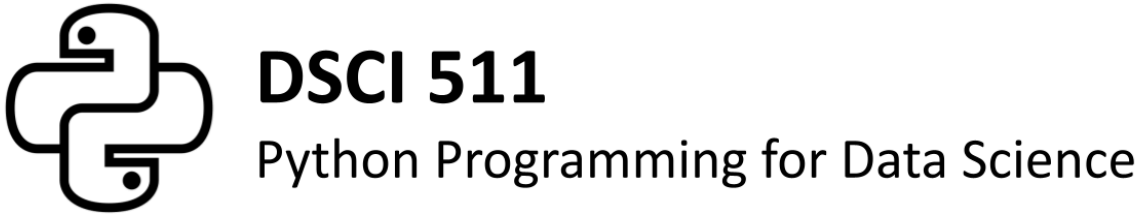
Lecture 2: Reshaping data

Print to PDF

Contents



- Filter columns in a DataFrame using logical operators or the `.query()` method
- Remove or fill missing values in a dataframe with `.dropna()` and `.fillna()`.
- Rename columns of a dataframe using the `.rename()` function or by accessing the `.columns` attribute.
- Use `.melt()`, `.pivot()` and `.pivot_table()` to reshape dataframes, specifically to make tidy dataframes.

```
import pandas as pd
```

- In lecture 1 you learned about the following types: DataFrame (table), Series (column), integer (whole number), and string (text)
- A new type for this lecture is 'boolean'. There are only two boolean values: True or False (note the first letter is capitalized in Python code).
- Booleans are useful when you want to search/filter through a DataFrame, i.e. find all the data where some particular condition is True. Load the file `data/YVR_weather_data.csv` into pandas to experiment with some filters.

Each row is a day, and each column is a weather measurement. Let's find all the days when the mean temperature is greater than 15. We'll start by creating a Series of boolean values like this:

```
weather = pd.read_csv('data/YVR_weather_data.csv')
weather.head()
```

	Date/Time	Year	Month	Mean Max Temp (°C)	Mean Max Temp Flag	Mean Min Temp (°C)	Mean Min Temp Flag	Mean Temp (°C)	Mean Temp Flag	Extr Max Temp (°C)	...	Total Snow (cm)	Total Snow Flag	Total Precip (mm)
0	Jan-37	1937	1	0.6	NaN	-8.1	NaN	-3.8	NaN	6.1	...	NaN	M	NaN
1	Feb-37	1937	2	5.2	NaN	-1.3	NaN	2.0	NaN	10.0	...	NaN	M	NaN
2	Mar-37	1937	3	11.7	NaN	2.9	NaN	7.3	NaN	17.2	...	0.0	NaN	59.7
3	Apr-37	1937	4	11.9	NaN	4.8	NaN	8.4	NaN	16.1	...	0.0	NaN	114.0
4	May-37	1937	5	16.3	NaN	6.6	NaN	11.5	NaN	20.6	...	0.0	NaN	44.2

5 rows × 25 columns

```
condition = weather['Mean Temp (°C)'] > 15
condition
```

```
0      False
1      False
2      False
3      False
4      False
       ...
912    False
913    False
914    False
915    False
916     True
Name: Mean Temp (°C), Length: 917, dtype: bool
```

The Series has True if that row has a value of greater than 15 in the mean temperature column, and False otherwise. Take a moment to compare the output of the code with the actual CSV file, to convince yourself this is what's happening.

You can then 'apply' this to your DataFrame using the `[ ]` notation you learned in Lecture 1.

```
weather[condition]
```

	Date/Time	Year	Month	Mean Max Temp (°C)	Mean Max Temp Flag	Mean Min Temp (°C)	Mean Min Temp Flag	Mean Temp (°C)	Mean Temp Flag	Extr Max Temp (°C)	...	Total Snow (cm)	Total Snow Flag	Tota Precip (mm)
5	Jun-37	1937	6	20.1	NaN	10.4	NaN	15.3	NaN	26.1	...	0.0	NaN	100.6
6	Jul-37	1937	7	22.5	NaN	11.6	NaN	17.1	NaN	27.8	...	0.0	NaN	1.8
7	Aug-37	1937	8	20.6	NaN	10.7	NaN	15.7	NaN	24.4	...	0.0	NaN	58.4
17	Jun-38	1938	6	20.8	NaN	9.8	NaN	15.3	NaN	25.6	...	0.0	NaN	14.7
18	Jul-38	1938	7	22.8	NaN	12.1	NaN	17.5	NaN	28.9	...	0.0	NaN	10.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	..
895	Sep-11	2011	9	20.4	NaN	12.3	NaN	16.4	NaN	25.2	...	0.0	NaN	69.6
905	Jul-12	2012	7	21.7	NaN	13.7	NaN	17.7	NaN	26.0	...	0.0	NaN	27.8
906	Aug-12	2012	8	23.5	NaN	14.4	NaN	19.0	NaN	28.6	...	0.0	NaN	2.9
907	Sep-12	2012	9	19.6	NaN	11.1	NaN	15.4	NaN	25.3	...	0.0	NaN	5.0
916	Jun-13	2013	6	19.3	I	11.5	I	15.4	I	22.0	...	0.0	I	2.8

216 rows × 25 columns

The following comparisons are supported:

- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

- `==` exactly equal to
- `!=` not equal to

```
condition = weather['Mean Temp (°C)'] == 19.0
exactly_19_degrees = weather[condition]
exactly_19_degrees[['Mean Temp (°C)']]
```

