DSS5201 Data Visualization
Week 6

Yuting Huang

NUS DSDS

2024-09-16

Midterm exam on Sep 30

Time: Monday Sep 30 7-8pm

Venue: MPSH 2B

Things to bring on the exam day:

- A laptop with the latest Python, VS Code, and Examplify installed.
- The laptop charger.
- Your NUS matriculation card.

Arrive at least 15 minutes early at the venue for necessary setups (download data sets, check Python packages, etc).

- General information:

  https://nus.atlassian.net/wiki/spaces/DAstudent/pages/22511642/Device+Minimum+System+Requirements

- Register & join a briefing session on Zoom by Sep 26.

  https://nus.atlassian.net/wiki/spaces/DAstudent/pages/22511675/Common+Briefing+Sessions

Contact CIT via **citbox25@nus.edu.sg** for any technical issues.

- **Content of exam:** All materials covered from Week 1 to Week 6 (inclusive).
- **Format: Open-book, block-internet on Examplify**.
- You can refer to materials from and beyond our course, but will **not** have access to the internet throughout the exam.
- Make sure you have downloaded data and installed necessary packages before the exam begins.

The total marks available are 30.

- **Part I:** Multiple choice + Fill-in-the-blank questions.
    - Answer questions directly on Examplify.
    - No submission of Python code is needed.
- **Part II:** Coding questions.
    - Answer questions in a Python Notebook (`.ipynb`) and render it to HTML (`.html`).
    - Submit exam on Examplify, **AND**
    - Submit the Notebook and HTML files to Canvas **immediately after** the exam.

Logistics

1. The exam will be available on **Examplify** from **Sunday, Sep 29 at 8pm**.
   - Only one download is allowed. So ensure you download the exam to the same laptop that you will use during the exam.
2. Exam data will be available on **Canvas** on **Monday, Sep 30 6:45pm**.
3. The following packages are required for the exam:
   - `pandas`, `numpy`, `matplotlib`, `json`, `pydataset`, `nycflights13`
   - You may use additional packages, but also need to **ensure** that they are installed properly beforehand.
   - You won't have access to internet once the exam begins.

4. The exam begins on **Monday 7pm**, sharp.
5. During the exam, save your Python Notebook frequently.
6. The exam **ends at 8pm on Examplify**.
    - Submit your exam on Examplify, **AND**
    - Both your python notebook and the rendered HTML to **Canvas** immediately after the exam.
    - On Canvas, the submission window closes at **8:15pm**.

Week 6 & Recess

- Install and update **Python, VSCode**, and **Examplify**
- Verify you've installed and are able load the required packages

`pandas, numpy, matplotlib, json, pydataset, nycflights13`

Week 7
Sunday 8pm

Download exam from Examplify

Monday 6:45pm

Download exam data from Canvas

7:00pm
Exam begins

No internet access once exam begins on Examplify.

Exam begins

Submit exam on Examplify

8:00pm
Exam ends

Internet access resumes after exam ends on Examplify.

- Submit **`ipynb`** and **`HTML`** to Canvas **<u>immediately after</u>** the exam.
- The Canvas submission folder will close at **8:15pm**

PSet 1: Practice questions

PSet 1 will be available on Canvas.

- Practice coding questions on data import, data manipulation, and data joins.
- Use Python notebook to answer questions and render it into HTML at the end.
- Due on **this Friday, Sep 20 by 11:59pm**.

Best to complete them independently, in an environment similar to the exam setting.

Data transformation Week 4

- `query()`, `sort_values()`, `rename()`, `groupby()`, ...

Tidy data (data reshaping) Week 5

- `melt()`, `stack()`, `pivot()`, and `pivot_table()`.

Relational data Week 6

Data never arrive in the condition that we need them.

They need to be reshaped and reformatted.

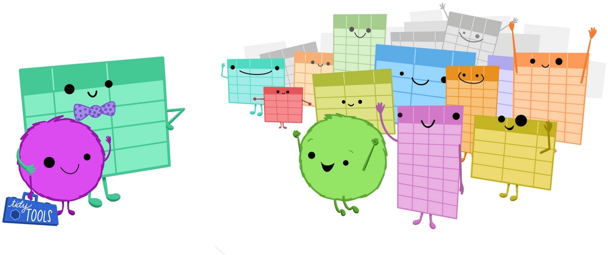
"Tidy" Table

Business Unit	Year	Quarter	Budget
Sales	2000	Q1	2,500,000
Marketing	2000	Q1	1,000,000
Sales	2000	Q2	2,750,000
Marketing	2000	Q2	1,250,000
Sales	2000	Q3	3,000,000
Marketing	2000	Q3	4,000,000
Sales	2000	Q4	2,000,000
Marketing	2000	Q4	500,000
Sales	2001	Q1	2,500,000
Marketing	2001	Q1	1,500,000

"UnTidy" Table

Past and projected budgets for WidgetCo.'s Sales and Marketing Org.						
Contact JDoe@widgets.ca for more information.						
	Year					
Business Unit	2000				2001	
	Q1	Q2	Q3	Q4	Q1	Q2*
Sales	2,500,000	2,750,000	3,000,000	2,000,000	2,500,000	3,000,000
Marketing	1,000,000	1,250,000	4,000,000	500,000	1,500,000	1,750,000
double check this number - JD						
*Projected Numbers						

- The tidy table is ready for use. The untidy table is not.



When working with real-world data, you will often find that data are stored across **multiple** files or data frames.

- Typically, these tables have to be combined to answer the questions we are interested in.
- Many tables of data are called **relational data**.

