CM4125 Topic 6: Data Manipulation (Part 2)

Joins and Aggregations

Let's create two small dataset to use as example

Let's create two small dataset to use as example

In [1]:
```
import pandas as pd
books = pd.DataFrame({'Author' : ['J. R. R. Tolkien',
                                  'George R. R. Martin',
                                  'J. K. Rowling',
                                  'Suzanne Collins']},
                     index = ['The Lord of the Rings',
                              'Game of Thrones',
                              'Harry Potter',
                              'The Hunger Games'])
books
```

Out[1]:

	Author
The Lord of the Rings	J. R. R. Tolkien
Game of Thrones	George R. R. Martin
Harry Potter	J. K. Rowling
The Hunger Games	Suzanne Collins

Let's create two small dataset to use as example

In [1]:
```
import pandas as pd
books = pd.DataFrame({'Author' : ['J. R. R. Tolkien',
                                  'George R. R. Martin',
                                  'J. K. Rowling',
                                  'Suzanne Collins']},
                     index = ['The Lord of the Rings',
                              'Game of Thrones',
                              'Harry Potter',
                              'The Hunger Games'])
books
```

Out[1]:

	Author
The Lord of the Rings	J. R. R. Tolkien
Game of Thrones	George R. R. Martin
Harry Potter	J. K. Rowling
The Hunger Games	Suzanne Collins

In [2]:
```
films = pd.DataFrame({'Year of First Film' : [1999, 2001, 2001, 2012],
                      'Number of Films' : [3, 2, 8, 4]},
                     index = ['The Matrix',
                              'The Lord of the Rings',
                              'Harry Potter',
                              'The Hunger Games'])


films
```

Out[2]:

	Year of First Film	Number of Films
The Matrix	1999	3
The Lord of the Rings	2001	2
Harry Potter	2001	8
The Hunger Games	2012	4

The `.join` operation is used to join the columns of two different datasets based on matching index

The `.join` operation is used to join the columns of two different datasets based on matching index

In [3]:
```
books.join(films)
```

Out[3]:

	Author	Year of First Film	Number of Films
The Lord of the Rings	J. R. R. Tolkien	2001.0	2.0
Game of Thrones	George R. R. Martin	NaN	NaN
Harry Potter	J. K. Rowling	2001.0	8.0
The Hunger Games	Suzanne Collins	2012.0	4.0

Types of Joins

In the previous example, the `books` dataset is the '*left*' dataset and `films` is the '*right*' dataset

In the previous example, the `books` dataset is the '*left*' dataset and `films` is the '*right*' dataset

A left join keeps all of the data from the '*left*' dataset and adds in the applicable data from the '*right*' dataset where the keys match up

In the previous example, the `books` dataset is the '*left*' dataset and `films` is the '*right*' dataset

A left join keeps all of the data from the '*left*' dataset and adds in the applicable data from the '*right*' dataset where the keys match up

If there is no film, the cells are populated with `NaN` (e.g. Game of Thrones)

In the previous example, the `books` dataset is the '*left*' dataset and `films` is the '*right*' dataset

A left join keeps all of the data from the '*left*' dataset and adds in the applicable data from the '*right*' dataset where the keys match up

If there is no film, the cells are populated with `NaN` (e.g. Game of Thrones)

The above operations is equivalent to

```
books.join(films, how='left')
```

We can also do a right join. This keeps all of the films, adding the book data where applicable. If there is no book, the cells are populated with `NaN`.

We can also do a right join. This keeps all of the films, adding the book data where applicable. If there is no book, the cells are populated with `NaN`.

In [4]:
```
books.join(films, how='right')
```

Out[4]:

	Author	Year of First Film	Number of Films
The Matrix	NaN	1999	3
The Lord of the Rings	J. R. R. Tolkien	2001	2
Harry Potter	J. K. Rowling	2001	8
The Hunger Games	Suzanne Collins	2012	4

We can also do a right join. This keeps all of the films, adding the book data where applicable. If there is no book, the cells are populated with `NaN`.

In [4]:
```
books.join(films, how='right')
```

Out[4]:

	Author	Year of First Film	Number of Films
The Matrix	NaN	1999	3
The Lord of the Rings	J. R. R. Tolkien	2001	2
Harry Potter	J. K. Rowling	2001	8
The Hunger Games	Suzanne Collins	2012	4

This is almost equivalent to `films.join(books)` (or `films.join(books, how='left')`), with the exception of the order in which the columns appear

If we want to keep all of the data (book **OR** film), we can use an outer join

If we want to keep all of the data (book **OR** film), we can use an outer join

In [5]:
```
films.join(books, how='outer')
```

Out[5]:

	Year of First Film	Number of Films	Author
Game of Thrones	NaN	NaN	George R. R. Martin
Harry Potter	2001.0	8.0	J. K. Rowling
The Hunger Games	2012.0	4.0	Suzanne Collins
The Lord of the Rings	2001.0	2.0	J. R. R. Tolkien
The Matrix	1999.0	3.0	NaN

And if we only want to keep the data (book **AND** film), we can use an inner join

And if we only want to keep the data (book **AND** film), we can use an inner join

In [6]:
```
films.join(books, how='inner')
```

Out[6]:

	Year of First Film	Number of Films	Author
The Lord of the Rings	2001	2	J. R. R. Tolkien
Harry Potter	2001	8	J. K. Rowling
The Hunger Games	2012	4	Suzanne Collins

