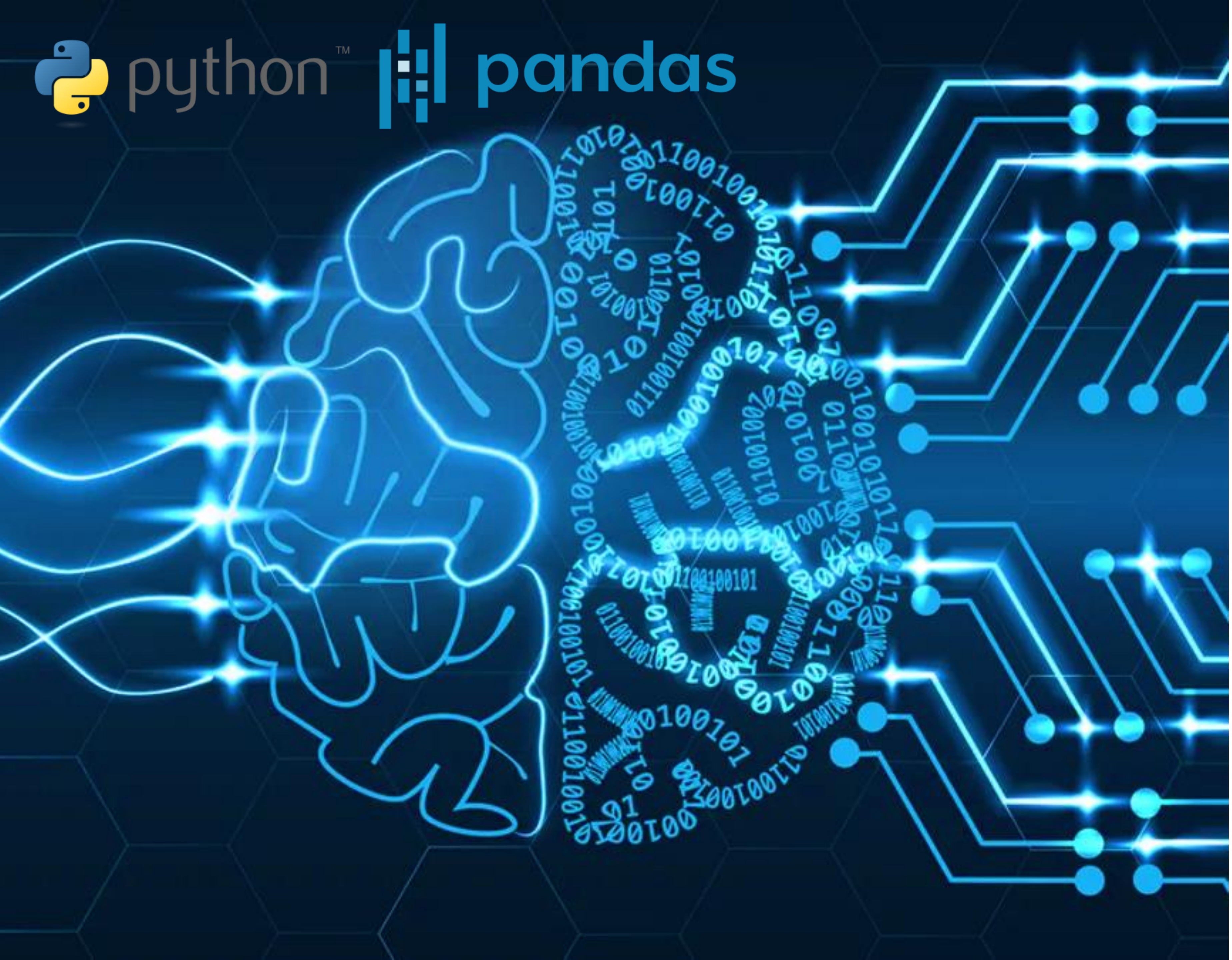


کتابخانه pandas در پایتون برای یادگیری ماشین

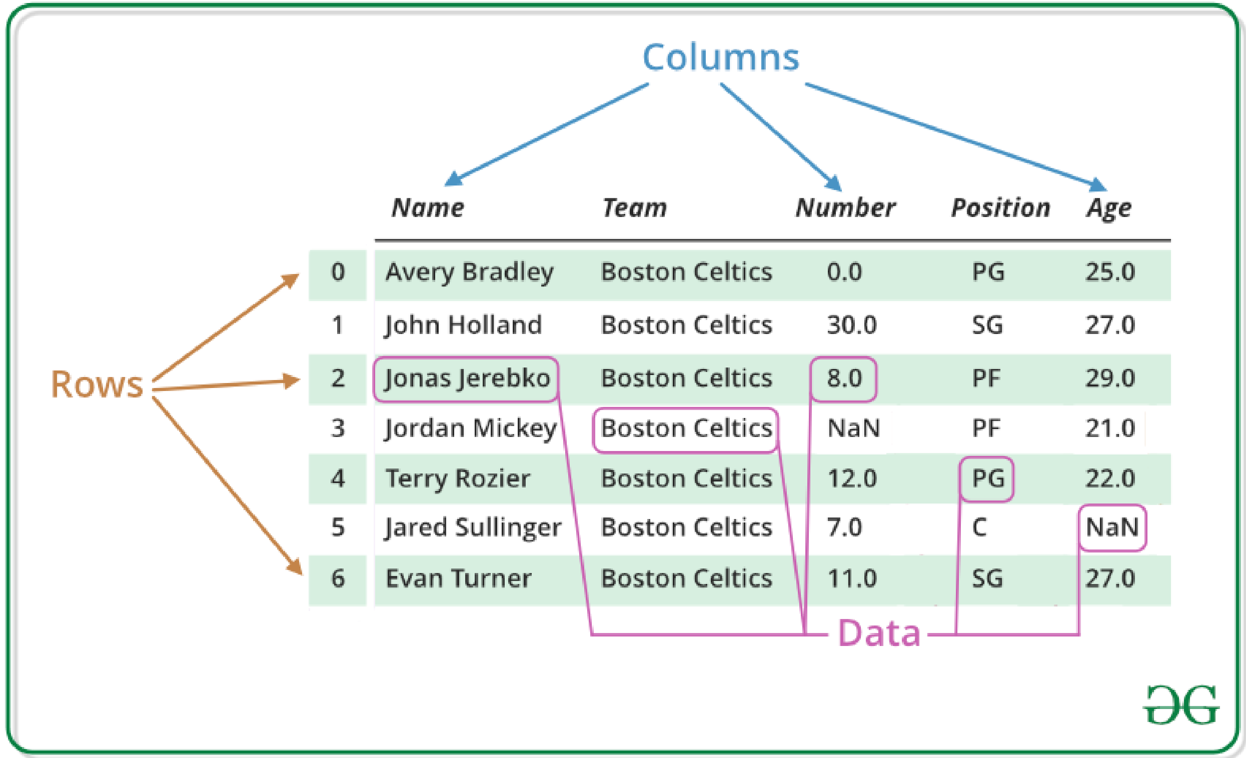
دکتر مهدی شریف‌زاده
امیرحسین محمودی
بهمن ۱۴۰۱

بسم الله الرحمن الرحیم

Pandas

Pandas is an important Python library for data manipulation, wrangling, and analysis. It functions as an intuitive and easy-to-use set of tools for performing operations on any kind of data.

Pandas allows you to work with both cross sectional data and time series based data.



	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Pandas > Data Structures of Pandas

All the data representation in pandas is done using two primary data structures:

- Series
- Dataframes

Pandas > Data Structures of Pandas >Series

Series in pandas is a one-dimensional ndarray with an axis label. It means that in functionality, it is almost similar to a simple array. Series objects can be used to represent time series data also. In this case, the index is a datetime object

Dataframe is the most important and useful data structure, which is used for almost all kind of data representation and manipulation in pandas. Unlike numpy arrays (in general) a dataframe can contain heterogeneous data. Typically tabular data is represented using dataframes, which is analogous to an Excel sheet or a SQL table. This is extremely useful in representing raw datasets as well as processed feature sets in Machine Learning and Data Science. All the operations can be performed along the axes, rows, and columns, in a dataframe. This will be the primary data structure which we will leverage, in most of the use cases in our later chapters.