Artwork by Allison Horst

Relational data

When one table is not enough

restaurant

name	id	address	type
Taco Stand	AH13JK	1 Main St.	Mexican
Pho Place	**JJ29JJ**	192 Street Rd.	Vietnamese
Taco Stand	XJ11AS	18 W. East St.	Fusion
Pizza Heaven	CI21AA	711 K Ave.	Italian

health inspections

name	id	inspection_date	inspector	score
Taco Stand	AH13JK	2018-08-21	Sheila	97
Pho Place	**JJ29JJ**	2018-03-12	D'eonte	98
Pho Place	**JJ29JJ**	2018-01-02	Monica	66
Taco Stand	XJ11AS	2018-12-16	Mark	43
Pizza Heaven	CI21AA	2018-08-21	Anh	99

rating

name	id	stars
Taco Stand	AH13JK	4.9
Pho Place	**JJ29JJ**	4.8
Taco Stand	XJ11AS	4.2
Pizza Heaven	CI21AA	4.7

Consider a town with a number of restaurants.

Across multiple data files, we have information on

- Location and type of cuisine.
- Health and safety inspections results.
- Online ratings on the restaurant.

Advantages of relational data

Storing data across multiple files has a number of benefits:

- **Efficient data storage:** Limit the need to repeat information.
- **Easier data updates:** If we need to update information, we can make the change in a single file.
- **Privacy:** We can restrict access to some of the data to ensure only those who should have access are able to read the data.

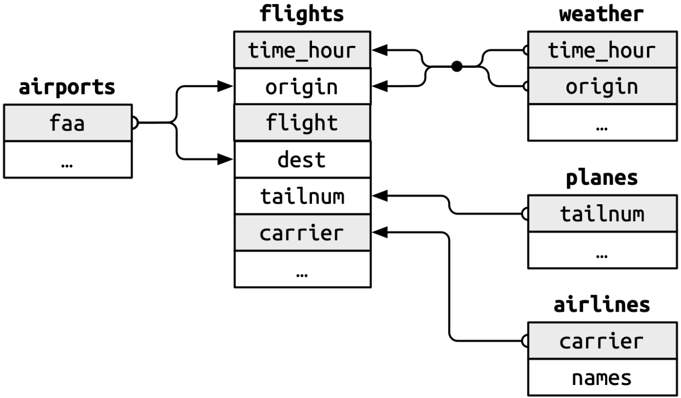
Today, we work with **five** related tables from about flights in the New York City in 2013.

1. `flights`: All flights that departed New York City in 2013.
2. `airlines`: Carrier name and its abbreviated code.
3. `airports`: Information about airports.
4. `planes`: Plane's `tailnum` found in the FAA aircraft registry.
5. `weather`: Weather at each airport in New York at each hour.

The data are available in the nycflights13 package.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import nycflights13
from nycflights13 import flights
from nycflights13 import airlines
from nycflights13 import airports
from nycflights13 import planes
from nycflights13 import weather
```

Here is a diagram (database schema) that identifies the connections between tables:



In `airlines`, we can look up the full carrier name from its abbreviated code.

```
airlines.head()
```

```
  carrier                    name
0      9E       Endeavor Air Inc.
1      AA  American Airlines Inc.
2      AS    Alaska Airlines Inc.
3      B6         JetBlue Airways
4      DL    Delta Air Lines Inc.
```

`airports` gives information about each airport, with a unique `faa` airport code.

```
airports.head()
```

```
   faa                           name        lat  ...  tz  dst
0  04G              Lansdowne Airport  41.130472  ...  -5    A  America/Ne
1  06A  Moton Field Municipal Airport  32.460572  ...  -6    A   America/Ch
2  06C            Schaumburg Regional  41.989341  ...  -6    A   America/Ch
3  06N                Randall Airport  41.431912  ...  -5    A  America/Ne
4  09J          Jekyll Island Airport  31.074472  ...  -5    A  America/Ne

[5 rows x 8 columns]
```

`planes` provides information about each plane, with a unique `tailnum` (tail number).

```
planes.head()
```

```
  tailnum    year                      type  ... seats speed     engine
0  N10156  2004.0  Fixed wing multi engine  ...    55   NaN  Turbo-fan
1  N102UW  1998.0  Fixed wing multi engine  ...   182   NaN  Turbo-fan
2  N103US  1999.0  Fixed wing multi engine  ...   182   NaN  Turbo-fan
3  N104UW  1999.0  Fixed wing multi engine  ...   182   NaN  Turbo-fan
4  N10575  2002.0  Fixed wing multi engine  ...    55   NaN  Turbo-fan

[5 rows x 9 columns]
```

weather provides hourly meterological data for the three airports in New York.

```python
weather.head()
```

```
  origin  year  month  day  ...  precip  pressure  visib              time_l
0    EWR  2013      1    1  ...     0.0    1012.0   10.0  2013-01-01T06:00
1    EWR  2013      1    1  ...     0.0    1012.3   10.0  2013-01-01T07:00
2    EWR  2013      1    1  ...     0.0    1012.5   10.0  2013-01-01T08:00
3    EWR  2013      1    1  ...     0.0    1012.2   10.0  2013-01-01T09:00
4    EWR  2013      1    1  ...     0.0    1011.9   10.0  2013-01-01T10:00

[5 rows x 15 columns]
```

The variable that connects a pair of data sets are called **keys**.

- A variable (or a *minimal* set of variables) that uniquely identifies an observation in a data frame.

In the database schema,

- In the `planes` table, `tailnum` is the key variable.
- In the `weather` table, each observation is uniquely identified by a set of two variables: `time_hour`, and `origin`.

Each data join involves a pair of keys: Primary key and foreign key.

- **Primary key** uniquely identifies an observation in the same table.
- For example, `carrier` is the primary key for `airlines`:

```
airlines.head()
```

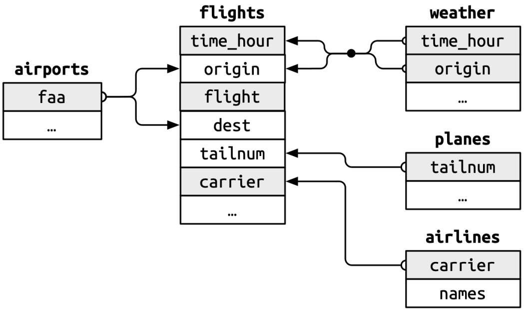
```
  carrier                    name
0      9E       Endeavor Air Inc.
1      AA  American Airlines Inc.
2      AS    Alaska Airlines Inc.
3      B6         JetBlue Airways
4      DL    Delta Air Lines Inc.
```

When more than one variable is needed, the key is called a **compound key**.

- `origin` and `time_hour` are the compound key for the `weather` table:

```
weather.head()
```

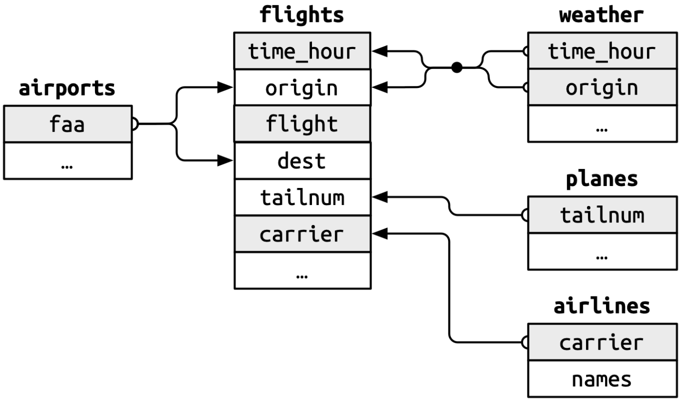
```
  origin  year  month  day  ...  precip  pressure  visib            time_
0    EWR  2013      1    1  ...     0.0    1012.0   10.0  2013-01-01T06:00
1    EWR  2013      1    1  ...     0.0    1012.3   10.0  2013-01-01T07:00
2    EWR  2013      1    1  ...     0.0    1012.5   10.0  2013-01-01T08:00
3    EWR  2013      1    1  ...     0.0    1012.2   10.0  2013-01-01T09:00
4    EWR  2013      1    1  ...     0.0    1011.9   10.0  2013-01-01T10:00

[5 rows x 15 columns]
```



Foreign key is the counterpart of primary key. It uniquely identifies an observation in a different table.

- `flights["carrier"]` is a foreign key that corresponds to the primary key `airlines["carrier"]`.
- A variable can be a primary and a foreign key at the same time.

These relationship can be summarized visually in the database schema.



Once we've identified primary key(s), it is a good practice to verify that they can indeed **uniquely** identify each observation.

- One way to do that is to `value_counts()` the key and look for entries where the count is greater than 1.

```
counts = airlines["carrier"].value_counts().reset_index(name = "n")
counts[counts["n"] > 1]
```

```
Empty DataFrame
Columns: [carrier, n]
Index: []
```

- That means, `carrier` can uniquely identify observations in `airlines`.

Let's turn to check the compound key for the `weather` table.

- From the schema, it appears that `time_hour` and `origin` can identify an observation in `weather`.
- We can check if that's true.

```
counts = weather[["time_hour","origin"]].value_counts()
counts = counts.reset_index(name = "n")
counts[counts["n"] > 1]
```

```
Empty DataFrame
Columns: [time_hour, origin, n]
Index: []
```

We should also check for missing values in primary keys.

- If the key is missing, the variable cannot identify any observation.

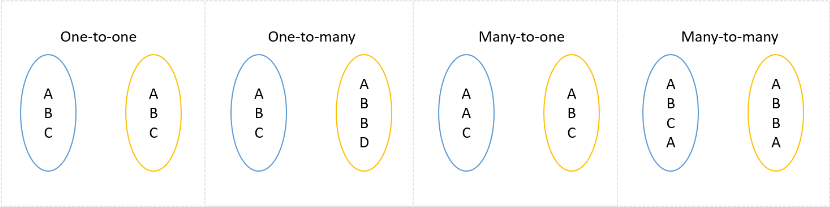
```
checks = planes[planes["tailnum"].isna()]
checks.head()
```

```
Empty DataFrame
Columns: [tailnum, year, type, manufacturer, model, engines, seats, speed,
Index: []
```

```python
checks = weather[weather["time_hour"].isna() | weather["origin"].isna()]
checks.head()
```

