

# Introduction to Python pandas

**Dial-In: 1-800-853-0223**  
**Code: 963825**

Hannah Kronenberg (RADAR, FRB Philadelphia)

[Hannah.Kronenberg@phil.frb.org](mailto:Hannah.Kronenberg@phil.frb.org)

January 29<sup>th</sup>, 2020

# Data for Today's Presentation

Below is a list of some of the data we will be using in today's presentation.

**pew.csv** : This is a csv file of income distribution by religion based off of data made available by the Pew Research Center.

**freddie\_static.csv** : This is training data based off of Freddie Mac single family performance loan-level data.

**loans\_values.csv** : This is a dataset with names, loan amounts, and loan values

**radar\_wants\_candy.csv** : This is a toy dataset of radar teams and their favorite candies.

**Zip\_Zhvi\_AllHomes.csv** : Zipcode level Zillow Home Value Index.

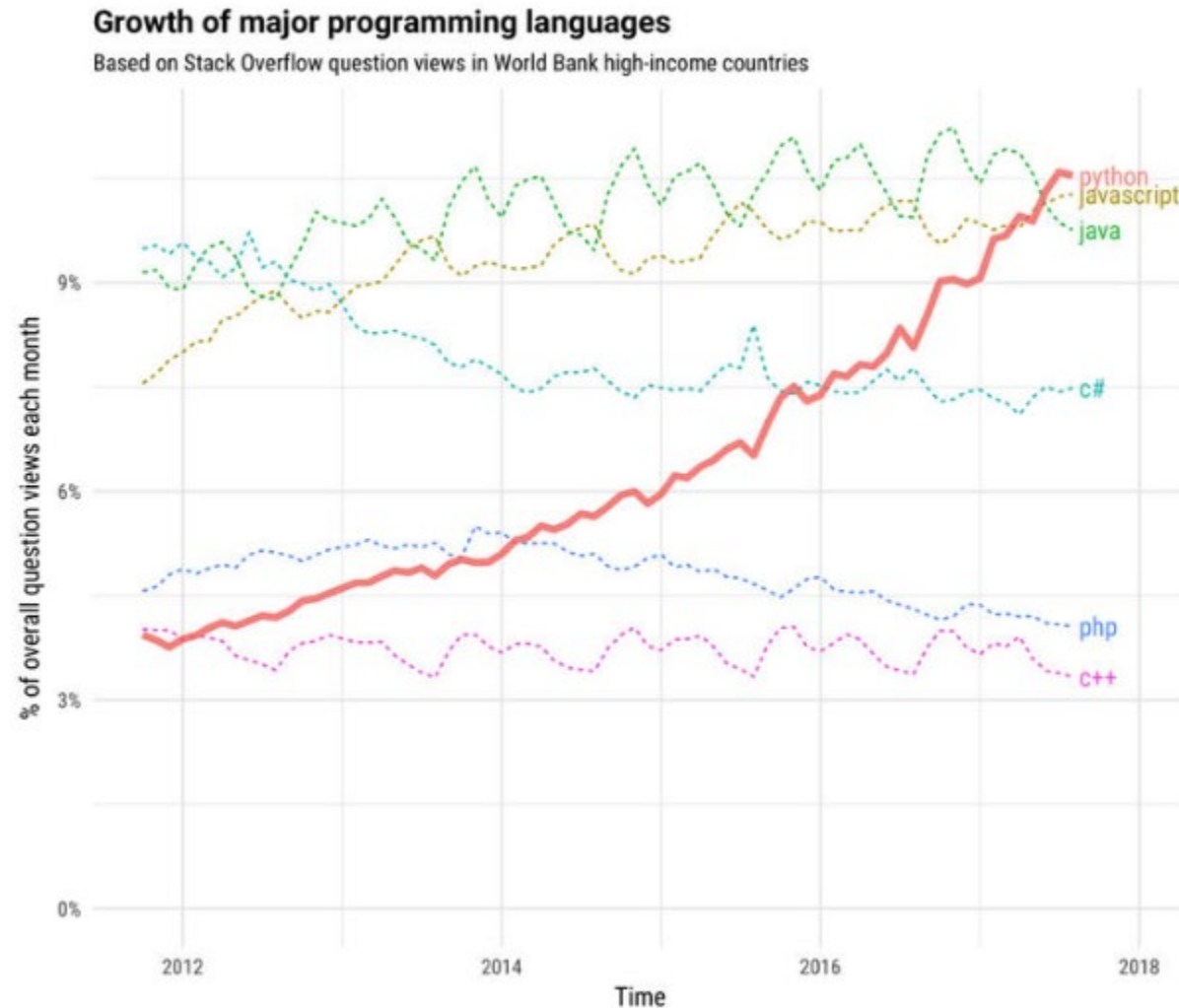
The data in today's training are specifically for the purpose of learning how to use Python pandas. The data should not be used for anything apart from trainings and playing around in Python.

# Why Use Python?

- Python is open source and free!
- Python is very popular for machine learning and webscraping techniques.
- Python is not only a useful data analysis tool, it is also popular for software development.