Data Retrieval

Pandas provides numerous ways to retrieve and read in data. We can convert data from CSV files, databases, flat files, and so on into dataframes. We can also convert a list of dictionaries (Python dict) into a dataframe. The sources of data which pandas allows us to handle cover almost all the major data sources. For our introduction, we will cover three of the most important data sources:

- List of dictionaries
- CSV files
- Databases

List of Dictionaries to Dataframe

This is one of the simplest methods to create a dataframe. It is useful in scenarios where we arrive at the data we want to analyze, after performing some computations and manipulations on the raw data. This allows us to integrate a pandas based analysis into data being generated by other Python processing pipelines.

List of Dictionaries to Dataframe

In [2]:

```
import pandas as pd
d =  [{'city':'Delhi',"data":1000},
      {'city':'Banglaore',"data":2000},
      {'city':'Mumbai',"data":1000}]
pd.DataFrame(d)
```

Out[2]:

	city	data
0	Delhi	1000
1	Banglaore	2000
2	Mumbai	1000

In [3]:

```
df = pd.DataFrame(d)
```

Here we provided a list of Python dictionaries to the DataFrame class of the pandas library and the dictionary was converted into a DataFrame.

Two important things to note here: first the keys of dictionary are picked up as the column names in the dataframe, secondly we didn't supply an index and hence it picked up the default index of normal arrays.

CSV Files to Dataframe

CSV (Comma Separated Files) files are perhaps one of the most widely used ways of creating a dataframe. We can easily read in a CSV, or any delimited file (like TSV), using pandas and convert into a dataframe. For our example we will read in the following file and convert into a dataframe by using Python. The data in figure below is a sample slice of a CSV file containing the data of cities of the world from http://simplemaps.com/data/world-cities.

```
city,city_ascii,lat,lng,pop,country,iso2,iso3,province
Qal eh-ye Now,Qal eh-ye,34.98300013,63.13329964,2997,Afghanistan,AF,AFG,Badghis
Chaghcharan,Chaghcharan,34.5167011,65.25000063,15000,Afghanistan,AF,AFG,Ghor
Lashkar Gah,Lashkar Gah,31.58299802,64.35999955,201546,Afghanistan,AF,AFG,Hilmand
Zaranj,Zaranj,31.11200108,61.88699752,49851,Afghanistan,AF,AFG,Nimroz
Tarin Kowt,Tarin Kowt,32.63329815,65.86669865,10000,Afghanistan,AF,AFG,Uruzgan
Zareh Sharan,Zareh Sharan,32.85000016,68.41670453,13737,Afghanistan,AF,AFG,Paktika
Asadabad,Asadabad,34.86600004,71.15000459,48400,Afghanistan,AF,AFG,Kunar
Taloqan,Taloqan,36.72999904,69.54000364,64256,Afghanistan,AF,AFG,Takhar
Mahmud-E Eraqi,Mahmud-E Eraqi,35.01669608,69.33330065,7407,Afghanistan,AF,AFG,Kapisa
Mehtar Lam,Mehtar Lam,34.65000001,70.16670052,17345,Afghanistan,AF,AFG,Laghman
Baraki Barak,Baraki Barak,33.9667021,68.96670354,22305,Afghanistan,AF,AFG,Logar
Aybak,Aybak,36.26100015,68.04000051,24000,Afghanistan,AF,AFG,Samangan
```

CSV Files to Dataframe

We can convert this file into a dataframe with the help of the following code leveraging pandas.

In [4]:
```
city_data = pd.read_csv(filepath_or_buffer='simplemaps-worldcities-basic.csv')
```

In [5]:
```
city_data.head(n=10)
```

Out[5]:

	city	city_ascii	lat	lng	pop	country	iso2	iso3	province
0	Qal eh-ye Now	Qal eh-ye	34.983000	63.133300	2997.0	Afghanistan	AF	AFG	Badghis
1	Chaghcharan	Chaghcharan	34.516701	65.250001	15000.0	Afghanistan	AF	AFG	Ghor
2	Lashkar Gah	Lashkar Gah	31.582998	64.360000	201546.0	Afghanistan	AF	AFG	Hilmand
3	Zaranj	Zaranj	31.112001	61.886998	49851.0	Afghanistan	AF	AFG	Nimroz
4	Tarin Kowt	Tarin Kowt	32.633298	65.866699	10000.0	Afghanistan	AF	AFG	Uruzgan
5	Zareh	Zareh	32.850000	68.416705	13737.0	Afghanistan	AF	AFG	Paktika