```
Empty DataFrame
Columns: [origin, year, month, day, hour, temp, dewp, humid, wind_dir, winc
Index: []
```

- We should also check these for other tables in the database!

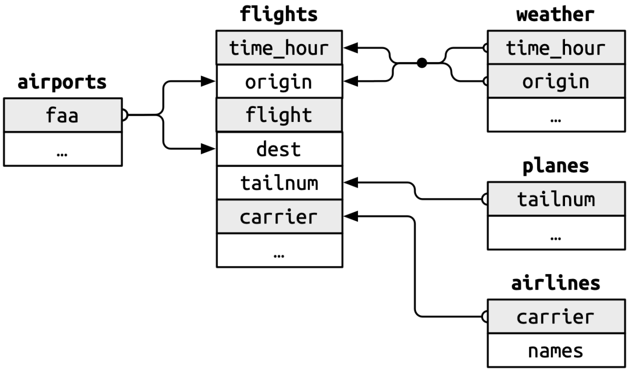


A primary key and the corresponding foreign key forms a **relation**.

- Ideally, relationships are **one-to-one**.
- In real-life data sets, relations are typically **one-to-many** or **many-to-one**:
  - E.g., each flight has one plane, but each plane flies many flights.
- Relations can also be **many-to-many**:
  - Each airline flies to many airports, each airport hosts many airlines.

To work with relational data, we need functions that works with a pair of tables.

- **Mutating joins**: Add new variables to one data frame from matching observations in another data frame.
- **Filtering joins**: Filter observations from one data frame based on whether they can be matched to an observation in another data frame.
- **Inequality joins**.



Let's combine a pair of tables using **mutating join**.

- `flights` and `airlines` via `carrier`.

To ease demonstration, let's first create a narrower data frame that contains fewer variables.

- We name it as `flights2`.

```
flights2 = flights[["time_hour", "origin", "dest", "tailnum", "carrier"]]
flights2.head()
```

```
             time_hour origin dest tailnum carrier
0  2013-01-01T10:00:00Z    EWR  IAH  N14228      UA
1  2013-01-01T10:00:00Z    LGA  IAH  N24211      UA
2  2013-01-01T10:00:00Z    JFK  MIA  N619AA      AA
3  2013-01-01T10:00:00Z    JFK  BQN  N804JB      B6
4  2013-01-01T11:00:00Z    LGA  ATL  N668DN      DL
```

Let's also take a look at the `airlines` table.

```
airlines.head()
```

```
  carrier                          name
0      9E             Endeavor Air Inc.
1      AA        American Airlines Inc.
2      AS          Alaska Airlines Inc.
3      B6               JetBlue Airways
4      DL          Delta Air Lines Inc.
```

- `carrier` is a primary key in `airlines`.
- We can join the `airlines` and `flights2` tables via `carrier`.

Notice that we use a left join (`how = "left"`) on `carrier`.

- The name of the airline is added to the right of the `flights2` table.

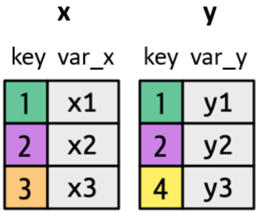
```
df = flights2.merge(airlines, how = "left", on = "carrier")
df.head()
```

```
             time_hour origin dest tailnum carrier                    name
0  2013-01-01T10:00:00Z    EWR  IAH  N14228      UA   United Air Lines Inc
1  2013-01-01T10:00:00Z    LGA  IAH  N24211      UA   United Air Lines Inc
2  2013-01-01T10:00:00Z    JFK  MIA  N619AA      AA  American Airlines Inc
3  2013-01-01T10:00:00Z    JFK  BQN  N804JB      B6        JetBlue Airways
4  2013-01-01T11:00:00Z    LGA  ATL  N668DN      DL   Delta Air Lines Inc
```

In the following, we will learn four types of **mutating joins**.

- All of them can be performed using the `merge()` function. We just need to specify the `how` argument inside of it.
  - `how = "left"`: Left join
  - `how = "right"`: Right join
  - `how = "inner"`: Inner join
  - `how = "outer"`: Outer join

To understand how they work, let's create simpler data sets and use visual representations.




```python
# Create data frames x and y
x = pd.DataFrame({
    "key": [1, 2, 3],
    "val_x": ["x1", "x2", "x3"]
})

y = pd.DataFrame({
    "key": [1, 2, 4],
    "val_y": ["y1", "y2", "y3"]
})
```

x

key	var_x
1	x1
2	x2
3	x3

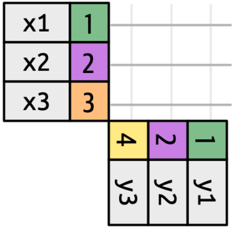
y

key	var_y
1	y1
2	y2
4	y3

- The colored column represents the **key** variable.
- The grey column represents the **value**.

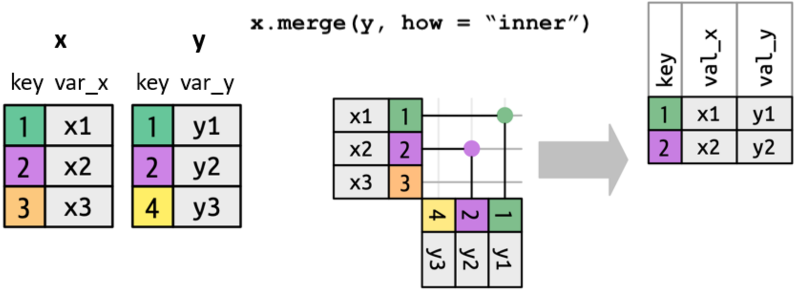
For simplicity, we only show the case with one key variable. But the idea generalizes to multiple keys and multiple values.

A join is a way of connecting each row in table `x` to zero, one, or more rows in table `y`.



- If you look closely, you may notice that we switched the order of the key and value columns in table `x`.
- This is to emphasize that joins matches based on the **key** variable.

In an actual join, matches will be indicated with dots.



- Number of dots = number of matches.
- Different types of joins will result in different number of rows.

Inner join is the simplest type of data joins.

- Matches pairs of observations whenever their keys are equal.
- Keeps observations that appear in **both** tables, and remove the unmatched ones.