	Mean Temp (°C)
295	19.0
643	19.0
727	19.0
823	19.0
906	19.0

```
condition = weather['Month'] != 1
not_january = weather[condition]
not_january[['Month']]
```

	Month
1	2
2	3
3	4
4	5
5	6
...	...
912	2
913	3
914	4
915	5
916	6

840 rows × 1 columns

In practice, most people do not write out the condition on its own line. Code is usually combined into one line like this:

```
not_january = weather[weather['Month'] != 1]
```

Python ignores extra blank spaces inside brackets, and some people prefer to spread their code over multiple lines for increased readability.

```
not_january = weather[
                  weather['Month'] != 1
                  ]
```

You can combine multiple conditions using the symbols `&`, `|`, and `^`. Note that each condition must be enclosed in parentheses.

```
#Use & to match BOTH conditions
#find every day in May when it snowed
condition = (weather['Month'] == 5) & (weather['Total Snow (cm)'] > 0)
weather[condition]
```

	Date/Time	Year	Month	Mean Max Temp (°C)	Mean Max Temp Flag	Mean Min Temp (°C)	Mean Min Temp Flag	Mean Temp (°C)	Mean Temp Flag	Extr Max Temp (°C)	...	Total Snow (cm)	Total Snow Flag	Total Precip (mm)	Pı

0 rows × 25 columns

```
#Use | to match AT LEAST ONE condition
#find every day when it rained or snowed or both
condition = (weather['Total Snow (cm)'] > 0) | (weather['Total Rain (mm)'] > 0)
weather[condition]
```

	Date/Time	Year	Month	Mean Max Temp (°C)	Mean Max Temp Flag	Mean Min Temp (°C)	Mean Min Temp Flag	Mean Temp (°C)	Mean Temp Flag	Extr Max Temp (°C)	...	Total Snow (cm)	Total Snow Flag	Total Precip (mm)
0	Jan-37	1937	1	0.6	NaN	-8.1	NaN	-3.8	NaN	6.1	...	NaN	M	NaN
1	Feb-37	1937	2	5.2	NaN	-1.3	NaN	2.0	NaN	10.0	...	NaN	M	NaN
2	Mar-37	1937	3	11.7	NaN	2.9	NaN	7.3	NaN	17.2	...	0.0	NaN	59.7
3	Apr-37	1937	4	11.9	NaN	4.8	NaN	8.4	NaN	16.1	...	0.0	NaN	114.0
4	May-37	1937	5	16.3	NaN	6.6	NaN	11.5	NaN	20.6	...	0.0	NaN	44.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
912	Feb-13	2013	2	7.8	NaN	3.0	NaN	5.4	NaN	10.4	...	0.0	NaN	74.4
913	Mar-13	2013	3	10.5	NaN	3.9	NaN	7.2	NaN	15.7	...	0.0	NaN	108.0
914	Apr-13	2013	4	12.8	NaN	6.2	NaN	9.5	NaN	17.2	...	0.0	T	115.8
915	May-13	2013	5	17.1	NaN	9.5	NaN	13.3	NaN	22.2	...	0.0	NaN	66.0
916	Jun-13	2013	6	19.3	I	11.5	I	15.4	I	22.0	...	0.0	I	2.8

912 rows × 25 columns

```
#Use ^ to match EXACTLY ONE condition
#find every day when it was either very windy or very cold, but not both
condition = (weather['Spd of Max Gust (km/h)'] > 75) ^ (weather['Mean Temp (°C)'] <= -10)
weather[condition]
```

	Date/Time	Year	Month	Mean Max Temp (°C)	Mean Max Temp Flag	Mean Min Temp (°C)	Mean Min Temp Flag	Mean Temp (°C)	Mean Temp Flag	Extr Max Temp (°C)	...	Total Snow (cm)	Total Snow Flag	Total Precip (mm)
241	Feb-57	1957	2	5.9	NaN	-1.5	NaN	2.2	NaN	11.1	...	23.6	NaN	84.3
243	Apr-57	1957	4	13.8	NaN	5.9	NaN	9.9	NaN	21.7	...	0.0	NaN	44.7
249	Oct-57	1957	10	13.7	NaN	6.1	NaN	9.9	NaN	20.6	...	0.0	T	81.0
250	Nov-57	1957	11	9.4	NaN	2.5	NaN	6.0	NaN	14.4	...	0.0	NaN	76.2
251	Dec-57	1957	12	8.5	NaN	2.8	NaN	5.7	NaN	13.3	...	0.0	T	181.4
...	...	...	...	...	...	...	...	...	...	...	...	...	...	..
899	Jan-12	2012	1	6.3	NaN	0.8	NaN	3.6	NaN	11.3	...	5.6	NaN	135.6
900	Feb-12	2012	2	7.7	NaN	2.0	NaN	4.8	NaN	12.6	...	1.8	NaN	133.6
901	Mar-12	2012	3	8.7	NaN	2.3	NaN	5.6	NaN	14.3	...	0.0	T	111.6
913	Mar-13	2013	3	10.5	NaN	3.9	NaN	7.2	NaN	15.7	...	0.0	NaN	108.0
914	Apr-13	2013	4	12.8	NaN	6.2	NaN	9.5	NaN	17.2	...	0.0	T	115.8

124 rows × 25 columns

As an alternative, you can use the `.query()` function on your DataFrame. In this case, you write the condition out as a string, enclosed in double-quotes.

```
#Find days with cool temperatures and with trace amounts of snowfall in either January or February
condition = "`Total Snow Flag` == 'T' &  `Mean Temp (°C)` <= 5 & Month == 1 | Month == 2"
snowy_days = weather.query(condition)
snowy_days.tail(5)
```