# Python is Rapidly Becoming More Popular



# Jupyter Notebooks

*“The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more.”*

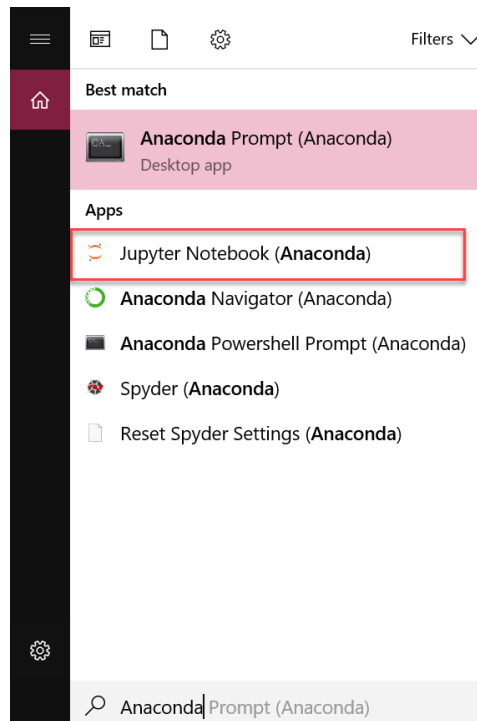
– [Project Jupyter](#)

- Jupyter notebooks are great for performing exploratory analysis.
- It is also a convenient tool for creating a data analysis narrative.
- There is a jupyter notebook that pairs with this presentation!



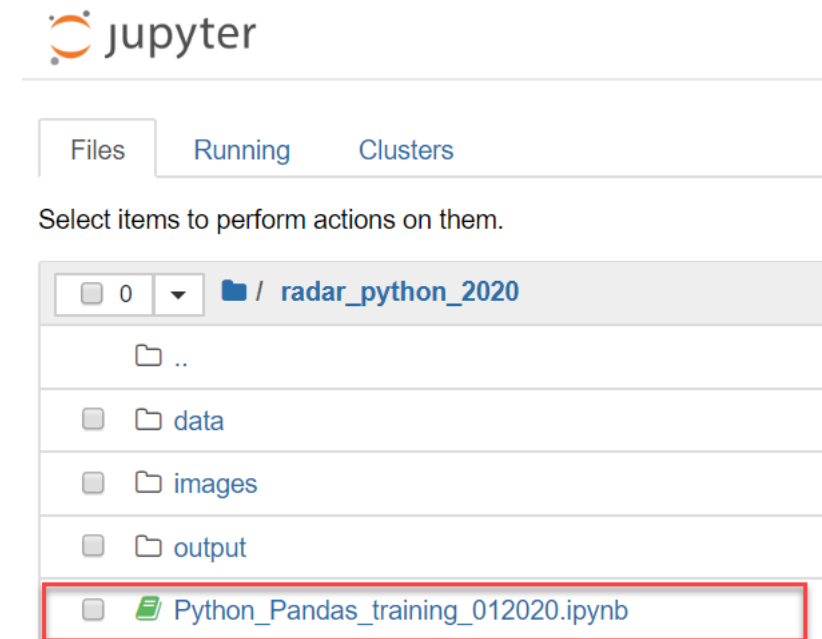
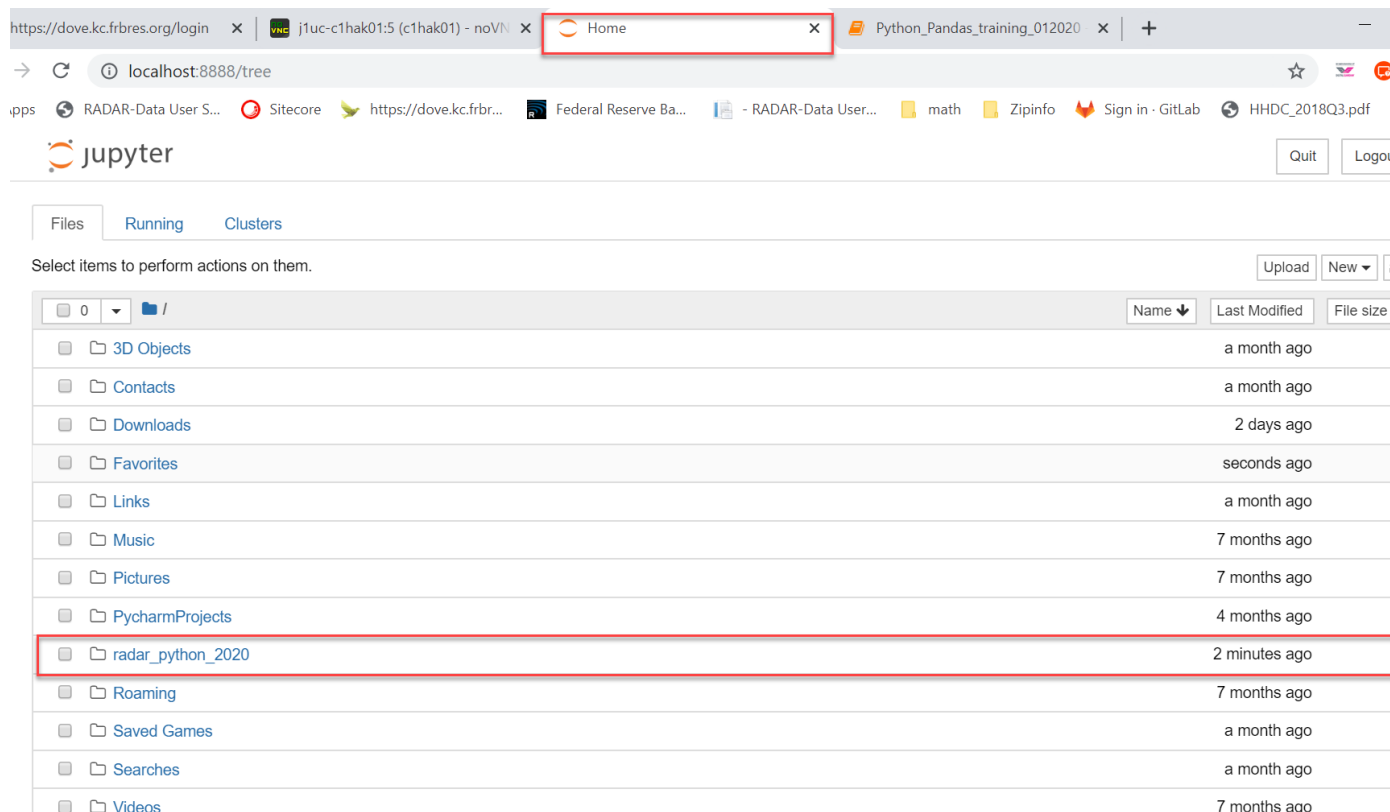
# Accessing Jupyter Notebooks/Anaconda

- If you want to use Jupyter Notebooks you need to get Anaconda installed on your computer.
  - Anaconda should be available for all RBs, contact your local IT for installation options.
- Once you have Anaconda you can open a number of interfaces including Jupyter.



