pandas

Lukas Hager

2024-04-08

Overview

pandas

- `pandas` is an open-source library that makes working with tabular data easy
  - `numpy`: good at arrays/matrix operations (e.g. all float values)
  - `pandas`: good at heterogeneous data (e.g. some float, some character, etc.)
  - Primary pythonic tool for cleaning data for analysis

Learning Objectives

- Understand how to load, slice, filter, join, reshape, and clean data using `pandas`
- Feel comfortable with the basics of regular expressions

pandas Data Structures

Importing pandas[2]

We'll always use the `pd` prefix for `pandas`. We'll also import `numpy` with its standard prefix[1]

```
import pandas as pd
import numpy as np
```

[1] Remember, this keeps us from confusing ourselves about which functions we're using when there's overlap.

[2] As always, much of the material here is based on Wes McKinney's excellent book's chapter on `pandas`

Series

pandas has a similar object to np.array – Series:

```
obj = pd.Series([4, 7, -5, 3])
obj
```

	0
0	4
1	7
2	-5
3	3

Series

pandas has a similar object to `np.array` – `Series`:

```
obj = pd.Series([4, 7, -5, 3])
obj
```

	0
0	4
1	7
2	-5
3	3

Note that the object has an index – these are like `dict` keys in base python:

```
obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])
obj2
```

	0
d	4

Indexing Series

The index allows us more descriptive access to values:

```
obj2['a']
```

```
-5
```

Indexing Series

The index allows us more descriptive access to values:

```
obj2['a']
```

-5

```
obj2[['a','c']]
```

	0
a	-5
c	3

Indexing Series

The index allows us more descriptive access to values:

```
obj2['a']
```

-5

```
obj2[['a','c']]
```

	0
a	-5
c	3

```
obj2[3]
```

3

Indexing Series

The index allows us more descriptive access to values:

```
obj2['a']
```

-5

```
obj2[['a','c']]
```

	0
a	-5
c	3

```
obj2[3]
```

3

```
obj2['d'] = 6
obj2[['a', 'd']]
```

Boolean Indexing

We can still use boolean indexing without removing the index link:

```
obj2[obj2>2]
```

	0
d	6
b	7
c	3

Boolean Indexing

We can still use boolean indexing without removing the index link:

```
obj2[obj2>2]
```

	0
d	6
b	7
c	3

Note that the index is still there, and both array and index are accessible with `Series` methods:

```
obj2[obj2>2].array
```

```
<PandasArray>
[6, 7, 3]
Length: 3, dtype: int64
```

Boolean Indexing

We can still use boolean indexing without removing the index link:

```
obj2[obj2>2]
```

	0
d	6
b	7
c	3

Note that the index is still there, and both array and index are accessible with `Series` methods:

```
obj2[obj2>2].array
```

```
<PandasArray>
[6, 7, 3]
Length: 3, dtype: int64
```

```
obj2[obj2>2].index
```

Creating `Series` from `dict`

Since we're arguing that `Series` is very similar to a `dict`, it's logical that we can create one from a `dict`:

```
sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utal
obj3 = pd.Series(sdata)
obj3
```

	0
Ohio	35000
Texas	71000
Oregon	16000
Utah	5000

Creating Series from dict

Since we're arguing that Series is very similar to a dict, it's logical that we can create one from a dict:

```
sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utal
obj3 = pd.Series(sdata)
obj3
```

	0
Ohio	35000
Texas	71000
Oregon	16000
Utah	5000

And we can also do the reverse operation:

```
obj3.to_dict()
```

```
{'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

Providing Index to Series

We can pass an index directly in the creation of a Series object:

```
states = ["California", "Ohio", "Oregon", "Texas"]
obj4 = pd.Series(sdata, index=states)
obj4
```

	0
California	NaN
Ohio	35000.0
Oregon	16000.0
Texas	71000.0

Providing Index to Series

We can pass an index directly in the creation of a `Series` object:

```
states = ["California", "Ohio", "Oregon", "Texas"]
obj4 = pd.Series(sdata, index=states)
obj4
```

	0
California	NaN
Ohio	35000.0
Oregon	16000.0
Texas	71000.0

Why do we have a `NaN` (not a number)? Where is Utah's entry?

NaN

Use `pd.isna` function to identify these values:

```
pd.isna(obj4)
```

	0
California	True
Ohio	False
Oregon	False
Texas	False

NaN

Use `pd.isna` function to identify these values:

```
pd.isna(obj4)
```

	0
California	True
Ohio	False
Oregon	False
Texas	False

We often want to look at arrays that have null values removed:

```
obj4[~pd.isna(obj4)]
```

	0
Ohio	35000.0
Oregon	16000.0
Texas	71000.0

NaN

Use `pd.isna` function to identify these values:

```
pd.isna(obj4)
```

	0
California	True
Ohio	False
Oregon	False
Texas	False

We often want to look at arrays that have null values removed:

```
obj4[~pd.isna(obj4)]
```

	0
Ohio	35000.0
Oregon	16000.0
Texas	71000.0

NaN

Use `pd.isna` function to identify these values:

```
pd.isna(obj4)
```

	0
California	True
Ohio	False
Oregon	False
Texas	False

We often want to look at arrays that have null values removed:

```
obj4[~pd.isna(obj4)]
```

	0
Ohio	35000.0
Oregon	16000.0
Texas	71000.0

DataFrame

The power of pandas is in the DataFrame object. There are many ways to construct a DataFrame but a common one is passing a dict:

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada",
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
frame
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

head and tail

```
frame.head(2)
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7

head and tail

```
frame.head(2)
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7

```
frame.tail(2)
```

	state	year	pop
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Exercise: `numpy` Array as `DataFrame`

Look up `pd.DataFrame`'s documentation and create a 1000×26 `DataFrame` where each column has a letter of the alphabet as its header, and its rows are 1000 Poisson random variables.

Solutions: numpy Array as DataFrame