	Date/Time	Year	Month	Mean Max Temp (°C)	Mean Max Temp Flag	Mean Min Temp (°C)	Mean Min Temp Flag	Mean Temp (°C)	Mean Temp Flag	Extr Max Temp (°C)	...	Total Snow (cm)	Total Snow Flag	Tota Precip (mm)
877	Feb-10	2010	2	10.3	NaN	3.9	NaN	7.1	NaN	13.1	...	0.0	NaN	102.2
888	Feb-11	2011	2	6.3	NaN	0.5	NaN	3.4	NaN	13.4	...	7.0	NaN	89.4
900	Feb-12	2012	2	7.7	NaN	2.0	NaN	4.8	NaN	12.6	...	1.8	NaN	133.6
911	Jan-13	2013	1	5.4	NaN	0.2	NaN	2.8	NaN	10.0	...	0.0	T	100.4
912	Feb-13	2013	2	7.8	NaN	3.0	NaN	5.4	NaN	10.4	...	0.0	NaN	74.4

5 rows × 25 columns

Notes on using `.query()`

- You don't have to mention your DataFrame specifically inside the condition, you only need the column names
- If a column name has spaces in it (like Total Snow Flag), you must enclose it in backticks (*not* quotes!)
- If you are trying to match against a string (like 'T') you must enclose it in single-quotes
- Parentheses are not required around each condition, though you can choose to add them if you need to group certain conditions together.

Practice

- Filter for every day in 1937.
- Filter for months from August to December (inclusive)
- Find days when it snowed less than 10cm

- Find all the days when it was at least 20 degrees and it didn't rain
- Filter for days when the mean temperature is between 5 and 15 degrees (exclusive)
- Find days when it was either windy (max gust of at least 75) or rainy (more than 50mm) but not both
- Filter for anything that's not in 2012

```
#PRACTICE CELL
```

You can combine boolean conditionals with `.loc[]` from Lecture 1 to update your DataFrame. Let's open up the language database from the previous lecture and make a change to it. Some languages have an unknown genetic grouping ('family'), which is indicated by a '?' in the data. Let's find all instances of '?' and replace it with the string 'Isolate', which is the more common technical term for such languages.

```
languages = pd.read_csv('data/WACL.csv')
languages.tail(5) #Check the 'family' column of the Zuni language (row 3336)
```

	iso_code	language_name	longitude	latitude	area	continent	status	family	source
3333	zom	Zou	93.9253	24.0649	Eurasia	Asia	threatened	Sino-Tibetan	NaN
3334	gnd	Zulgo-Gemzek	14.0578	10.827	Africa	Africa	not endangered	Afro-Asiatic	NaN
3335	zul	Zulu	31.3512	-25.3305	Africa	Africa	not endangered	Atlantic-Congo	NaN
3336	zun	Zuni	-108.782	35.0056	North America	Americas	shifting	?	NaN
3337	zzj	Zuojiang Zhuang	107.3622	21.83753	Eurasia	Asia	not endangered	Tai-Kadai	NaN

```

```

```
column = 'family'
condition = languages[column] == '?'
languages.loc[condition, column] = 'Isolate'
languages.tail(5) #Now check the 'family' column for Zuni
```

	iso_code	language_name	longitude	latitude	area	continent	status	family	source
3333	zom	Zou	93.9253	24.0649	Eurasia	Asia	threatened	Sino-Tibetan	NaN
3334	gnd	Zulgo-Gemzek	14.0578	10.827	Africa	Africa	not endangered	Afro-Asiatic	NaN
3335	zul	Zulu	31.3512	-25.3305	Africa	Africa	not endangered	Atlantic-Congo	NaN
3336	zun	Zuni	-108.782	35.0056	North America	Americas	shifting	Isolate	NaN
3337	zzj	Zuojiang Zhuang	107.3622	21.83753	Eurasia	Asia	not endangered	Tai-Kadai	NaN

This can be condensed to a single line:

```
languages.loc[languages['family'] == '?', 'family'] = 'Isolate'
```

Sometimes a cell in a spreadsheet will be missing values. In the language data, some rows have an entry for source material, but others do not. Pandas represents empty cells with the special value `NaN`, meaning “not a number”. You can check if your dataset has missing values like this:

```
#Check the entire DataFrame for Nan
languages.isna()
```

	iso_code	language_name	longitude	latitude	area	continent	status	family	source
0	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...
3333	False	False	False	False	False	False	False	False	True
3334	False	False	False	False	False	False	False	False	True
3335	False	False	False	False	False	False	False	False	True
3336	False	False	False	False	False	False	False	False	True
3337	False	False	False	False	False	False	False	False	True

3338 rows × 9 columns

```
#Check specific columns
languages['source'].isna()
```

```
0       False
1       False
2       False
3       False
4       False
        ...
3333     True
3334     True
3335     True
3336     True
3337     True
Name: source, Length: 3338, dtype: bool
```

```
#Use Python's any() function to find out if there is at least one NaN value in the DataFrame
any(languages.isna())
```

```
True
```

```
#Or check just a specific column
any(languages['family'].isna())
```

```
False
```