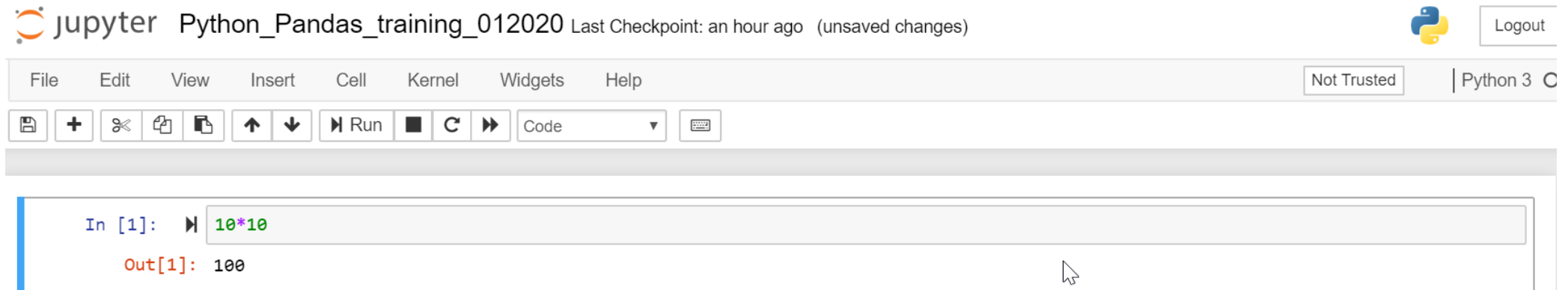
# Accessing Jupyter Notebooks/Anaconda (cont'd)

- Jupyter notebooks will open in a browser. Navigate to wherever you saved your .ipynb file.





# Jupyter Notebook Example!



## Introduction to Python Pandas

Hannah Kronenberg (RADAR, FRB Philadelphia)

January 29th 2020

## Data for Today's Presentation

**Below is a list of some of the data we will be using in today's presentation.**

- pew.csv : This is a csv file of income distribution by religious distribution based off of data made available by the Pew Research Center.
- freddie\_static.csv : This is training data based off of Freddie Mac single family performance loan-level data.



# Common Data Types in Python

- **Integers:** An integer is a whole number

```
an_int = 10
```

- You can perform basic math on integers

```
In [44]: an_int*2
```

```
Out[44]: 20
```

- **Floats:** A float is a number with a decimal.

```
my_float = 10.1
```

- An integer times a float is a float.
- 2 divided by 3 is a float (it would not result in rounding or truncation)

```
In [45]: 2/3
```

```
Out[45]: 0.6666666666666666
```

- **Strings:** A string is a sequence of characters surrounded by quotes.

```
cool_String = 'Hannah is awesome'
```

- You can multiply and add strings
- You can index and slice strings

```
cool_String[0:6]
```

```
'Hannah'
```

You can use single ' ' or double " " quotes to represent strings in python

# Common Data Types in Python

- **Lists:** Create a list by specifying comma separated values in square brackets.

- Values in a list do not need to be the same type.

```
ls_1 = ['Hannah', 1, 5.5, 'Talks too much']
```

```
ls_2 = list(range(10))
```

- **Booleans:** A boolean is either True or False.

- You can either specify True or False or give a variable a condition that can be evaluated as either True or False.

```
false_bool = len(ls_1) == 3
```

```
true_bool = True
```

- **Dictionaries:** A dictionary is a set of keys and corresponding values.

- Keys must be unique
  - Values can repeat

```
my_dict = {'a':'Boston', 'b':'New York', 'c':'Philadelphia', 'd':'Cleveland', 'e':'Richmond', 'f':'Atlanta',  
          'g':'Chicago', 'h':'St.Louis', 'i':'Minneapolis', 'j':'Kansas City', 'k':'Dallas', 'l':'San Francisco',  
          'm':'Board'}
```

```
my_dict['a']
```

```
Out[32]: 'Boston'
```

# Find Out a Data Type

- You can find out the data type of an object by using the following command:
  - `type(object)`

```
In [10]: type(an_int)
```

```
Out[10]: int
```

```
In [11]: type(10==10)
```

```
Out[11]: bool
```

```
In [12]: type('Hannah')
```

```
Out[12]: str
```

You can use similar syntax for SOME type casting. For example `float(an_int)` will turn “an\_int” into a float.

```
an_int = float(an_int)
print(type(an_int))

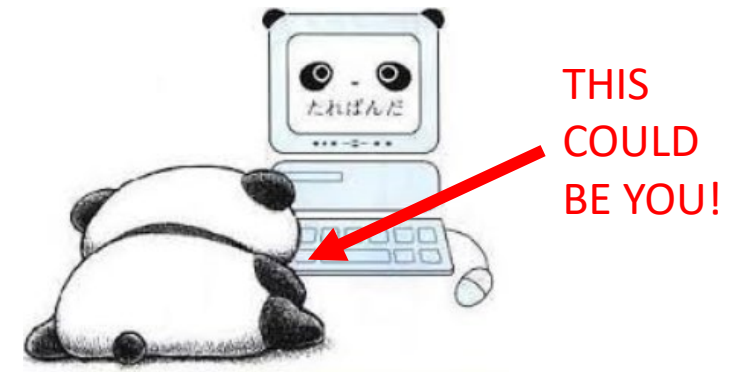
<class 'float'>
```

Note: Python will let you name an object a data type like “int” or “bool” BUT YOU SHOULD NOT!