You should choose the join type based on what your resulting data table is intended to describe

You should choose the join type based on what your resulting data table is intended to describe

For instance, the inner join gave us a table of films based on books

You should choose the join type based on what your resulting data table is intended to describe

For instance, the inner join gave us a table of films based on books

Contrarily, the left join gave us a list of books with additional information on the film (if any)

Once again, remember that `.join` is not modifying the dataframe, so if you want to save the result, assign it to either the same or a different variable with an appropriate name

```
films = films.join(books, how='left')

books = films.join(books, how='right')

films_on_books = films.join(books, how='inner')

favourite_series = films.join(books, how='outer')
```

This table may be used as a reminder of the difference between the joins

This table may be used as a reminder of the difference between the joins

Type of Join	Keeps Rows of Left Data	Keeps Rows of Right Data
left (default)	yes	only if matching left
right	only if matching right	yes
outer	yes	yes
inner	only if matching right	only if matching left

Joining Different Columns

`.join` joins by comparing indexes of each dataframe

`.join` joins by comparing indexes of each dataframe

Sometimes the key column(s) is not the index (particularly if you are using default indexing)

`.join` joins by comparing indexes of each dataframe

Sometimes the key column(s) is not the index (particularly if you are using default indexing)

If you need to join based on columns other than the index, you should use `merge`

For example, it is possible that we would encounter data with default indexes as follows:

For example, it is possible that we would encounter data with default indexes as follows:

In [7]:
```
books = books.reset_index()
books = books.rename(columns={'index' : 'Book Series Title'})
books
```

Out[7]:

	Book Series Title	Author
0	The Lord of the Rings	J. R. R. Tolkien
1	Game of Thrones	George R. R. Martin
2	Harry Potter	J. K. Rowling
3	The Hunger Games	Suzanne Collins

For example, it is possible that we would encounter data with default indexes as follows:

In [7]:
```
books = books.reset_index()
books = books.rename(columns={'index' : 'Book Series Title'})
books
```

Out[7]:

	Book Series Title	Author
0	The Lord of the Rings	J. R. R. Tolkien
1	Game of Thrones	George R. R. Martin
2	Harry Potter	J. K. Rowling
3	The Hunger Games	Suzanne Collins

In [8]:
```
films = films.reset_index()
films = films.rename(columns={'index' : 'Film Series Title'})
films
```

Out[8]:

	Film Series Title	Year of First Film	Number of Films
0	The Matrix	1999	3
1	The Lord of the Rings	2001	2
2	Harry Potter	2001	8
3	The Hunger Games	2012	4

If we join on the index, the result is nonsense!

If we join on the index, the result is nonsense!

In [9]:
```
books.join(films) # WRONG
```

Out[9]:

	Book Series Title	Author	Film Series Title	Year of First Film	Number of Films
0	The Lord of the Rings	J. R. R. Tolkien	The Matrix	1999	3
1	Game of Thrones	George R. R. Martin	The Lord of the Rings	2001	2
2	Harry Potter	J. K. Rowling	Harry Potter	2001	8
3	The Hunger Games	Suzanne Collins	The Hunger Games	2012	4

We could change the `Book Series Title` and `Film Series Title` to indexes and join with `.join`

We could change the `Book Series Title` and `Film Series Title` to indexes and join with `.join`

Or we can use `.merge`, in which left, right, outer, and inner joins work the same way

We could change the `Book Series Title` and `Film Series Title` to indexes and join with `.join`

Or we can use `.merge`, in which left, right, outer, and inner joins work the same way

However, it is not the index we are comparing, it is the column specified with `left_on=` and `right_on=`

In [10]:
```
books.merge(films,
            how='inner',
            left_on='Book Series Title',
            right_on='Film Series Title')
```

Out[10]:

	Book Series Title	Author	Film Series Title	Year of First Film	Number of Films
0	The Lord of the Rings	J. R. R. Tolkien	The Lord of the Rings	2001	2
1	Harry Potter	J. K. Rowling	Harry Potter	2001	8
2	The Hunger Games	Suzanne Collins	The Hunger Games	2012	4

Data Aggregation

Before continuing, we will reset the index of the books

Before continuing, we will reset the index of the books

In [11]:
```
books = books.set_index('Book Series Title', drop=True)
books
```

Out[11]:

Book Series Title	Author
The Lord of the Rings	J. R. R. Tolkien
Game of Thrones	George R. R. Martin
Harry Potter	J. K. Rowling
The Hunger Games	Suzanne Collins

Let's import a dataset of a list of books

Let's import a dataset of a list of books

In [12]:
```
volumes = pd.read_csv('https://www.dropbox.com/s/9flqjjvetgbex97/volumes.csv?r
volumes
```

Out[12]:

	Series	Title	Rating	Year
0	Harry Potter	Harry Potter and the Philosopher's Stone	4.47	1997
1	Harry Potter	Harry Potter and the Chamber of Secrets	4.42	1998
2	Harry Potter	Harry Potter and the Prisoner of Azkaban	4.56	1999
3	Harry Potter	Harry Potter and the Goblet of Fire	4.55	2000
4	Harry Potter	Harry Potter and the Order of the Phoenix	4.49	2003
5	Harry Potter	Harry Potter and the Half-Blood Prince	4.57	2005
6	Harry Potter	Harry Potter and the Deathly Hallows	4.61	2007
7	The Lord of the Rings	The Fellowship of the Ring	4.36	1954
8	The Lord of the Rings	The Two Towers	4.44	1954
9	The Lord of the Rings	The Return of the King	4.53	1955
10	Game of Thrones	A Game of Thrones	4.45	1996
11	Game of Thrones	A Clash of Kings	4.41	1998
12	Game of Thrones	A Storm of Swords	4.54	2000
13	Game of Thrones	A Feast for Crows	4.13	2005
14	Game of Thrones	A Dance with Dragons	4.33	2011
15	The Hunger Games	The Hunger Games	4.33	2008
16	The Hunger Games	Catching Fire	4.29	2009
17	The Hunger Games	Mockingjay	4.03	2010

We want to summarise by series, the `.groupby` method gives you the name of the column which has the groups

We want to summarise by series, the `.groupby` method gives you the name of the column which has the groups

The result is a Python object which we will use for the next step

We want to summarise by series, the `.groupby` method gives you the name of the column which has the groups

The result is a Python object which we will use for the next step

In [13]:
```
volumes_notitle = volumes.drop(columns=['Title']) # Remove title, otherwise wa
groups = volumes_notitle.groupby('Series')
groups
```

Out[13]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001D0A04
36B10>
```

We will use `count` and `mean` to work out the number of books and the average rating, and use `min` to work out the first publication year.

We will use `count` and `mean` to work out the number of books and the average rating, and use `min` to work out the first publication year.

Other operations available include `sum` and `max` (you can check others here)

Now we have our data summary by groups

Now we have our data summary by groups

In [14]:
```
summary = groups.agg({'count', 'mean', 'min'})
summary
```

Out[14]:

	Rating			Year		
	count	min	mean	count	min	mean
Series						
Game of Thrones	5	4.13	4.372000	5	1996	2002.000000
Harry Potter	7	4.42	4.524286	7	1997	2001.285714
The Hunger Games	3	4.03	4.216667	3	2008	2009.000000
The Lord of the Rings	3	4.36	4.443333	3	1954	1954.333333

Joining Aggregated Data

We now have a dataframe with two sub-frames (one for `Rating` and one for `Year`) we can easily separate them

We now have a dataframe with two sub-frames (one for `Rating` and one for `Year`) we can easily separate them

In [15]:
```
rating_summary = summary['Rating']
rating_summary
```

Out[15]:

Series	count	min	mean
Game of Thrones	5	4.13	4.372000
Harry Potter	7	4.42	4.524286
The Hunger Games	3	4.03	4.216667
The Lord of the Rings	3	4.36	4.443333

We now have a dataframe with two sub-frames (one for `Rating` and one for `Year`) we can easily separate them

In [15]:
```
rating_summary = summary['Rating']
rating_summary
```

Out[15]:

Series	count	min	mean
Game of Thrones	5	4.13	4.372000
Harry Potter	7	4.42	4.524286
The Hunger Games	3	4.03	4.216667
The Lord of the Rings	3	4.36	4.443333

In [16]:
```
year_summary = summary['Year']
year_summary
```

Out[16]:

Series	count	min	mean
Game of Thrones	5	1996	2002.000000
Harry Potter	7	1997	2001.285714
The Hunger Games	3	2008	2009.000000
The Lord of the Rings	3	1954	1954.333333

Let's rename the columns for the `Ratings` aggregation, and remove the unneeded `min` column

Let's rename the columns for the `Ratings` aggregation, and remove the unneeded `min` column

In [17]:
```
rating_summary = summary['Rating'].rename(
    columns={'mean' : 'Average Rating',
             'count' : 'Number of Books'})
rating_summary = rating_summary.drop(columns={'min'})
rating_summary
```

Out[17]:

Series	Number of Books	Average Rating
Game of Thrones	5	4.372000
Harry Potter	7	4.524286
The Hunger Games	3	4.216667
The Lord of the Rings	3	4.443333

We may also want to round off the average ratings

We may also want to round off the average ratings

In [18]:
```
rating_summary['Average Rating'] = rating_summary['Average Rating'].round(2)