```
rng = np.random.default_rng(seed=481)
random_array = rng.poisson(size = (1000,26))
headers = list('abcdefghijklmnopqrstuvwxyz')
pois = pd.DataFrame(data = random_array, columns = headers)
pois.head(3)
```

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
0	2	1	0	0	1	0	0	1	1	0	0	2	2	2	3	2	1	0	4
1	1	0	1	2	0	0	1	1	0	0	2	1	3	2	0	2	4	2	3
2	1	0	2	0	1	2	0	3	0	0	0	2	1	1	1	0	0	2	3

Selecting a Column

```
frame.state
```

	state
0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada
5	Nevada

Selecting a Column

```
frame.state
```

	state
0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada
5	Nevada

```
frame['state']
```

	state
0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada

Selecting a Column

```
frame.state
```

	state
0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada
5	Nevada

```
frame['state']
```

	state
0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada

Selecting Columns

```
frame[['state', 'year']]
```

	state	year
0	Ohio	2000
1	Ohio	2001
2	Ohio	2002
3	Nevada	2001
4	Nevada	2002
5	Nevada	2003

Selecting Columns

```
frame[['state', 'year']]
```

	state	year
0	Ohio	2000
1	Ohio	2001
2	Ohio	2002
3	Nevada	2001
4	Nevada	2002
5	Nevada	2003

Note that this is a `DataFrame`, while the previous objects were `Series`

Selecting Column as `DataFrame`

```
frame[['state']]
```

	state
0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada
5	Nevada

Creating Variables

```
frame['debt'] = 1.5
frame['debt2'] = np.arange(len(frame))
frame
```

	state	year	pop	debt	debt2
0	Ohio	2000	1.5	1.5	0
1	Ohio	2001	1.7	1.5	1
2	Ohio	2002	3.6	1.5	2
3	Nevada	2001	2.4	1.5	3
4	Nevada	2002	2.9	1.5	4
5	Nevada	2003	3.2	1.5	5

Reindexing

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a",
obj
```

	0
d	4.5
b	7.2
a	-5.3
c	3.6

Reindexing

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a",
obj
```

	0
d	4.5
b	7.2
a	-5.3
c	3.6

Suppose we want to set a new index on this `DataFrame`. What happens?

```
obj.reindex(["a", "b", "c", "d", "e"])
```

	0
a	-5.3
b	7.2
c	3.6
d	4.5
e	NaN

Reindexing

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a",
obj
```

	0
d	4.5
b	7.2
a	-5.3
c	3.6

Suppose we want to set a new index on this `DataFrame`. What happens?

```
obj.reindex(["a", "b", "c", "d", "e"])
```

	0
a	-5.3
b	7.2
c	3.6
d	4.5
e	NaN

`reindex` for Economic Data

- One useful application of reindexing for Economics is when we work with time-series data
- We may want to interpolate incomplete time-series data (fill it forward)

```
og_df = pd.DataFrame(
    {
        'price': np.random.uniform(size=3),
        'quantity': np.random.uniform(size=3)
    },
    index=[1,3,5]
)
og_df
```

	price	quantity
1	0.179027	0.817546
3	0.746673	0.004064
5	0.335627	0.100192

Default `reindex` Behavior

What should we do if we want one observation per day?

Default `reindex` Behavior

What should we do if we want one observation per day?

```
og_df.reindex(np.arange(6))
```

	price	quantity
0	NaN	NaN
1	0.179027	0.817546
2	NaN	NaN
3	0.746673	0.004064
4	NaN	NaN
5	0.335627	0.100192

`ffill`

How should we interpolate day 2's data?

ffill

How should we interpolate day 2's data?

```
filled_df = og_df.reindex(np.arange(6), method='ffill')
filled_df
```

	price	quantity
0	NaN	NaN
1	0.179027	0.817546
2	0.179027	0.817546
3	0.746673	0.004064
4	0.746673	0.004064
5	0.335627	0.100192

pandas Indexing (numpy Notation)

We can filter data via the same notation as in numpy:

```
filled_df[~pd.isna(filled_df['price'])]
```

	price	quantity
1	0.179027	0.817546
2	0.179027	0.817546
3	0.746673	0.004064
4	0.746673	0.004064
5	0.335627	0.100192

pandas Indexing (.loc)

The preferred method for subsetting data in pandas is using the `.loc` method[3]:

```
filled_df.loc[~pd.isna(filled_df['price'])]
```

	price	quantity
1	0.179027	0.817546
2	0.179027	0.817546
3	0.746673	0.004064
4	0.746673	0.004064
5	0.335627	0.100192

[3] If you're curious about why, check out this example.

Indexing using Index Values

Note that we previously were using Boolean indexing – we can of course also index using the index itself

```
filled_df.loc[[1,3,5]]
```

	price	quantity
1	0.179027	0.817546
3	0.746673	0.004064
5	0.335627	0.100192

Select on Both Axes with `.loc`

```
filled_df.loc[~pd.isna(filled_df['price']), 'quantity']
```

	quantity
1	0.817546
2	0.817546
3	0.004064
4	0.004064
5	0.100192

Value Assignment with `.loc`

```
filled_df.loc[pd.isna(filled_df['price'])] = 0.
filled_df
```

	price	quantity
0	0.000000	0.000000
1	0.179027	0.817546
2	0.179027	0.817546
3	0.746673	0.004064
4	0.746673	0.004064
5	0.335627	0.100192

ufuncs with pandas

```
frame = pd.DataFrame(
    np.random.standard_normal((4, 3)),
    columns=list("bde"),
    index=["Utah", "Ohio", "Texas", "Oregon"]
)
frame
```

	b	d	e
Utah	-0.790449	-0.031260	-0.403217
Ohio	-0.952796	1.228350	-1.315987
Texas	0.929138	0.635936	-0.585285
Oregon	1.120469	0.086756	0.757068

np.abs

```
np.abs(frame)
```

	b	d	e
Utah	0.790449	0.031260	0.403217
Ohio	0.952796	1.228350	1.315987
Texas	0.929138	0.635936	0.585285
Oregon	1.120469	0.086756	0.757068

apply

```
def f1(x: pd.Series) -> float:
    """
    Return the difference between the largest and smallest
    value of a `Series`.
    """
    return x.max() - x.min()

frame.apply(f1)
```