# pandas!

- Base Python can be really useful but it's not the best way to do data analysis or data visualization
- pandas is the most popular Python library for data analysis.
- It makes tasks such as loading, transforming, summarizing and visualizing data much easier than in base Python.
- Import pandas using the line below:

```
import pandas as pd
```



The “as pd” is optional but it is a common abbreviation used when importing pandas

# Two Important Objects in pandas

Series and DataFrames are the two key objects in pandas.


- Series: Represents a column of data.

```
city_names = pd.Series(['San Francisco','San Jose','Sacramento'])
```

```
population = pd.Series([852469, 1015785,485199])
```

- DataFrame: A table structure with rows and columns.
  - A DataFrame contains one or more Series

Column Names



```
city_df = pd.DataFrame({'Name': city_names, 'Population': population})
```

# Access a Column in A data Frame

city\_df

Row numbers

	Name	Population
0	San Francisco	852469
1	San Jose	1015785
2	Sacramento	485199

city\_df['Population']

```
0      852469
1    1015785
2     485199
Name: Population, dtype: int64
```

city\_df[['Population']]

	Population
0	852469
1	1015785
2	485199

## Loading a CSV in pandas

- You will very rarely be creating your own data frames from scratch.
- You need to be able to load some data from a csv.

Basic Syntax for importing a csv:

```
df = pd.read_csv('filepath/data.csv', sep = 'separator')
```



The separator option is only necessary if the file is delimited by something besides a comma



# Loading a CSV in pandas

- Let's take a look at our static Freddie Mac data frame

```
freddie = pd.read_csv('data/freddie_static.csv',sep=';')
```

```
freddie.head()
```

	id	creditbureau_score_orig	first_pmt_dt	first_homebuyer_flg	maturity_dt	mi_pct_orig	units_no	occupancy_type	cltv_ratio_orig	dti_ratio_orig	loan_ar
0	2435	790	199905	Y	202904	30	1	P	95	27	
1	2576	795	199905	N	202904	999	1	P	73	29	
2	2671	749	199903	N	202902	25	1	P	90	24	
3	2766	657	199904	N	202903	0	1	P	76	23	
4	2860	702	199904	N	202903	0	1	S	56	38	

## Loading a CSV in pandas

- Sometimes you will be working with data that doesn't have a header row when it comes in.

Will	100	199
Matt	25	78
Jennifer	132	210
Meera	76	222
Aaron	46	145

- You will want to set column names and types before you bring in your data.

# Loading a CSV in pandas

**Put your types in a list in the order that they appear.**

```
origclass = ['str', 'float64', 'float64']
```

**Put your column names in a list in the order that they appear.**

```
origcols = ['Name', 'Loan', 'Value']
```

**Use a list comprehension to assign data types in the origclass list to the columns in the origcols list.**

```
origtypes = {origcols[i]: origclass[i] for i in range(len(origcols))}
```

**Load your data using your lists of data types and column names.**

```
loans_values = pd.read_csv('data/loans_values.csv', header=None, dtype = origtypes, names = origcols)
```

# What is a list comprehension???

- List comprehension are a cool way to work with lists in python!
- Let's look at our example from the previous slide:

```
origtypes = {origcols[i]: origclass[i] for i in range(len(origcols))}
```

- We could have instead used a for loop which would have been:

```
origtypes = {}  
for i in range(len(origcols)):  
    origtypes[origcols[i]] = origclass[i]
```

- Both methods create a dictionary **origtypes** and then say:
  - Map each value in origcols to the value in origclass that is at the same index.

# Renaming Variables in pandas

- You also might want to rename a column in pandas

```
df.rename(columns={'original_name' : 'new_name'}, inplace = True)
```

What if I want to rename the variable “id” in the static Freddie data to “loan\_id” ?

```
freddie.rename(columns={'id' : 'loan_id'}, inplace = True)
```

	loan_id	creditbureau_score_orig	first_pmt_dt	first_homebuyer_flg	maturity_dt	mi_pct_orig	units_no
0	2435	790	199905	Y	202904	30	1
1	2576	795	199905	N	202904	999	1
2	2671	749	199903	N	202902	25	1
3	2766	657	199904	N	202903	0	1
4	2860	702	199904	N	202903	0	1

# Date Formats in pandas

- Sometimes a data type won't be in the format you need it to be.
- This can especially be true when working with dates.
- If you want to change a variable to “date” format, use the following syntax:

```
data['date_var'] = pd.to_datetime(data['date_var'], format = 'date format')
```

Let's try changing first\_pmt\_dt to a date formatted variable in the Freddie Data.

# Date Formats in pandas

- First payment date in the Freddie data is in the format 'YYYYMM.' In python this is represented as %Y%m.

	id	creditbureau_score_orig	first_pmt_dt	first_homebuyer_flg	maturity_dt	m
0	2435	790	199905	Y	202904	
1	2576	795	199905	N	202904	
2	2671	749	199903	N	202902	
3	2766	657	199904	N	202903	
4	2860	702	199904	N	202903	

```
freddie['first_pmt_dt'] = pd.to_datetime(freddie['first_pmt_dt'], format = '%Y%m')
```

	id	creditbureau_score_orig	first_pmt_dt	first_homebuyer_flg	maturity_dt	m
0	2435	790	1999-05-01	Y	202904	
1	2576	795	1999-05-01	N	202904	
2	2671	749	1999-03-01	N	202902	
3	2766	657	1999-04-01	N	202903	
4	2860	702	1999-04-01	N	202903	

- What's the point?