Databases to Dataframe

The pandas.from_sql function combined with Python's powerful database library implies that the task of getting data from DBs is simple and easy. Due to this capability, no intermediate steps of data extraction are required. We will now take an example of reading data from a Microsoft SQL Server database. The following code will achieve this task.

```
server = 'xxxxxxxx' # Address of the database server
user = 'xxxxxx' # the username for the database server
password = 'xxxxx' # Password for the above user
database = 'xxxxx' # Database in which the table is present
conn = pymssql.connect(server=server, user=user, password=password, database=database)
query = "select * from some_table"
df = pd.read_sql(query, conn)
```

Data Access > Head and Tail

In [55]:

```
city_data.tail()
```

Out[55]:

	city	city_ascii	lat	lng	pop	country	iso2	iso3	prov
7317	Mutare	Mutare	-18.970019	32.650038	216785.0	Zimbabwe	ZW	ZWE	Manica
7318	Kadoma	Kadoma	-18.330006	29.909947	56400.0	Zimbabwe	ZW	ZWE	Mashona
7319	Chitungwiza	Chitungwiza	-18.000001	31.100003	331071.0	Zimbabwe	ZW	ZWE	H
7320	Harare	Harare	-17.817790	31.044709	1557406.5	Zimbabwe	ZW	ZWE	H
7321	Bulawayo	Bulawayo	-20.169998	28.580002	697096.0	Zimbabwe	ZW	ZWE	Bula

Data Access > Slicing and Dicing

In [56]:
```
series_es = city_data.lat
```

In [57]:
```
type(series_es)
```

Out[57]:
```
pandas.core.series.Series
```

In [58]:
```
series_es[1:10:2]
```

Out[58]:
```
1    34.516701
3    31.112001
5    32.850000
7    36.729999
9    34.650000
Name: lat, dtype: float64
```

In [59]:
```
series_es[:7]
```

Out[59]:
```
0    34.983000
1    34.516701
2    31.582998
3    31.112001
4    32.633298
5    32.850000
6    34.866000
Name: lat, dtype: float64
```

In [60]:
```
series_es[:-7315]
```

Out[60]:
```
0    34.983000
1    34.516701
2    31.582998
3    31.112001
4    32.633298
5    32.850000
6    34.866000
Name: lat, dtype: float64
```

Data Access > Slicing and Dicing in Dataframes

In [61]:
```
city_data[:7]
```

Out[61]:

	city	city_ascii	lat	lng	pop	country
0	Qal eh-ye Now	Qal eh-ye	34.983000	63.133300	2997.0	Afghanistan
1	Chaghcharan	Chaghcharan	34.516701	65.250001	15000.0	Afghanistan
2	Lashkar Gah	Lashkar Gah	31.582998	64.360000	201546.0	Afghanistan
3	Zaranj	Zaranj	31.112001	61.886998	49851.0	Afghanistan
4	Tarin Kowt	Tarin Kowt	32.633298	65.866699	10000.0	Afghanistan
5	Zareh Sharan	Zareh Sharan	32.850000	68.416705	13737.0	Afghanistan
6	Asadabad	Asadabad	34.866000	71.150005	48400.0	Afghanistan

Data Access

For providing access to specific rows and specific columns, pandas provides useful functions like iloc and loc which can be used to refer to specific rows and columns in a dataframe. There is also the ix function but we recommend using either loc or iloc. The following examples leverages the iloc function provided by pandas. This allows us to select the rows and columns using structure similar to array slicing In the example, we will only pick up the first five rows and the first four columns.

In [62]:

```
city_data.iloc[:5,:4]
```

Out[62]:

	city	city_ascii	lat	lng
0	Qal eh-ye Now	Qal eh-ye	34.983000	63.133300
1	Chaghcharan	Chaghcharan	34.516701	65.250001
2	Lashkar Gah	Lashkar Gah	31.582998	64.360000
3	Zaranj	Zaranj	31.112001	61.886998
4	Tarin Kowt	Tarin Kowt	32.633298	65.866699

Data Access

Another access mechanism is Boolean based access to the dataframe rows or columns. This is particularly important for dataframes, as it allows us to work with a specific set of rows and columns. Let's consider the following example in which we want to select cities that have population of more than 10 million and select columns that start with the letter l:

In [63]:
```
city_data[city_data['pop'] > 10000000][city_data.columns[pd.Series(city_data.columns).str.startswith('l')]]
```