If you want to get rid of any rows with `NaN` use `.dropna()`

```
languages.dropna() #This returns a copy, it doesn't modify your original!
```

	iso_code	language_name	longitude	latitude	area	continent	status	family	
0	aiw	Aari	36.5721	5.95034	Africa	Africa	not endangered	South Omotic	
1	kbt	Abadi	146.992	-9.03389	Papunesia	Pacific	not endangered	Austronesian	
2	mij	Mungbam	10.2267	6.5805	Africa	Africa	shifting	Atlantic-Congo	
3	aau	Abau	141.324	-3.97222	Papunesia	Pacific	shifting	Sepik	
4	abq	Abaza	42.7273	41.1214	Eurasia	Europe	threatened	Abkhaz-Adyge	ket
...	...	...	...	...	...	...	...	...	
3287	zyn	Yongnan Zhuang	107.3622	21.83753	Eurasia	Asia	not endangered	Tai-Kadai	
3296	buh	Younuo Bunu	110.474	25.7638	Eurasia	Asia	shifting	Hmong-Mien	
3302	yuk	Yuki	-123.296	39.8389	North America	Americas	extinct	Yuki-Wappo	
3318	kxk	Lahta-Zayein Karen	96.908	20.16	Eurasia	Asia	not endangered	Sino-Tibetan	
3328	zil	Zialo	-9.51	8.43	Africa	?	threatened	Mande	

1754 rows × 9 columns

You can replace all occurences of `NaN` with a different value using `.fillna()`

```
languages.fillna('No source') #This returns a copy, it doesn't modify the original!
```

	iso_code	language_name	longitude	latitude	area	continent	status	family	
0	aiw	Aari	36.5721	5.95034	Africa	Africa	not endangered	South Omotic	
1	kbt	Abadi	146.992	-9.03389	Papunesia	Pacific	not endangered	Austronesian	
2	mij	Mungbam	10.2267	6.5805	Africa	Africa	shifting	Atlantic-Congo	
3	aau	Abau	141.324	-3.97222	Papunesia	Pacific	shifting	Sepik	
4	abq	Abaza	42.7273	41.1214	Eurasia	Europe	threatened	Abkhaz-Adyge	ket
...	...	...	...	...	...	...	...	...	
3333	zom	Zou	93.9253	24.0649	Eurasia	Asia	threatened	Sino-Tibetan	
3334	gnd	Zulgo-Gemzek	14.0578	10.827	Africa	Africa	not endangered	Afro-Asiatic	
3335	zul	Zulu	31.3512	-25.3305	Africa	Africa	not endangered	Atlantic-Congo	
3336	zun	Zuni	-108.782	35.0056	North America	Americas	shifting	Isolate	
3337	zzj	Zuojiang Zhuang	107.3622	21.83753	Eurasia	Asia	not endangered	Tai-Kadai	

3338 rows × 9 columns

Renaming columns

To rename all the columns at once, you can directly set the `columns` attribute of your DataFrame. You'll need to use a data type called a *list* which contains all of your column names between square brackets.

```
scores = pd.read_csv('data/student_scores.csv', index_col='Student_ID')
scores.columns.to_list() #check the original columns
```

```
['Biology', 'Chemistry', 'Physics', 'English', 'Drama', 'Art']
```

```
#Create a list of new names
scores.columns = ['BIOL-101', 'CHEM-200', 'PHYS-132', 'ENGL-100', 'DRAM-301', 'ART-216'] #this is a lis
scores
```

Student_ID	BIOL-101	CHEM-200	PHYS-132	ENGL-100	DRAM-301	ART-216
23583	97	71	67	62	72	54
69204	88	74	97	53	67	97
74763	74	64	68	57	70	64
79080	81	25	81	94	91	88
61824	71	69	83	97	98	75
49915	58	54	54	78	87	74
16055	63	72	58	67	91	92
47192	70	54	70	94	94	52
48641	70	66	96	91	53	64
65083	90	27	93	76	72	88
56645	99	37	66	76	86	71
62012	92	59	86	63	96	81
92865	67	29	50	74	75	78
13367	99	44	79	90	61	51
69346	58	42	73	60	76	98
41169	56	69	75	55	85	68
68390	79	62	94	96	55	79
98353	68	38	70	84	68	51
58887	82	54	92	53	81	56

	BIOL-101	CHEM-200	PHYS-132	ENGL-100	DRAM-301	ART-216
Student_ID						
69618	94	32	88	81	96	62
51213	87	34	50	69	75	52
79018	96	64	55	59	99	54
46410	56	30	77	99	92	55
90278	74	73	82	98	89	82
34549	93	65	61	79	59	92
52230	58	70	59	55	99	54
13918	98	51	77	76	75	81
14379	77	56	55	87	57	77
54578	94	45	51	88	58	70
79496	80	66	51	69	66	53

You can choose to rename only some columns with the `rename()` function. To illustrate this, let's rename some columns from the languges dataset.

```
languages = pd.read_csv('data/WACL.csv')
languages.columns.to_list() #remind yourself what the current headers are
```

```
['iso_code',
 'language_name',
 'longitude',
 'latitude',
 'area',
 'continent',
 'status',
 'family',
 'source']
```

For this function, you need to create a *dictionary* which is another basic type of object in Python. This type is used when you want to map, associate, or pair together two peices of information. Dictionaries are surrounded by curly braces, and each pair in the dictionary is separated by a colon. For example:

```
old_to_new = {"language_name": "name",
              "area": "region",
              "source": "bibliography"}
```

The first item in each pair is called a 'key', and second item is called a 'value'. Here the keys are `language_name`, `area`, and `source`, and the corresponding values are `name`, `region`, and `bibliography`.