# Date Formats in pandas

- If you want to extract “date” objects from a variable, it needs to be in date format.
- The basic syntax for extracting date objects from a date variable is:

```
data['new varname'] = data['Date Variable'].dt.dateobject
```

- Let's extract month from the Freddie data.

```
freddie['first_pmt_month'] = freddie['first_pmt_dt'].dt.month
```

	id	creditbureau_score_orig	first_pmt_dt	first_pmt_month
0	2435	790	1999-05-01	5
1	2576	795	1999-05-01	5
2	2671	749	1999-03-01	3
3	2766	657	1999-04-01	4
4	2860	702	1999-04-01	4

# Filter

- You will often want to cut down a data frame.
- You can use `.loc` to filter your data.

```
df_filtered = df.loc[(condition)]
```

- Common operators for filter conditions include: `>`, `>=`, `<`, `<=`, `!=` (not equals), `==` (equals).
- If you want to combine conditions in a filter you can use `&`, `&&` (and) or `|`, `||` (or).

What if I want to look at loans in the `freddie_static` data with a `loan_amt_orig > $100,000`?

## Filtering your data

- Look at loans in the Freddie training data with origination amount greater than \$100,000

```
freddie_filtered_amt = freddie.loc[freddie['loan_amt_orig']>100000]
```

- Look at loans with first payment date equal to 201612.

```
freddie_filtered_dt = freddie.loc[freddie['first_pmt_dt']= 201612]
```

## Filter (Cont'd)

```
freddie_filtered_dt = freddie.loc[freddie['first_pmt_dt'] = 201612]
```

```
File "<ipython-input-50-6e18eff6da6a>", line 1  
    freddie_filtered_dt = freddie.loc[freddie['first_pmt_dt'] = 201612]  
                                ^
```

SyntaxError: invalid syntax

- Tests for equivalency MUST use **==**
- **=** in Python is used for assignment.

The correct syntax is:

```
freddie_filtered_dt = freddie.loc[freddie['first_pmt_dt'] == 201612]
```

# Summarize data

- When we have new data we often will want to find out certain summary statistics.
- The basic syntax for summarizing data is:

```
dataframe.groupby([groupby variables])[variable you want to summarize].summaryfunction()
```

- Let's get the average loan amount at origination in the Freddie Mac Data by first payment date.

# Summarize data

- Average loan amount at origination from the Freddie Mac data.

```
freddie.groupby(['first_pmt_dt'])['loan_amt_orig'].mean()
```

```
first_pmt_dt
199902    115413.793183
199903    126369.450715
199904    127642.752562
199905    125431.023182
199906    125386.786949
199907    128995.801847
199908    127333.000000
199909    121189.349112
199910    124426.564496
199911    122146.687697
199912    121123.809524
200001    120204.761905
200002    126414.089347
200003    126585.882353
200004    125546.539379
200005    126648.601399
200006    127023.809524
200007    128807.795699
200008    127470.149254
200009    129316.113161
200010    131974.301676
200011    134422.554348
200012    131537.558685
200101    138276.110444
200102    143894.610778
200103    149631.982475
200104    147374.622356
200105    146631.333722
200106    146382.655827
200107    147984.210526
...
201708    274500.000000
201709    193000.000000
201710    227500.000000
201711    338000.000000
201712    144500.000000
201801    249250.000000
Name: loan_amt_orig, Length: 228, dtype: float64
```

# Transform Data

- Sometimes you will want to perform operations on a variable but preserve the number of rows originally in your data.
- One way to do this is by using “transform”.
- This functionality is similar to window functions in SQL and “mutate” in R.
- In the radar\_wants\_candy table we want to see how many people there are in each team. But we also want to keep the favorite\_ct by candy and radar\_group.

	radar_group	candy	favorite_ct
0	FMG	Twix	2
1	FMG	Snickers	3
2	FMG	Kitkat	2
3	SMT	Twix	1
4	SMT	Snickers	5
5	SMT	Kitkat	2
6	DW	Twix	3
7	DW	Snickers	3
8	DW	Kitkat	3
9	SES	Twix	1
10	SES	Snickers	1
11	SES	Kitkat	3



# Team Size with Transform

The code below returns the full original table with a new column “team\_size” that contains the size of each radar\_group.

```
radar_wants_candy['team_size']=radar_wants_candy.groupby(['radar_group'])['favorite_ct'].transform(sum)
```

Note: Transform does not change the number of rows in the table.

	radar_group	candy	favorite_ct	team_size
0	FMG	Twix	2	7
1	FMG	Snickers	3	7
2	FMG	Kitkat	2	7
3	SMT	Twix	1	8
4	SMT	Snickers	5	8
5	SMT	Kitkat	2	8
6	DW	Twix	3	9
7	DW	Snickers	3	9
8	DW	Kitkat	3	9
9	SES	Twix	1	5
10	SES	Snickers	1	5
11	SES	Kitkat	3	5

# Tidy vs. Untidy Data

Similar to working in the R Tidyverse, it is best to do analysis in Python pandas using “tidy” data.