rating_summary
```

Out[18]:

Series	Number of Books	Average Rating
Game of Thrones	5	4.37
Harry Potter	7	4.52
The Hunger Games	3	4.22
The Lord of the Rings	3	4.44

Let's also rename the column from the `Year` aggregation and drop the other columns

Let's also rename the column from the `Year` aggregation and drop the other columns

In [19]:
```
year_summary = year_summary.rename(columns={'min' : 'First Published'})
year_summary = year_summary.drop(columns={'mean', 'count'})
year_summary
```

Out[19]:

Series	First Published
Game of Thrones	1996
Harry Potter	1997
The Hunger Games	2008
The Lord of the Rings	1954

Now we can join the `books`, `rating_summary` and `year_summary` dataframes

Now we can join the `books`, `rating_summary` and `year_summary` dataframes

Since all have the same keys we don't need to worry about join type, but this is a left join so will keep everything in the `books` data frame if it didn't match

Now we can join the `books`, `rating_summary` and `year_summary` dataframes

Since all have the same keys we don't need to worry about join type, but this is a left join so will keep everything in the `books` data frame if it didn't match

In [20]:
```
books.join(rating_summary).join(year_summary)
```

Out[20]:

Book Series Title	Author	Number of Books	Average Rating	First Published
The Lord of the Rings	J. R. R. Tolkien	3	4.44	1954
Game of Thrones	George R. R. Martin	5	4.37	1996
Harry Potter	J. K. Rowling	7	4.52	1997
The Hunger Games	Suzanne Collins	3	4.22	2008

How are aggregations ”done” in Tableau?

Melt and Pivoting

Wide vs Long Data (a real-life example)

Wide data contains a column for each variable, and a row for each entity

Wide data contains a column for each variable, and a row for each entity

The "entity" ID (in this case `Name`, but it could be an ID, a number, etc.) is in the first column, or it could be the index

Wide data contains a column for each variable, and a row for each entity

The "entity" ID (in this case `Name`, but it could be an ID, a number, etc.) is in the first column, or it could be the index

Name	Age	Height	Hair Colour
Alice	36	1.68	Blonde
Bob	28	1.73	Red
Charlie	29	1.60	-

Long data contains a row for each observation of a variable

Long data contains a row for each observation of a variable

This is also called entity-attribute-value data

Long data contains a row for each observation of a variable

This is also called entity-attribute-value data

Note that rows can be omitted if there is missing data

Long data contains a row for each observation of a variable

This is also called entity-attribute-value data

Note that rows can be omitted if there is missing data

Entity ID	Attribute / Variable	Value
Alice	Age	36
Bob	Age	28
Charlie	Age	29
Alice	Height	1.68
Bob	Height	1.73
Charlie	Height	1.60
Alice	Hair Colour	Blonde
Bob	Hair Colour	Red

Tidy Data

Tidy Data

Defined by Hadley Wickham in this article, it describes long and wide data, and gives more examples on how to better work with both

Tidy Data

Defined by Hadley Wickham in this article, it describes long and wide data, and gives more examples on how to better work with both

It is written for R users!

Melting Wide to Long Data

Once again, let's import a new dataset

Once again, let's import a new dataset

In [21]:
```
stock = pd.read_csv('https://www.dropbox.com/s/dl0bz061v5biv4k/stock.csv?raw=1
stock
```

Out[21]:

	Company	Symbol	1980	1990	2000	2010	2020
0	Apple	AAPL	0.51	1.22	3.88	37.53	318.73
1	Google	GOOGL	NaN	NaN	NaN	312.54	1518.73
2	Microsoft	MSFT	NaN	1.03	53.31	28.21	185.38

Once again, let's import a new dataset

In [21]:
```
stock = pd.read_csv('https://www.dropbox.com/s/dl0bz061v5biv4k/stock.csv?raw=1
stock
```

Out[21]:

	Company	Symbol	1980	1990	2000	2010	2020
0	Apple	AAPL	0.51	1.22	3.88	37.53	318.73
