

Pandas

- It is an IPython library for data manipulation
- Has functions to work with any kind of data such as tabular data

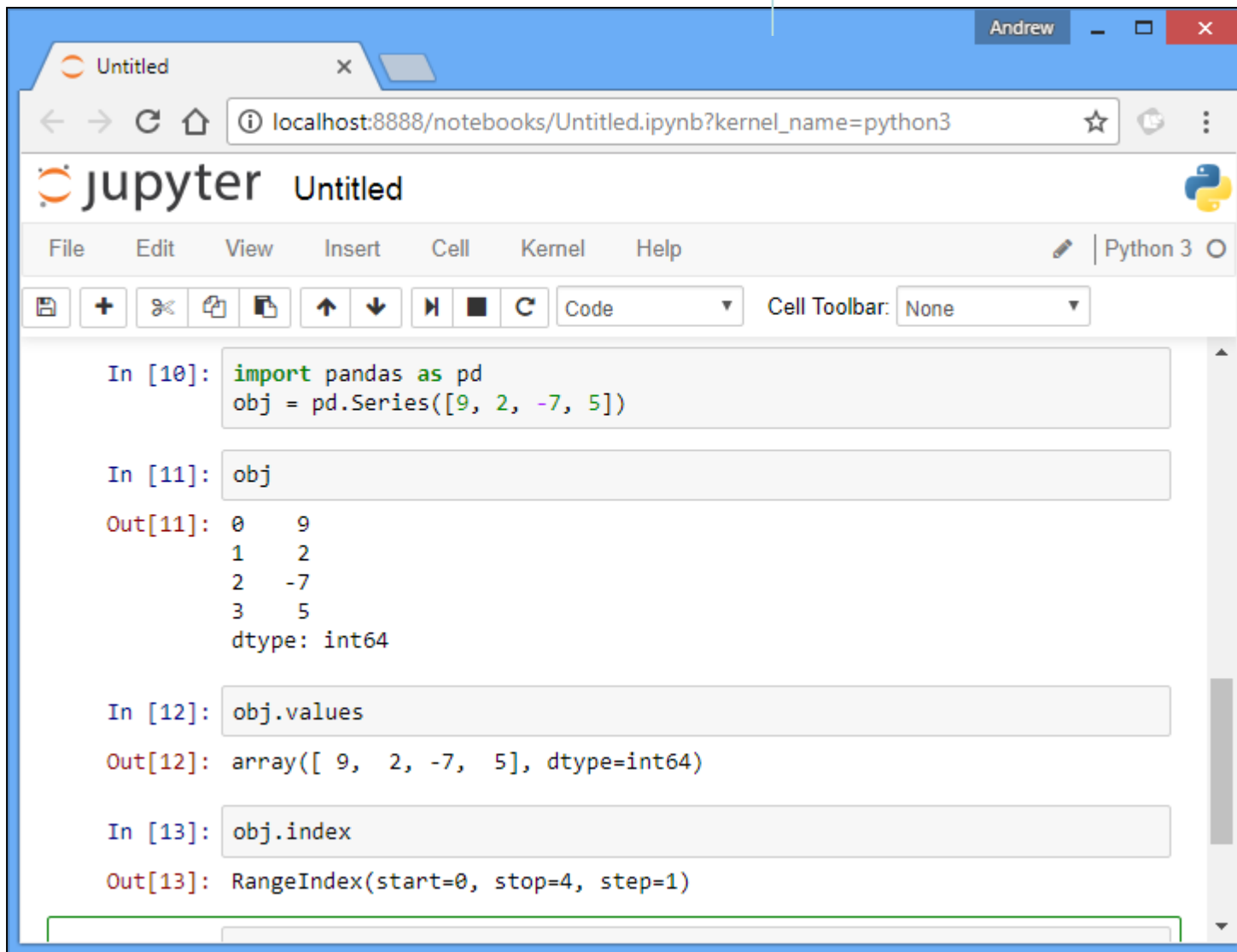
Data Structures of Pandas

- All data representation in Pandas is done using one of two data structures:
 - Series
 - DataFrame

Series

- A Series is a one-dimensional array-like **object** containing a sequence of values and an associated array of data labels, called its *index*
- A Series object wraps both its values and indices

Series



The screenshot shows a Jupyter Notebook window titled "Untitled" with a browser address bar at `localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3`. The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations and cell execution. The code cells and their outputs are as follows:

```
In [10]: import pandas as pd
         obj = pd.Series([9, 2, -7, 5])

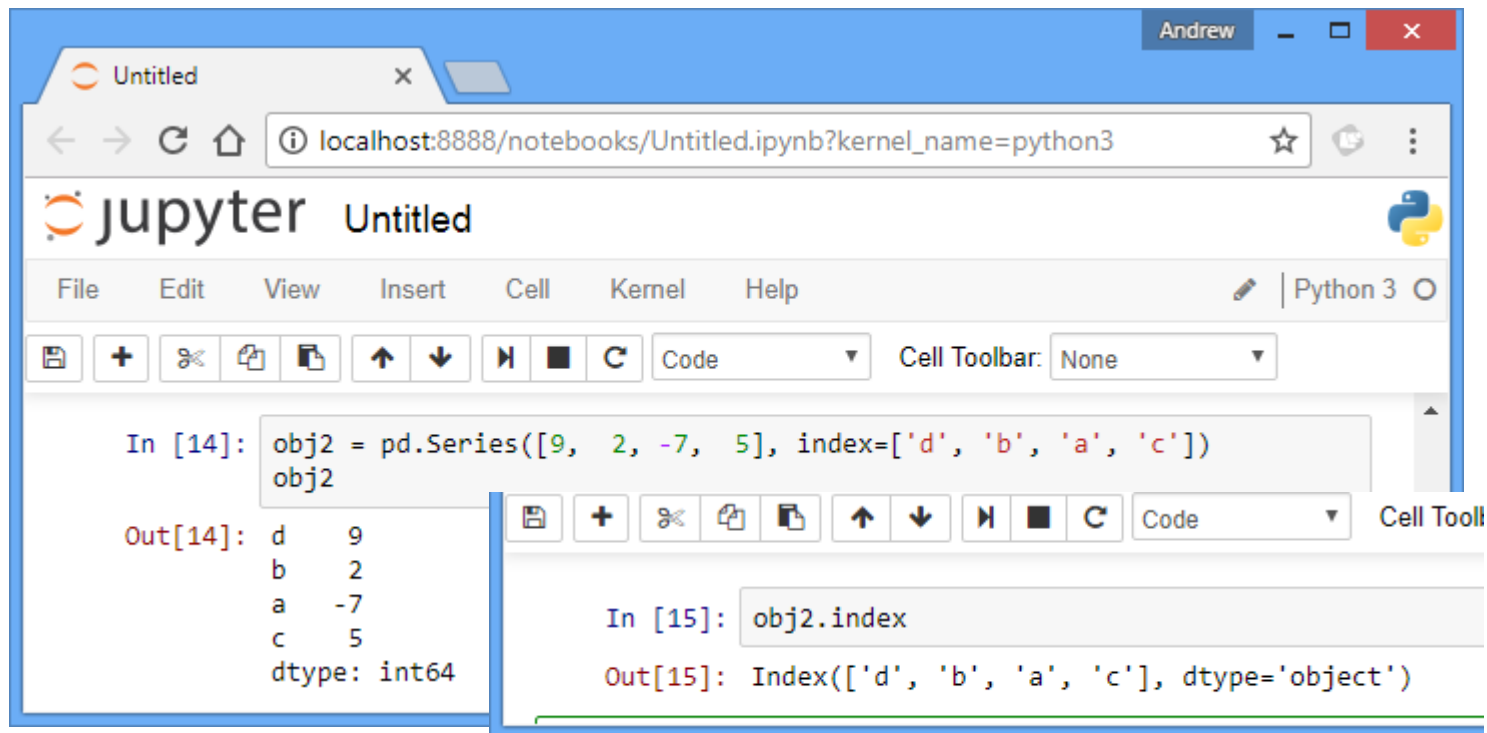
In [11]: obj
Out[11]: 0    9
         1    2
         2   -7
         3    5
         dtype: int64

In [12]: obj.values
Out[12]: array([ 9,  2, -7,  5], dtype=int64)

In [13]: obj.index
Out[13]: RangeIndex(start=0, stop=4, step=1)
```

Series

- You can create a Series with an index identifying each data point with a label



The screenshot shows a Jupyter Notebook window titled 'Untitled' with a browser address bar at `localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3`. The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar. The first code cell (In [14]) contains the following Python code:

```
In [14]: obj2 = pd.Series([9, 2, -7, 5], index=['d', 'b', 'a', 'c'])
obj2
```

The output (Out[14]) displays the Series data:

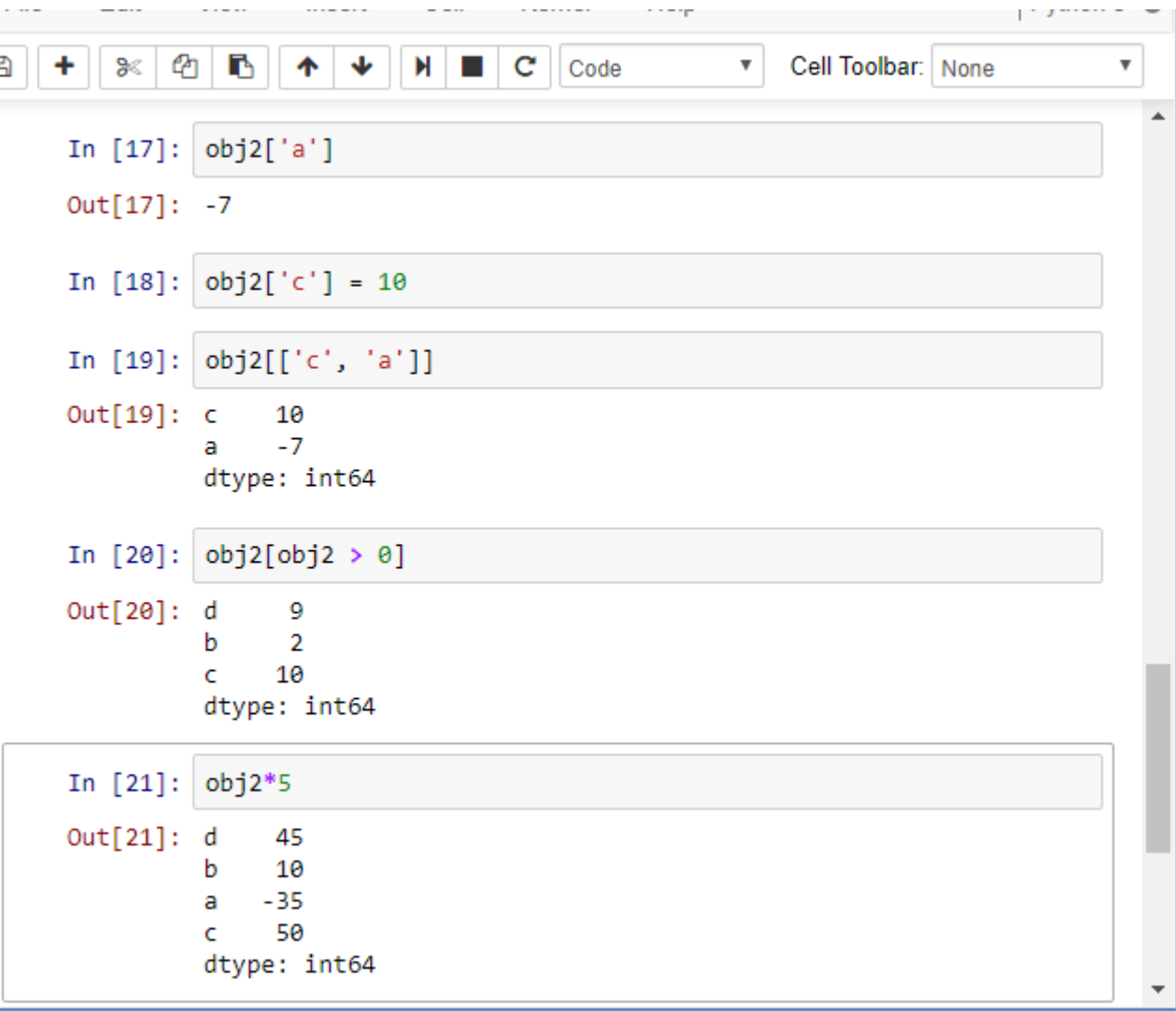
```
Out[14]: d    9
         b    2
         a   -7
         c    5
         dtype: int64
```

The second code cell (In [15]) contains the following Python code:

```
In [15]: obj2.index
```

The output (Out[15]) displays the index:

```
Out[15]: Index(['d', 'b', 'a', 'c'], dtype='object')
```



The image shows a Jupyter Notebook interface with a toolbar at the top containing icons for adding, deleting, and running code cells, as well as a 'Code' dropdown and a 'Cell Toolbar' dropdown set to 'None'. Below the toolbar are four code cells, each followed by its output.

```
In [17]: obj2['a']  
Out[17]: -7
```

```
In [18]: obj2['c'] = 10
```

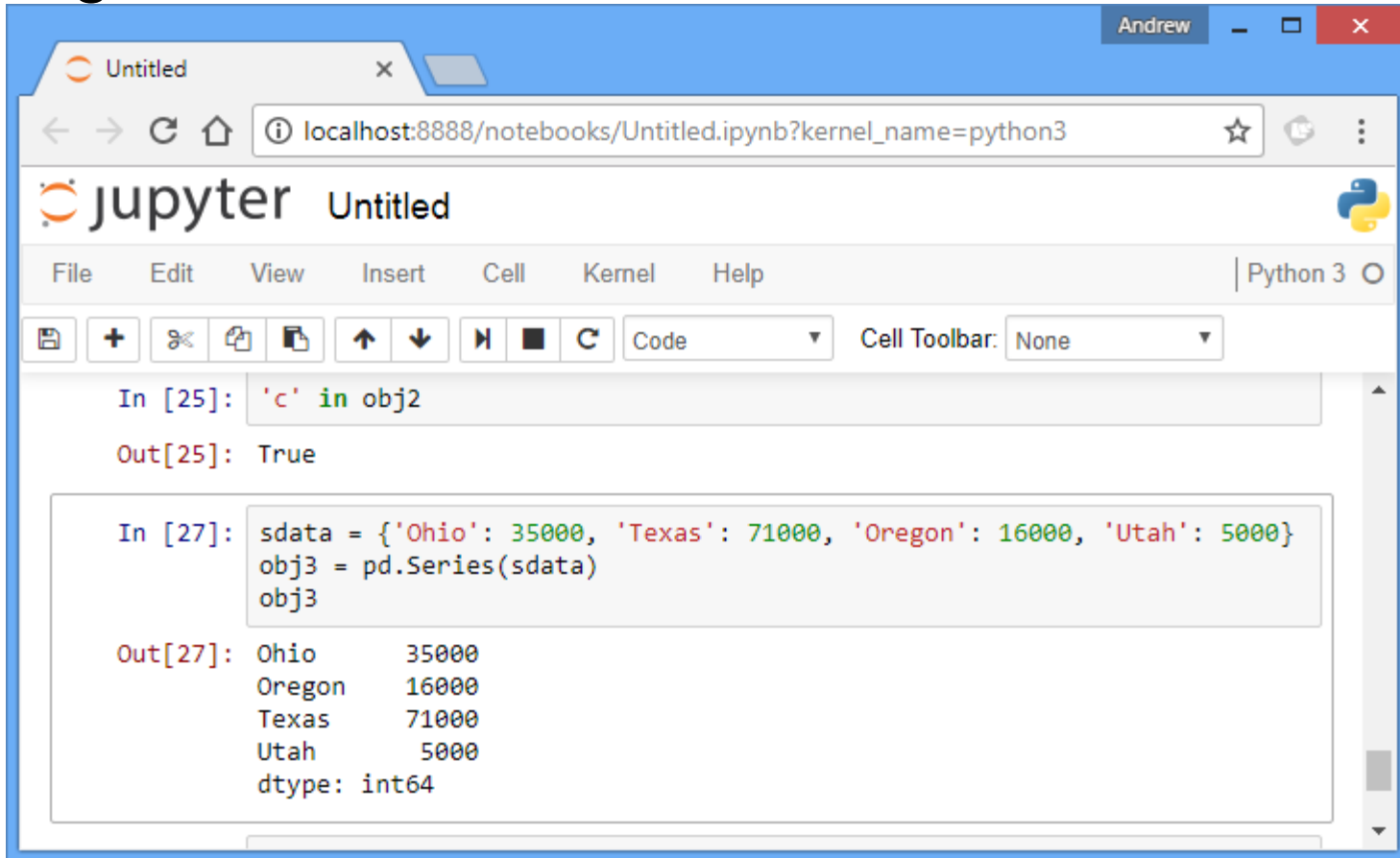
```
In [19]: obj2[['c', 'a']]  
Out[19]: c    10  
         a    -7  
         dtype: int64
```

```
In [20]: obj2[obj2 > 0]  
Out[20]: d     9  
         b     2  
         c    10  
         dtype: int64
```

```
In [21]: obj2*5  
Out[21]: d    45  
         b    10  
         a   -35  
         c    50  
         dtype: int64
```