	0
b	2.073265
d	1.259610
e	2.073055

apply along rows

```
frame.apply(f1, axis = 'columns')
```

	0
Utah	0.759189
Ohio	2.544337
Texas	1.514424
Oregon	1.033714

Note on `apply`

- Apply will inherently be *inefficient* relative to a vectorized function
- For example – don't use `apply` to take the logarithm of a column in your `DataFrame`. Use `np.log`
  - See the `.map` method for how you could do this if you wanted to

Exercise: Distributions

Use and your 1000×26 `DataFrame` and return the sample mean and variance for each column. Use `apply` and `numpy` for one and only `numpy` for the other. Is there a time difference?

Solutions: Distributions

```
%%time
mean_pois = pois.apply(np.mean, axis = 0)
print(mean_pois.shape)
```

```
(26,)
CPU times: user 1.5 ms, sys: 136 µs, total: 1.63 ms
Wall time: 1.55 ms
```

Solutions: Distributions

```
%%time
mean_pois = pois.apply(np.mean, axis = 0)
print(mean_pois.shape)
```

```
(26,)
CPU times: user 1.5 ms, sys: 136 µs, total: 1.63 ms
Wall time: 1.55 ms
```

```
%%time
var_pois = np.var(pois, axis = 0)
print(var_pois.shape)
```

```
(26,)
CPU times: user 334 µs, sys: 91 µs, total: 425 µs
Wall time: 555 µs
```

Sorting

- We often want to sort data by some column or set of columns
  - Useful in time-series analysis
  - Also useful when we need to compute lagged variables
- We can do this a few ways in `pandas`

Sorting by Index

```
obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])
obj
```

	0
d	0
a	1
b	2
c	3

Sorting by Index

```
obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])
obj
```

	0
d	0
a	1
b	2
c	3

```
obj.sort_index()
```

	0
a	1
b	2
c	3
d	0

Sorting by Index

```
obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])
obj
```

	0
d	0
a	1
b	2
c	3

```
obj.sort_index()
```

	0
a	1
b	2
c	3
d	0

Note that this is sorting *lexicographically*

Sorting by Variables

We also want to sort data by specific variables, that are not the index

```
obj.sort_values()
```

	0
d	0
a	1
b	2
c	3

Sorting by Variables

We also want to sort data by specific variables, that are not the index

```
obj.sort_values()
```

	0
d	0
a	1
b	2
c	3

`NaN` values are sorted to the end by default

```
obj = pd.Series([3., 2., np.nan, 5., np.nan], index=["d", "a"
obj.sort_values()
```

	0
a	2.0
d	3.0
c	5.0

Summary Statistics

As we've seen, we can call aggregating functions on `DataFrame` objects to get some summary stats

```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
                   [np.nan, np.nan], [0.75, -1.3]],
                  index=["a", "b", "c", "d"],
                  columns=["one", "two"])

df
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Summary Stats

```
df.sum()
```

	0
one	9.25
two	-5.80

Summary Stats

```
df.sum()
```

	0
one	9.25
two	-5.80

```
df.mean()
```

	0
one	3.083333
two	-2.900000

Summary Stats

```
df.sum()
```

	0
one	9.25
two	-5.80

```
df.mean()
```

	0
one	3.083333
two	-2.900000

What are these methods doing with `NaN` values?

Summary Stats

```
df.sum()
```

	0
one	9.25
two	-5.80

```
df.mean()
```

	0
one	3.083333
two	-2.900000

What are these methods doing with `NaN` values?

A fuller list of summary statistics methods for `pandas` is available here.

`DataFrame.sum` versus `np.sum`

Note that `np.sum` will have different default behavior depending on the object that it's called on:

```
np.sum(random_array)
```

25866

DataFrame.sum versus np.sum

Note that `np.sum` will have different default behavior depending on the object that it's called on:

```
np.sum(random_array)
```

25866

```
np.sum(pois).head()
```

	0
a	1001
b	950
c	1029
d	1003
e	1019

describe

We can also use the built-in `describe` method:

```
df.describe()
```

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

`describe` with Non-numeric Data

```
obj = pd.Series(["a", "a", "b", "c"] * 4)
obj.describe()
```

	0
count	16
unique	3
top	a
freq	8

unique

We often want to know the unique (or distinct) values in a `Series` (equivalently, column in our `DataFrame`)

```
obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
obj.unique()
```

```
array(['c', 'a', 'd', 'b'], dtype=object)
```

unique

We often want to know the unique (or distinct) values in a `Series` (equivalently, column in our `DataFrame`)

```
obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
obj.unique()
```

```
array(['c', 'a', 'd', 'b'], dtype=object)
```

We also will want to know how often distinct values arise in a `Series`

```
obj.value_counts()
```

	0
c	3
a	3
b	2
d	1

isin

We also can check which elements of a `Series` object match certain values:

```
obj[obj.isin(["b", "c"])]
```

	0
0	c
5	b
6	b
7	c
8	c

Loading Data

Loading Data

- One of the nicest features of `pandas` is that it has methods for reading an assortment of stored data into a `DataFrame`
- We'll focus on some of the most common ones here, but all the loading methods are relatively similar

pd.read_csv

Reads data from a comma-separated value file (CSV). The data looks like this

```
from io import StringIO # you will not need StringIO if readi
raw_csv_file = StringIO("""
a, b, c
1, 2, europe
3, 5, asia
-2, 6, north america
""")
pd.read_csv(raw_csv_file)
```

	a	b	c
0	1	2	europe
1	3	5	asia
2	-2	6	north america

Useful `pd.read_csv` Arguments

- `sep`: what the delineator in the file is (“,” and “|” are common)
- `na_values`: list of strings to coerce to `NaN` (e.g. “NULL”, “N/A”, “NA”)
- `names`: pass your own column names instead of inferring from the data
- `skiprows`: tell `pandas` to not read in certain rows of the file
- `index_col`: tell `pandas` which column to use as the index

pd.read_excel

Reads in data from a specific Excel *sheet* (remember you can have multiple sheets in a workbook)