## Tidy data has 3 main principles<sup>1</sup>:

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

### Untidy Data

	Religious_tradition	lt_30K	30K_49K	50K_99K	100K_plus
0	Buddhist	0.36	0.18	0.32	0.13
1	Catholic	0.36	0.19	0.26	0.19
2	Evangelical Protestant	0.35	0.22	0.28	0.14
3	Hindu	0.17	0.13	0.34	0.36
4	Historically Black Protestant	0.53	0.22	0.17	0.08
5	Jehovah's Witness	0.48	0.25	0.22	0.04
6	Jewish	0.16	0.15	0.24	0.44
7	Mainline Protestant	0.29	0.20	0.28	0.23
8	Mormon	0.27	0.20	0.33	0.20
9	Muslim	0.34	0.17	0.29	0.20
10	Orthodox Christian	0.18	0.17	0.36	0.29
11	Unaffiliated (religious "nones")	0.33	0.20	0.26	0.21

### Tidy Data

	Religious_tradition	income	pct
0	Buddhist	lt_30K	0.36
1	Catholic	lt_30K	0.36
2	Evangelical Protestant	lt_30K	0.35
3	Hindu	lt_30K	0.17
4	Historically Black Protestant	lt_30K	0.53
5	Jehovah's Witness	lt_30K	0.48
6	Jewish	lt_30K	0.16
7	Mainline Protestant	lt_30K	0.29
8	Mormon	lt_30K	0.27
9	Muslim	lt_30K	0.34
10	Orthodox Christian	lt_30K	0.18
11	Unaffiliated (religious "nones")	lt_30K	0.33
12	Buddhist	30K_49K	0.18
13	Catholic	30K_49K	0.19
14	Evangelical Protestant	30K_49K	0.22
15	Hindu	30K_49K	0.13
16	Historically Black Protestant	30K_49K	0.22
17	Jehovah's Witness	30K_49K	0.25
18	Jewish	30K_49K	0.15
19	Mainline Protestant	30K_49K	0.20

1. <https://tidyr.tidyverse.org/articles/tidy-data.html>

2. Data source: <https://www.pewforum.org/religious-landscape-study/income-distribution/>

# How do we “Tidy” our Data?

- We first bring in our “untidy” dataset. This dataset is not tidy because **column headers are values**, in this case income brackets, **as opposed to variable names**.

	Religious_tradition	lt_30K	30K_49K	50K_99K	100K_plus
0	Buddhist	0.36	0.18	0.32	0.13
1	Catholic	0.36	0.19	0.26	0.19
2	Evangelical Protestant	0.35	0.22	0.28	0.14
3	Hindu	0.17	0.13	0.34	0.36
4	Historically Black Protestant	0.53	0.22	0.17	0.08
5	Jehovah's Witness	0.48	0.25	0.22	0.04
6	Jewish	0.16	0.15	0.24	0.44
7	Mainline Protestant	0.29	0.20	0.28	0.23
8	Mormon	0.27	0.20	0.33	0.20
9	Muslim	0.34	0.17	0.29	0.20
10	Orthodox Christian	0.18	0.17	0.36	0.29
11	Unaffiliated (religious "nones")	0.33	0.20	0.26	0.21

## How do we “Tidy” our Data? (cont'd)

- We can use the **melt** command to gather the 4 income bracket value columns into a 2 column **key-value** pair. Within the melt command we need to name our two new columns and specify **which column(s) do not** feed into the **key-value** pair.

First we specify the **columns that don't feed** into the pair

Then we name the **value column** Finally, we name the **key column**

`pew_long = pd.melt(pew_wide, id_vars = ['Religious_tradition'], value_name = 'pct', var_name = 'income')`



	Religious_tradition	income	pct
0	Buddhist	lt_30K	0.36
1	Catholic	lt_30K	0.36
2	Evangelical Protestant	lt_30K	0.35
3	Hindu	lt_30K	0.17
4	Historically Black Protestant	lt_30K	0.53
5	Jehovah's Witness	lt_30K	0.48
6	Jewish	lt_30K	0.16
7	Mainline Protestant	lt_30K	0.29
8	Mormon	lt_30K	0.27
9	Muslim	lt_30K	0.34
10	Orthodox Christian	lt_30K	0.18
11	Unaffiliated (religious "nones")	lt_30K	0.33
12	Buddhist	30K_49K	0.18
13	Catholic	30K_49K	0.19
14	Evangelical Protestant	30K_49K	0.22
15	Hindu	30K_49K	0.13
16	Historically Black Protestant	30K_49K	0.22
17	Jehovah's Witness	30K_49K	0.25
18	Jewish	30K_49K	0.15
19	Mainline Protestant	30K_49K	0.20

## How do we “Untidy” our Data?

- Calculations are often more easily done using “tidy” data. However, we might prefer to view our data in the “untidy” or “wide” format. We can use the **pivot** command to convert our table back.

```
pew_wide_again = pew_long.pivot(index='Religious_tradition', columns='income', values='pct').reset_index()
```

	Religious_tradition	lt_30K	30K_49K	50K_99K	100K_plus
0	Buddhist	0.36	0.18	0.32	0.13
1	Catholic	0.36	0.19	0.26	0.19
2	Evangelical Protestant	0.35	0.22	0.28	0.14
3	Hindu	0.17	0.13	0.34	0.36
4	Historically Black Protestant	0.53	0.22	0.17	0.08
5	Jehovah's Witness	0.48	0.25	0.22	0.04
6	Jewish	0.16	0.15	0.24	0.44
7	Mainline Protestant	0.29	0.20	0.28	0.23
8	Mormon	0.27	0.20	0.33	0.20
9	Muslim	0.34	0.17	0.29	0.20
10	Orthodox Christian	0.18	0.17	0.36	0.29
11	Unaffiliated (religious "nones")	0.33	0.20	0.26	0.21

**MR. MESSY**



# The OS Module

- Sometimes in Python you will want to:
  - Find out what directory you are in
  - Change directories
  - Perform command line operations
- In order to do these things you need to import the os module.

```
import os
```

- Some common commands include:  
os.getcwd() to return your current working directory.

```
os.getcwd()
```

os.chdir('filepath') to change your working directory to a specified 'filepath'.

```
os.chdir('filepath')
```

- For more info, check out Python's documentation on the os module: <https://docs.python.org/3/library/os.html>.