Series

- Another way to think about a Series is as a dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict

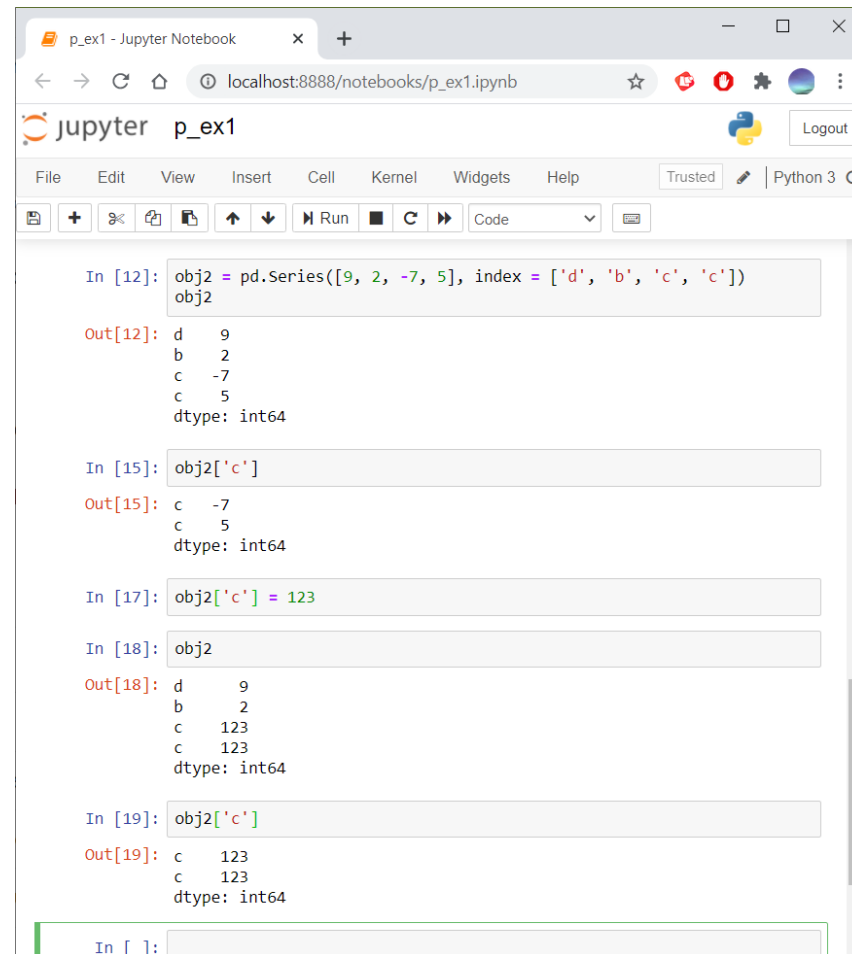


The screenshot shows a Jupyter Notebook window titled 'Untitled' with a browser address bar at 'localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3'. The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar. The code cell shows the following execution:

```
In [25]: 'c' in obj2
Out[25]: True

In [27]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
         obj3 = pd.Series(sdata)
         obj3
Out[27]: Ohio      35000
         Oregon   16000
         Texas    71000
         Utah      5000
         dtype: int64
```

The index/label values in a Series do not have to be unique



The screenshot shows a Jupyter Notebook interface with a browser window at localhost:8888/notebooks/p_ex1.ipynb. The notebook title is 'p_ex1'. The code and output are as follows:

```
In [12]: obj2 = pd.Series([9, 2, -7, 5], index = ['d', 'b', 'c', 'c'])
obj2
```

```
Out[12]: d    9
         b    2
         c   -7
         c    5
         dtype: int64
```

```
In [15]: obj2['c']
```

```
Out[15]: c   -7
         c    5
         dtype: int64
```

```
In [17]: obj2['c'] = 123
```

```
In [18]: obj2
```

```
Out[18]: d    9
         b    2
         c  123
         c  123
         dtype: int64
```

```
In [19]: obj2['c']
```

```
Out[19]: c  123
         c  123
         dtype: int64
```

```
In [ ]:
```


DataFrame

- Dataframe (DF) is the most important and useful data structure in pandas
- It is used for almost all kind of data representation and manipulation in pandas
- A DF is a table containing rows (representing objects/observations) and columns (representing features/variables/attributes)
- Unlike numpy arrays, it can contain heterogeneous data
- DataFrames are useful for representing raw datasets and processed datasets in ML and data science

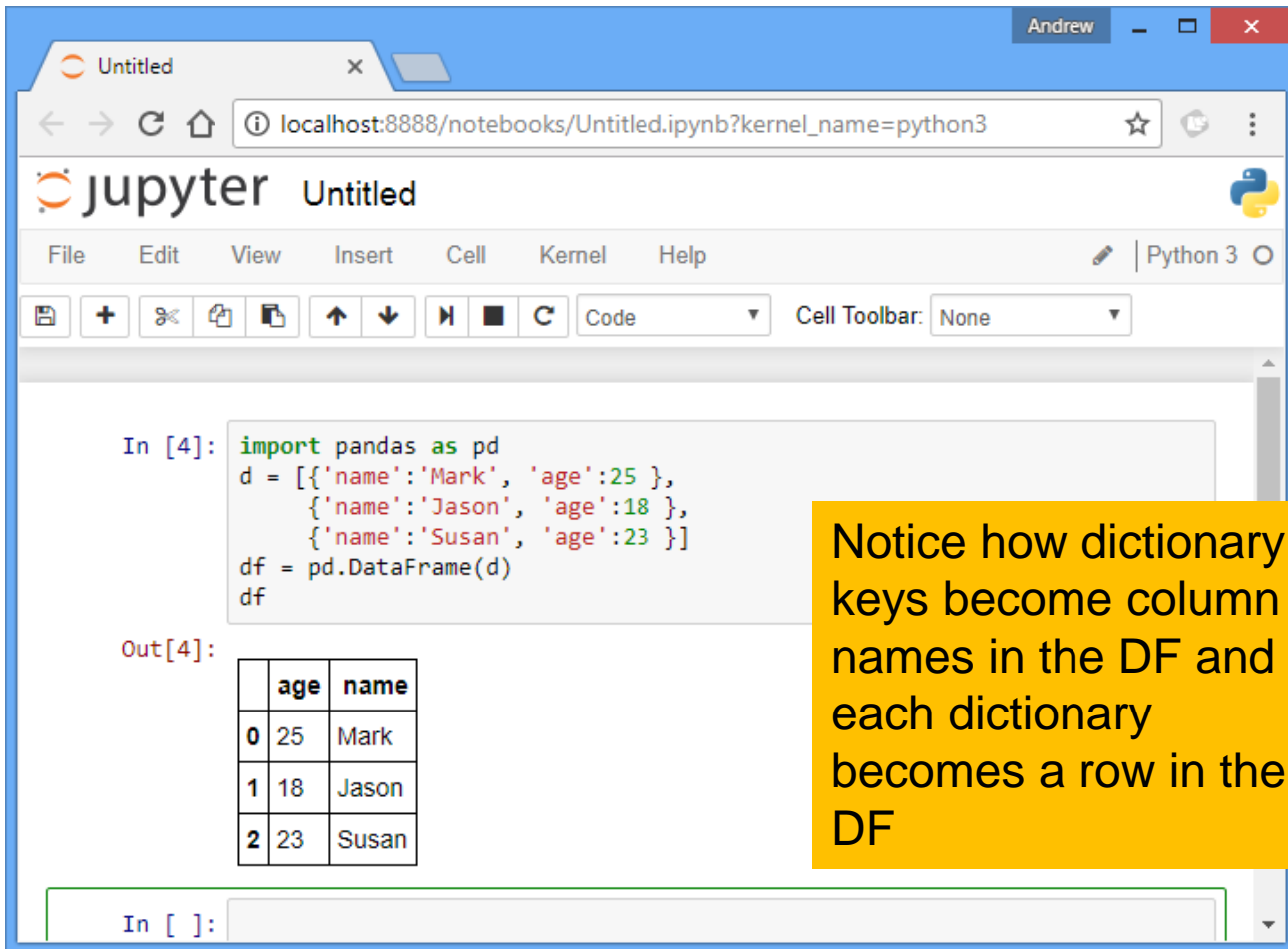
Relationship between DataFrames and Series

- We can think of a DataFrame as a sequence of aligned Series objects
 - aligned means that they share the same index

Constructing Data Frame Objects

- A DataFrame object can be constructed in different ways, including:
 - from a list of dicts
 - from a dict of dicts
 - from a list of Series objects
 - from a dict of Series objects
 - from a 2-D NumPy array
 - from a 2-D Python list

List of Dictionaries to a Dataframe



A screenshot of a Jupyter Notebook interface. The browser address bar shows `localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3`. The notebook title is "Untitled". The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. The toolbar shows various icons for file operations and code execution. The code cell (In [4]:) contains the following Python code:

```
import pandas as pd
d = [{'name': 'Mark', 'age': 25 },
     {'name': 'Jason', 'age': 18 },
     {'name': 'Susan', 'age': 23 }]
df = pd.DataFrame(d)
df
```

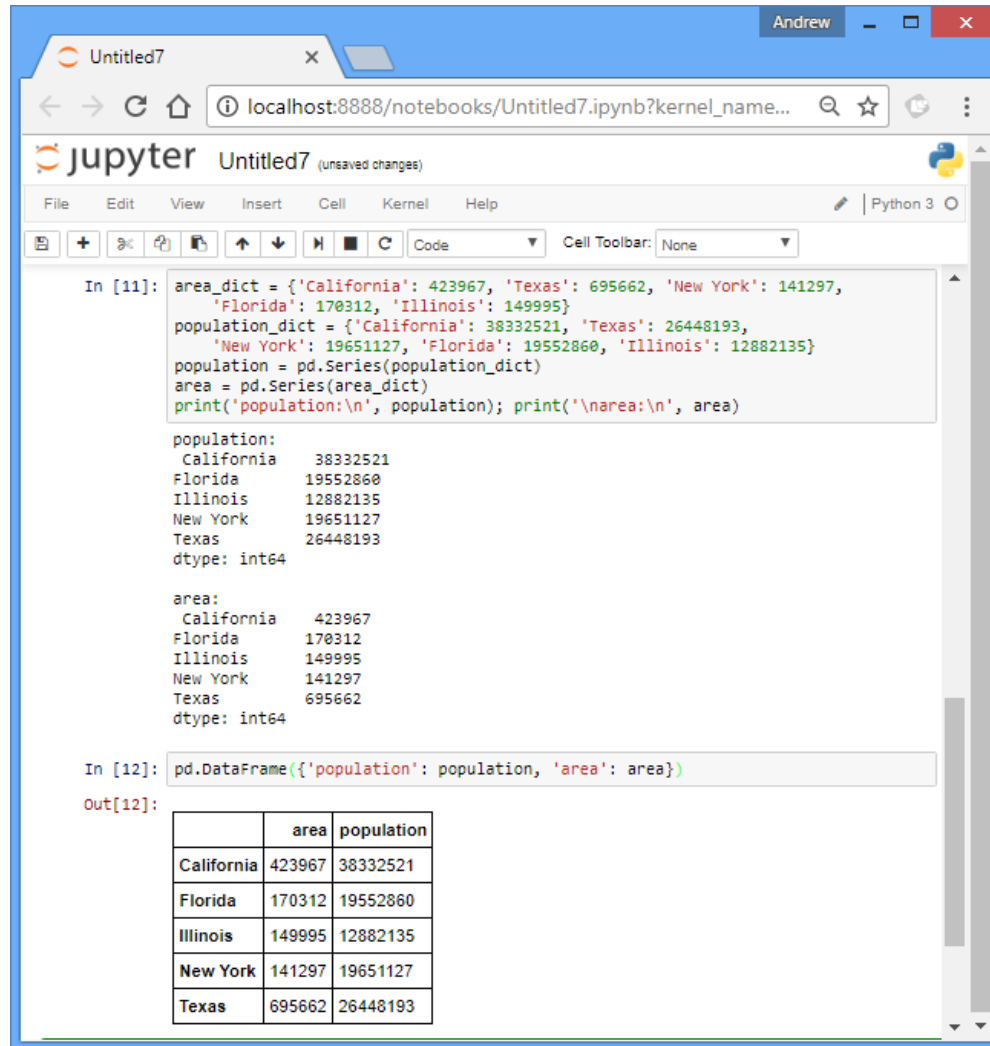
The output (Out[4]:) is a DataFrame table:

	age	name
0	25	Mark
1	18	Jason
2	23	Susan

Below the table is an input prompt `In []:`.

Notice how dictionary keys become column names in the DF and each dictionary becomes a row in the DF

A Dictionary of Series Objects to a Dataframe



The screenshot shows a Jupyter Notebook interface with a browser window at localhost:8888. The notebook has a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar. The code in the first cell (In [11]) defines two dictionaries, 'area_dict' and 'population_dict', for five US states. These are converted into pandas Series objects and then printed. The second cell (In [12]) uses these Series to create a pandas DataFrame. The output (Out[12]) is a table with columns 'area' and 'population'.

```
In [11]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
'Florida': 170312, 'Illinois': 149995}
population_dict = {'California': 38332521, 'Texas': 26448193,
'New York': 19651127, 'Florida': 19552860, 'Illinois': 12882135}
population = pd.Series(population_dict)
area = pd.Series(area_dict)
print('population:\n', population); print('\narea:\n', area)

population:
California    38332521
Florida      19552860
Illinois     12882135
New York     19651127
Texas        26448193
dtype: int64

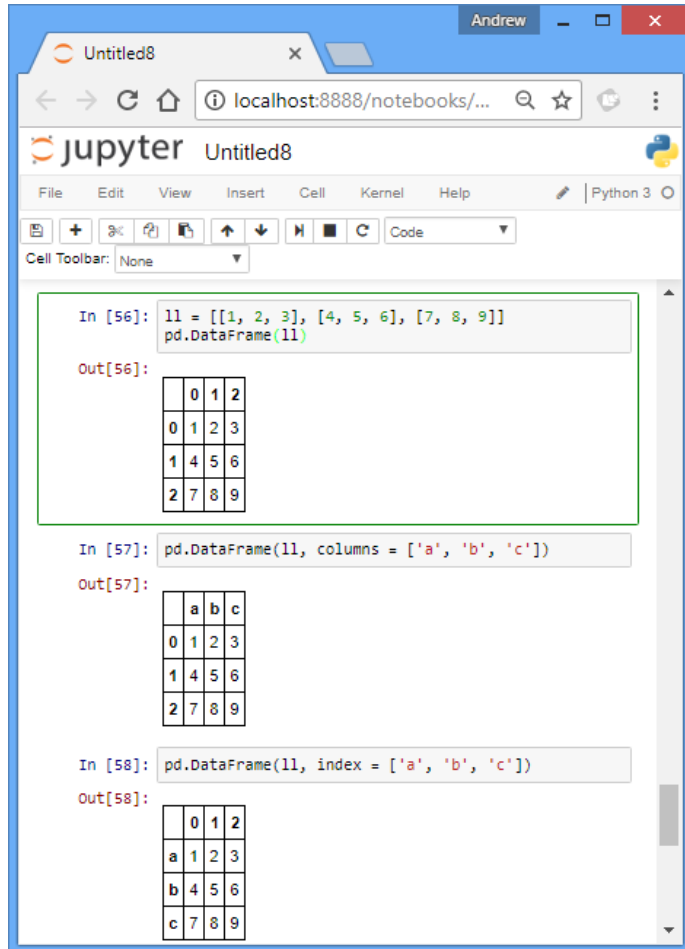
area:
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
dtype: int64

In [12]: pd.DataFrame({'population': population, 'area': area})

Out[12]:
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

A 2-D List to a DataFrame



Untitled8

localhost:8888/notebooks/...

jupyter Untitled8

File Edit View Insert Cell Kernel Help Python 3

Cell Toolbar: None

```
In [56]: ll = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
         pd.DataFrame(ll)
```

Out[56]:

0	1	2
0	1	2
1	4	5
2	7	8