Once you have the dictionary, you pass it to the `columns` argument of the `.rename()` function on your DataFrame.

```
languages = pd.read_csv('data/WACL.csv')
languages = languages.rename(columns = old_to_new) #this returns a copy!
languages.head()
```

	iso_code	name	longitude	latitude	region	continent	status	family	
0	aiw	Aari	36.5721	5.95034	Africa	Africa	not endangered	South Omotic	dani
1	kbt	Abadi	146.992	-9.03389	Papunesia	Pacific	not endangered	Austronesian	
2	mij	Mungbam	10.2267	6.5805	Africa	Africa	shifting	Atlantic-Congo	d
3	aau	Abau	141.324	-3.97222	Papunesia	Pacific	shifting	Sepik	
4	abq	Abaza	42.7273	41.1214	Eurasia	Europe	threatened	Abkhaz-Adyge	ketevan_lomta

We will discuss how to use dictionaries in more detail in a future lecture. For now, you'll just need to memorize this syntax for renaming columns.

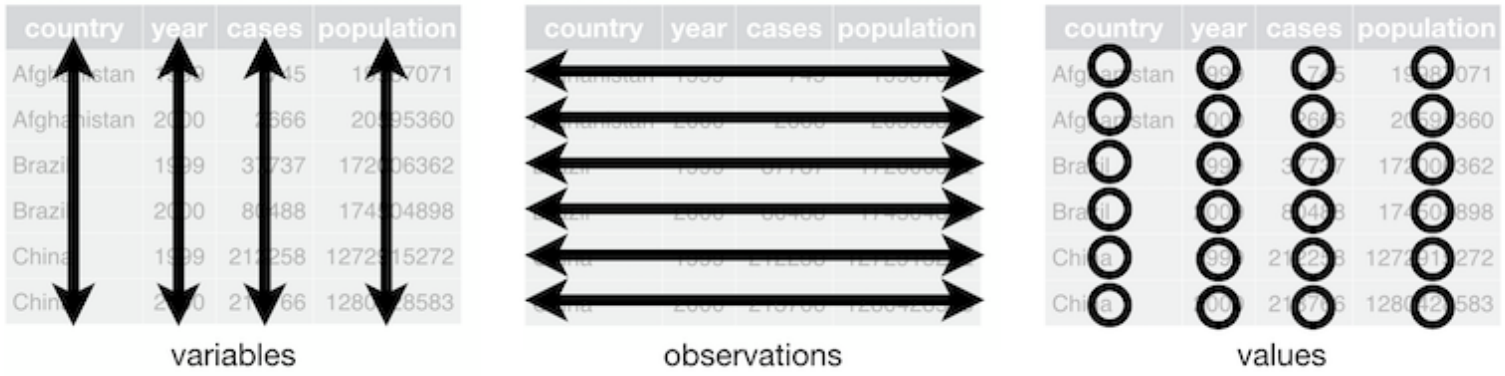
Data type cheatsheet

Type name	Data format	Example
string	text	`words = 'this is a string'`
integer	whole number	`fifteen = 15`
float	decimal number	`two_and_a_half = 2.5`
list	sequence	`groceries = ['apples', 'pears', 'oranges']`
dictionary	mapping	`english_to_french = {'house': 'maison', 'chair': 'chaise'}`
boolean	true/false	`not_false = True`
DataFrame	table	`results = pd.read_csv('experiment.csv')`
Series	column	`scores = results['scores']`

Tidy data

- Tidy data is about “linking the structure of a dataset with its semantics (its meaning)”
- It is defined by:
    1. Each variable forms a column
    2. Each observation forms a row
    3. Each type of observational unit forms a table
- Often you’ll need to reshape a dataframe to make it tidy (or for some other purpose)

- We'll look at four different ways to do this: `.transpose()`, `.pivot()`, `.pivot_table()`, and `.melt()`



Source: r4ds

Wide vs. long tables

Wide Format

Team	Points	Assists	Rebounds
A	88	12	22
B	91	17	28
C	99	24	30
D	94	28	31

Long Format

Team	Variable	Value
A	Points	88
A	Assists	12
A	Rebounds	22
B	Points	91
B	Assists	17
B	Rebounds	28
C	Points	99
C	Assists	24
C	Rebounds	30
D	Points	94
D	Assists	28
D	Rebounds	31

Image source

Transpose

Transposing a table means swapping the rows and columns. DataFrames have a `.transpose()` function for this. Run the next two cells and compare the output.

```
scores = pd.read_csv('data/student_scores.csv')
scores.head(5)
```

	Student_ID	Biology	Chemistry	Physics	English	Drama	Art
0	23583	97	71	67	62	72	54
1	69204	88	74	97	53	67	97
2	74763	74	64	68	57	70	64
3	79080	81	25	81	94	91	88
4	61824	71	69	83	97	98	75

```
scores_T = scores.transpose() #This returns a copy!
scores_T.head(5)
```

	0	1	2	3	4	5	6	7	8	9	...	20	21
Student_ID	23583	69204	74763	79080	61824	49915	16055	47192	48641	65083	...	51213	79018
Biology	97	88	74	81	71	58	63	70	70	90	...	87	96
Chemistry	71	74	64	25	69	54	72	54	66	27	...	34	64
Physics	67	97	68	81	83	54	58	70	96	93	...	50	55
English	62	53	57	94	97	78	67	94	91	76	...	69	59

5 rows × 30 columns

Pivot

Pivoting a table means changing it from long to wide format. To illustrate, we will explore a dataset of the results from the 2024 Olympics.

```
olympics = pd.read_csv('data/Olympics_2024.csv')
olympics
```

	Competitions	Rank	NOC	Gold	Silver	Bronze	Total
0	Archery	1	South Korea	5	1	1	7
1	Archery	2	France*	0	1	1	2
2	Archery	3	United States	0	1	1	2
3	Archery	4	China	0	1	0	1
4	Archery	5	Germany	0	1	0	1
...	...	...	...	...	...	...	...
449	Wrestling	22	Denmark	0	0	1	1
450	Wrestling	23	Greece	0	0	1	1
451	Wrestling	24	India	0	0	1	1
452	Wrestling	25	Norway	0	0	1	1
453	Wrestling	26	Puerto Rico	0	0	1	1

454 rows × 7 columns

Let's reshape the data so each row is a country ("NOC"), each column is a sport ("Competitions"), and the cell values show the total medal count for that country in that sport.