# Exercise Using Zillow ZHVI Data!



# Our Task

1. Load the Zillow Zip Code ZHVI Data
2. Filter our data to only Philadelphia, PA ZIP Codes
3. Reshape the data from wide to long format
4. Calculate the average ZHVI by ZIP Code for each year in the data
5. Identify the ZIP Code with the maximum average ZHVI for each year
6. Cut down your data to the maximum average ZHVI and zip code with the max by year for Philadelphia zip codes
7. Export a csv with your results

## Read in Zillow ZHVI Data

- Read in the Zillow ZHVI Data
- To read in this data you need to add in the argument engine = 'python'.

```
zillow = pd.read_csv('data/Zip_Zhvi_AllHomes.csv', engine = 'python')
```

```
zillow.head()
```

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	1996-05	1996-06	...	2018-02	2018-03	2018-04	2018-05	2018-06
0	61639	10025	New York	NY	New York-Newark-Jersey City	New York County	1	171600.0	171600.0	171400.0	...	1130500	1123700	1119500	1116900	1113000
1	84654	60657	Chicago	IL	Chicago-Naperville-Elgin	Cook County	2	158400.0	159700.0	160700.0	...	351600	352900	351900	350400	348000
2	61637	10023	New York	NY	New York-Newark-Jersey City	New York County	3	347900.0	349600.0	351100.0	...	1516000	1497900	1497800	1504600	1480000
3	91982	77494	Katy	TX	Houston-The Woodlands-Sugar Land	Harris County	4	210400.0	212200.0	212200.0	...	326600	330400	332600	334500	331000

## Filter to Philadelphia, Pennsylvania

First, I filter down to observations where the city name is Philadelphia.

```
phil_hv = zillow.loc[(zillow['City']=='Philadelphia')]
```

To make sure I filtered correctly I print out unique cities and states in my new dataframe

```
print(phil_hv['City'].unique() , phil_hv['State'].unique())
```

```
['Philadelphia'] ['PA' 'MS' 'TN' 'NY']
```



## Filter to Philadelphia, Pennsylvania

I need to filter by both City AND State.

```
phil_hv = zillow.loc[(zillow['City']=='Philadelphia') & (zillow['State']=='PA')]
```

That's more like it!

```
print(phil_hv['City'].unique() , phil_hv['State'].unique())
```

```
['Philadelphia'] ['PA']
```

# Dropping Columns

A lot of columns are no longer necessary now that I have cut down my data frame.

First, I make a list of the columns I no longer need:

```
unwanted = ['RegionID', 'City', 'State', 'Metro', 'CountyName', 'SizeRank']
```

Then I drop all the columns in the list from my data frame using the “drop” command:

```
phil_hv_final = phil_hv.drop(unwanted, axis=1)
```



# Renaming a Column

We know this is zipcode level data, therefore “RegionName” signifies zipcode\*.

```
phil_hv_final.rename(columns={'RegionName':'zip'}, inplace=True)
```

	zip	1996-04	1996-05	1996-06	1996-07	1996-08	1996-09	1996-10	1996-11	1996-12	...	2018-02	2018-03	2018-04	2018-05	2018-06	2018-07	1
214	19111	84900.0	84700.0	84500.0	84300.0	84200.0	84100.0	83900.0	83800.0	83700.0	...	174300	175500	175700	175800	176400	177300	
300	19124	43100.0	43000.0	42900.0	42700.0	42500.0	42400.0	42300.0	42300.0	42300.0	...	86200	88000	88900	89900	90800	91400	
377	19120	46100.0	46100.0	46100.0	46100.0	46000.0	45900.0	45900.0	45800.0	45700.0	...	91100	92100	92400	92400	92600	92800	
542	19148	41100.0	41100.0	41000.0	40900.0	40700.0	40600.0	40500.0	40400.0	40400.0	...	205700	209500	211800	213400	214900	215500	
690	19145	41000.0	41000.0	41000.0	40900.0	40800.0	40700.0	40700.0	40800.0	40900.0	...	204100	208400	209500	209100	209600	211100	

5 rows × 273 columns

But we still have  
nearly 300  
columns!!!



\*Zillow has a number of datasets [available online](#). For the dataset Zip\_Zhvi\_AllHomes “RegionName” signifies zipcode. On the other hand for City\_Zhvi\_AllHomes “RegionName” signifies City.

# Let's Reshape Our Data

We can use melt to reshape our data from wide to long.

- Our id variable is “zip”
- The variable whose value we want to see by zipcode is ZHVI
- The data we are reshaping are all dates so our new variable is “Date”

```
phil_ZHVI = pd.melt(phil_hv_final, id_vars=['zip'], value_name='ZHVI', var_name='Date')
```

	zip	Date	ZHVI
0	19111	1996-04	84900.0
1	19124	1996-04	43100.0
2	19120	1996-04	46100.0
3	19148	1996-04	41100.0
4	19145	1996-04	41000.0