```
In [57]: pd.DataFrame(ll, columns = ['a', 'b', 'c'])
```

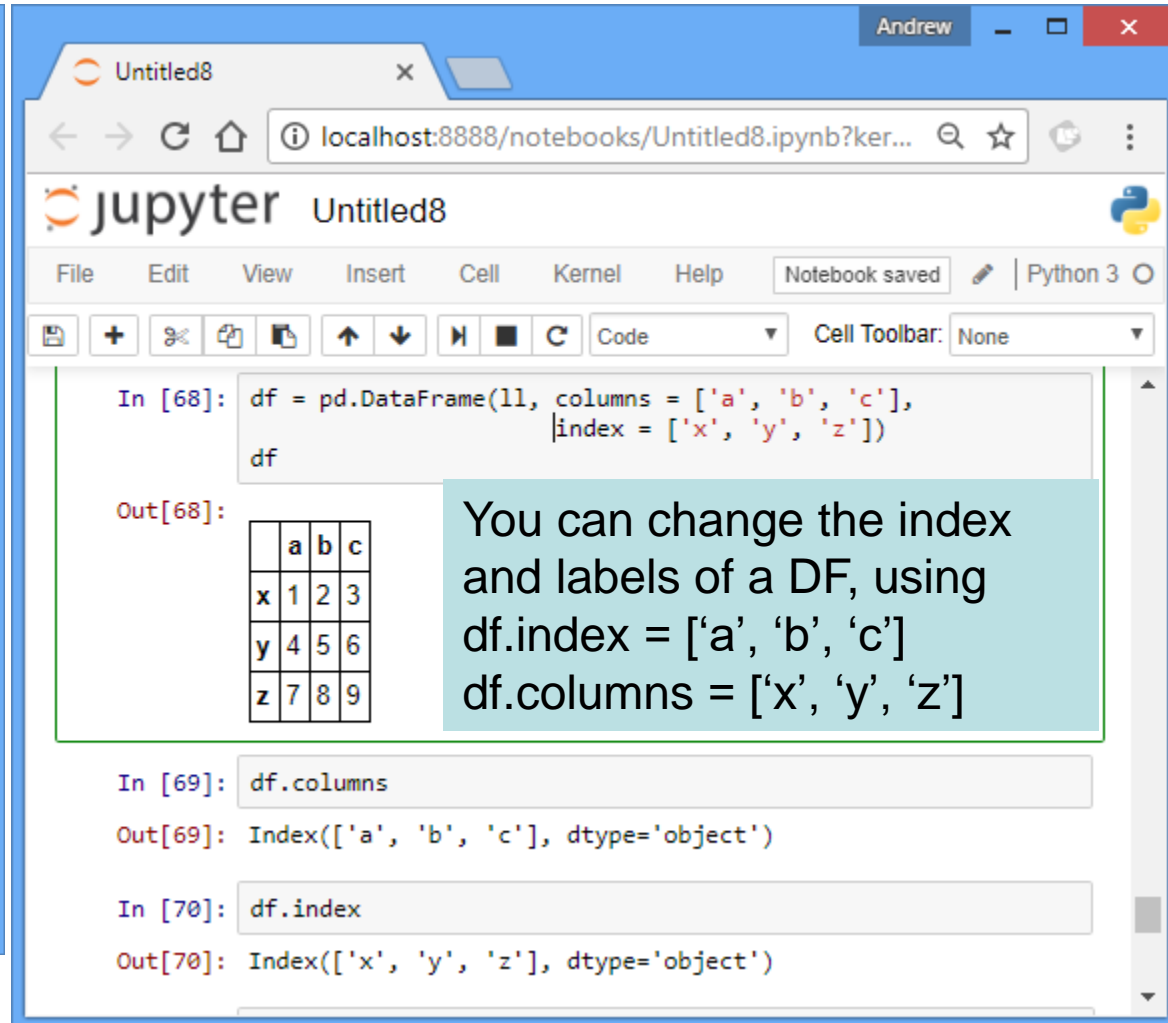
Out[57]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

```
In [58]: pd.DataFrame(ll, index = ['a', 'b', 'c'])
```

Out[58]:

0	1	2
a	1	2
b	4	5
c	7	8



Untitled8

localhost:8888/notebooks/Untitled8.ipynb?ker...

jupyter Untitled8

File Edit View Insert Cell Kernel Help Notebook saved Python 3

Cell Toolbar: None

```
In [68]: df = pd.DataFrame(ll, columns = ['a', 'b', 'c'],
         df
         index = ['x', 'y', 'z'])
```

Out[68]:

	a	b	c
x	1	2	3
y	4	5	6
z	7	8	9

You can change the index and labels of a DF, using
`df.index = ['a', 'b', 'c']`
`df.columns = ['x', 'y', 'z']`

```
In [69]: df.columns
```

Out[69]: Index(['a', 'b', 'c'], dtype='object')

```
In [70]: df.index
```

Out[70]: Index(['x', 'y', 'z'], dtype='object')

Data Retrieval

- Pandas provides different ways to retrieve data into a DataFrame. For example, we can read flat files, csv files, database tables, ...etc.

CSV Files to a Dataframe

- We can read a CSV, or any delimited file, using pandas and get the result as a DF
- This is done using panda's `read_csv()` function
- Usually, the only parameter needed is the name of the input file but sometimes you may need to add more parameters

Examples

- We are using the files simplemaps-worldcities-basic.csv and nyc_taxi data

Examples

Code Cell Toolbar: None

```
In [30]: import pandas as pd  
city_data = pd.read_csv('simplemaps-worldcities-basic.csv')  
city_data.head(3)
```

Out[30]:

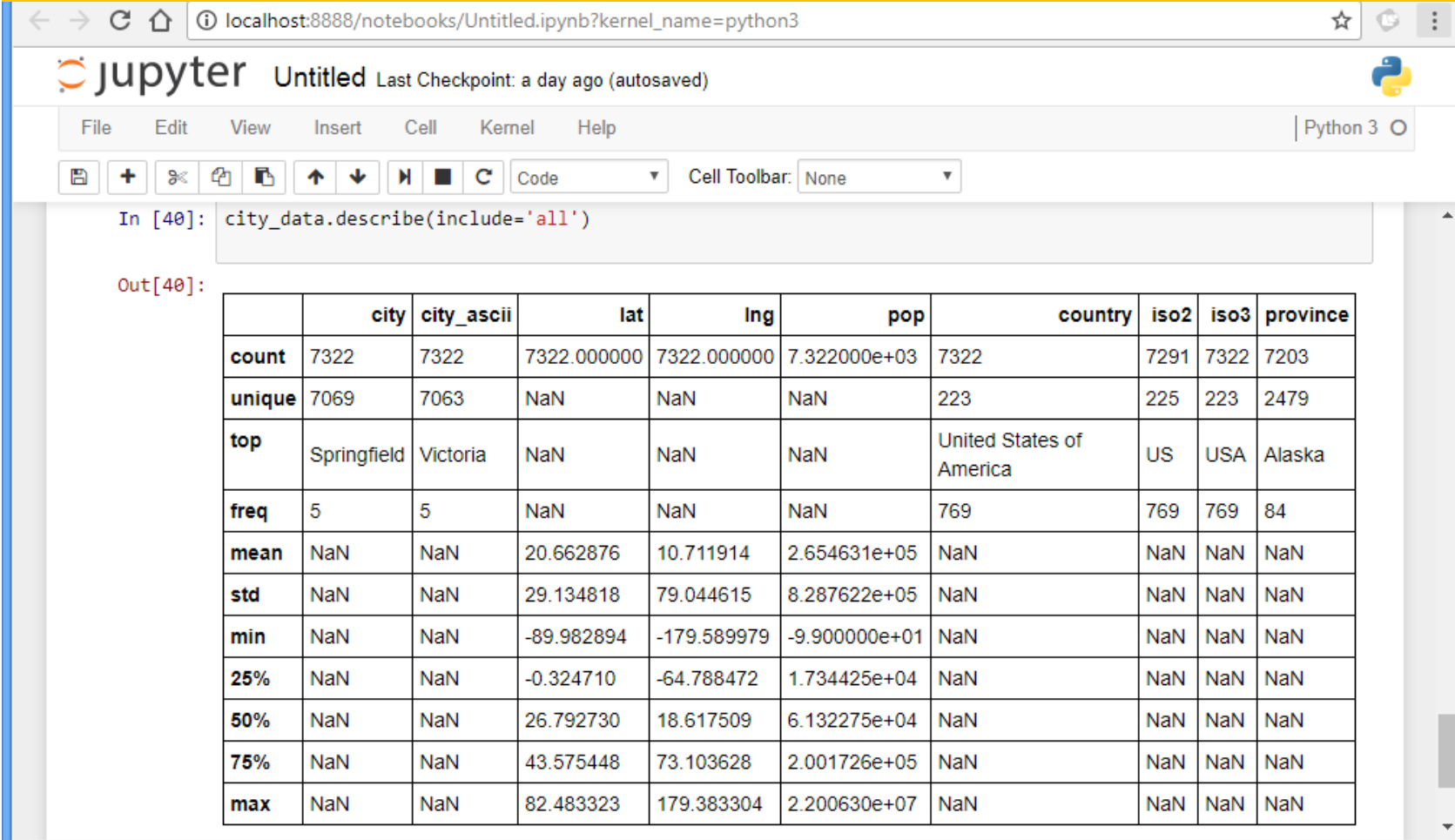
	city	city_ascii	lat	lng	pop	country	iso2
0	Qal eh-ye Now	Qal eh-ye	34.983000	63.133300	2997.0	Afghanistan	AF
1	Chaghcharan	Chaghcharan	34.516701	65.250001	15000.0	Afghanistan	AF
2	Lashkar Gah	Lashkar Gah	31.582998	64.360000	201546.0	Afghanistan	AF

```
In [32]: city_data.tail()
```

Out[32]:

	city	city_ascii	lat	lng	pop	country	iso2
7317	Mutare	Mutare	-18.970019	32.650038	216785.0	Zimbabwe	ZW
7318	Kadoma	Kadoma	-18.330006	29.909947	56400.0	Zimbabwe	ZW
7319	Chitungwiza	Chitungwiza	-18.000001	31.100003	331071.0	Zimbabwe	ZW
7320	Harare	Harare	-17.817790	31.044709	1557406.5	Zimbabwe	ZW
7321	Bulawayo	Bulawayo	-20.169998	28.580002	697096.0	Zimbabwe	ZW

DF.describe() gives basic statistics of the columns
w/o param include='all', function shows only columns with numeric types



The screenshot shows a Jupyter Notebook interface with a browser address bar at localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3. The notebook title is 'Untitled' and it shows 'Last Checkpoint: a day ago (autosaved)'. The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. The toolbar shows various icons for file operations and cell execution. The code cell contains the command `city_data.describe(include='all')`. The output cell displays a table with 10 columns: count, unique, top, freq, mean, std, min, 25%, 50%, 75%, and max. The table provides statistical data for the columns city, city_ascii, lat, lng, pop, country, iso2, iso3, and province.

	city	city_ascii	lat	lng	pop	country	iso2	iso3	province
count	7322	7322	7322.000000	7322.000000	7.322000e+03	7322	7291	7322	7203
unique	7069	7063	NaN	NaN	NaN	223	225	223	2479
top	Springfield	Victoria	NaN	NaN	NaN	United States of America	US	USA	Alaska
freq	5	5	NaN	NaN	NaN	769	769	769	84
mean	NaN	NaN	20.662876	10.711914	2.654631e+05	NaN	NaN	NaN	NaN
std	NaN	NaN	29.134818	79.044615	8.287622e+05	NaN	NaN	NaN	NaN
min	NaN	NaN	-89.982894	-179.589979	-9.900000e+01	NaN	NaN	NaN	NaN
25%	NaN	NaN	-0.324710	-64.788472	1.734425e+04	NaN	NaN	NaN	NaN
50%	NaN	NaN	26.792730	18.617509	6.132275e+04	NaN	NaN	NaN	NaN
75%	NaN	NaN	43.575448	73.103628	2.001726e+05	NaN	NaN	NaN	NaN
max	NaN	NaN	82.483323	179.383304	2.200630e+07	NaN	NaN	NaN	NaN

Keyword argument **parse_dates** is used to specify the columns that contain dates so that pandas can parse them correctly

The screenshot shows a Jupyter Notebook running in a web browser. The browser tabs include 'Untitled', 'panda.describe - Google', and 'pandas.DataFrame.describe'. The address bar shows 'localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3'. The Jupyter interface has a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations and code execution. The notebook content shows two code cells. The first cell (In [41]) runs '%pwd' and returns the current directory. The second cell (In [44]) reads a CSV file 'data/nyc_data.csv' using 'pd.read_csv' with 'parse_dates' set to ['pickup_datetime', 'dropoff_datetime'], and then displays the first five rows of the data using 'data.head()'. The output is a DataFrame with columns: medallion, hack_license, vendor_id, rate_code, and store_id.

```
In [41]: %pwd
Out[41]: 'c:\\TEACHING\\Summer\\17\\minibook-2nd\\chapter2'
```

```
In [44]: data_filename = 'data/nyc_data.csv'
data = pd.read_csv(data_filename, parse_dates=['pickup_datetime', 'dropoff_datetime'])
data.head()
```

```
Out[44]:
```

	medallion	hack_license	vendor_id	rate_code	store_id
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN
1	517C6B330DBB3F055D007B07512628B3	2C19FBEE1A6E05612EFE4C958C14BC7F	VTS	1	NaN
2	ED15611F168E41B33619C83D900FE266	754AEBD7C80DA17BA1D81D89FB6F4D1D	CMT	1	N
3	B33E704CC189E80C9671230C16527BBC	6789C77E1CBDC850C450D72204702976	VTS	1	NaN
4	BD5CC6A22D05EB2D5C8235526A2A4276	5E8F2C93B5220A922699FEB AFC2F7A54	VTS	1	NaN

Untitled panda.describe - Google pandas.DataFrame.descri

localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3

jupyter Untitled Last Checkpoint: a day ago (autosaved)

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [41]: %pwd
```

```
Out[41]: 'c:\\TEACHING\\Summer\\17\\minibook-2nd\\chapter2'
```

```
In [44]: data_filename = 'data/nyc_data.csv'
data = pd.read_csv(data_filename, parse_dates=['pickup_datetime', 'dropoff_datetime'])
data.head()
```

```
Out[44]:
```

enger_count	trip_time_in_secs	trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude
300	0.61	-73.955925	40.781887	-73.963181	40.777832	
960	3.28	-74.005501	40.745735	-73.964943	40.755722	
386	1.50	-73.969955	40.799770	-73.954567	40.787392	
0	0.00	-73.991432	40.755081	-73.991417	40.755085	
360	1.31	-73.966225	40.773716	-73.955399	40.782597	



```
In [45]: data.columns
```

```
Out[45]: Index(['medallion', 'hack_license', 'vendor_id', 'rate_code',  
              'store_and_fwd_flag', 'pickup_datetime', 'dropoff_datetime',  
              'passenger_count', 'trip_time_in_secs', 'trip_distance',  
              'pickup_longitude', 'pickup_latitude', 'dropoff_longitude',  
              'dropoff_latitude'],  
             dtype='object')
```

```
In [50]: p_lat = data.pickup_latitude  
p_lat.head()
```

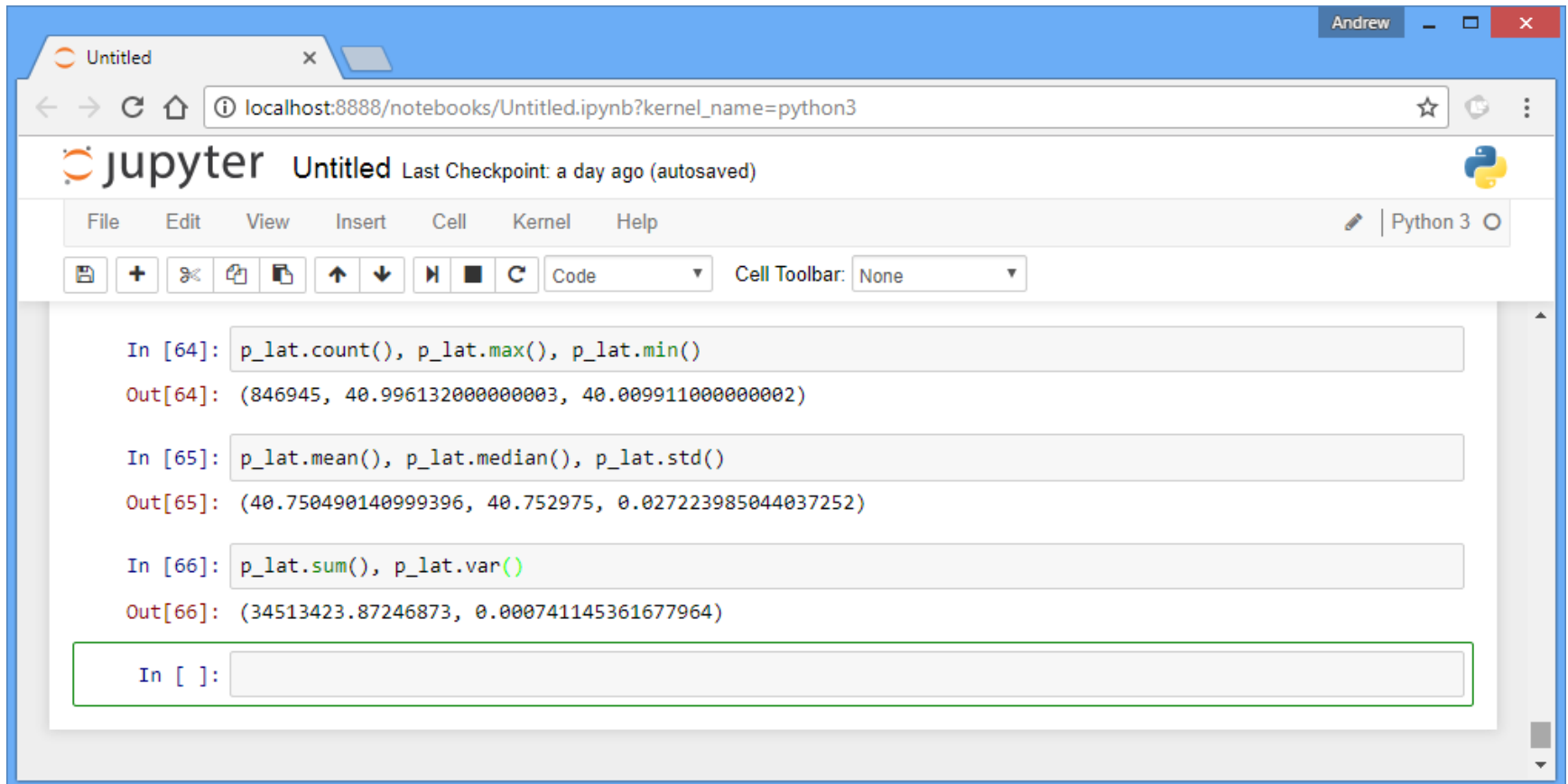
```
Out[50]: 0    40.781887  
        1    40.745735  
        2    40.799770  
        3    40.755081  
        4    40.773716  
        Name: pickup_latitude, dtype: float64
```

```
In [52]: p_long = data['pickup_longitude']  
p_long.tail()
```

```
Out[52]: 846940    -73.992058  
        846941    -73.994949  
        846942    -73.993492  
        846943    -73.978477  
        846944    -73.987206  
        Name: pickup_longitude, dtype: float64
```