To do this, we need to use `.pivot()`, which take three arguments:

- `index` is the name of an existing column that should serve as your new row index (NOC)
- `columns` is the name of an existing column whose values will be converted to new columns headers (Competitions)

- `values` is the name of an existing column whose values will be used to fill the new columns (Total)

The table has a lot of `NaN` values (because not all countries win medals in all sports). We'll replace those with zeroes for readability.

```
olympics.pivot(index='NOC', columns='Competitions', values='Total').fillna(0).head()
```

Competitions	Archery	Artistic swimming	Athletics	Badminton	Basketball	Boxing	Breaking	Canoeing	Cyclinç
NOC									
Albania	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.(
Algeria	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.(
Argentina	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.(
Armenia	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.(
Australia	0.0	0.0	7.0	0.0	1.0	2.0	0.0	5.0	8.(

5 rows × 35 columns

Pandas has to do some calculations when reshaping the table, which results in decimal numbers ("floats"). This is a little strange, since you can't win a partial medal. We can ask Pandas to change the type of the entire DataFrame with the `.astype()` function.

```
olympics.pivot(index='NOC', columns='Competitions', values='Total').fillna(0).astype(int)
```

Competitions	Archery	Artistic swimming	Athletics	Badminton	Basketball	Boxing	Breaking	Canoeing	Cyclinç
NOC									
Albania	0	0	0	0	0	0	0	0	(
Algeria	0	0	1	0	0	1	0	0	(
Argentina	0	0	0	0	0	0	0	0	'
Armenia	0	0	0	0	0	0	0	0	(
Australia	0	0	7	0	1	2	0	5	8
...	...	...	...	...	...	...	...	...	..
Uganda	0	0	2	0	0	0	0	0	(
Ukraine	0	0	3	0	0	1	0	1	(
United States	2	1	34	0	3	1	1	2	6
Uzbekistan	0	0	0	0	0	5	0	0	(
Zambia	0	0	1	0	0	0	0	0	(

93 rows × 35 columns

```
#This format makes it easy to extract information about individual countries
o = olympics.pivot(index='NOC', columns='Competitions', values='Total').fillna(0).astype(int)
o.loc['Canada']
```

```
Competitions
Archery               0
Artistic swimming     0
Athletics             5
Badminton             0
Basketball            0
Boxing                1
Breaking              1
Canoeing              2
Cycling               0
Diving                1
Equestrian            0
Fencing               1
Field hockey          0
Football              0
Golf                  0
Gymnastics            1
Handball              0
Judo                  1
Modern pentathlon     0
Rowing                1
Rugby sevens          1
Sailing               0
Shooting              0
Skateboarding         0
Sport climbing        0
Surfing               0
Swimming              8
Table tennis          0
Taekwondo             1
Tennis                1
Triathlon             0
Volleyball            1
Water polo            0
Weightlifting         1
Wrestling             0
Name: Canada, dtype: int64
```

Now let's pivot the other direction, so that each row is a sport, and each column is a country

```
olympics.pivot(index='Competitions', columns='NOC', values='Total').fillna(0).astype(int).head()
```

NOC	Albania	Algeria	Argentina	Armenia	Australia	Austria	Azerbaijan	Bahrain	Belgium	Bot
Competitions										
Archery	0	0	0	0	0	0	0	0	0	
Artistic swimming	0	0	0	0	0	0	0	0	0	
Athletics	0	1	0	0	7	0	0	2	3	
Badminton	0	0	0	0	0	0	0	0	0	
Basketball	0	0	0	0	1	0	0	0	0	

5 rows × 93 columns

```
#This format makes it easy to extract data about individual sports
o = olympics.pivot(index='Competitions', columns='NOC', values='Total').fillna(0).astype(int)
o.loc['Fencing']
```

```
NOC
Albania          0
Algeria          0
Argentina        0
Armenia          0
Australia        0
                ..
Uganda           0
Ukraine          2
United States    4
Uzbekistan       0
Zambia           0
Name: Fencing, Length: 93, dtype: int64
```

Practice

Take a few moment to play with `.pivot()` and change the parameters around

```
#PRACTICE CELL
```

`.pivot_table()`

You can only pivot a table if it has unique combinations of row and column values. If you have duplicates, then you'll need to use the `pivot_table()` function instead. To illustrate this, load the accent study dataset:

```
results = pd.read_csv('data/accentStudyDataset.csv')
results.head()
```

	ID	FirstLanguage	Test	Accent	Item	ItemScore	TestScore	TestPct
0	P01	Turkish	Baikal	Chinese	Baikal1	1	5	62.5
1	P01	Turkish	Baikal	Chinese	Baikal2	1	5	62.5
2	P01	Turkish	Baikal	Chinese	Baikal3	0	5	62.5
3	P01	Turkish	Baikal	Chinese	Baikal4	0	5	62.5
4	P01	Turkish	Baikal	Chinese	Baikal5	1	5	62.5

These are partial results from an experiment studying the effects of accent on listening comprehension. Participants with different langauge backgrounds listened to passages recorded in English, spoken by people with different accents (British, Chinese, or Spanish). Participants then completed a short test, answering questions about the passage they had listened to.

