=.5,
        linewidth=0)
```



Seaborn with *matplotlib*

```
# Customize using Matplotlib
sns.histplot(data=(data_x, data_y), fill=True,
        kde="True", palette="crest", alpha=.5,
        linewidth=0)
plt.title('Distributions of Random Data')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.grid(True, which='both', linestyle='--',
       linewidth=0.5)
```