Descriptive Statistics with Pandas

- The following functions do as their names indicate



The screenshot shows a Jupyter Notebook window titled 'Untitled' with a browser address bar at 'localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3'. The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations and cell execution. The notebook contains three code cells, each followed by its output:

```
In [64]: p_lat.count(), p_lat.max(), p_lat.min()
Out[64]: (846945, 40.996132000000003, 40.009911000000002)

In [65]: p_lat.mean(), p_lat.median(), p_lat.std()
Out[65]: (40.750490140999396, 40.752975, 0.027223985044037252)

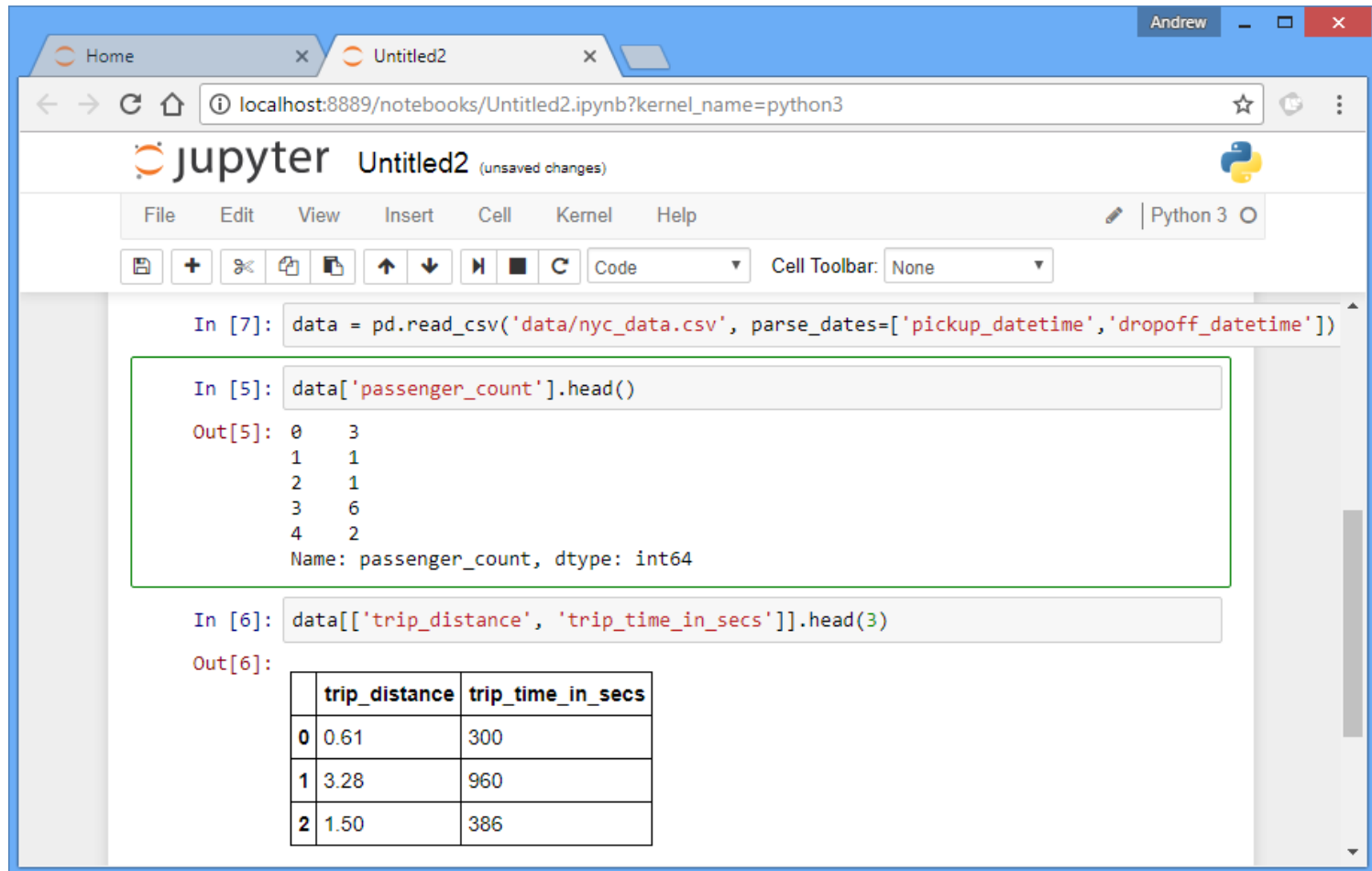
In [66]: p_lat.sum(), p_lat.var()
Out[66]: (34513423.87246873, 0.000741145361677964)
```

Below the last output, there is an empty input cell labeled 'In []:'.

Selecting Data

- Pandas allows us to select rows, columns, and combinations of rows and columns, ... etc.

Selecting Columns



The screenshot shows a Jupyter Notebook window titled "Untitled2" with a browser address bar at `localhost:8889/notebooks/Untitled2.ipynb?kernel_name=python3`. The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for saving, adding cells, and running code. The code area shows three input cells:

```
In [7]: data = pd.read_csv('data/nyc_data.csv', parse_dates=['pickup_datetime', 'dropoff_datetime'])
```

```
In [5]: data['passenger_count'].head()
```

The output for In [5] is:

```
Out[5]: 0    3
        1    1
        2    1
        3    6
        4    2
        Name: passenger_count, dtype: int64
```

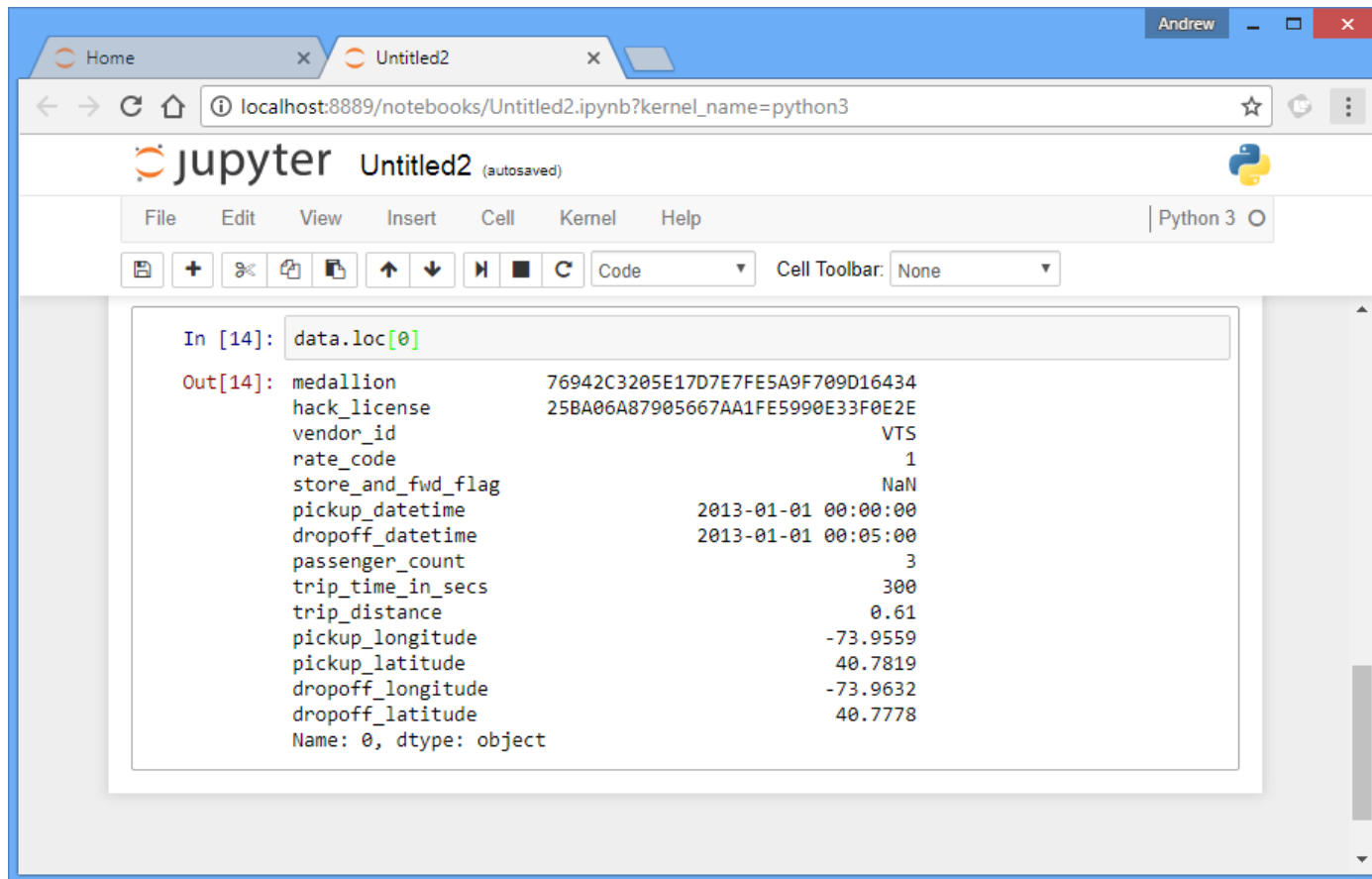
```
In [6]: data[['trip_distance', 'trip_time_in_secs']].head(3)
```

The output for In [6] is a table:

	trip_distance	trip_time_in_secs
0	0.61	300
1	3.28	960
2	1.50	386

Selecting Rows

- Rows can be retrieved by position or name
 - Use loc attribute to select row(s) from their labels
 - Use iloc attribute to select row(s) from their positions
- The following example show how to select the first row:



The screenshot shows a Jupyter Notebook window titled 'Untitled2' with a Python 3 kernel. The code cell contains the command `data.loc[0]`. The output displays the first row of a dataset with various attributes and their values.

```
In [14]: data.loc[0]
```

```
Out[14]: medallion          76942C3205E17D7E7FE5A9F709D16434  
hack_license      25BA06A87905667AA1FE5990E33F0E2E  
vendor_id          VTS  
rate_code          1  
store_and_fwd_flag      NaN  
pickup_datetime      2013-01-01 00:00:00  
dropoff_datetime      2013-01-01 00:05:00  
passenger_count        3  
trip_time_in_secs      300  
trip_distance         0.61  
pickup_longitude      -73.9559  
pickup_latitude        40.7819  
dropoff_longitude      -73.9632  
dropoff_latitude        40.7778  
Name: 0, dtype: object
```

Remark: The **row selection** syntax `data[:2]` is provided as a convenience.
Passing a single element or a list to the `[]` operator selects columns

In [27]: `data[0:2]`

Out[27]:

	medallion	hack_license	vendo
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS
1	517C6B330DBB3F055D007B07512628B3	2C19FBEE1A6E05612EFE4C958C14BC7F	VTS

In [29]: `data['passenger_count'].head(3)`

Out[29]:

0	3
1	1
2	1

Name: passenger_count, dtype: int64

In [30]: `data.iloc[0:2]`

Out[30]:

	medallion	hack_license	vendo
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS
1	517C6B330DBB3F055D007B07512628B3	2C19FBEE1A6E05612EFE4C958C14BC7F	VTS

Using `DF.loc[rows_labels_names]` or `DF.iloc[rows_indexes]` is the proper way to select rows.
The slice notation, `DF[:]`, is provided for convenience.

```
In [31]: data.loc[0:2]
```

```
Out[31]:
```

	medallion	hack_license	vendor
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VT
1	517C6B330DBB3F055D007B07512628B3	2C19FBEE1A6E05612EFE4C958C14BC7F	VT
2	ED15611F168E41B33619C83D900FE266	754AEBD7C80DA17BA1D81D89FB6F4D1D	CMT

```
In [39]: data[['trip_distance', 'trip_time_in_secs']].head(3)
```

```
Out[39]:
```

	trip_distance	trip_time_in_secs
0	0.61	300
1	3.28	960
2	1.50	386

Here, you needed to give the column labels, because writing `DF[]` is the notation to select columns.

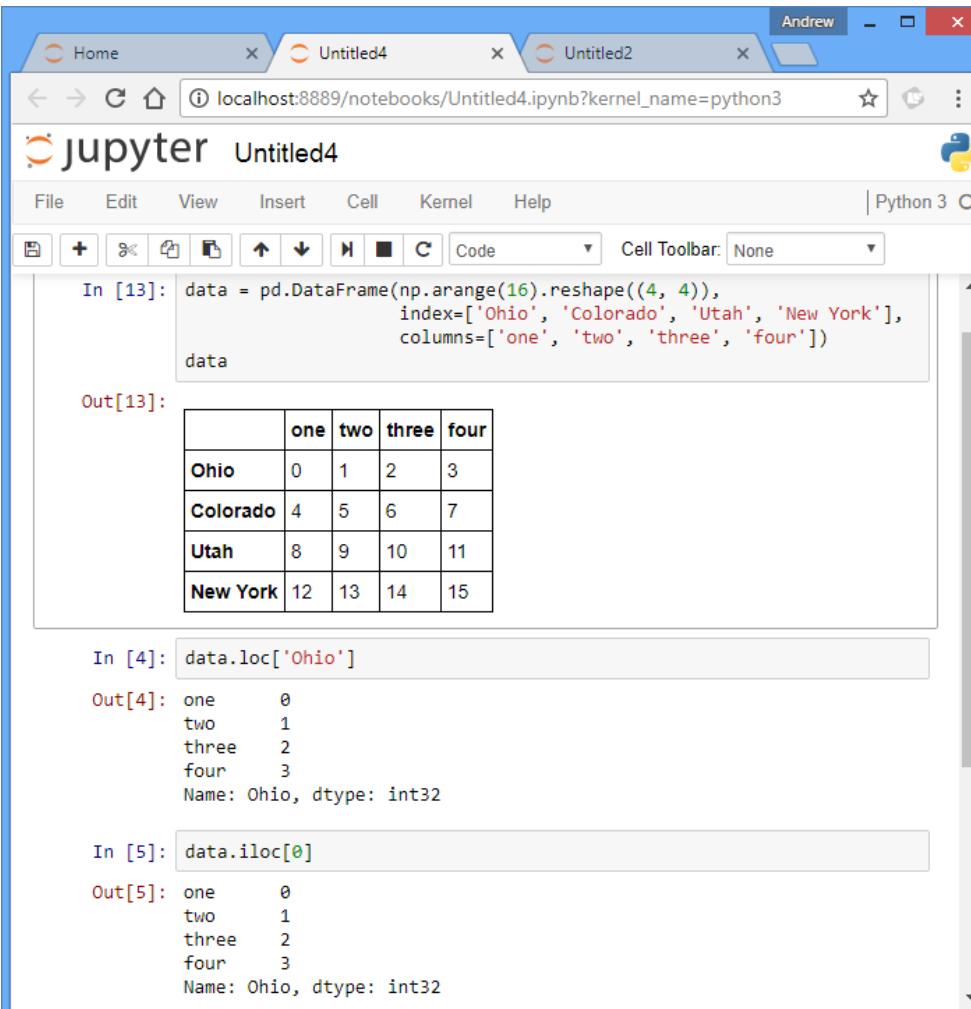
```
In [40]: data[[1,2]].head(3)
```

```
KeyError
```

```
Traceback (most recent call last)
```

```
(In Python input 40: b80a7080012f) in <module>():
```