It would be useful to reshape this so that rows are organized by first language, columns by accent condition, with cell values showing the quiz scores. That would help us see interactions, e.g. how did first-language Turkish speakers do when listening to British accents?

However, if we try to do this with `.pivot()`, we get an error.

```
results.pivot(index='FirstLanguage', columns='Accent', values='TestPct')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[38], line 1
----> 1 results.pivot(index='FirstLanguage', columns='Accent', values='TestPct')

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/frame.py:9339, in DataFrame.pivot(self, colum
   9332 @Substitution("")
   9333 @Appender(_shared_docs["pivot"])
   9334 def pivot(
   9335     self, *, columns, index=lib.no_default, values=lib.no_default
   9336 ) -> DataFrame:
   9337     from pandas.core.reshape.pivot import pivot
-> 9339     return pivot(self, index=index, columns=columns, values=values)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/reshape/pivot.py:570, in pivot(data, columns,
    566         indexed = data._constructor_sliced(data[values]._values, index=multiindex)
    567 # error: Argument 1 to "unstack" of "DataFrame" has incompatible type "Union
    568 # [List[Any], ExtensionArray, ndarray[Any, Any], Index, Series]"; expected
    569 # "Hashable"
--> 570 result = indexed.unstack(columns_listlike)  # type: ignore[arg-type]
    571 result.index.names = [
    572     name if name is not lib.no_default else None for name in result.index.names
    573 ]
    575 return result

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/series.py:4615, in Series.unstack(self, level
   4570 """
   4571 Unstack, also known as pivot, Series with MultiIndex to produce DataFrame.
   4572
   (...)
   4611 b    2    4
   4612 """
   4613 from pandas.core.reshape.reshape import unstack
-> 4615 return unstack(self, level, fill_value, sort)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/reshape/reshape.py:517, in unstack(obj, level
    515 if is_1d_only_ea_dtype(obj.dtype):
    516     return _unstack_extension_series(obj, level, fill_value, sort=sort)
--> 517 unstacker = _Unstacker(
    518     obj.index, level=level, constructor=obj._constructor_expanddim, sort=sort
    519 )
```

```
    520 return unstacker.get_result(
    521     obj._values, value_columns=None, fill_value=fill_value
    522 )

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/reshape/reshape.py:154, in _Unstacker.__init_
    146 if num_cells > np.iinfo(np.int32).max:
    147     warnings.warn(
    148         f"The following operation may generate {num_cells} cells "
    149         f"in the resulting pandas object.",
    150         PerformanceWarning,
    151         stacklevel=find_stack_level(),
    152     )
--> 154 self._make_selectors()

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/reshape/reshape.py:210, in _Unstacker._make_s
    207 mask.put(selector, True)
    209 if mask.sum() < len(self.index):
--> 210     raise ValueError("Index contains duplicate entries, cannot reshape")
    212 self.group_index = comp_index
    213 self.mask = mask

ValueError: Index contains duplicate entries, cannot reshape
```

The error message says there are duplicate entries, so the table can't be reshaped. This means that there are some row/column combinations that would appear twice in the new pivoted table. For example, there's more than one person with a first language of Turkish in the Chinese accent condition.

In this case, we need to use `pivot_table()`, which is similar to `.pivot()` but it can handle the duplicates.

```
results.pivot_table(index='FirstLanguage', columns='Accent', values='TestPct') #index, columns, and valu
```

Accent	British	Chinese	Spanish
FirstLanguage			
Arabic	37.500000	62.500000	50.000000
Bengali	59.375000	46.875000	62.500000
Burmese	50.000000	50.000000	25.000000
Burmese	37.500000	50.000000	50.000000
Cantonese	46.428571	53.571429	55.357143
English	58.333333	70.833333	68.750000
Estonian	62.500000	37.500000	75.000000
French	81.250000	81.250000	75.000000
Hindi	75.000000	25.000000	62.500000
Italian	75.000000	75.000000	87.500000
Korean	50.000000	50.000000	50.000000
Luganda	62.500000	75.000000	50.000000
Malayalam	62.500000	25.000000	25.000000
Mandarin	50.961538	49.519231	42.788462
Montenegrin	50.000000	37.500000	50.000000
Pashto	68.750000	31.250000	50.000000
Portuguese	52.083333	35.416667	60.416667
Russian	62.500000	50.000000	62.500000
Serbian	75.000000	87.500000	37.500000

Accent	British	Chinese	Spanish
FirstLanguage			
Spanish	51.562500	51.562500	59.375000
Spanish	62.500000	37.500000	37.500000
Swahili	68.750000	75.000000	62.500000
Tamil	87.500000	87.500000	87.500000
Turkish	56.250000	75.000000	37.500000
Vietnamese	37.500000	50.000000	62.500000

To find the results for a particular first language, use `.loc[]`

```
r = results.pivot_table(index='FirstLanguage', columns='Accent', values='TestPct')
r.loc['Turkish']
```

```
Accent
British    56.25
Chinese    75.00
Spanish    37.50
Name: Turkish, dtype: float64
```

By default, `.pivot_table()` combines ("aggregates") all the duplicate values and calculates the average. The cell above shows that the average test scores for first-language Turkish speakers for passages read with British accent is 56.25% You can do other types of calculations as well, but we'll leave that for a later lesson on grouping and aggregation.