```
# Inner join
x.merge(y, how = "inner", on = "key")
```

```
   key val_x val_y
0    1    x1    y1
1    2    x2    y2
```

Left, right, and outer joins

These join types keeps observations that appear in **at least one** of the two tables.

- Left join: Keeps all rows in `x`, including those unmatched to `y`.
- Right join: Keeps all rows in `y`, including those unmatched to `x`.
- Outer join: Keeps all rows in both tables, regardless of matches.
- Cross join: Creates the cartesian product from both `x` and `y`.

These joins work by adding "virtual" observations to each table. The matched observations have their original values, the unmatched ones are filled with `NaN`.

```python
# Left join
x.merge(y, how = "left", on = "key")
```

```
   key val_x val_y
0    1    x1    y1
1    2    x2    y2
2    3    x3   NaN
```

```python
# Right join
x.merge(y, how = "right", on = "key")
```

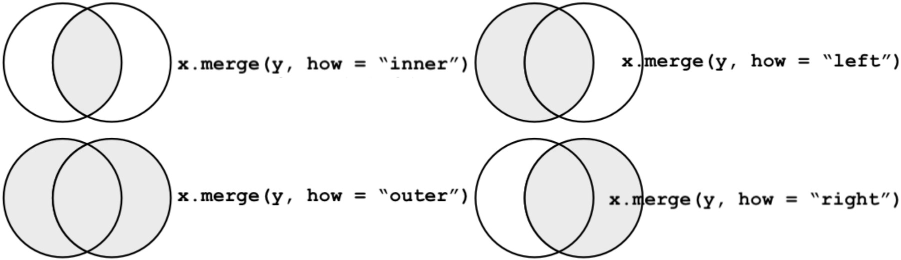
```
   key val_x val_y
0    1    x1    y1
1    2    x2    y2
2    4   NaN    y3
```

```python
# Outer join
x.merge(y, how = "outer", on = "key")
```

```
   key val_x val_y
0    1    x1    y1
1    2    x2    y2
2    3    x3   NaN
3    4   NaN    y3
```

```python
# Cross join (much less often used)
x.merge(y, how = "cross")
```

```
   key_x val_x  key_y val_y
0      1    x1      1    y1
1      1    x1      2    y2
2      1    x1      4    y3
3      2    x2      1    y1
4      2    x2      2    y2
5      2    x2      4    y3
```



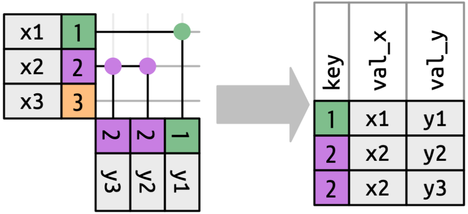
The most common join is **left join**, as it preserves the original observation even when there isn't a match.

- **Left join should be your default join**, unless you have a strong reason to prefer one of the others.

So far, we've explored what happens if a row in x matches zero or one row in y.

This is not always the case.

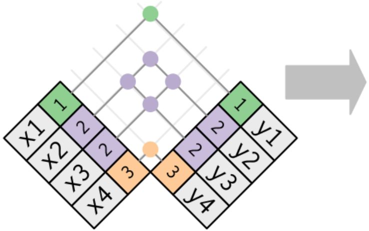
1. If one table has duplicated keys, then the matching row will be duplicated as well.



```python
x = pd.DataFrame({
    "key": [1, 2, 3],
    "val_x": ["x1", "x2", "x3"]
})
y = pd.DataFrame({
    "key": [1, 2, 2],
    "val_y": ["y1", "y2", "y3"]
})
x.merge(y, how = "inner", on = "key")
```

```
   key val_x val_y
0    1    x1    y1
1    2    x2    y2
2    2    x2    y3
```

2. If both table have duplicated keys, you get all possible combinations, the Cartesian product:
   - However, this is usually a data error.
   - In most cases, need to have **unique keys** for at least one of your tables.



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

```python
x = pd.DataFrame({
    "key": [1, 2, 2, 3],
    "val_x": ["x1", "x2", "x3", "x4"]
})
y = pd.DataFrame({
    "key": [1, 2, 2, 3],
    "val_y": ["y1", "y2", "y3", "y4"]
})
x.merge(y, how = "left", on = "key")
```

```
   key val_x val_y
0    1    x1    y1
1    2    x2    y2
2    2    x2    y3
3    2    x3    y2
4    2    x3    y3
5    3    x4    y4
```

Many-to-many joins are particularly **problematic**

- Because they can result in a **size explosion** of the object returned from the join.
- This will have a large impact on the performance of your code.

Let's return to the flights data, `flights2`.

```
flights2.head(10)
```

```
             time_hour origin dest tailnum carrier
0 2013-01-01T10:00:00Z    EWR  IAH  N14228      UA
1 2013-01-01T10:00:00Z    LGA  IAH  N24211      UA
2 2013-01-01T10:00:00Z    JFK  MIA  N619AA      AA
3 2013-01-01T10:00:00Z    JFK  BQN  N804JB      B6
4 2013-01-01T11:00:00Z    LGA  ATL  N668DN      DL
5 2013-01-01T10:00:00Z    EWR  ORD  N39463      UA
6 2013-01-01T11:00:00Z    EWR  FLL  N516JB      B6
7 2013-01-01T11:00:00Z    LGA  IAD  N829AS      EV
8 2013-01-01T11:00:00Z    JFK  MCO  N593JB      B6
9 2013-01-01T11:00:00Z    LGA  ORD  N3ALAA      AA
```

There are several ways to specify the key variables.

1. Specify the argument `on = "key_variable"`.