1	Google	GOOGL	NaN	NaN	NaN	312.54	1518.73
2	Microsoft	MSFT	NaN	1.03	53.31	28.21	185.38

What kind of data is this?

Once again, let's import a new dataset

In [21]:
```
stock = pd.read_csv('https://www.dropbox.com/s/dl0bz061v5biv4k/stock.csv?raw=1
stock
```

Out[21]:

	Company	Symbol	1980	1990	2000	2010	2020
0	Apple	AAPL	0.51	1.22	3.88	37.53	318.73
1	Google	GOOGL	NaN	NaN	NaN	312.54	1518.73
2	Microsoft	MSFT	NaN	1.03	53.31	28.21	185.38

What kind of data is this?

Which are the entities/ID columns?

Once again, let's import a new dataset

In [21]:
```
stock = pd.read_csv('https://www.dropbox.com/s/dl0bz061v5biv4k/stock.csv?raw=1
stock
```

Out[21]:

	Company	Symbol	1980	1990	2000	2010	2020
0	Apple	AAPL	0.51	1.22	3.88	37.53	318.73
1	Google	GOOGL	NaN	NaN	NaN	312.54	1518.73
2	Microsoft	MSFT	NaN	1.03	53.31	28.21	185.38

What kind of data is this?

Which are the entities/ID columns?

Which are the attributes (variables) in this case?

Let's melt this data using the `.melt` method

Let's melt this data using the `.melt` method

In [22]:
```
stock.melt(id_vars = ['Company', 'Symbol'])
```

Out[22]:

	Company	Symbol	variable	value
0	Apple	AAPL	1980	0.51
1	Google	GOOGL	1980	NaN
2	Microsoft	MSFT	1980	NaN
3	Apple	AAPL	1990	1.22
4	Google	GOOGL	1990	NaN
5	Microsoft	MSFT	1990	1.03
6	Apple	AAPL	2000	3.88
7	Google	GOOGL	2000	NaN
8	Microsoft	MSFT	2000	53.31
9	Apple	AAPL	2010	37.53
10	Google	GOOGL	2010	312.54
11	Microsoft	MSFT	2010	28.21
12	Apple	AAPL	2020	318.73
13	Google	GOOGL	2020	1518.73
14	Microsoft	MSFT	2020	185.38

We can ensure that the new columns get named correctly by using the `var_name` and `value_name` options

We can ensure that the new columns get named correctly by using the `var_name` and `value_name` options

In [23]:
```
stock = stock.melt(id_vars = ['Company', 'Symbol'], var_name='Year', value_nam
stock
```

Out[23]:

	Company	Symbol	Year	Price (USD)
0	Apple	AAPL	1980	0.51
1	Google	GOOGL	1980	NaN
2	Microsoft	MSFT	1980	NaN
3	Apple	AAPL	1990	1.22
4	Google	GOOGL	1990	NaN
5	Microsoft	MSFT	1990	1.03
6	Apple	AAPL	2000	3.88
7	Google	GOOGL	2000	NaN
8	Microsoft	MSFT	2000	53.31
9	Apple	AAPL	2010	37.53
10	Google	GOOGL	2010	312.54
11	Microsoft	MSFT	2010	28.21
12	Apple	AAPL	2020	318.73
13	Google	GOOGL	2020	1518.73
14	Microsoft	MSFT	2020	185.38

The `NaN` values aren't really contributing anything and are just there because they were in the wide dataset, so let's remove them!

The `NaN` values aren't really contributing anything and are just there because they were in the wide dataset, so let's remove them!

In [24]:
```
stock = stock[stock['Price (USD)'].notnull()]
stock
```

Out[24]:

	Company	Symbol	Year	Price (USD)
0	Apple	AAPL	1980	0.51
3	Apple	AAPL	1990	1.22
5	Microsoft	MSFT	1990	1.03
6	Apple	AAPL	2000	3.88
8	Microsoft	MSFT	2000	53.31
9	Apple	AAPL	2010	37.53
10	Google	GOOGL	2010	312.54
11	Microsoft	MSFT	2010	28.21
12	Apple	AAPL	2020	318.73
13	Google	GOOGL	2020	1518.73
14	Microsoft	MSFT	2020	185.38

If you prefer the data sorted by entity, then variable, you can re-sort

If you prefer the data sorted by entity, then variable, you can re-sort

In [25]:
```
stock.sort_values(['Symbol', 'Year'])
```

Out[25]:

	Company	Symbol	Year	Price (USD)
0	Apple	AAPL	1980	0.51
3	Apple	AAPL	1990	1.22
6	Apple	AAPL	2000	3.88
9	Apple	AAPL	2010	37.53
12	Apple	AAPL	2020	318.73
10	Google	GOOGL	2010	312.54
13	Google	GOOGL	2020	1518.73
5	Microsoft	MSFT	1990	1.03
8	Microsoft	MSFT	2000	53.31
11	Microsoft	MSFT	2010	28.21
14	Microsoft	MSFT	2020	185.38

Once again, Python has not detected that the columns were integer types

Once again, Python has not detected that the columns were integer types

In [26]:
```
stock[stock['Year'] >= 2000]  # Error!
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
Cell In[26], line 1
----> 1 stock[stock['Year'] >= 2000]