# Get the Mean ZHVI by Year

- Now we want to get the mean ZHVI for each zipcode by year
- The first thing we will need to do is create a year variable that contains the year extracted from our “Date” variable.

```
phil_ZHVI['Year'] = phil_ZHVI['Date'].dt.year
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-74-40a4ed0bd8ab> in <module>()
----> 1 phil_ZHVI['Year'] = phil_ZHVI['Date'].dt.year

/software/anaconda/3/5.3.0/lib/python3.7/site-packages/pandas/core/generic.py in __getattr__(self, name)
   4370     if (name in self._internal_names_set or name in self._metadata or
   4371         name in self._accessors):
-> 4372         return object.__getattribute__(self, name)
   4373     else:
   4374         if self._info_axis._can_hold_identifiers_and_holds_name(name):

/software/anaconda/3/5.3.0/lib/python3.7/site-packages/pandas/core/accessor.py in __get__(self, obj, cls)
   131         # we're accessing the attribute of the class, i.e., Dataset.geo
   132         return self._accessor
-> 133     accessor_obj = self._accessor(obj)
   134     # Replace the property with the accessor object. Inspired by:
   135     # http://www.pydanny.com/cached-property.html
                                     ^
/software/anaconda/3/5.3.0/lib/python3.7/site-packages/pandas/core/indexes/accessors.py in __new__(cls, data)
   323     pass # we raise an attribute error anyway
   324
-> 325     raise AttributeError("Can only use .dt accessor with datetimelike "
   326                          "values")

AttributeError: Can only use .dt accessor with datetimelike values
```



# Get the Mean ZHVI by Year

- The “Date” variable was not in a “datetime” format.
- We need to convert “Date” to a “datetime” format if we want to perform date operations (such as extract date values) on it.

```
phil_ZHVI['Date'] = pd.to_datetime(phil_ZHVI['Date'])
```

Now we can extract year!

```
phil_ZHVI['Year'] = phil_ZHVI['Date'].dt.year
```

	zip	Date	ZHVI	Year
0	19111	1996-04-01	84900.0	1996
1	19124	1996-04-01	43100.0	1996
2	19120	1996-04-01	46100.0	1996
3	19148	1996-04-01	41100.0	1996
4	19145	1996-04-01	41000.0	1996

## Now Let's Take The Mean

- We now can take the mean ZHVI by year and zipcode.

```
avg_annual_phil_ZHVI = phil_ZHVI.groupby(['zip', 'Year'])['ZHVI'].mean()
```

```
zip    Year  
19102  1996    79966.666667  
       1997    83166.666667  
       1998    92550.000000  
       1999   114358.333333  
       2000   145175.000000  
Name: ZHVI, dtype: float64
```

# Don't Forget to Reset the Index

- As we saw on the previous slide our data does not look the way we might expect it to.
- We need to reset the index.
- If you do not reset the index after performing a groupby calculation, you will not be able to perform additional calculations on the dataframe.

```
avg_annual_phil_ZHVI = avg_annual_phil_ZHVI.reset_index()
```

	zip	Year	ZHVI
0	19102	1996	79966.666667
1	19102	1997	83166.666667
2	19102	1998	92550.000000
3	19102	1999	114358.333333
4	19102	2000	145175.000000

# Getting the Max ZHVI By Year

- I want my final data frame to be the zip codes with the maximum ZHVI value in each year
- The first step is to calculate the maximum ZHVI by year across the entire data frame while maintaining the original table.

```
avg_annual_phil_ZHVI['zhvi_max']=avg_annual_phil_ZHVI.groupby(['Year'])['ZHVI'].transform(max)
```

	zip	Year	ZHVI	zhvi_max
0	19102	1996	79966.666667	182400.000000
1	19102	1997	83166.666667	184358.333333
2	19102	1998	92550.000000	187950.000000
3	19102	1999	114358.333333	207350.000000
4	19102	2000	145175.000000	239900.000000

## Final Result

Finally we need to filter down to observations where the ZHVI for that year is equal to the max across zip codes.

```
phil_maxZHVI=avg_annual_phil_ZHVI.loc[avg_annual_phil_ZHVI['ZHVI']==avg_annual_phil_ZHVI['zhvi_max']]
```

	zip	Year	ZHVI	zhvi_max
30	19103	2003	353258.333333	353258.333333
34	19103	2007	492158.333333	492158.333333
35	19103	2008	489900.000000	489900.000000
36	19103	2009	497866.666667	497866.666667
37	19103	2010	466416.666667	466416.666667

## Let's Clean it Up!

- ZHVI and zhvi\_max are the same thing so I can drop one of them

```
phil_maxZHVI_final=phil_maxZHVI [['Year','zip','ZHVI']]
```

- Finally I can sort my final table by year

```
phil_maxZHVI_final=phil_maxZHVI_final.sort_values('Year')
```



# Final Result



	Year	zip	ZHVI
184	1996	19118	182400.000000
185	1997	19118	184358.333333
186	1998	19118	187950.000000
187	1999	19118	207350.000000
188	2000	19118	239900.000000
189	2001	19118	274108.333333
190	2002	19118	311075.000000
30	2003	19103	353258.333333
192	2004	19118	411558.333333
193	2005	19118	474808.333333
194	2006	19118	489216.666667
34	2007	19103	492158.333333
35	2008	19103	489900.000000
36	2009	19103	497866.666667
37	2010	19103	466416.666667
199	2011	19118	436083.333333
200	2012	19118	425241.666667
40	2013	19103	425908.333333
41	2014	19103	449225.000000
203	2015	19118	472150.000000
204	2016	19118	499550.000000
205	2017	19118	520166.666667
206	2018	19118	535990.909091



# Export Our Results

- We can export our results using the to\_csv command.

```
df.to_csv('filepath/filename.csv', index=False)
```



```
phil_maxZHVI_final.to_csv('output/phil_zhvi_result.csv', index=False)
```

- Set index=False unless you want to retain row numbers.



# Thank you

- **Questions about this presentation?**

Contact [Hannah Kronenberg](#)

- **Trainings don't stop here!**

## **February 5: [Special Topics in SQL](#)**

- This training will focus on ways to use SQL as well as best practices for users to remember when using SQL. Topics covered will include **connecting to databases using R** and **debugging code**.
- **Stay tuned for more Python trainings in 2020**
- Visit our [Training Page](#) for slide decks and to sign up for future trainings in FedLearn
- Recordings available for all training series events effective May 2019