```
url = 'https://www.blm.gov/sites/blm.gov/files/Table2_Number_
pd.read_excel(url, usecols = ['FY 2014', 'FY 2015']).head()
```

	FY 2014	FY 2015
0	80540.0	78570.00
1	1813246.0	1813246.00
2	39560.0	19995.65
3	465407.0	460904.00
4	223206.0	208056.00

Useful `pd.read_excel` Arguments

- `sheet_name`: if a workbook has multiple sheets, access the one that you want
- `names`: pass your own column names instead of inferring from the data
- `thousands/decimal`: tell `pandas` what the separator for thousands or decimals is if you want to parse a text field to numeric
- `index_col`: tell `pandas` which column to use as the index

pd.read_html

- If a table on a webpage is sufficiently simple[4], we can use pandas to read it directly
- Note that this method returns a list of DataFrame objects
- Often need to specify the match argument to get the table you want

```
econ_courses = pd.read_html('https://econ.washington.edu/cours
econ_courses[0].head(3)
```

	Course	Course Title (click for details)	SLN	Instructor
0	ECON 200 A	Introduction to Microeconomics	13588	Melissa Knox
1	ECON 200 AA	Introduction to Microeconomics	13589	Roy Sonn
2	ECON 200 AB	Introduction to Microeconomics	13590	Erik Anderse

[4] More on this when we discuss web scraping, but you really want the table to be a <table> object in the HTML

Useful `pd.read_html` Arguments

- All the previous ones from `pd.read_csv` and `pd.read_excel`
- `attrs`: a dictionary of attributes to identify a table in the webpage's HTML
  - We'll discuss this a little more in webscraping, but you should be aware of it if you're using this method on a page where there are a lot of tables

Data Cleaning in pandas

Duplicate Data

- Depending on context, we may want to remove duplicate observations from a dataset
- Recall that we have the `.unique()` method to get the unique values from a `Series`
- If we want to filter out duplicates, we have some direct options for `DataFrames`

Dealing with Duplicate Data

Creating Data

```
data = pd.DataFrame({"k1": ["one", "two"] * 3 + ["two"],
                     "k2": [1, 1, 2, 3, 3, 4, 4]})
data
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

.duplicated()

```
data.duplicated()
```

Discretizing

If we want to bin data, it can be surprisingly tricky to code – luckily, pandas has a built-in method for us:

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
bins = [18, 25, 35, 60, 100]
age_categories = pd.cut(ages,bins)
age_categories
```

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60]
```

String Manipulation

- A **huge** benefit of the package
- Working with text data is crucial in Economic research
- Let's begin with an example

Class List

Let's return to our Econ department data

```
econ_courses[0]['Course']
```

	Course
0	ECON 200 A
1	ECON 200 AA
2	ECON 200 AB
3	ECON 200 AC
4	ECON 200 AD
5	ECON 200 AE
6	ECON 200 AF
7	ECON 200 AH
8	ECON 200 AI
9	ECON 200 AJ
10	ECON 200 B
11	ECON 201 A
12	ECON 201 AA
13	ECON 201 AB
14	ECON 201 AC

Class List

Let's return to our Econ department data

```
econ_courses[0]['Course']
```

	Course
0	ECON 200 A
1	ECON 200 AA
2	ECON 200 AB
3	ECON 200 AC
4	ECON 200 AD
5	ECON 200 AE
6	ECON 200 AF
7	ECON 200 AH
8	ECON 200 AI
9	ECON 200 AJ
10	ECON 200 B
11	ECON 201 A
12	ECON 201 AA
13	ECON 201 AB
14	ECON 201 AC

First option: `str.slice`

```
slice_df = econ_courses[0][['Course']]
slice_df['course_no'] = slice_df['Course'].str.slice(5,8)
slice_df.head(4)
```

	Course	course_no
0	ECON 200 A	200
1	ECON 200 AA	200
2	ECON 200 AB	200
3	ECON 200 AC	200

First option: `str.slice`

```
slice_df = econ_courses[0][['Course']]
slice_df['course_no'] = slice_df['Course'].str.slice(5,8)
slice_df.head(4)
```

	Course	course_no
0	ECON 200 A	200
1	ECON 200 AA	200
2	ECON 200 AB	200
3	ECON 200 AC	200

How can we check?

Check: `str.slice`

```
pd.value_counts(slice_df['course_no'])
```

	course_no
201	12
200	11
300	6
301	4
401	2
345	2
599	1
485	1
408	1
421	1
601	1
399	1
230	1
235	1
593	1
484	1

Check: `str.slice`

```
pd.value_counts(slice_df['course_no'])
```

	course_no
201	12
200	11
300	6
301	4
401	2
345	2
599	1
485	1
408	1
421	1
601	1
399	1
230	1
235	1
593	1
484	1

Classification: np.where

```
slice_df['classification'] = np.where(
    slice_df['course_no'].astype('int') >= 500,
    'grad',
    np.where(
        slice_df['course_no'].astype('int') >= 400,
        '400 level',
        np.where(
            slice_df['course_no'].astype('int') >= 300,
            '300 level',
            '200 level'
        )
    )
)
slice_df.iloc[[30,50]]
```

	Course	course_no	classification
30	ECON 300 D	300	300 level
50	ECON 454 A	454	400 level

Ideal Approach

- It would be cool if we could just tell `pandas` to grab the 3-digit substring within any larger string
- Then we could apply this to any course catalog (where the prefix isn't just "ECON")
- This is done using regular expressions

Regular Expressions (regex)