```
df = flights2.merge(airlines, how = "left", on = "carrier")
df.head()
```

```
            time_hour origin dest tailnum carrier                   name
0  2013-01-01T10:00:00Z    EWR  IAH  N14228      UA   United Air Lines Inc
1  2013-01-01T10:00:00Z    LGA  IAH  N24211      UA   United Air Lines Inc
2  2013-01-01T10:00:00Z    JFK  MIA  N619AA      AA  American Airlines Inc
3  2013-01-01T10:00:00Z    JFK  BQN  N804JB      B6        JetBlue Airways
4  2013-01-01T11:00:00Z    LGA  ATL  N668DN      DL   Delta Air Lines Inc
```

2. Leave the on argument empty. Then the merge() function will use the common variable(s) in the two tables.

   - In the example below, the two tables are joined via carrier.

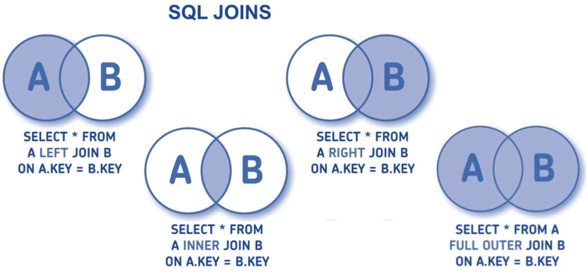
```
df = flights2.merge(airlines, how = "left")
df.head()
```

```
            time_hour origin dest tailnum carrier                    name
0  2013-01-01T10:00:00Z    EWR  IAH  N14228      UA   United Air Lines Inc
1  2013-01-01T10:00:00Z    LGA  IAH  N24211      UA   United Air Lines Inc
2  2013-01-01T10:00:00Z    JFK  MIA  N619AA      AA  American Airlines Inc
3  2013-01-01T10:00:00Z    JFK  BQN  N804JB      B6        JetBlue Airways
4  2013-01-01T11:00:00Z    LGA  ATL  N668DN      DL   Delta Air Lines Inc.
```

3. If the names of the key variables are different in two tables, specify the left_on and right_on parameters.

```
df = flights2.merge(
    airports, how = "left", left_on = "dest", right_on = "faa"
)
df.head()
```

```
              time_hour origin dest tailnum  ...     alt   tz  dst
0  2013-01-01T10:00:00Z    EWR  IAH  N14228  ...    97.0 -6.0    A   Ameri
1  2013-01-01T10:00:00Z    LGA  IAH  N24211  ...    97.0 -6.0    A   Ameri
2  2013-01-01T10:00:00Z    JFK  MIA  N619AA  ...     8.0 -5.0    A  America
3  2013-01-01T10:00:00Z    JFK  BQN  N804JB  ...     NaN  NaN  NaN
4  2013-01-01T11:00:00Z    LGA  ATL  N668DN  ...  1026.0 -5.0    A  America

[5 rows x 13 columns]
```



The pandas user guide provides the full documentation for `merge()`.

The translation to SQL is straightforward:

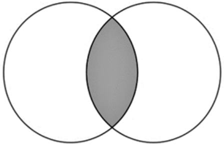
Pandas	**SQL**
x.merge(y, on = "key", how = "left")	SELECT * FROM x LEFT JOIN y ON (key)
x.merge(y, on = "key", how = "right")	SELECT * FROM x RIGHT JOIN y ON (key)
x.merge(y, on = "key", how = "inner")	SELECT * FROM x INNER JOIN y ON (key)
x.merge(y, on = "key", how = "outer")	SELECT * FROM x FULL OUTER JOIN y ON (key)

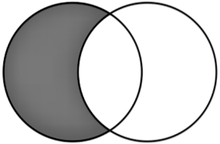
Filtering joins

Filtering joins match observations in the same way as mutating joins, but affect the observations in the final table.

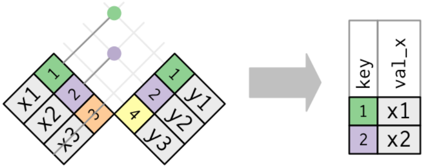
There are two types:

- `isin()`: Keep all observation in `x` that has a match in `y`.
- Negation of `isin()`: Remove all observation in `x` that has a match in `y`.

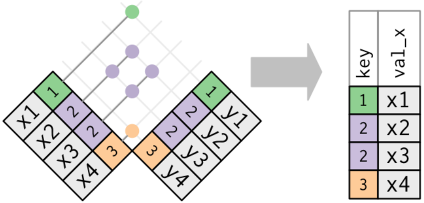




`isin()` keeps only the **matched** observations in `x`.



If there are duplicated keys in `x`, all those rows will be kept.

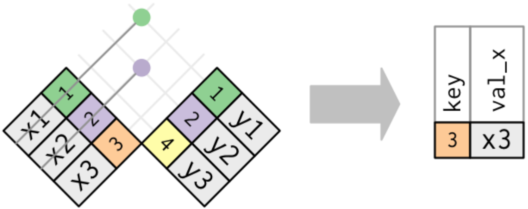


- Find all flights that flew to the most popular destinations:

```
# The most popular destination airports
top = flights["dest"].value_counts().nlargest(1).reset_index(name = "n")
# Keeping flights that flew to that destination
flights2[flights2["dest"].isin(top["dest"])]
```