Aside: More about loc and iloc



Home x Untitled4 x Untitled2 x Andrew

localhost:8889/notebooks/Untitled4.ipynb?kernel_name=python3

jupyter Untitled4

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [13]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                             index=['Ohio', 'Colorado', 'Utah', 'New York'],
                             columns=['one', 'two', 'three', 'four'])
```

data

```
Out[13]:
```

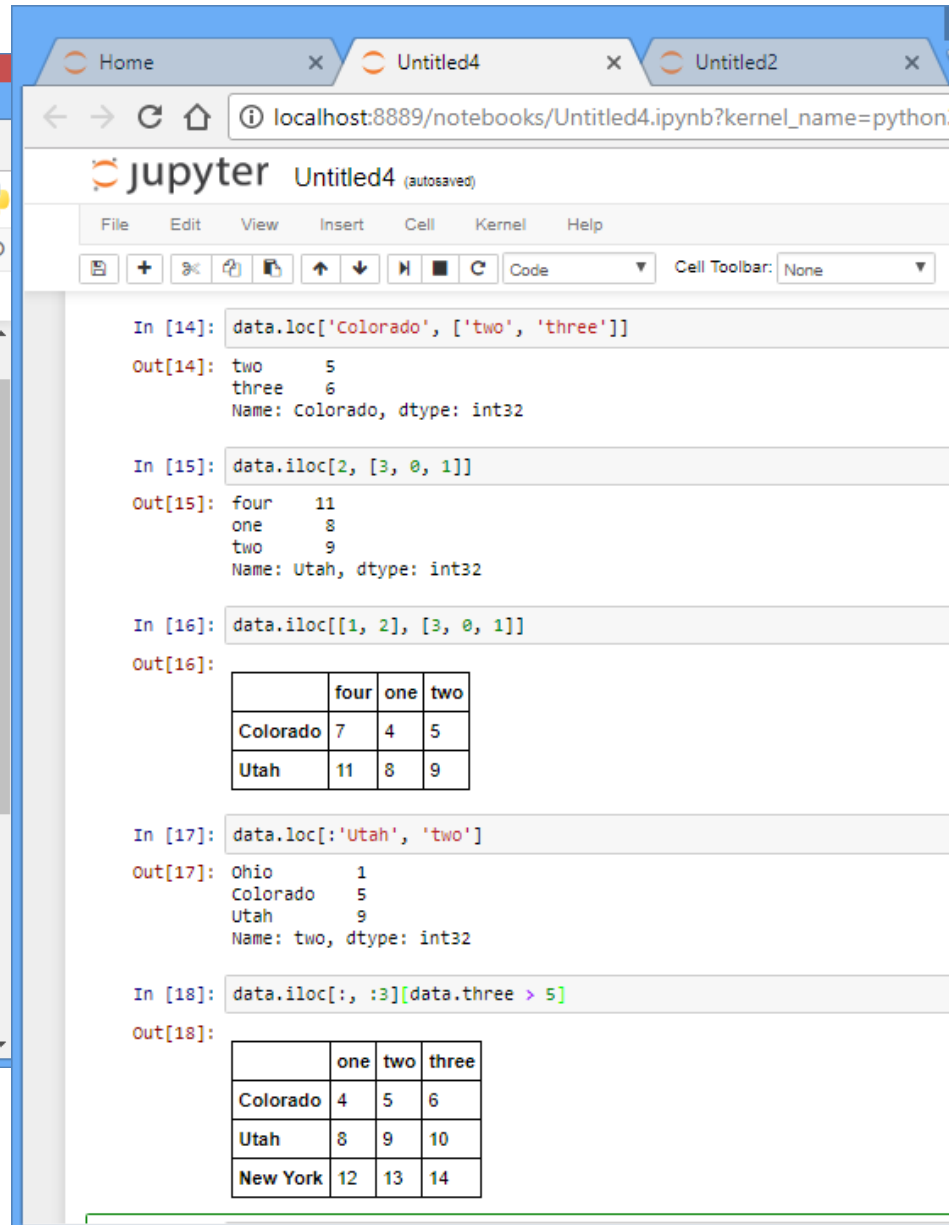
	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [4]: data.loc['Ohio']
```

```
Out[4]: one      0
        two      1
        three    2
        four     3
        Name: Ohio, dtype: int32
```

```
In [5]: data.iloc[0]
```

```
Out[5]: one      0
        two      1
        three    2
        four     3
        Name: Ohio, dtype: int32
```



Home x Untitled4 x Untitled2

localhost:8889/notebooks/Untitled4.ipynb?kernel_name=python3

jupyter Untitled4 (autosaved)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

```
In [14]: data.loc['Colorado', ['two', 'three']]
```

```
Out[14]: two      5
        three    6
        Name: Colorado, dtype: int32
```

```
In [15]: data.iloc[2, [3, 0, 1]]
```

```
Out[15]: four     11
        one       8
        two       9
        Name: Utah, dtype: int32
```

```
In [16]: data.iloc[[1, 2], [3, 0, 1]]
```

```
Out[16]:
```

	four	one	two
Colorado	7	4	5
Utah	11	8	9

```
In [17]: data.loc[:, 'Utah', 'two']
```

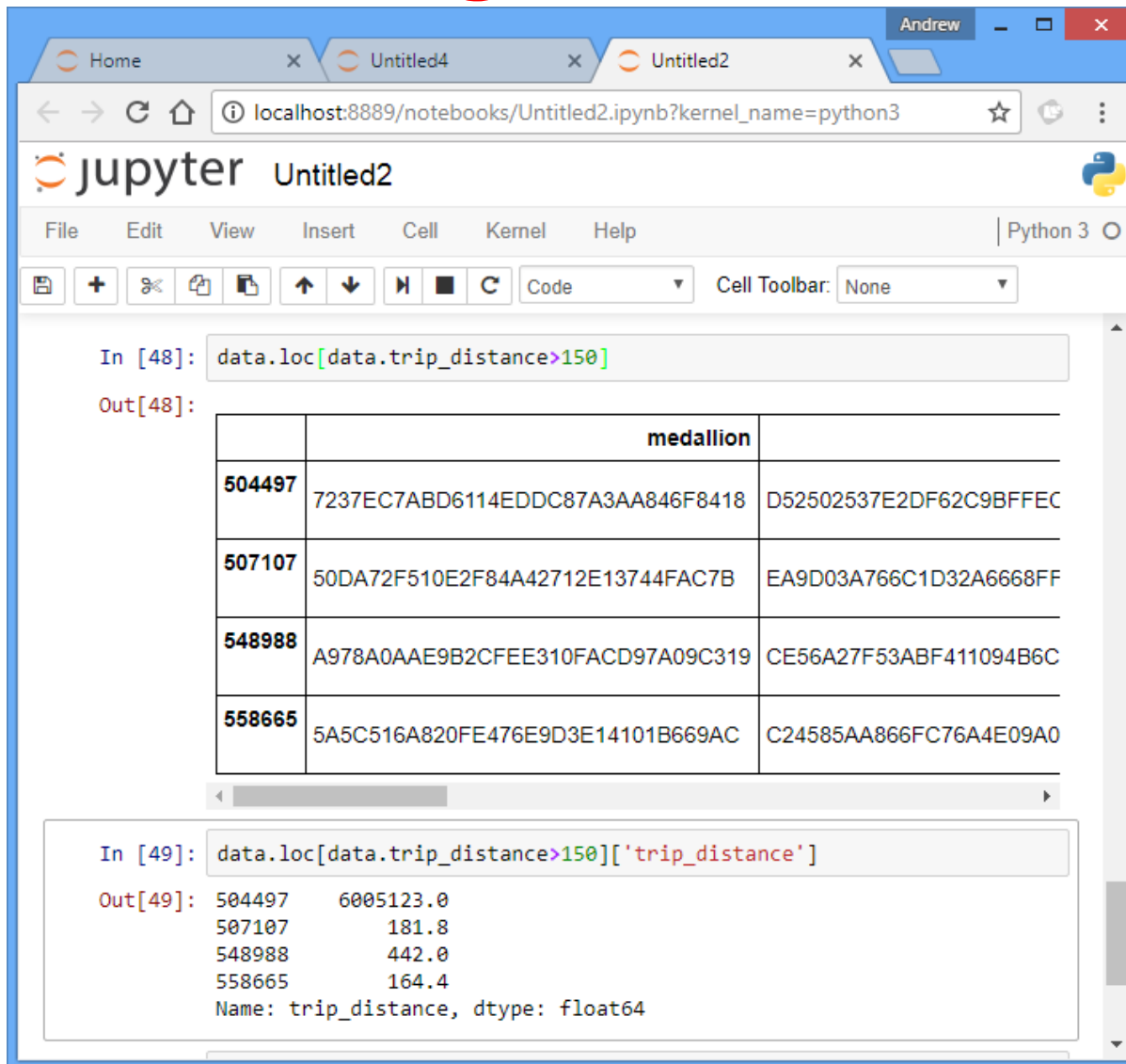
```
Out[17]: Ohio      1
        Colorado   5
        Utah       9
        Name: two, dtype: int32
```

```
In [18]: data.iloc[:, :3][data.three > 5]
```

```
Out[18]:
```

	one	two	three
Colorado	4	5	6
Utah	8	9	10
New York	12	13	14

Filtering with Boolean Indexing



The screenshot shows a Jupyter Notebook interface with a browser window at localhost:8889. The notebook has two tabs: 'Untitled4' and 'Untitled2'. The 'Untitled2' tab is active, showing a code cell with the following input and output:

```
In [48]: data.loc[data.trip_distance>150]
```

The output is a DataFrame with three columns: 'medallion', 'trip_distance', and 'fare_amount'. The first column is highlighted in yellow. The data is as follows:

	medallion	trip_distance	fare_amount
504497	7237EC7ABD6114EDDC87A3AA846F8418	D52502537E2DF62C9BFFEC	
507107	50DA72F510E2F84A42712E13744FAC7B	EA9D03A766C1D32A6668FF	
548988	A978A0AAE9B2CFEE310FACD97A09C319	CE56A27F53ABF411094B6C	
558665	5A5C516A820FE476E9D3E14101B669AC	C24585AA866FC76A4E09A0	

Below this, another code cell is shown with the following input and output:

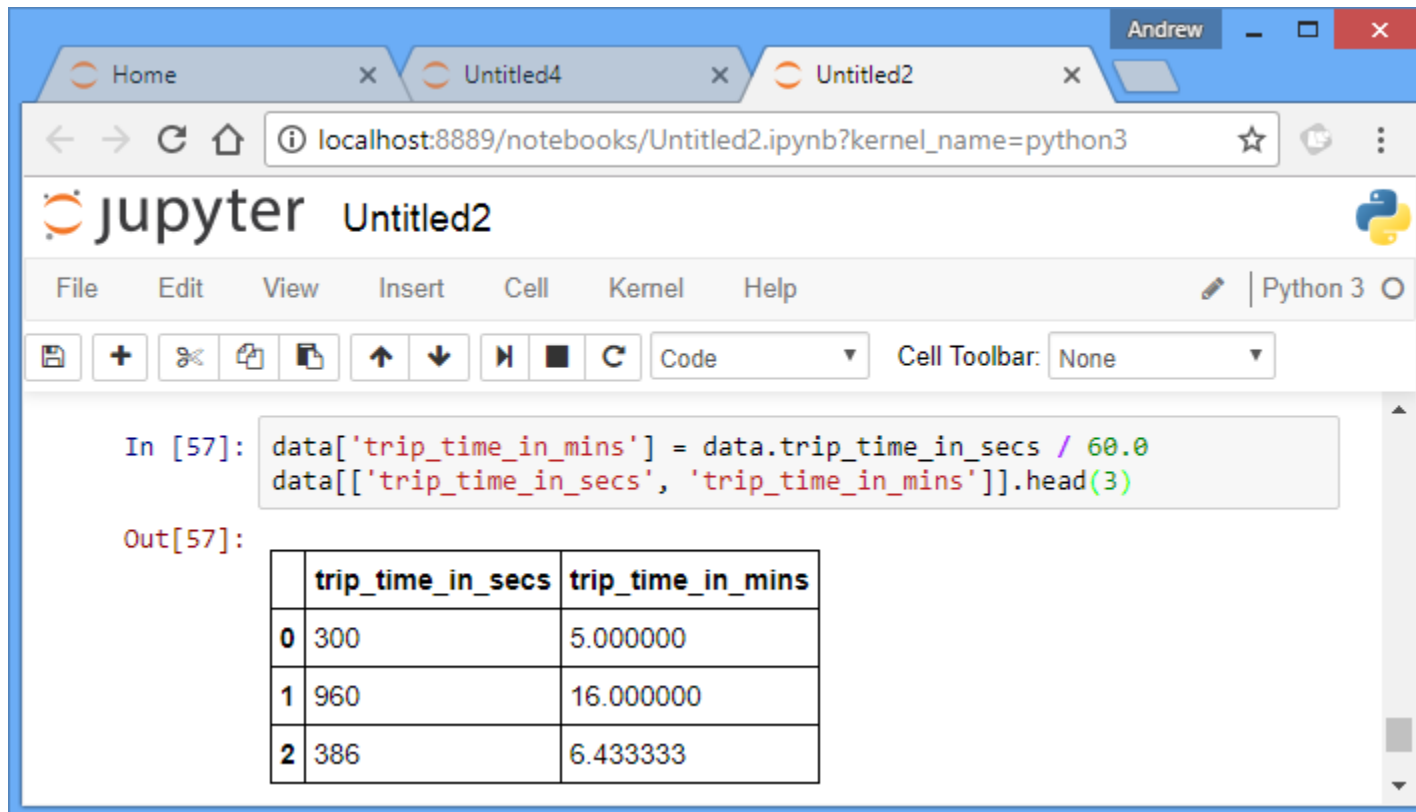
```
In [49]: data.loc[data.trip_distance>150]['trip_distance']
```

The output is a Series of trip distances for the filtered rows:

```
Out[49]: 504497    6005123.0
507107      181.8
548988      442.0
558665      164.4
Name: trip_distance, dtype: float64
```

Operating on Data

- Vector arithmetic can be used with DataFrames and Series objects
- Example: adding a new column to the DF with the trip duration in minutes



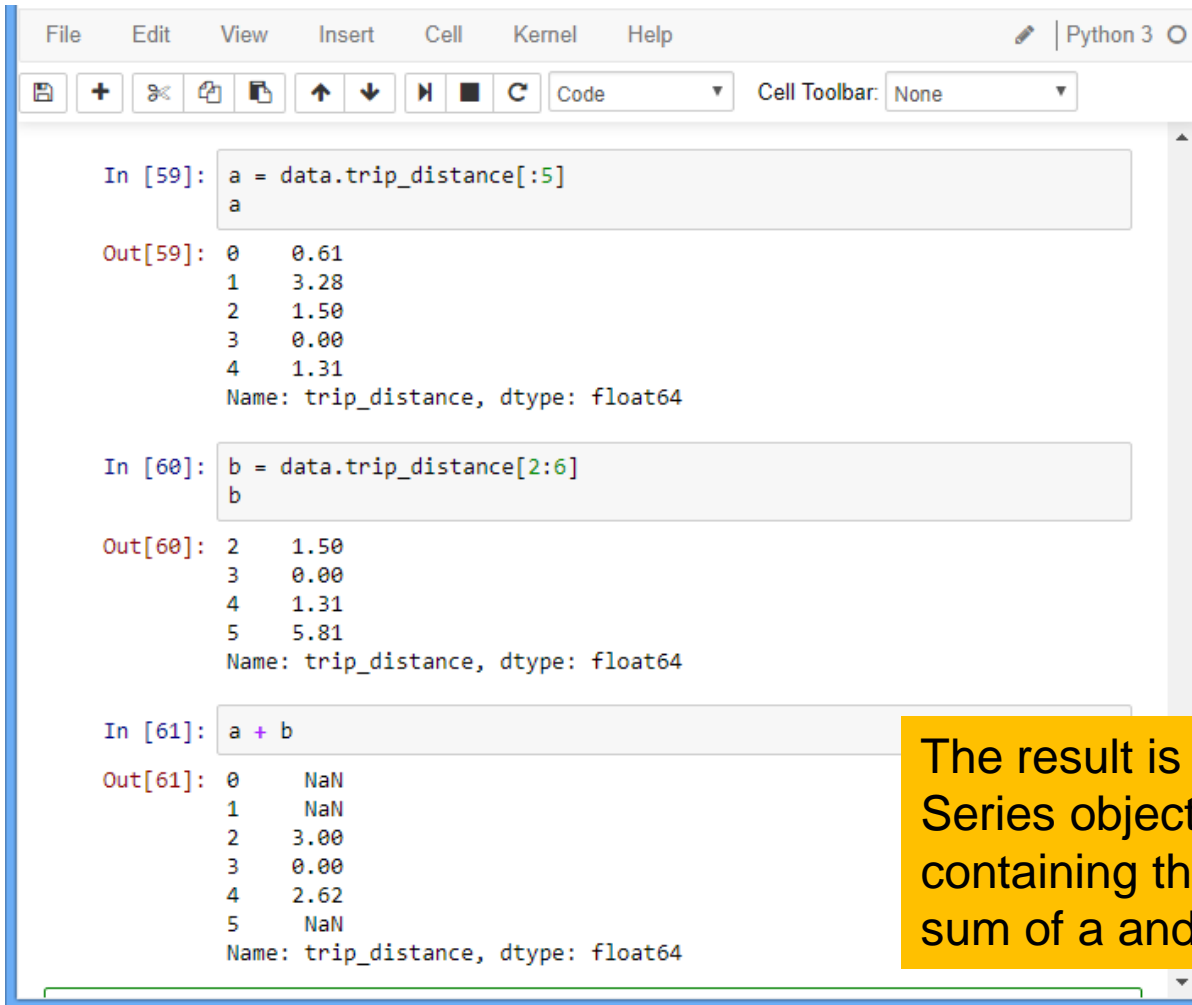
The screenshot shows a Jupyter Notebook interface with a browser window at localhost:8889. The notebook has a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar. The active cell is a code cell with the following Python code:

```
In [57]: data['trip_time_in_mins'] = data.trip_time_in_secs / 60.0  
data[['trip_time_in_secs', 'trip_time_in_mins']].head(3)
```