- Strings that are used to match patterns instead of explicit combinations of characters in text
- In our example, we want to match numbers and no characters
  - In regex, a general digit (0-9) can be represented using "\d"
  - In regex, we can specify the exact number of matches we want via "{3}"
- So a regex that would work in this application looks like this: `\d{3}` (match three digits)

Regular Expressions in pandas

These are *extremely* useful in working with string data in pandas

```
regex_df = econ_courses[0][['Course']]
regex_df['course_no'] = regex_df['Course'].str.extract('(\d{3
regex_df.head(4)
```

	Course	course_no
0	ECON 200 A	200
1	ECON 200 AA	200
2	ECON 200 AB	200
3	ECON 200 AC	200

Generally Useful Regular Expressions

- Match any one upper-case letter: `[A-Z]`
- Match any one lower-case letter: `[a-z]`
- Match any one letter: `[A-Za-z]`
- Match any one character: `.`
- Match multiple of a character in a row (greedy, zero or more): `*`
- Match multiple of a character in a row (non-greedy, one or more): `+`
- Match a space: `\s` or a literal space
- Match any text between “a” and “b”: `(?<=a).+(?=b)`
- Match “a” or “b”: `|`

Example: Match Phone Numbers

```
messy_phone_nos = pd.Series(['410-662-6967', '292-330-2492',
```

Example: Match Phone Numbers

```
messy_phone_nos = pd.Series(['410-662-6967', '292-330-2492',
```

```
messy_phone_nos.str.match('\d+-\d+-\d+')
```

	0
0	True
1	True
2	True

Example: Match Phone Numbers

```
messy_phone_nos = pd.Series(['410-662-6967', '292-330-2492',
```

```
messy_phone_nos.str.match('\d+-\d+-\d+')
```

	0
0	True
1	True
2	True

```
messy_phone_nos.str.match('\d{3}-\d{3}-\d{4}')
```

	0
0	True
1	True
2	False

Example: Match Washington Cities

```
wa_cities = pd.Series(
    ['Seattle, WA', 'Bellingham, Washington', 'Portland, Oreg
)
```

Example: Match Washington Cities

```
wa_cities = pd.Series(
    ['Seattle, WA', 'Bellingham, Washington', 'Portland, Oreg
)
```

```
wa_cities.str.match('[A-Za-z]+, (WA|Washington)')
```

	0
0	True
1	True
2	False

Example: Match Washington Cities

```
wa_cities = pd.Series(
    ['Seattle, WA', 'Bellingham, Washington', 'Portland, Oreg
)
```

```
wa_cities.str.match('[A-Za-z]+, (WA|Washington)')
```

	0
0	True
1	True
2	False

```
wa_cities.str.match('[A-Za-z]+, W[Aa]+?')
```

	0
0	True
1	True
2	False

Exercise: Matching

Write a regular expression that will **match** each element in its entirety in these arrays:

1. “285a”, “2a86”, “44b”, “abc”
2. “1 + 1 = 2”, “10 + 2 = 12”, “3+11=14”
3. Match any numpy method, but no pandas methods (assume `np` and `pd` prefixes)

A useful engine for practice: regex101

Hint: check out what ? does.

Matching Solutions

```
pd.Series(["285a", "2a86", "44b", "abc"]).str.extract('([a-z\
```

	0
0	285a
1	2a86
2	44b
3	abc

Matching Solutions

```
pd.Series(["285a", "2a86", "44b", "abc"]).str.extract('([a-z\
```

	0
0	285a
1	2a86
2	44b
3	abc

```
pd.Series(["1 + 1 = 2", "10 + 2 = 12", "3+11=14"])\
    .str.extract('(\d+\s?\+\s?\d+\s?\=\s?\d+)')[0]
```

	0
0	$1 + 1 = 2$
1	$10 + 2 = 12$
2	$3+11=14$

Matching Solutions

```
pd.Series(["285a", "2a86", "44b", "abc"]).str.extract('([a-z\
```

	0
0	285a
1	2a86
2	44b
3	abc

```
pd.Series(["1 + 1 = 2", "10 + 2 = 12", "3+11=14"])\
    .str.extract('(\d+\s?\+\s?\d+\s?\=\s?\d+)')[0]
```

	0
0	1 + 1 = 2
1	10 + 2 = 12
2	3+11=14

```
pd.Series(['np.inner', 'pd.read_html']).str.extract('(np\.[A-
```

Wrangling Data in pandas

Database-Style Joining

When we want to link data in two `DataFrames` by keys, we will `join` or `merge` them

```
df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "a", "b"]
                    "data1": pd.Series(range(7), dtype="Int64
df2 = pd.DataFrame({"key": ["a", "b", "d"],
                    "data2": pd.Series(range(3), dtype="Int64
df1,df2
```

```
(  key  data1
 0   b      0
 1   b      1
 2   a      2
 3   c      3
 4   a      4
 5   a      5
 6   b      6,
   key  data2
 0   a      0
 1   b      1
```

pd.merge

```
pd.merge(df1, df2)
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

pd.merge

```
pd.merge(df1, df2)
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

How does it know how to join? Do we have all of our original observations?

Specifying Merge Keys

A best practice is to explicitly state the columns you want to join on

```
pd.merge(df1, df2, on="key")
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

Different Names in Merge

```
df1.rename(columns={'key': 'key1'}, inplace=True)
df2.rename(columns={'key': 'key2'}, inplace=True)
pd.merge(df1, df2, left_on='key1', right_on='key2')
```

	key1	data1	key2	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

Outer Join

```
pd.merge(df1, df2, left_on='key1', right_on='key2', how='oute
```

	key1	data1	key2	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0
6	c	3	NaN	<NA>
7	NaN	<NA>	d	2

Left Join

```
pd.merge(df1, df2, left_on='key1', right_on='key2', how='left
```

	key1	data1	key2	data2
0	b	0	b	1
1	b	1	b	1
2	a	2	a	0
3	c	3	NaN	<NA>
4	a	4	a	0
5	a	5	a	0
6	b	6	b	1

Right Join