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\ops
\common.py:76, in _unpack_zerodim_and_defer.<locals>.new_method(self,
other)
     72             return NotImplemented
     74 other = item_from_zerodim(other)
---> 76 return method(self, other)

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\arra
ylike.py:60, in OpsMixin.__ge__(self, other)
     58 @unpack_zerodim_and_defer("__ge__")
     59 def __ge__(self, other):
---> 60     return self._cmp_method(other, operator.ge)

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\seri
es.py:6119, in Series._cmp_method(self, other, op)
   6116 lvalues = self._values
   6117 rvalues = extract_array(other, extract_numpy=True, extract_ran
ge=True)
-> 6119 res_values = ops.comparison_op(lvalues, rvalues, op)
   6121 return self._construct_result(res_values, name=res_name)

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\ops
\array_ops.py:344, in comparison_op(left, right, op)
    341     return invalid_comparison(lvalues, rvalues, op)
```

As you can see, the data type is `object`, meaning these numbers are stored as `str`

As you can see, the data type is `object`, meaning these numbers are stored as `str`

In [27]:
```
stock['Year']
```

Out[27]:
```
0     1980
3     1990
5     1990
6     2000
8     2000
9     2010
10    2010
11    2010
12    2020
13    2020
14    2020
Name: Year, dtype: object
```

The columns which was originally headings has become `object` (`str`) types

The columns which was originally headings has become `object` (`str`) types

We can change the data type of the `Year` column using `astype`

The columns which was originally headings has become `object` (`str`) types

We can change the data type of the `Year` column using `astype`

In [28]:
```
stock = stock.astype({'Year' : 'int64'})
stock
```

Out[28]:

	Company	Symbol	Year	Price (USD)
0	Apple	AAPL	1980	0.51
3	Apple	AAPL	1990	1.22
5	Microsoft	MSFT	1990	1.03
6	Apple	AAPL	2000	3.88
8	Microsoft	MSFT	2000	53.31
9	Apple	AAPL	2010	37.53
10	Google	GOOGL	2010	312.54
11	Microsoft	MSFT	2010	28.21
12	Apple	AAPL	2020	318.73
13	Google	GOOGL	2020	1518.73
14	Microsoft	MSFT	2020	185.38

Now we are able to do numerical comparisons on the `Year` column

Now we are able to do numerical comparisons on the `Year` column

In [29]:
```
stock[stock['Year'] >= 2000]
```

Out[29]:

	Company	Symbol	Year	Price (USD)
6	Apple	AAPL	2000	3.88
8	Microsoft	MSFT	2000	53.31
9	Apple	AAPL	2010	37.53
10	Google	GOOGL	2010	312.54
11	Microsoft	MSFT	2010	28.21
12	Apple	AAPL	2020	318.73
13	Google	GOOGL	2020	1518.73
14	Microsoft	MSFT	2020	185.38

Pivoting Long to Wide Data

A new dataset...

A new dataset...

In [30]:
```
weather = pd.read_csv('https://www.dropbox.com/s/1pyru339tll1njf/weather-canad
weather
```

Out[30]:

	Station Name	Province	Year	Mean Temperature (C)	Total Precipitation (mm)
0	BEAR CREEK	BC	1971	15.4	20.9
1	COWICHAN BAY CHERRY POINT	BC	1971	17.4	12.8
2	COWICHAN LAKE FORESTRY	BC	1971	18.8	21.3
3	COWICHAN LAKE VILLAGE	BC	1971	17.7	36.4
4	DUNCAN FORESTRY	BC	1971	17.7	18.1
...	...	...	...	...	...
81757	GOOSE A	NL	2017	15.8	109.0
81758	HOPEDALE (AUT)	NL	2017	11.6	83.2
81759	MARY'S HARBOUR A	NL	2017	14.5	56.9
81760	NAIN	NL	2017	10.6	38.3
81761	WABUSH A	NL	2017	12.9	129.0

81762 rows × 5 columns

This was long data, however we have multiple variables per observation

This was long data, however we have multiple variables per observation

We can pivot the table to see each station with the entry corresponding to different years

This was long data, however we have multiple variables per observation

We can pivot the table to see each station with the entry corresponding to different years

In [31]:
```
weather_p = weather.pivot(index='Station Name',
                          columns='Year',
                          values=['Mean Temperature (C)',
                                  'Total Precipitation (mm)'])
weather_p
```

Out[31]:

	Mean Temperature (C)										...
Year	**1971**	**1972**	**1973**	**1974**	**1975**	**1976**	**1977**	**1978**	**1979**	**1980**	**...**
Station Name											
(AE) BOW SUMMIT	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
100 MILE HOUSE	16.0	15.1	15.1	13.7	17.2	14.5	13.5	15.5	16.0	14.1	...
100 MILE HOUSE 6NE	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
108 MILE HOUSE	NaN	NaN	15.9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
108 MILE HOUSE ABEL LAKE	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
...	...	...	...	...	...	...	...	...	...	...	...
YOYO	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
ZAMA LO	15.7	14.3	15.3	13.5	17.5	14.3	13.6	14.6	18.1	16.0	...
ZEBALLOS MURAUDE CREEK	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...

You can see that the method was capable of "grouping" data, either for mean temp or precipitations!

You can see that the method was capable of "grouping" data, either for mean temp or precipitations!

This allows us to create new dataframes based only on the required info

You can see that the method was capable of "grouping" data, either for mean temp or precipitations!

This allows us to create new dataframes based only on the required info

In [32]:
```
temperature = weather_p['Mean Temperature (C)']
temperature
```

Out[32]:

Year	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	...
Station Name											
(AE) BOW SUMMIT	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
100 MILE HOUSE	16.0	15.1	15.1	13.7	17.2	14.5	13.5	15.5	16.0	14.1	...
100 MILE HOUSE 6NE	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
108 MILE HOUSE	NaN	NaN	15.9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
108 MILE HOUSE ABEL LAKE	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
...	...	...	...	...	...	...	...	...	...	...	...
YOYO	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
ZAMA LO	15.7	14.3	15.3	13.5	17.5	14.3	13.6	14.6	18.1	16.0	...
ZEBALLOS MURAUDE CREEK	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
ZEHNER	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...

In [33]:
```
temp2010 = temperature[[2010]]
temp2010 = temp2010[temp2010[2010].notnull()]
temp2010
```

Out[33]:

Year	2010
Station Name	
100 MILE HOUSE 6NE	14.6
108 MILE HOUSE ABEL LAKE	16.8
ABBOTSFORD A	18.3
ABEE AGDM	15.3
ACADIA VALLEY	17.8
...	...
YOHIN	17.7
YOHO NP OHARA LAKE	10.0
YOHO PARK	12.0
YORKTON	18.5
ZAMA LO	9.5

1323 rows × 1 columns

In [33]:
```
temp2010 = temperature[[2010]]
temp2010 = temp2010[temp2010[2010].notnull()]
temp2010
```

Out[33]:

Year	2010
Station Name	
100 MILE HOUSE 6NE	14.6
108 MILE HOUSE ABEL LAKE	16.8
ABBOTSFORD A	18.3
ABEE AGDM	15.3
ACADIA VALLEY	17.8
...	...
YOHIN	17.7
YOHO NP OHARA LAKE	10.0
YOHO PARK	12.0
YORKTON	18.5
ZAMA LO	9.5

1323 rows × 1 columns

In [34]:
```
temp2010.mean()
```

Out[34]:

```
Year
2010    17.429932
dtype: float64
```

Let's pivot again, this time with the `Year` as index

Let's pivot again, this time with the `Year` as index

In [35]:
```
weather_p2 = weather.pivot(index='Year',
                           columns='Station Name',
                           values=['Mean Temperature (C)',
                                   'Total Precipitation (mm)'])

weather_p2
```

Out[35]:

Station Name	(AE) BOW SUMMIT	100 MILE HOUSE	100 MILE HOUSE 6NE	108 MILE HOUSE	108 MILE HOUSE ABEL LAKE	150 MILE HOUSE 7N	70 MILE HOUSE	ABBEY	AB
Year									
1971	NaN	16.0	NaN	NaN	NaN	15.5	NaN	17.6	
1972	NaN	15.1	NaN	NaN	NaN	14.1	NaN	NaN	
1973	NaN	15.1	NaN	15.9	NaN	14.1	NaN	NaN	
1974	NaN	13.7	NaN	NaN	NaN	13.0	11.9	NaN	
1975	NaN	17.2	NaN	NaN	NaN	NaN	15.2	NaN	
1976	NaN	14.5	NaN	NaN	NaN	NaN	12.8	NaN	
1977	NaN	13.5	NaN	NaN	NaN	NaN	13.0	18.3	
1978	NaN	15.5	NaN	NaN	NaN	NaN	15.6	18.3	
1979	NaN	16.0	NaN	NaN	NaN	NaN	15.0	19.3	
1980	NaN	14.1	NaN	NaN	NaN	NaN	13.1	18.4	
1981	NaN	15.1	NaN	NaN	NaN	NaN	14.1	18.6	
1982	NaN	14.8	NaN	NaN	NaN	NaN	13.1	17.7	
1983	NaN	14.4	NaN	NaN	NaN	NaN	12.9	18.6	
1984	NaN	14.7	NaN	NaN	NaN	NaN	13.3	20.2	

1985	NaN	16.6	NaN	NaN	NaN	NaN	15.3	19.6
1986	NaN	13.5	NaN	NaN	NaN	NaN	NaN	17.3
1987	NaN	15.4	15.1	NaN	15.4	NaN	NaN	18.6
1988	NaN	14.9	14.7	NaN	15.1	NaN	NaN	19.9
1989	NaN	15.0	14.8	NaN	14.6	NaN	NaN	21.0
1990	NaN	16.3	16.0	NaN	15.4	NaN	NaN	18.3
1991	NaN	14.9	14.5	NaN	14.3	NaN	NaN	19.0
1992	NaN	15.7	15.3	NaN	15.3	NaN	NaN	16.5
1993	NaN	13.5	13.3	NaN	12.9	NaN	NaN	15.8
1994	NaN	16.7	16.6	NaN	16.6	NaN	NaN	19.5
1995	NaN	15.9	15.5	NaN	15.3	NaN	NaN	18.6
1996	NaN	15.3	15.4	NaN	14.3	NaN	NaN	19.2
1997	NaN	14.9	14.5	NaN	14.1	NaN	NaN	19.3
1998	12.1	18.1	17.4	NaN	17.3	NaN	NaN	21.1
1999	NaN	14.9	14.1	NaN	14.0	NaN	NaN	17.2
2000	9.1	NaN	14.6	NaN	15.2	NaN	NaN	20.3
2001	9.6	NaN	14.3	NaN	14.3	NaN	NaN	20.5
2002	10.0	NaN	15.2	NaN	16.3	NaN	NaN	20.9
2003	11.1	NaN	16.2	NaN	16.3	NaN	NaN	20.7
2004	NaN	NaN	NaN	NaN	NaN	NaN	NaN	18.5
2005	NaN	NaN	14.4	NaN	14.2	NaN	NaN	19.4

This allows us to get the mean of **all** stations!

This allows us to get the mean of **all** stations!

In [36]:
```
weather_p2['Mean Temperature (C)'].mean()
```

Out[36]:
```
Station Name
(AE) BOW SUMMIT              10.380000
100 MILE HOUSE               15.217241
100 MILE HOUSE 6NE           15.243333
108 MILE HOUSE               15.900000
108 MILE HOUSE ABEL LAKE     15.883333
                                ...
YOYO                         13.350000
ZAMA LO                      15.392500
ZEBALLOS MURAUDE CREEK       17.028571
ZEHNER                       18.083333
ZHODA                        18.825000
Length: 4808, dtype: float64
```

Pivoting with Aggregation

This next cell will give an error! **WHY**

This next cell will give an error! **WHY**

In [37]:
```
weather.pivot(index='Province',
              columns='Year',
              values=['Mean Temperature (C)',
                      'Total Precipitation (mm)'])  # Error!
```

```
---------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Cell In[37], line 1
----> 1 weather.pivot(index='Province',
      2                columns='Year',
      3                values=['Mean Temperature (C)',
      4                        'Total Precipitation (mm)'])

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\fram
e.py:9339, in DataFrame.pivot(self, columns, index, values)
   9332 @Substitution("")
   9333 @Appender(_shared_docs["pivot"])
   9334 def pivot(
   9335     self, *, columns, index=lib.no_default, values=lib.no_defa
ult
   9336 ) -> DataFrame:
   9337     from pandas.core.reshape.pivot import pivot
-> 9339     return pivot(self, index=index, columns=columns, values=va
lues)

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\resh
ape\pivot.py:570, in pivot(data, columns, index, values)
    566         indexed = data._constructor_sliced(data[values]._value
s, index=multiindex)
    567 # error: Argument 1 to "unstack" of "DataFrame" has incompatib
le type "Union
    568 # [List[Any], ExtensionArray, ndarray[Any, Any], Index, Serie
s]"; expected
    569 # "Hashable"
```

```
--> 570 result = indexed.unstack(columns_listlike)  # type: ignore[ar
g-type]
    571 result.index.names = [
    572     name if name is not lib.no_default else None for name in r
esult.index.names
    573 ]
    575 return result