The output of the cell is a DataFrame with 3 rows and 2 columns:

	trip_time_in_secs	trip_time_in_mins
0	300	5.000000
1	960	16.000000
2	386	6.433333

More on Vector Operations



The screenshot shows a Jupyter Notebook interface with three code cells. The first cell defines a Series 'a' from index 0 to 4. The second cell defines a Series 'b' from index 2 to 5. The third cell performs an addition 'a + b', which results in a new Series with NaN values at indices 0, 1, and 5 due to misalignment.

```
File Edit View Insert Cell Kernel Help Python 3
+ < > < > < > < > < > < >
In [59]: a = data.trip_distance[:5]
a
Out[59]: 0    0.61
         1    3.28
         2    1.50
         3    0.00
         4    1.31
         Name: trip_distance, dtype: float64

In [60]: b = data.trip_distance[2:6]
b
Out[60]: 2    1.50
         3    0.00
         4    1.31
         5    5.81
         Name: trip_distance, dtype: float64

In [61]: a + b
Out[61]: 0    NaN
         1    NaN
         2    3.00
         3    0.00
         4    2.62
         5    NaN
         Name: trip_distance, dtype: float64
```

The result is a new Series object containing the aligned sum of a and b

More on Vector Operations

```
In [135]: a = np.arange(12).reshape(3, 4)
df = pd.DataFrame(a, columns=list('ABCD'))
df
```

Out[135]:

	A	B	C	D
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
In [138]: b = df.iloc[0]
b
```

Out[138]:

A	0
B	1
C	2
D	3

Name: 0, dtype: int32


```
In [139]: df - b
```

Out[139]:

	A	B	C	D
0	0	0	0	0
1	4	4	4	4
2	8	8	8	8

```
[[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]]
```

```
[[0, 1, 2, 3],
 [0, 1, 2, 3],
 [0, 1, 2, 3]]
```



Handling Missing Data

- In pandas, missing data is represented by NaN (Not a Number) or None
- Pandas provides several Series and DataFrame methods to deal with missing data, including:
 - `isnull()` indicates whether values are null or not
 - `notnull()` the opposite of `isnull()`
 - `dropna()` removes missing data
 - `fillna(some_default_value)` replaces missing data with a default value

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_name=python3

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [28]: import numpy as np
string_data = pd.Series(['apples', 'oranges', np.nan, None])
string_data
```

```
Out[28]: 0    apples
1    oranges
2         NaN
3         None
dtype: object
```

```
In [29]: string_data.isnull()
```

```
Out[29]: 0    False
1    False
2     True
3     True
dtype: bool
```

```
In [30]: string_data.fillna('blank')
```

```
Out[30]: 0    apples
1    oranges
2     blank
3     blank
dtype: object
```

Working with a Series

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_name=pytho...

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [33]: string_data.dropna()

Out[33]: 0    apples
         1    oranges
         dtype: object

In [34]: string_data

Out[34]: 0    apples
         1    oranges
         2     NaN
         3     None
         dtype: object

In [35]: cleaned = string_data.dropna()
         cleaned

Out[35]: 0    apples
         1    oranges
         dtype: object

In [36]: string_data

Out[36]: 0    apples
         1    oranges
         2     NaN
         3     None
         dtype: object
```

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_name=p...

jupyter Untitled5 (autosaved)

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [38]: from numpy import nan as NA
df = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                  [NA, NA, NA], [NA, 6.5, 3.]])
df
```

Out[38]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
In [42]: df.isnull()
```

Out[42]:

	0	1	2
0	False	False	False
1	False	True	True
2	True	True	True
3	True	False	False

```
In [43]: df.fillna(0)
```

Out[43]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	0.0	0.0
2	0.0	0.0	0.0
3	0.0	6.5	3.0

Working with a
DataFrame

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_name=p...

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

In [45]: `df`

Out[45]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

In [46]: `cleaned = df.dropna()
cleaned`

Out[46]:

	0	1	2
0	1.0	6.5	3.0

In [47]: `df.dropna(how='all')`

Out[47]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

Passing how='all'
will only drop rows
that are all NA

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_name=p...

jupyter Untitled5 (unsaved changes)

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

In [64]: `df[3] = NA`
`df`

Out[64]:

	0	1	2	3
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

In [65]: `df.dropna(axis=1, how='all')`

Out[65]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

In [66]: `df.dropna(thresh=2)`

Out[66]:

	0	1	2	3
0	1.0	6.5	3.0	NaN
3	NaN	6.5	3.0	NaN

In [67]: `df.dropna(axis=1, thresh=2)`

Out[67]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

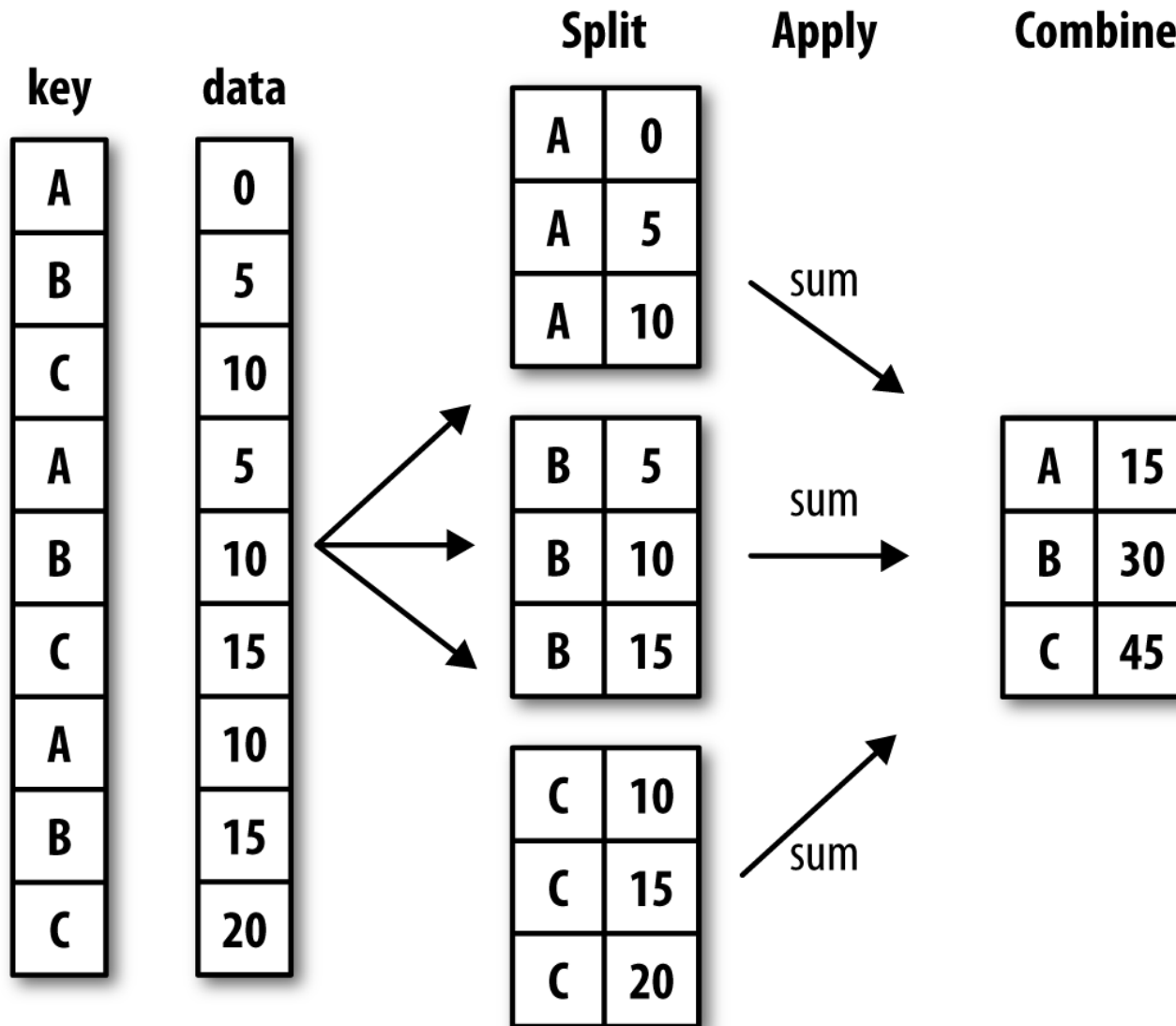
To drop columns, use the argument **axis=1** or **axis = 'columns'**

The **thresh** argument is used to keep rows/columns **containing** a certain number of observations.

The groupby Operation

- Similar to the *group by* operation in SQL where:
 - data is split into groups that share common attributes
 - a function is applied to each group
 - function results are combined into a result object

Illustration of a group aggregation



Grouping Key

- Data is split into groups based on the values of a key or several keys
- Each grouping key can be
 - a list, array, dict, or series object
 - a column name in a DataFrame
- The keys do not have to be all of the same type

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_n...

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [74]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                             'key2' : ['one', 'two', 'one', 'two', 'one'],
                             'data1' : np.random.randn(5),
                             'data2' : np.random.randn(5)})
df
```

Out[74]:

	data1	data2	key1	key2
0	0.705617	-0.124217	a	one
1	-0.447389	-1.989662	a	two
2	1.404804	-0.045252	b	one
3	-1.561116	1.593082	b	two
4	-0.823505	-1.901100	a	one

```
In [75]: grouped = df['data1'].groupby(df['key1'])
grouped
```

Out[75]: <pandas.core.groupby.SeriesGroupBy object at 0x00000099A5DBBB70>

```
In [76]: grouped.mean()
```

Out[76]: key1
a -0.188426
b -0.078156
Name: data1, dtype: float64

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_n...

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

In [81]: df

Out[81]:

	data1	data2	key1	key2
0	0.705617	-0.124217	a	one
1	-0.447389	-1.989662	a	two
2	1.404804	-0.045252	b	one
3	-1.561116	1.593082	b	two
4	-0.823505	-1.901100	a	one

In [82]: grouped2 = df['data1'].groupby([df['key1'], df['key2']]).mean()
grouped2

Out[82]:

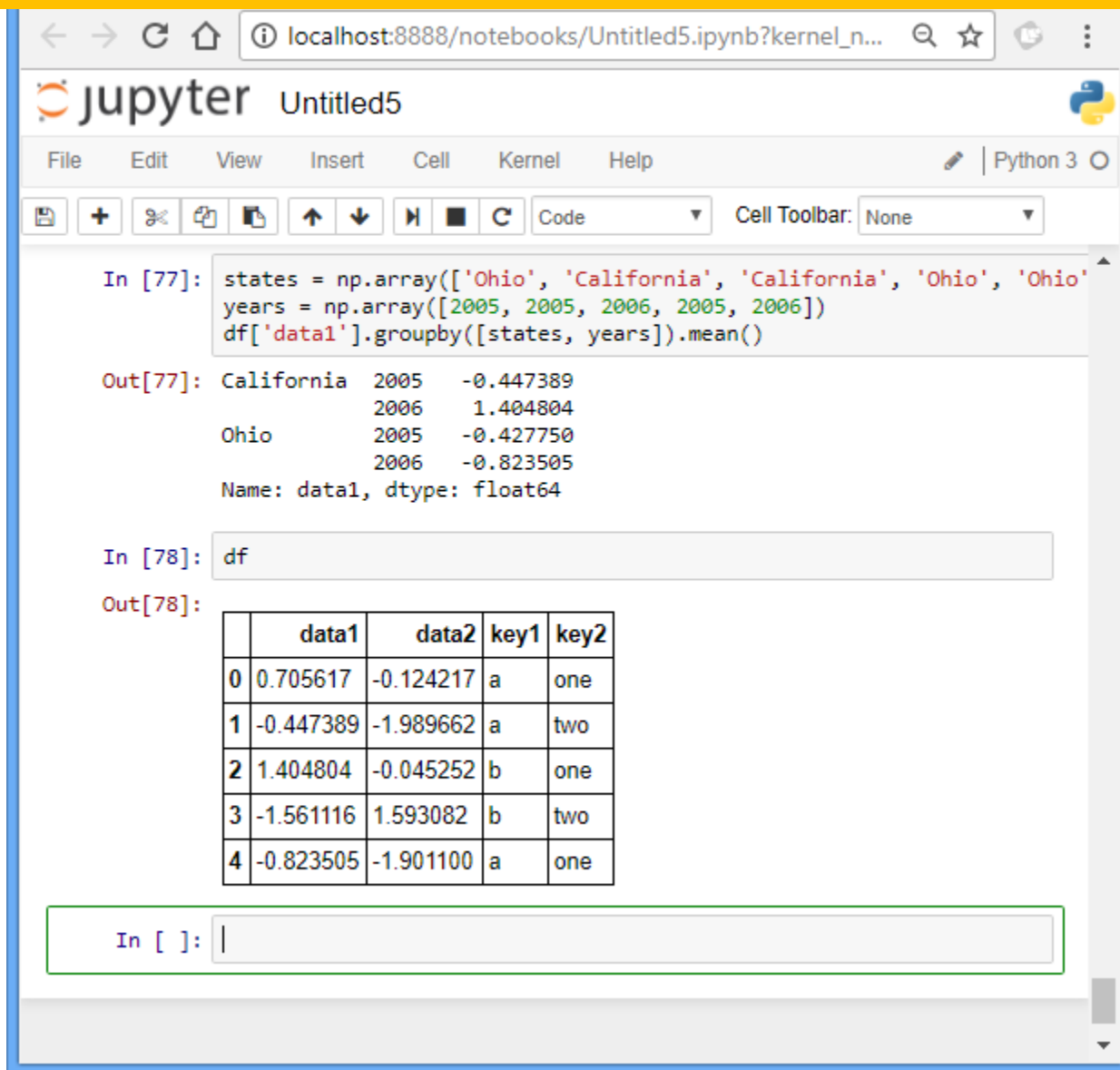
key1	key2	data1
a	one	-0.058944
a	two	-0.447389
b	one	1.404804
b	two	-1.561116

Name: data1, dtype: float64

In [84]: type(grouped2)

Out[84]: pandas.core.series.Series

Using arrays as the grouping keys



The image shows a Jupyter Notebook interface with a browser address bar at the top displaying `localhost:8888/notebooks/Untitled5.ipynb?kernel_n...`. The notebook title is "Untitled5". The menu bar includes "File", "Edit", "View", "Insert", "Cell", "Kernel", and "Help". The toolbar contains icons for saving, adding cells, undo, redo, and running code. The current cell is a code cell with the following Python code:

```
In [77]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
years = np.array([2005, 2005, 2006, 2005, 2006])
df['data1'].groupby([states, years]).mean()
```

The output of this code is displayed below the cell:

```
Out[77]: California 2005 -0.447389
          2006  1.404804
          Ohio    2005 -0.427750
          2006 -0.823505
          Name: data1, dtype: float64
```

Below this, a second code cell is shown with the input `df`:

```
In [78]: df
```

The output of this cell is a table:

```
Out[78]:
```

	data1	data2	key1	key2
0	0.705617	-0.124217	a	one
1	-0.447389	-1.989662	a	two
2	1.404804	-0.045252	b	one
3	-1.561116	1.593082	b	two
4	-0.823505	-1.901100	a	one

At the bottom, there is an empty code cell with the prompt `In []:`.

Untitled5

localhost:8888/notebooks/Untitled...