```
pd.merge(df1, df2, left_on='key1', right_on='key2', how='righ
```

	key1	data1	key2	data2
0	a	2	a	0
1	a	4	a	0
2	a	5	a	0
3	b	0	b	1
4	b	1	b	1
5	b	6	b	1
6	NaN	<NA>	d	2

Non-distinct Join Keys

```
df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
                    "data1": pd.Series(range(6), dtype="Int64
df2 = pd.DataFrame({"key": ["a", "b", "a", "b", "d"],
                    "data2": pd.Series(range(5), dtype="Int64
df1,df2
```

```
(  key  data1
 0   b      0
 1   b      1
 2   a      2
 3   c      3
 4   a      4
 5   b      5,
   key  data2
 0   a      0
 1   b      1
 2   a      2
 3   b      3
 4   d      4)
```

Cartesian Product

```
print(pd.merge(df1, df2, on='key', how='left'))
```

```
   key  data1  data2
0    b      0      1
1    b      0      3
2    b      1      1
3    b      1      3
4    a      2      0
5    a      2      2
6    c      3   <NA>
7    a      4      0
8    a      4      2
9    b      5      1
10   b      5      3
```

Inner Joins Don't Help

```
print(pd.merge(df1, df2, on='key', how='inner'))
```

```
  key  data1  data2
0   b      0      1
1   b      0      3
2   b      1      1
3   b      1      3
4   b      5      1
5   b      5      3
6   a      2      0
7   a      2      2
8   a      4      0
9   a      4      2
```

join

To merge on the index of a DataFrame, we can use join[5]

```
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
                     index=["a", "c", "e"],
                     columns=["Ohio", "Nevada"]).astype("Int64
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13,
                      index=["b", "c", "d", "e"],
                      columns=["Missouri", "Alabama"]).astype
left2,right2
```

```
(   Ohio  Nevada
 a     1       2
 c     3       4
 e     5       6,
    Missouri  Alabama
 b         7        8
 c         9       10
 d        11       12
 e        13       14)
```

[5] This can also be done with the left_index and right_index arguments in

Using join

```
print(left2.join(right2, how = 'inner'))
```

```
   Ohio  Nevada  Missouri  Alabama
c     3       4         9       10
e     5       6        13       14
```

Using join

```
print(left2.join(right2, how = 'inner'))
```

```
   Ohio  Nevada  Missouri  Alabama
c     3       4         9       10
e     5       6        13       14
```

```
print(left2.join(right2, how = 'outer'))
```

```
   Ohio  Nevada  Missouri  Alabama
a     1       2      <NA>     <NA>
b  <NA>    <NA>         7        8
c     3       4         9       10
d  <NA>    <NA>        11       12
e     5       6        13       14
```

Exercise: Joins/Merges

- Create a `DataFrame` that has every combination of the integers from 1-10 and their names as strings (i.e. “one”, “two”, “three”, etc) – you should have 100 combinations.
- Then create a new column which contains the computation of the integer exponentiated to the number of letters in the name for all 100 computations.
- Finally, filter to rows where that value is divisible by 27.

Solutions: Joins

```
df1 = pd.DataFrame({'ints': np.arange(1,11), 'key': 1}).set_i
char_list = ['one', 'two', 'three', 'four', 'five', 'six', 's
df2 = pd.DataFrame({'chars': char_list, 'key': 1}).set_index(
cj_data = df1.join(df2, how = 'left')
cj_data.shape
```

```
(100, 2)
```

Solutions: Joins

```
df1 = pd.DataFrame({'ints': np.arange(1,11), 'key': 1}).set_i
char_list = ['one', 'two', 'three', 'four', 'five', 'six', 's
df2 = pd.DataFrame({'chars': char_list, 'key': 1}).set_index(
cj_data = df1.join(df2, how = 'left')
cj_data.shape
```

```
(100, 2)
```

```
cj_data['n_chars'] = cj_data['chars'].str.len()
cj_data['exp'] = cj_data['ints'] ** cj_data['n_chars']
print(cj_data.loc[cj_data['exp'] % 27 == 0].shape)
print(cj_data.loc[cj_data['exp'] % 27 == 0].head(5))
```

```
(30, 4)
     ints  chars  n_chars  exp
key
1       3    one        3   27
1       3    two        3   27
1       3  three        5  243
```

Stacking Data

- A common occurrence is that data is stored in individual files, and we want to combine all the files into one dataset
  - E.g. a folder full of CSV files
- Remember DRY: we don't want to copy-paste 15 versions of `pd.read_csv(...)` and then combine them

pd.concat

```
s1 = pd.Series([0, 1], index=["a", "b"], dtype="Int64")
s2 = pd.Series([2, 3, 4], index=["c", "d", "e"], dtype="Int64
s3 = pd.Series([5, 6], index=["f", "g"], dtype="Int64")
pd.concat([s1, s2, s3])
```

	0
a	0
b	1
c	2
d	3
e	4
f	5
g	6

Note that we passed a list to `pd.concat`

Example: Economics Courses in 2023

- Suppose we wanted to create a `DataFrame` of all the Economics courses UW offered in 2023
- The URLs are in the format

```
https://econ.washington.edu/courses/<year>/<quarter>/all
```

- We already know how to download the tables with `pd.read_html`

Powerful Combination: List Comprehension and pd.concat

```
def download_econ_courses(year:int, quarter: str) -> pd.DataF
    """
    Download Economics Course Offerings for `year` and `quart
    """
    url = f'https://econ.washington.edu/courses/{year}/{quart
    df = pd.read_html(url, match = 'ECON Courses')[0]
    df['quarter'] = quarter
    return df

quarters = ['winter', 'spring', 'summer', 'autumn']

data_2023 = pd.concat([download_econ_courses(2023, x) for x i
print(data_2023.head(2))
```