Each accent condition in this experiment contained 4 different test passages, and the code above aggregates over all of them. To see the finer details of each test within in each accent, simply provide both as column names:

```
results.pivot_table(index='FirstLanguage', columns=['Accent', 'Test'], values='TestPct').head()
```

Accent	British			Chinese			Spanish		
Test	Baikal	CompSci	balloon	Baikal	CompSci	balloon	Baikal	CompSci	balloon
FirstLanguage									
Arabic	NaN	NaN	37.5	62.5	NaN	NaN	NaN	50.00	NaN
Bengali	62.5	58.333333	NaN	NaN	NaN	46.875	58.333333	75.00	NaN
Burmese	50.0	NaN	NaN	NaN	50.00	NaN	NaN	NaN	25.000000
Burmese	NaN	NaN	37.5	NaN	50.00	NaN	50.000000	NaN	NaN
Cantonese	62.5	50.000000	37.5	50.0	56.25	NaN	68.750000	56.25	45.833333

Note that this creates a large number of `NaN` values because there are some combinations of language/accent/test that did not occur in the original data (e.g. no Burmese speakers heard the CompSci passage with a Britsh accent)

Melt

`.melt()` works as the opposite of `.pivot()`, and converts data from a wide format to a long format. To illustrate this, we'll return to the Internet Movie Database.

```
imdb = pd.read_csv('data/imdb.csv')
imdb.head(2) #remind yourself what this looks like
```

	Series_Title	Released_Year	Certificate	Runtime	Genre	IMDB_Rating	Meta_score	Director	Star1
0	The Shawshank Redemption	1994	A	142 min	Drama	9.3	80.0	Frank Darabont	Tim Robbins
1	The Godfather	1972	A	175 min	Crime, Drama	9.2	100.0	Francis Ford Coppola	Marlon Brando

Let's reshape the table so that rows are organized by director, and there are only two columns: 'Actor' and 'Billing_Order'.

To do this we'll use `.melt()`, which requires up to four arguments:

- `id_vars` is a list of column names that will act as the index for creating new rows (Director)
- `value_vars` is a list of columns from your original data that you want melted (Star1, Star2, Star3, Star4)
- `var_name` is the header for a new column, which will contain the column names listed in `value_vars` (Billing_Order)
- `value_name` is the header for a new column which will contain the row values from the columns listed in `value_vars` (Actor)

`value_vars` is optional. If you omit it, Pandas will assume you want to melt everything that's not listed in `id_vars`

```
imdb.melt(id_vars=['Director'], value_vars=['Star1', 'Star2', 'Star3', 'Star4'], var_name='Billing_Orde
```

	Director	Billing_Order	Actor
0	Frank Darabont	Star1	Tim Robbins
1	Francis Ford Coppola	Star1	Marlon Brando
2	Christopher Nolan	Star1	Christian Bale
3	Francis Ford Coppola	Star1	Al Pacino
4	Sidney Lumet	Star1	Henry Fonda
...	...	...	...
3995	Blake Edwards	Star4	Buddy Ebsen
3996	George Stevens	Star4	Carroll Baker
3997	Fred Zinnemann	Star4	Donna Reed
3998	Alfred Hitchcock	Star4	William Bendix
3999	Alfred Hitchcock	Star4	Godfrey Tearle

4000 rows × 3 columns

```
#We can use this new format to quickly look up which actors have appears with which directors
i = imdb.melt(id_vars=['Director'], value_vars=['Star1', 'Star2', 'Star3', 'Star4'], var_name='Billing_(
i = i[['Actor', 'Billing_Order']] #I prefer this order
i.loc['Francis Ford Coppola']
```

	Actor	Billing_Order
Director		
Francis Ford Coppola	Marlon Brando	Star1
Francis Ford Coppola	Al Pacino	Star1
Francis Ford Coppola	Martin Sheen	Star1
Francis Ford Coppola	Gene Hackman	Star1
Francis Ford Coppola	Al Pacino	Star1
Francis Ford Coppola	Al Pacino	Star2
Francis Ford Coppola	Robert De Niro	Star2
Francis Ford Coppola	Marlon Brando	Star2
Francis Ford Coppola	John Cazale	Star2
Francis Ford Coppola	Diane Keaton	Star2
Francis Ford Coppola	James Caan	Star3
Francis Ford Coppola	Robert Duvall	Star3
Francis Ford Coppola	Robert Duvall	Star3
Francis Ford Coppola	Allen Garfield	Star3
Francis Ford Coppola	Andy Garcia	Star3
Francis Ford Coppola	Diane Keaton	Star4
Francis Ford Coppola	Diane Keaton	Star4
Francis Ford Coppola	Frederic Forrest	Star4
Francis Ford Coppola	Frederic Forrest	Star4

	Actor	Billing_Order
Director		
Francis Ford Coppola	Talia Shire	Star4

Reseting the index

Reshaping a table can change the way rows are indexed. Sometimes, you may want to reset the index to use the default ordinal numbers. For example, let's load a dataset about the Titanic, and use `.pivot_table()` to compare survival rates for men and women places in different passenger classes.

```
titanic = pd.read_csv('data/titanic.csv')
titanic.head() #get quick look
```

	PassengerId	Survived	Class	Name	Gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Eml
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	

```
survival_rates = titanic.pivot_table(index='Gender', values='Survived', columns='Class')
survival_rates
```

Class	1	2	3
Gender			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

This sets the `Gender` column as the row index, so it is no longer part of the table data. To restore it as a column, use the `.reset_index()` function:

```
survival_rates.reset_index() #This returns a copy!
```

Class	Gender	1	2	3
0	female	0.968085	0.921053	0.500000
1	male	0.368852	0.157407	0.135447