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Cell Toolbar: None

```
In [86]: df
```

```
Out[86]:
```

	data1	data2	key1	key2
0	0.705617	-0.124217	a	one
1	-0.447389	-1.989662	a	two
2	1.404804	-0.045252	b	one
3	-1.561116	1.593082	b	two
4	-0.823505	-1.901100	a	one

```
In [87]: df.groupby('key1').mean()
```

```
Out[87]:
```

	data1	data2
key1		
a	-0.188426	-1.338326
b	-0.078156	0.773915

Notice: there is no key2 column in the result as df['key2'] is not numeric data

```
In [88]: df.groupby(['key1', 'key2']).mean()
```

```
Out[88]:
```

		data1	data2
key1	key2		
a	one	-0.058944	-1.012658
	two	-0.447389	-1.989662
b	one	1.404804	-0.045252
	two	-1.561116	1.593082

Untitled5

localhost:8888/notebooks/Untitle...

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Cell Toolbar: None

```
In [90]: df.groupby('key1').size()
```

```
Out[90]: key1
a      3
b      2
dtype: int64
```

```
In [91]: df.groupby(['key1', 'key2']).size()
```

```
Out[91]: key1 key2
a      one    2
         two    1
b      one    1
         two    1
dtype: int64
```

The GroupBy method **size()** returns group sizes

Joins

- `pd.merge`
- The idea is similar to joins in databases. That is to join data from more than one table
- Consider the NY city taxi dataset
- The DFs `data` and `fare` have same number of rows (one row for each ride)
- We want to get the average tip from the `fare` DF
- We will then join the average tip into the `data` DF

Andrew

Untitled5

localhost:8888/notebooks/Untitled5.ipynb?kernel_name=python3

jupyter Untitled5

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [93]: data = pd.read_csv('data/nyc_data.csv', parse_dates=['pickup_datetime',
                                                         'dropoff_datetime'])
fare = pd.read_csv('data/nyc_fare.csv', parse_dates=['pickup_datetime'])

In [94]: tip = fare[['medallion', 'tip_amount']] \
          .loc[fare.tip_amount>0].groupby('medallion').mean()
print(len(tip))
tip.head(3)
```

13407

Out[94]:

	tip_amount
medallion	
00005007A9F30E289E760362F69E4EAD	1.815854
000318C2E3E6381580E5C99910A60668	2.857222
000351EDC735C079246435340A54C7C1	2.099111


```
In [13]: data_merged = pd.merge(data, tip, left_on='medallion', right_index=True)
data_merged.head(3)
```

Out[13]:

	medallion	hack_license	vendor_id	rate_code	store
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN
4761	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN
22991	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN

```
In [14]: data_merged[['medallion', 'tip_amount']].head(3)
```

Out[14]:

	medallion	tip_amount
0	76942C3205E17D7E7FE5A9F709D16434	3.180417
4761	76942C3205E17D7E7FE5A9F709D16434	3.180417
22991	76942C3205E17D7E7FE5A9F709D16434	3.180417

More about the merge Operation

- Idea is similar to the join operation in SQL where pandas will combine two DFs where values under common columns are the same

Andrew

Untitled8

localhost:8888/notebooks/Untitled8.ipynb?kernel_name=pyth...

jupyter Untitled8

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [71]: df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                             'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
print(df1)
```

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

```
In [73]: print(df2)
```

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

```
In [74]: pd.merge(df1, df2)
```

Out[74]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Andrew

Untitled8

localhost:8888/notebooks/Untitle...

jupyter Untitled8

File Edit View Insert Cell Kernel Help Python 3

Code Cell Toolbar: None

```
In [75]: print(pd.merge(df1, df2))
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Untitled5 - Jupyter Notebook

localhost:8888/notebooks/chapter2/Untitled5.ipynb

jupyter Untitled5 Last Checkpoint: 12/29/2017 (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

```
In [125]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df2 = pd.DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)})
print(df1, "\n"); print(df2, "\n");
print(pd.merge(df1, df2))
```

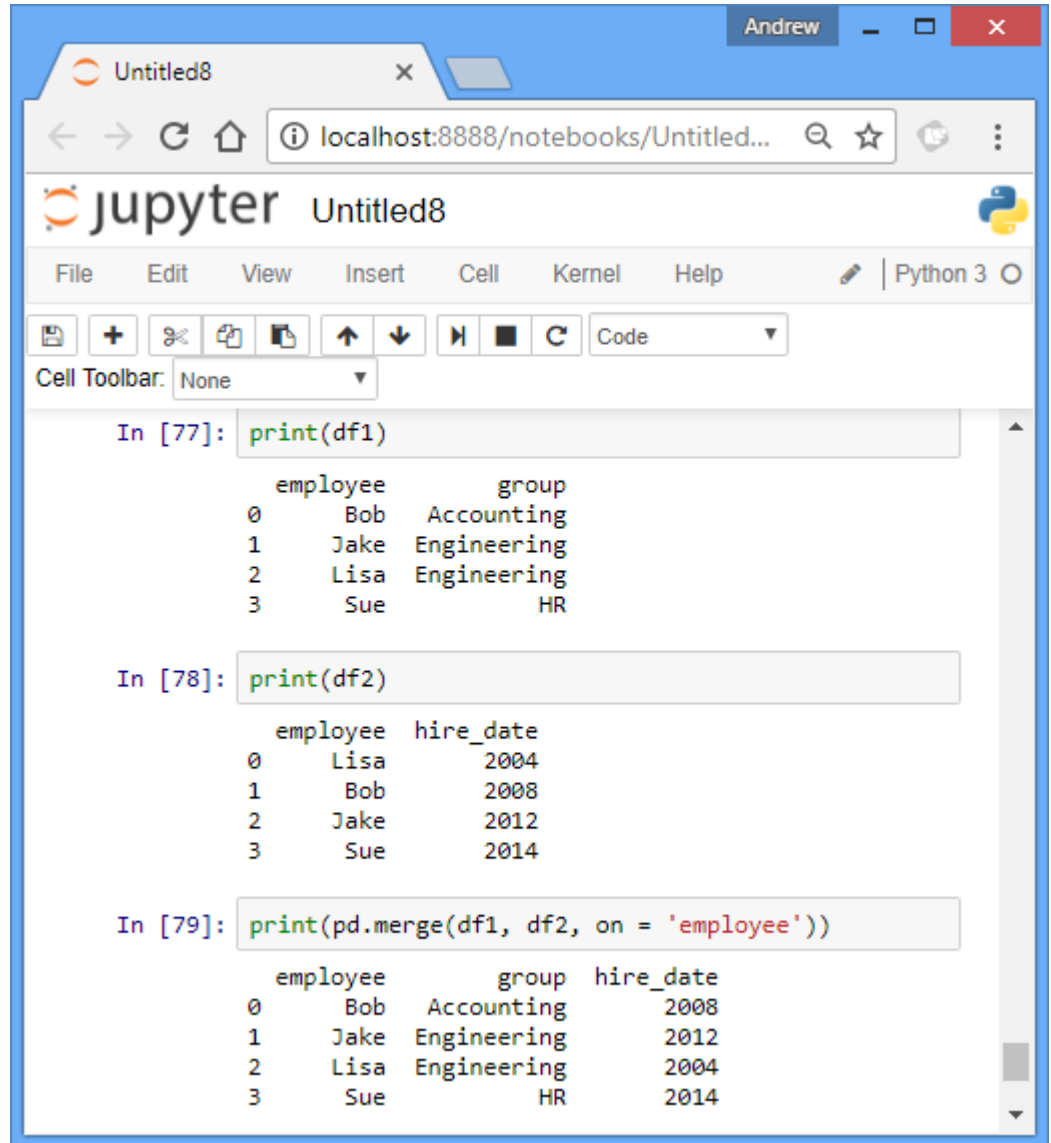
	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

	key	data2
0	a	0
1	b	1
2	d	2

	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

The on Keyword Argument

- The attribute **on** allows us to explicitly specify the name of the key column(s)
- It takes either a column name or a list of column names



The screenshot shows a Jupyter Notebook interface with a browser window at localhost:8888. The notebook has three code cells. The first cell prints df1, which is a DataFrame with columns 'employee' and 'group'. The second cell prints df2, which is a DataFrame with columns 'employee' and 'hire_date'. The third cell prints the result of pd.merge(df1, df2, on='employee'), which is a DataFrame with columns 'employee', 'group', and 'hire_date'.

```
In [77]: print(df1)
```

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

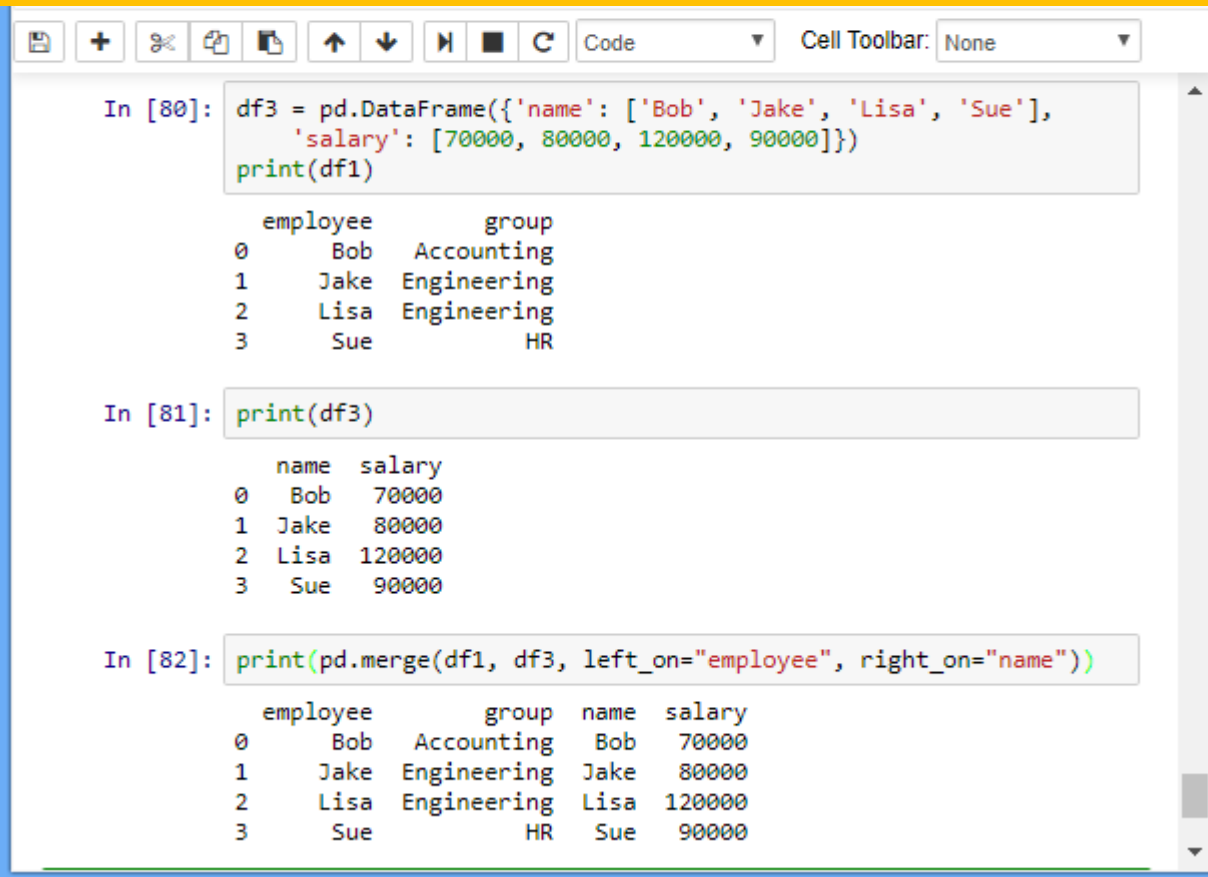
```
In [78]: print(df2)
```

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

```
In [79]: print(pd.merge(df1, df2, on = 'employee'))
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

If the columns to join on have different names, we can use the **left_on** and **right_on** parameters to specify the column names



```
In [80]: df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                             'salary': [70000, 80000, 120000, 90000]})
print(df1)
```

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

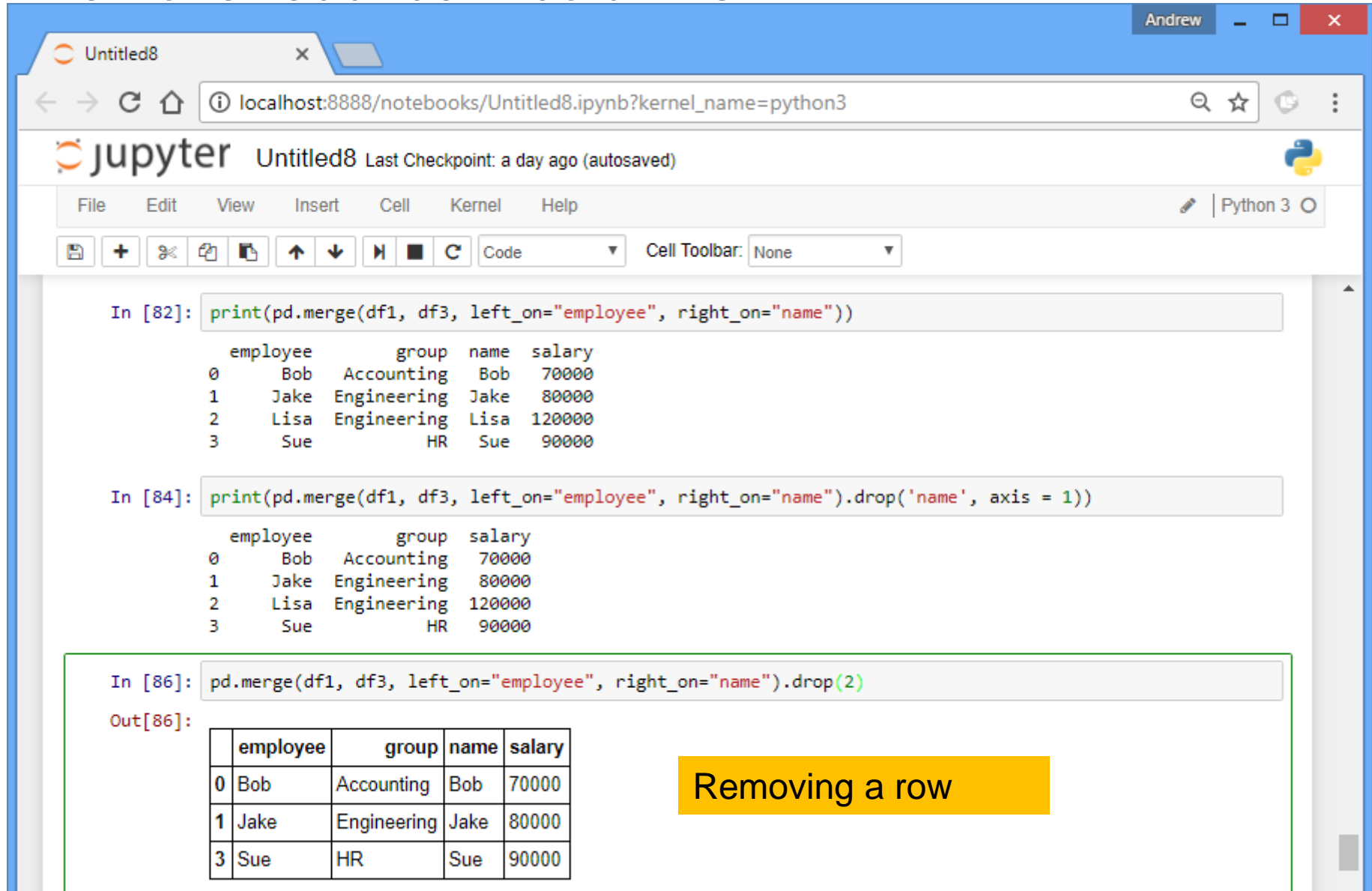
```
In [81]: print(df3)
```

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

```
In [82]: print(pd.merge(df1, df3, left_on="employee", right_on="name"))
```

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

We can use the DataFrame's **drop()** method to remove redundant columns



The screenshot shows a Jupyter Notebook window titled 'Untitled8' with a URL of `localhost:8888/notebooks/Untitled8.ipynb?kernel_name=python3`. The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for saving, adding cells, and running code. The code cells show the following:

```
In [82]: print(pd.merge(df1, df3, left_on="employee", right_on="name"))
```

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

```
In [84]: print(pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis = 1))
```

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

```
In [86]: pd.merge(df1, df3, left_on="employee", right_on="name").drop(2)
```

```
Out[86]:
```

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
3	Sue	HR	Sue	90000

A yellow box with the text "Removing a row" is positioned to the right of the final table output.

DataFrame.set_index

In [87]: `print(df1)`

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

In [88]: `print(df2)`

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

In [89]: `df1a = df1.set_index('employee')`
`print(df1a)`

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

In [92]: `df2a = df2.set_index('employee')`
`print(df2a)`

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

Untitled5 - Jupyter Notebook

localhost:8888/noteboo...