```
      Course Course Title (click for details)    SLN    Instru
0   ECON 200 A   Introduction to Microeconomics  13928  Melissa
1  ECON 200 AA   Introduction to Microeconomics  13929   Ken Ino
```

Pivoting Data

- We'll often want to change the fundamental form of the data from "wide" to "long" or the opposite
- For example, instead of having "year" as a column, we may want each year to be its own column
- This is always kind of a pain
  - `pandas` makes it relatively painless with `pivot` and `melt`

Econ Course Data

```
print(data_2023.tail(5))
```

```
       Course Course Title (click for details)    SLN          I
69  ECON 594 A                  Economic Growth  14241  Stephen
70  ECON 600 A    Independent Study or Research  14242
71  ECON 601 A                       Internship  14243
72  ECON 602 A  Teaching Introductory Economics  14244         Ya
73  ECON 800 A            Doctoral Dissertation  14245

            Meeting Time quarter
69  MW 10:30am - 11:50am  autumn
70       to be arranged  autumn
71       to be arranged  autumn
72    W 9:30am - 11:20am  autumn
73       to be arranged  autumn
```

Econ Course Data

```
print(data_2023.tail(5))
```

```
        Course Course Title (click for details)    SLN          I
69  ECON 594 A                    Economic Growth  14241  Stephen
70  ECON 600 A     Independent Study or Research  14242
71  ECON 601 A                         Internship  14243
72  ECON 602 A  Teaching Introductory Economics  14244         Ya
73  ECON 800 A              Doctoral Dissertation  14245

              Meeting Time quarter
69  MW 10:30am - 11:50am  autumn
70         to be arranged  autumn
71         to be arranged  autumn
72     W 9:30am - 11:20am  autumn
73         to be arranged  autumn
```

What if we wanted to know courses taught by a professor over the year (400 level)?

Data to Pivot

```
profs = ['Melissa Knox', 'Yael Jacobs', 'Fahad Khalil']
data_subset = data_2023.loc[
    data_2023['Instructor'].isin(profs) &
    (data_2023['Course'].str.get(5) == '4')
]
print(data_subset)
```

```
       Course                Course Title (click for details)    SL
42  ECON 400 A                          Advanced Microeconomics  1397
56  ECON 485 A  Game Theory with Applications to Economics  1398
48  ECON 422 A                Investment, Capital, and Finance  1367
54  ECON 448 A                     Population and Development  1368
44  ECON 422 A                Investment, Capital, and Finance  2356

      Instructor              Meeting Time quarter
42  Melissa Knox   TTh 1:30pm - 3:20pm  winter
56  Fahad Khalil  MW 10:30am - 12:20pm  winter
48   Yael Jacobs  TTh 8:30am - 10:20am  spring
54  Melissa Knox  TTh 8:30am - 10:20am  spring
44   Yael Jacobs   TTh 5:30pm - 7:20pm  autumn
```

Pivoting

```
pivoted = data_subset.pivot(
    index='Instructor', # what should the index be?
    columns='quarter', # what column should create the columns
    values='Course' # what values should fill in the new colu
)
pivoted
```

quarter Instructor	autumn	spring	winter
Fahad Khalil	NaN	NaN	ECON 485 A
Melissa Knox	NaN	ECON 448 A	ECON 400 A
Yael Jacobs	ECON 422 A	ECON 422 A	NaN

Full Data Pivot

Will this work for the full dataset?

```
data_2023.pivot(
    index='Instructor',
    columns='quarter',
    values='Course'
)
```

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

Full Data Pivot

Will this work for the full dataset?

```
data_2023.pivot(
    index='Instructor',
    columns='quarter',
    values='Course'
)
```

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

How can we see this?

```
dup_entries = data_2023.loc[
    data_2023[['Instructor', 'quarter']].duplicated(keep=Fals
    ~pd.isna(data_2023['Instructor']),
    ['Instructor', 'quarter', 'Course']
]
```

Duplicates

```
print(dup_entries.sort_values(['Instructor', 'quarter']))
```

```
        Instructor quarter      Course
57   Alan Griffith  autumn  ECON 496 A
67   Alan Griffith  autumn  ECON 587 A
44   Alan Griffith  winter  ECON 410 A
68   Alan Griffith  winter  ECON 590 A
64   Brian Greaney  spring  ECON 509 A
..             ...     ...         ...
74      Yanqin Fan  spring  ECON 599 A
47    Yu-chin Chen  autumn  ECON 426 A
60    Yu-chin Chen  autumn  ECON 502 A
67  Yuya Takahashi  spring  ECON 534 A
73  Yuya Takahashi  spring  ECON 596 B

[71 rows x 3 columns]
```

Resetting the Index

To unpivot (melt), we need the key variable to not be the index:

```
print(pivoted.reset_index())
```

```
quarter    Instructor       autumn       spring       winter
0        Fahad Khalil          NaN          NaN  ECON 485 A
1        Melissa Knox          NaN  ECON 448 A  ECON 400 A
2         Yael Jacobs  ECON 422 A  ECON 422 A          NaN
```

Melting

Then we can melt:

```
melted = pd.melt(
    pivoted.reset_index(),
    id_vars = 'Instructor'
)
print(melted)
```

```
     Instructor quarter       value
0  Fahad Khalil  autumn         NaN
1  Melissa Knox  autumn         NaN
2   Yael Jacobs  autumn  ECON 422 A
3  Fahad Khalil  spring         NaN
4  Melissa Knox  spring  ECON 448 A
5   Yael Jacobs  spring  ECON 422 A
6  Fahad Khalil  winter  ECON 485 A
7  Melissa Knox  winter  ECON 400 A
8   Yael Jacobs  winter         NaN
```

Aggregation

How Aggregation Works in pandas

groupby

The first step is telling pandas how to split (which groups to use):

```
grouped_data = data_2023.groupby('Course')
grouped_data
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fbf60
```

groupby

The first step is telling pandas how to split (which groups to use):

```
grouped_data = data_2023.groupby('Course')
grouped_data
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fbf60
```

This is our dataset, but with instructions about how to group

Aggregating

We can just call some methods directly on the grouped data (this will be fastest)