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\fram
e.py:9928, in DataFrame.unstack(self, level, fill_value, sort)
   9864 """
   9865 Pivot a level of the (necessarily hierarchical) index labels.
   9866
   (...)
   9924 dtype: float64
   9925 """
   9926 from pandas.core.reshape.reshape import unstack
-> 9928 result = unstack(self, level, fill_value, sort)
   9930 return result.__finalize__(self, method="unstack")

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\resh
ape\reshape.py:504, in unstack(obj, level, fill_value, sort)
    502 if isinstance(obj, DataFrame):
    503     if isinstance(obj.index, MultiIndex):
--> 504         return _unstack_frame(obj, level, fill_value=fill_valu
e, sort=sort)
    505     else:
    506         return obj.T.stack(future_stack=True)

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\resh
ape\reshape.py:529, in _unstack_frame(obj, level, fill_value, sort)
```

```
    525 def _unstack_frame(
    526     obj: DataFrame, level, fill_value=None, sort: bool = True
    527 ) -> DataFrame:
    528     assert isinstance(obj.index, MultiIndex)  # checked by cal
ler
--> 529     unstacker = _Unstacker(
    530         obj.index, level=level, constructor=obj._constructor,
sort=sort
    531     )
    533     if not obj._can_fast_transpose:
    534         mgr = obj._mgr.unstack(unstacker, fill_value=fill_valu
e)

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\resh
ape\reshape.py:154, in _Unstacker.__init__(self, index, level, constru
ctor, sort)
    146 if num_cells > np.iinfo(np.int32).max:
    147     warnings.warn(
    148         f"The following operation may generate {num_cells} cel
ls "
    149         f"in the resulting pandas object.",
    150         PerformanceWarning,
    151         stacklevel=find_stack_level(),
    152     )
--> 154 self._make_selectors()

File ~\AppData\Roaming\Python\Python311\site-packages\pandas\core\resh
ape\reshape.py:210, in _Unstacker._make_selectors(self)
    207 mask.put(selector, True)
    209 if mask.sum() < len(self.index):
--> 210     raise ValueError("Index contains duplicate entries, cannot
reshape")
```

If we use `pivot_table` with `aggfunc`, we can tell Pandas what to do with these values for instance we may want the `mean`

If we use `pivot_table` with `aggfunc`, we can tell Pandas what to do with these values for instance we may want the `mean`

In [38]:
```
weather.pivot_table(index='Province',
                    columns='Year',
                    values=['Mean Temperature (C)', 'Total Precipitation (mm)'
                    aggfunc='mean')
```

Out[38]:

Year	1971	1972	1973	1974	1975	1976	1
Province							
AB	15.057655	13.122741	15.160550	14.393491	17.158160	15.045833	13.898
BC	17.103300	15.994702	15.956347	14.695879	17.289267	14.965147	15.166
MB	16.250000	16.147368	17.946281	20.249194	20.560504	18.756303	18.335
NB	18.055172	17.520968	19.972581	17.357143	19.917742	17.650820	18.026
NL	14.676596	14.482456	16.917241	13.165000	17.165517	14.661818	14.688
NS	18.052174	17.608451	19.455844	16.082895	19.432877	17.419737	17.687
NT	13.304545	11.672727	14.507143	13.164286	14.565789	13.507692	12.472
NU	6.747222	4.506452	6.985294	7.532432	6.761765	6.460000	6.934
ON	18.109699	18.820209	19.890492	19.444156	20.575974	18.717377	19.830
PE	18.700000	18.006667	20.486667	17.133333	20.940000	18.200000	18.257
QC	17.190986	17.606268	19.279911	17.842729	19.930769	17.253037	17.449
SK	16.155233	15.250568	17.767895	19.021081	20.272826	18.302260	17.582
YT	13.860000	12.914286	12.732258	11.845714	13.639474	12.808333	12.702

13 rows × 94 columns

You can also set different aggregation functions for each variable:

You can also set different aggregation functions for each variable:

In [39]:
```
weather.pivot_table(index='Province',
                    columns='Year',
                    values=['Mean Temperature (C)', 'Total Precipitation (mm)'
                    aggfunc=['mean','sum'])
```

Out[39]:

Year	1971	1972	1973	1974	1975	1976	1
Province							
AB	15.057655	13.122741	15.160550	14.393491	17.158160	15.045833	13.898
BC	17.103300	15.994702	15.956347	14.695879	17.289267	14.965147	15.166
MB	16.250000	16.147368	17.946281	20.249194	20.560504	18.756303	18.335
NB	18.055172	17.520968	19.972581	17.357143	19.917742	17.650820	18.026
NL	14.676596	14.482456	16.917241	13.165000	17.165517	14.661818	14.688
NS	18.052174	17.608451	19.455844	16.082895	19.432877	17.419737	17.687
NT	13.304545	11.672727	14.507143	13.164286	14.565789	13.507692	12.472
NU	6.747222	4.506452	6.985294	7.532432	6.761765	6.460000	6.934
ON	18.109699	18.820209	19.890492	19.444156	20.575974	18.717377	19.830
PE	18.700000	18.006667	20.486667	17.133333	20.940000	18.200000	18.257
QC	17.190986	17.606268	19.279911	17.842729	19.930769	17.253037	17.449
SK	16.155233	15.250568	17.767895	19.021081	20.272826	18.302260	17.582
YT	13.860000	12.914286	12.732258	11.845714	13.639474	12.808333	12.702

13 rows × 188 columns

Now, harness the power of AI (in Google Colab) to get better explanations of the errors obtained before!

Lab