```
                  time_hour origin dest tailnum carrier
5      2013-01-01T10:00:00Z    EWR  ORD  N39463      UA
9      2013-01-01T11:00:00Z    LGA  ORD  N3ALAA      AA
25     2013-01-01T11:00:00Z    EWR  ORD  N9EAMQ      MQ
38     2013-01-01T11:00:00Z    LGA  ORD  N3CYAA      AA
57     2013-01-01T12:00:00Z    LGA  ORD  N4WNAA      AA
...                     ...    ...  ...     ...     ...
336645 2013-09-30T23:00:00Z    LGA  ORD  N4XBAA      AA
336669 2013-10-01T00:00:00Z    LGA  ORD  N853UA      UA
336675 2013-10-01T00:00:00Z    EWR  ORD  N511MQ      MQ
336696 2013-10-01T00:00:00Z    JFK  ORD  N298JB      B6
336709 2013-10-01T00:00:00Z    LGA  ORD  N434AA      AA
```

The function can also be used to keep only the **unmatched** records.



- It is useful for diagnosing join mismatches.

- Identify the `flights` that do not have a match in `planes`:

```
# Find the unmatched records
unmatched = flights[~flights["tailnum"].isin(planes["tailnum"])]
# Count each unmatched tailnum
counts = unmatched["tailnum"].value_counts().reset_index(name = "n")
counts.head()
```

```
  tailnum    n
0  N725MQ  575
1  N722MQ  513
2  N723MQ  507
3  N713MQ  483
4  N735MQ  396
```

The data we have seen in class have been cleaned up so you have as few problems as possible.

Your own data is unlikely to be so nice.

So there are a few things you should do with your own data to make your joins go more smoothly.

1. Identify the primary keys in each variable.
2. Check that none of the variables in the primary key are missing. If a value is missing, it cannot identify an observation.
3. Check that foreign keys match primary keys in another table.

So far, we've also only seen joins where rows are matched if the `x` key equals the `y` key.

Now we will relax this restriction.

- We shall introduce **inequality joins**.
- It matches rows based on an **inequality condition** between the keys.
- It is supported in SQL. Let's see how it can be done in `pandas`.

```python
# Create two data frames
sales = pd.DataFrame({
    "sales_date": pd.to_datetime(["2024-09-01", "2024-09-03",
                                  "2024-09-14", "2024-09-17"])
})

promos = pd.DataFrame({
    "promo_date": pd.to_datetime(["2024-09-09", "2024-09-15"]),
    "promo_price": [179, 179]
})
```

```
sales
```

```
  sales_date
0 2024-09-01
1 2024-09-03
2 2024-09-14
3 2024-09-17
```

```
promos
```

```
  promo_date  promo_price
0 2024-09-09          179
1 2024-09-15          179
```

- We've learned about **inner join** – matching rows where `sales_date` equals `promo_date`.

```
sales.merge(promos, how = "inner",
            left_on = "sales_date", right_on = "promo_date")
```

```
Empty DataFrame
Columns: [sales_date, promo_date, promo_price]
Index: []
```

- There is no match since `sales_date` and `promo_date` does not exactly equal.

- **Inequality join:** Matching rows where `sales_date` occurs after `promo_date`.

```
df = sales.merge(promos, how = "cross")
df.query("sales_date >= promo_date")
```

```
  sales_date promo_date  promo_price
4 2024-09-14 2024-09-09          179
6 2024-09-17 2024-09-09          179
7 2024-09-17 2024-09-15          179
```

A two-step operation:

- Uses a **cross join**, which joins all rows from both tables.
- Filter the resulting data frame to keep only the rows where `sales_date` occurs after `promo_date`.

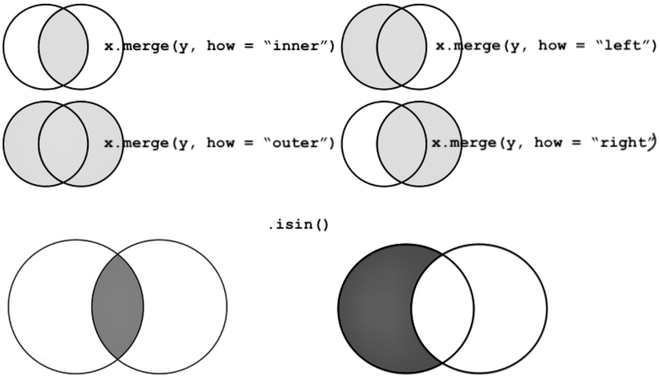
A more elegant (and efficient) way exists, but we need the `pyjanitor` package.

- The function we will be using is `conditional_join()`.

```
import janitor as jn
sales.conditional_join(promos, ("sales_date", "promo_date", ">="),
                       how = "inner")
```

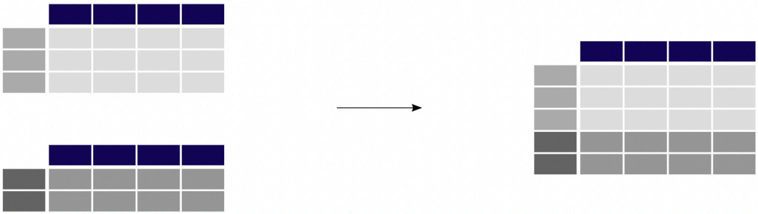
```
  sales_date promo_date  promo_price
0 2024-09-14 2024-09-09          179
1 2024-09-17 2024-09-09          179
2 2024-09-17 2024-09-15          179
```

- **Mutating joins**: Match by key variables and keep columns of both inputs.
- **Filtering joins**: Match by key variables and keep columns of the first input.
- **Inequality joins**: Relax the restrictions on keys being equal.



Concatination

If we have two or more data frames with the same index or the same columns, we can **concatinate** them using `pd.concat()`.



- Create two data frames with the same columns:

```python
# Create two data frames
sales = pd.DataFrame({
    'sales_date': pd.to_datetime(["2024-09-01", "2024-09-03",
                                  "2024-09-14", "2024-09-17"])
})

sales1 = pd.DataFrame({
    'sales_date': pd.to_datetime(["2024-08-01"])
})
```

1. Concatenate data frames **vertically** (i.e., appending data frames).
    - `axis = 0` specifies that the tables should be concatinated along rows.
    - `ignore_index = True` tells the function to ignore the original row indices, and create a new set of index starting from 0.