```
print(grouped_data.count().head(10))
```

```
            Course Title (click for details)  SLN  Instructor
Course
ECON 200 A                                     4    4           4
ECON 200 AA                                    3    3           3
ECON 200 AB                                    3    3           3
ECON 200 AC                                    3    3           3
ECON 200 AD                                    3    3           3
ECON 200 AE                                    3    3           3
ECON 200 AF                                    3    3           3
ECON 200 AG                                    3    3           3
ECON 200 AH                                    3    3           3
ECON 200 AI                                    3    3           3

            quarter
Course
```

Aggregating On Specific Columns

```
print(grouped_data['Instructor'].count().head(5))
```

```
Course
ECON 200 A     4
ECON 200 AA    3
ECON 200 AB    3
ECON 200 AC    3
ECON 200 AD    3
Name: Instructor, dtype: int64
```

Aggregating On Specific Columns

```
print(grouped_data['Instructor'].count().head(5))
```

```
Course
ECON 200 A     4
ECON 200 AA    3
ECON 200 AB    3
ECON 200 AC    3
ECON 200 AD    3
Name: Instructor, dtype: int64
```

```
print(grouped_data[['Instructor']].count().head(5))
```

```
            Instructor
Course
ECON 200 A           4
ECON 200 AA          3
ECON 200 AB          3
ECON 200 AC          3
ECON 200 AD          3
```

Load Apple Data

```
import json
f = open('resources/aapl_json.json')
aapl_dict = json.load(f)
times = pd.to_datetime(aapl_dict['TimeInfo']['Ticks'], unit='
times.name = 'time'
aapl_hfts = pd.DataFrame(
    data = aapl_dict['Series'][0]['DataPoints'],
    index = times
)
print(aapl_hfts.head(5))
```

```
                            0
time
2024-04-05 13:30:00  169.540
2024-04-05 13:31:00  169.505
2024-04-05 13:32:00  169.300
2024-04-05 13:33:00  169.132
2024-04-05 13:34:00  169.030
```

Clean Data

```
aapl_hfts.reset_index(inplace=True)
aapl_hfts.rename(columns={0: 'price'}, inplace=True)
aapl_hfts['date'] = aapl_hfts['time'].dt.date
print(aapl_hfts.head(5))
```

```
                  time    price        date
0 2024-04-05 13:30:00  169.540  2024-04-05
1 2024-04-05 13:31:00  169.505  2024-04-05
2 2024-04-05 13:32:00  169.300  2024-04-05
3 2024-04-05 13:33:00  169.132  2024-04-05
4 2024-04-05 13:34:00  169.030  2024-04-05
```

Aggregate

```
print(aapl_hfts.groupby('date')['price'].ohlc())
```

```
              open    high     low   close
date
2024-04-05  169.54  170.38  169.03  169.47
```

Aggregate

```
print(aapl_hfts.groupby('date')['price'].ohlc())
```

```
              open    high     low   close
date
2024-04-05  169.54  170.38  169.03  169.47
```

If we don't want the grouped columns as an index, we can pass that to groupby:

```
agg_data = aapl_hfts.groupby('date', as_index=False)[['price']
print(agg_data)
print(agg_data.columns)
```

```
         date       price
0  2024-04-05  169.681324
Index(['date', 'price'], dtype='object')
```

Renaming Aggregate Columns

If we're using agg, we often don't like the names it spits out – it's easy to change them, we just need to pass a tuple:

```
new_agg = aapl_hfts.groupby('date', as_index=False)[['price']]
    [('average_price', 'mean'),
     ('sd', np.std)]
)
new_agg
```

	price	
	average_price	sd
date		
2024-04-05	169.681324	0.281085

Aside: Hierarchical Columns

Note that the columns here are now hierarchical

```
new_agg.columns
```

```
MultiIndex([('price', 'average_price'),
            ('price',             'sd')],
           )
```

Aside: Hierarchical Columns

Note that the columns here are now hierarchical

```
new_agg.columns
```

```
MultiIndex([('price', 'average_price'),
            ('price',             'sd')],
           )
```

If we want to get columns from a level, we can do so:

```
new_agg.columns.get_level_values(1)
```

```
Index(['average_price', 'sd'], dtype='object')
```

Aside: Hierarchical Columns

Note that the columns here are now hierarchical

```
new_agg.columns
```

```
MultiIndex([('price', 'average_price'),
            ('price',             'sd')],
           )
```

If we want to get columns from a level, we can do so:

```
new_agg.columns.get_level_values(1)
```

```
Index(['average_price', 'sd'], dtype='object')
```

We can also just flatten the index:

```
new_agg.columns.to_flat_index()
```

```
Index([('price', 'average_price'), ('price', 'sd')], dtype='obje
```

Aside: Hierarchical Columns

Note that the columns here are now hierarchical

```
new_agg.columns
```

```
MultiIndex([('price', 'average_price'),
            ('price',             'sd')],
           )
```

If we want to get columns from a level, we can do so:

```
new_agg.columns.get_level_values(1)
```

```
Index(['average_price', 'sd'], dtype='object')
```

We can also just flatten the index:

```
new_agg.columns.to_flat_index()
```

```
Index([('price', 'average_price'), ('price', 'sd')], dtype='obje
```

Aggregate with Multiple Functions

```
print(aapl_hfts.groupby('date')['price'].agg(['ohlc', 'mean',
```

```
                ohlc                                    mean     min
                open    high     low   close          price   price
date
2024-04-05   169.54  170.38  169.03  169.47  169.681324  169.03
```

Aggregate with Multiple Functions

```
print(aapl_hfts.groupby('date')['price'].agg(['ohlc', 'mean',
```

```
                ohlc                                    mean     min
                open    high     low   close          price   price
date
2024-04-05   169.54   170.38   169.03   169.47   169.681324   169.03
```

This gives you the flavor and use cases that I've most encountered, but if you want to get really in-depth (pandas can do some crazy stuff), check out McKinney's chapter.