Jupyter Untitled5

Logout

Trusted Python 3

Edit View Insert Cell Kernel Widgets Help

+ ✂ 📄 📄 ⬆ ⬆ ▶ Run ■ ↺ ▶

Code

In [111]: df

Out[111]:

	month	year	sale
0	1	2012	55
1	4	2014	40
2	7	2013	84
3	10	2014	31

In [118]: df.set_index(pd.Index(["a", "b", "c", "d"]), inplace = True)

In [119]: df

Out[119]:

	month	year	sale
a	1	2012	55
b	4	2014	40
c	7	2013	84
d	10	2014	31

Untitled5 - Jupyter Notebook

localhost:8888/noteboo...

Jupyter Untitled5

Edit View Insert Cell Kernel W

+ ✂ 📄 📄 ⬆ ⬆ ▶ Run ■ ↺ ▶

In [120]: df.drop("a")

Out[120]:

	month	year	sale
b	4	2014	40
c	7	2013	84
d	10	2014	31

In [121]: df

Out[121]:

	month	year	sale
a	1	2012	55
b	4	2014	40
c	7	2013	84
d	10	2014	31

In [123]: df.drop('a', inplace = True)

In [124]: df

Out[124]:

	month	year	sale
b	4	2014	40
c	7	2013	84
d	10	2014	31

Joining on Indexes

The screenshot shows a Jupyter Notebook interface with a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar. The notebook is titled "Notebook saved" and is running Python 3. The main content area displays four code cells and their corresponding outputs.

Cell 1: `In [93]: print(df1a)`

Output 1:

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

Cell 2: `In [94]: print(df2a)`

Output 2:

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

Cell 3: `In [95]: print(pd.merge(df1a, df2a, left_index=True, right_index=True))`

Output 3:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Cell 4: `In [87]: print(df1)`

Output 4:

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

Cell 5: `In [88]: print(df2)`

Output 5:

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

Cell 6: `In [89]: df1a = df1.set_index('employee')
print(df1a)`

Output 6:

	group
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

Cell 7: `In [92]: df2a = df2.set_index('employee')
print(df2a)`

Output 7:

	hire_date
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

Andrew

Untitled8

localhost:8888/notebooks/Untitled8.ipynb?kernel_name=pyt...

jupyter Untitled8

File Edit View Insert

+

The **join()** method performs a merge that defaults to joining on indices

```
In [100]: print(df1a.join(df2a))
```

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

We can mix merging on indices and columns

```
In [101]: print(pd.merge(df1a, df2, left_index=True, right_on='employee'))
```

	group	employee	hire_date
1	Accounting	Bob	2008
2	Engineering	Jake	2012
0	Engineering	Lisa	2004
3	HR	Sue	2014

```
In [102]: print(pd.merge(df1, df2a, left_on='employee', right_index=True))
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Attributes lsuffix and rsuffix

The screenshot shows a Jupyter Notebook interface with the following content:

```
In [33]: df1 = pd.DataFrame({'name': ['Mike', 'Bob', 'Jim'], 'age': [21, 25, 17]})
df1
```

Out[33]:

	name	age
0	Mike	21
1	Bob	25
2	Jim	17

```
In [34]: df2 = pd.DataFrame({'name': ['Mike', 'Bob', 'Jim'], 'group': ['HR', 'Accounting', 'Sales']})
df2
```

Out[34]:

	name	group
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

```
In [47]: df1.join(df2)
```

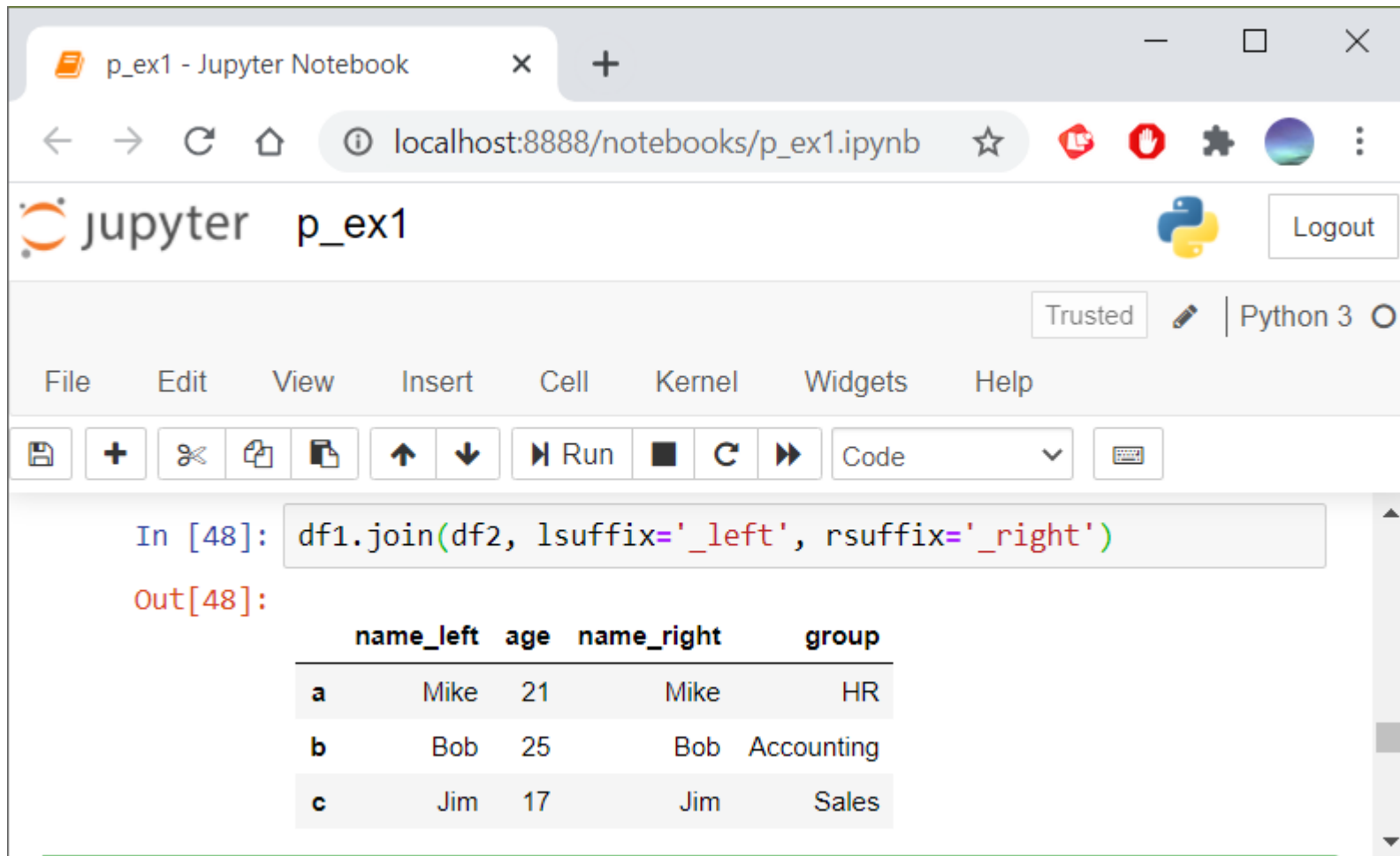
ValueError Traceback (most recent call last)

The browser window shows the Jupyter Notebook interface with the URL `localhost:8888/notebooks/p_ex1.ipynb`. The notebook title is `p_ex1 - Jupyter Notebook`. The last checkpoint was saved on Sunday at 3:28 PM (autosaved). The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The toolbar shows icons for saving, running, and other actions. The code editor shows the following code:

```
1973
1974 def lrenamer(x):
```

A `ValueError` is shown at the bottom: `ValueError: columns overlap but no suffix specified: Index(['name'], dtype='object')`

Attributes lsuffix and rsuffix



The screenshot shows a Jupyter Notebook window titled "p_ex1 - Jupyter Notebook" in a browser at "localhost:8888/notebooks/p_ex1.ipynb". The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding, deleting, and running cells. The current cell is a code cell containing the following Python code:

```
In [48]: df1.join(df2, lsuffix='_left', rsuffix='_right')
```

The output of the code is displayed below the cell:

```
Out[48]:
```

	name_left	age	name_right	group
a	Mike	21	Mike	HR
b	Bob	25	Bob	Accounting
c	Jim	17	Jim	Sales

The how Attribute

- Notice, by default pandas performs **inner** join (intersect)
- For full outer join (union), use the attribute `how = 'outer'`
- Inner join may be specified explicitly using `how = 'inner'`
- Other possible values for **how** are **left** and **right** for left/right outer joins



```
In [105]: df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'], 'food': ['fish', 'beans', 'bread']},  
                             columns=['name', 'food'])  
print(df6)
```

```
   name  food  
0  Peter  fish  
1   Paul  beans  
2   Mary  bread
```

```
In [106]: df7 = pd.DataFrame({'name': ['Mary', 'Joseph'], 'drink': ['water', 'juice']},  
                             columns=['name', 'drink'])  
print(df7)
```

```
   name  drink  
0  Mary  water  
1 Joseph  juice
```

```
In [107]: print(pd.merge(df6, df7))
```

```
   name  food  drink  
0  Mary  bread  water
```

```
In [108]: print(pd.merge(df6, df7, how = 'inner'))
```

```
   name  food  drink  
0  Mary  bread  water
```

Andrew

Untitled8

localhost:8888/notebooks/Untitled8.ipynb?k...

jupyter Untitled8

File Edit View Insert Cell Kernel Help Python 3

Cell Toolbar: None

```
In [109]: print(df6)
```

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

```
In [110]: print(df7)
```

	name	drink
0	Mary	water
1	Joseph	juice

```
In [111]: print(pd.merge(df6, df7, how = 'outer'))
```

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	water
3	Joseph	NaN	juice

Andrew

Untitled8

localhost:8888/notebooks/Untitled8.ipynb?k...

jupyter Untitled8

File Edit View Insert Cell Kernel Help Python 3

Cell Toolbar: None

```
In [109]: print(df6)
```

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

```
In [110]: print(df7)
```

	name	drink
0	Mary	water
1	Joseph	juice

```
In [112]: print(pd.merge(df6, df7, how = 'left'))
```

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	water

```
In [113]: print(pd.merge(df6, df7, how = 'right'))
```

	name	food	drink
0	Mary	bread	water
1	Joseph	NaN	juice

Renaming Columns in a DF – Function rename

jupyter p_ex1 Last Checkpoint: Last Sunday at 3:28 PM (autosaved)



Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted

Python 3

Run Code

```
In [61]: df1 = pd.DataFrame({'name': ['Mike', 'Bob', 'Jim'], 'group': ['HR', 'Accounting', 'Sales']})
df1
```

Out[61]:

	name	group
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

```
In [62]: # Renaming one column
df1.rename(columns = {'group': 'department'}, inplace = True)
df1
```

Out[62]:

	name	department
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

```
In [63]: # Renaming several columns
df1.rename(columns = {'name': 'person', 'group': 'department'}, inplace = True)
df1
```

Out[63]:

	person	department
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

DF.rename



Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted

Python 3

Save + Undo Copy Paste Up Down Run Stop Restart Code

```
In [74]: # Renaming several columns
df1.rename(columns = {'name': 'person', 'group': 'department'}, inplace = True)
df1
```

Out[74]:

	person	department
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

```
In [75]: df1.rename({'person': 'Preson-Name'}, axis = 'columns')
```

Out[75]:

	Preson-Name	department
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

```
In [76]: df1.rename({'person': 'Preson-Name'}, axis = 1)
```

Out[76]:

	Preson-Name	department
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

DF.rename

The screenshot shows a Jupyter Notebook interface with a menu bar (File, Edit, View, Insert, Cell, Format, Widgets, Help) and a toolbar with icons for saving, adding, zooming, and running code. The notebook contains four code cells, each showing a pandas DataFrame operation and its output.

Cell 1:

```
In [81]: df1
```

Out[81]:

	person	department
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

Cell 2:

```
In [77]: df1.rename({1:4, 2:8}, axis = 'index')
```

Out[77]:

	person	department
0	Mike	HR
4	Bob	Accounting
8	Jim	Sales

Cell 3:

```
In [78]: df1.rename({1:4, 2:8}, axis = 0)
```

Out[78]:

	person	department
0	Mike	HR
4	Bob	Accounting
8	Jim	Sales

Cell 4:

```
In [79]: df1.rename({1:4, 2:8})
```

Out[79]:

	person	department
0	Mike	HR
4	Bob	Accounting
8	Jim	Sales

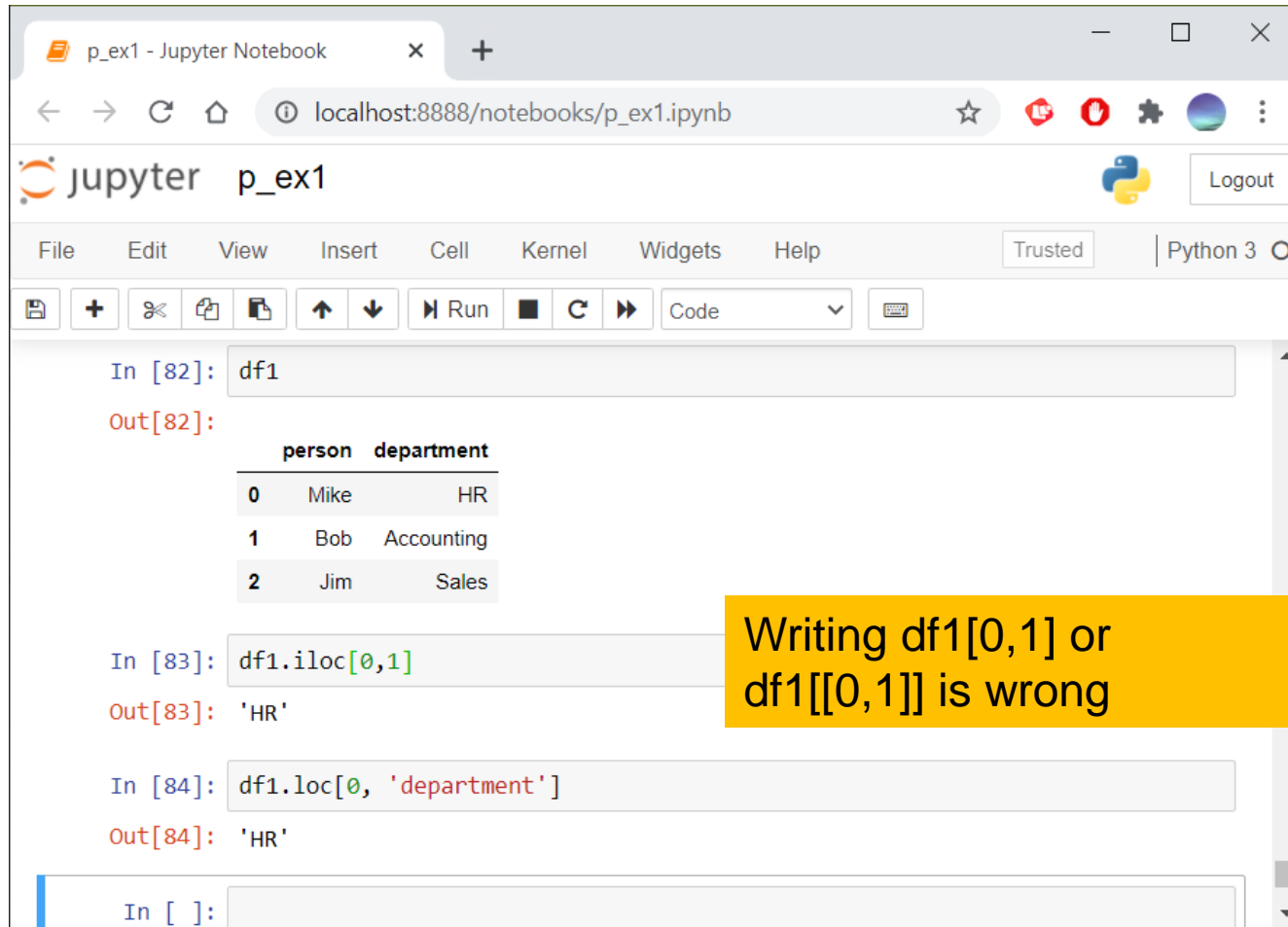
Cell 5:

```
In [80]: df1.rename({1:4, 2:8}, axis = 'rows')
```

Out[80]:

	person	department
0	Mike	HR
4	Bob	Accounting
8	Jim	Sales

Accessing an Element in a DF



The screenshot shows a Jupyter Notebook interface with the following content:

```
In [82]: df1
```

Out[82]:

	person	department
0	Mike	HR
1	Bob	Accounting
2	Jim	Sales

```
In [83]: df1.iloc[0,1]
```

Out[83]: 'HR'

```
In [84]: df1.loc[0, 'department']
```

Out[84]: 'HR'

```
In [ ]:
```

A yellow callout box on the right side of the notebook contains the text: "Writing df1[0,1] or df1[[0,1]] is wrong".