```
## Concatenate the data frames vertically
pd.concat([sales, sales1], axis = 0, ignore_index = True)
```

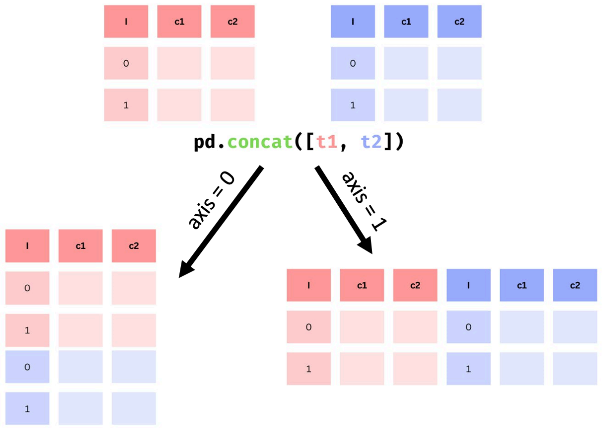
```
   sales_date
0  2024-09-01
1  2024-09-03
2  2024-09-14
3  2024-09-17
4  2024-08-01
```

2. Concatenate data frames **horizontally**.
   - `axis = 1` specifies that the tables should be concatinated along columns.

```
prices = pd.DataFrame({
    "quantity": [3, 9, 1, 2],
    "price": [209, 209, 179, 179]
})

## Concatenate the data frames horizontally
pd.concat([sales, prices], axis = 1)
```

```
  sales_date  quantity  price
0 2024-09-01         3    209
1 2024-09-03         9    209
2 2024-09-14         1    179
3 2024-09-17         2    179
```

Summary of concatination



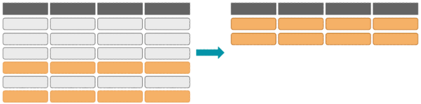
A review

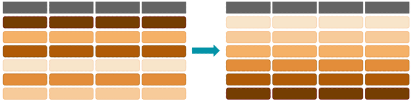
Customarily, we import them as:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

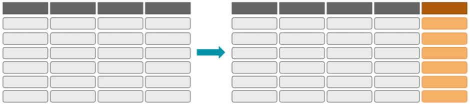
- `pandas` functions can handle many different types of files.
  - `read_csv()`, `read_excel()`, `read_html()`, ...
- For `json` files, we use the `json` package.
- For API requests, we use the `requests` package.
- Data can be stored in packages too. So far we've used `pydataset` and `nycflights13` for that.

Row manipulations



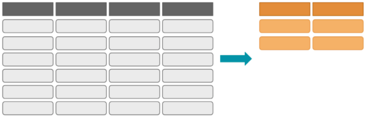


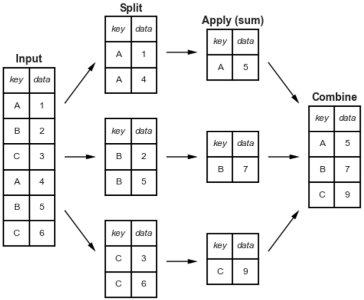
Column manipulations



Data manipulation

Groups and summaries



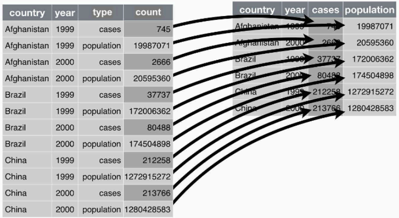


Wide to long: `melt()`

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

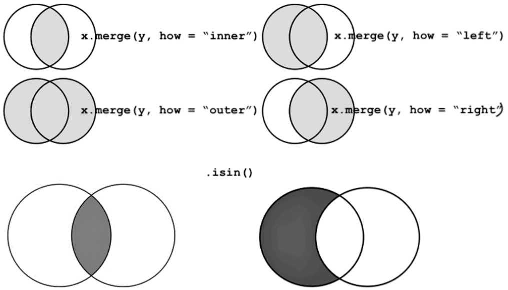
country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

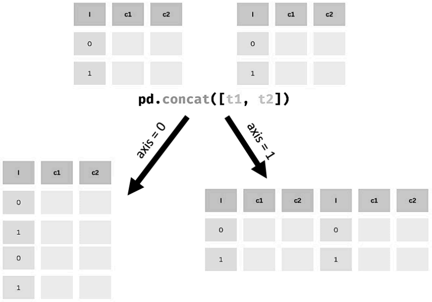
Long to wide: `pivot()`



country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583





- For specific tasks, functions in packages like `pyjanitor` can be useful as well.

So far, we've primarily used `pandas` and `matplotlib` for visualization.

- Histogram: For quantitative (continuous) variables.
- Bar chart: For qualitative (categorical) variables.
- Line chart: Time-series data.

To visualize more more than two variables on a chart, it's easier with `seaborn`, especially when we have **tidy data**.

- We will cover more about visualization after the midterm exam.