Out[63]:

	lat	lng
360	-34.602502	-58.397531
1171	-23.558680	-46.625020
2068	31.216452	121.436505
3098	28.669993	77.230004
3110	19.016990	72.856989
3492	35.685017	139.751407
4074	19.442442	-99.130988
4513	24.869992	66.990009
5394	55.752164	37.615523
6124	41.104996	29.010002
7071	40.749979	-73.980017

Data Access

When we select data based on some condition, we always get the part of dataframe that satisfies the condition supplied. Sometimes we want to test a condition against a dataframe but want to preserve the shape of the dataframe. In these cases, we can use the where function.We'll illustrate this function with an example in which we will try to select all the cities that have population greater than 15 million.

In [64]:
```
city_greater_10mil = city_data[city_data['pop'] > 10000000]
city_greater_10mil.rename(columns={'pop':'population'}, inplace=True)
city_greater_10mil.where(city_greater_10mil.population > 15000000)
```

```
C:\Users\sharmatu\AppData\Local\Continuum\Anaconda\envs\Python3.5\lib\site-packages\pandas\core\frame.py:2746: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy
  **kwargs)
```

Out[64]:

	city	city_ascii	lat	lng	population	country	iso2	iso3	province
360	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1171	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2068	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3098	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3110	Mumbai	Mumbai	19.016990	72.856989	15834918.0	India	IN	IND	Maharashtra
3492	Tokyo	Tokyo	35.685017	139.751407	22006299.5	Japan	JP	JPN	Tokyo
4074	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4513	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5394	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6124	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7071	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Here we see that we get the output dataframe of the same size but the rows that don't conform to the condition are replaced with NaN

Data Operations > Values Attribute

Each pandas dataframe will have certain attributes. One of the important attributes is values. It is important as it allows us access to the raw values stored in the dataframe and if they all homogenous i.e., of the same kind then we can use numpy operations on them. This becomes important when our data is a mix of numeric and other data types and after some selections and computations, we arrive at the required subset of numeric data. Using the values attribute of the output dataframe, we can treat it in the same way as anumpy array. This is very useful when working with feature sets in Machine Learning. Traditionally, numpy vectorized operations are much faster than function based operations on dataframes.

In [65]:
```
df = pd.DataFrame(np.random.randn(8, 3),
columns=['A', 'B', 'C'])
```

Operations on dataframes

In [66]:
```
nparray = df.values
type(nparray)
```

Out[66]:
```
numpy.ndarray
```

Data Operations > Missing Data and the fillna Function

In [67]:
```
from numpy import nan
df.iloc[4,2] = nan
```

In [68]:
```
df
```

Out[68]:

	A	B	C
0	-1.279701	-0.074395	-1.370447
1	1.536038	0.060453	0.856685
2	0.475407	1.029245	-0.420355
3	-1.636635	-0.385956	-0.261129
4	1.259545	1.916660	NaN
5	1.591468	0.813209	0.605695
6	-1.270361	0.200358	0.035595
7	-0.189060	-1.874718	-1.088224

In [69]:
```
df.fillna(0)
```

Out[69]:

	A	B	C
0	-1.279701	-0.074395	-1.370447
1	1.536038	0.060453	0.856685
2	0.475407	1.029245	-0.420355
3	-1.636635	-0.385956	-0.261129
4	1.259545	1.916660	0.000000
5	1.591468	0.813209	0.605695
6	-1.270361	0.200358	0.035595
7	-0.189060	-1.874718	-1.088224

Data Operations> Descriptive Statistics Functions

```
In [70]: columns_numeric = ['lat','lng','pop']
```

```
In [71]: city_data[columns_numeric].mean()

Out[71]: lat         20.662876
         lng         10.711914
         pop     265463.071633
         dtype: float64
```

```
In [72]: city_data[columns_numeric].sum()

Out[72]: lat    1.512936e+05
         lng    7.843263e+04
         pop    1.943721e+09
         dtype: float64
```

```
In [73]: city_data[columns_numeric].count()

Out[73]: lat    7322
         lng    7322
         pop    7322
         dtype: int64
```

```
In [74]: city_data[columns_numeric].median()

Out[74]: lat       26.792730
         lng       18.617509
         pop    61322.750000
         dtype: float64
```

```
In [75]: city_data[columns_numeric].quantile(0.8)

Out[75]: lat        46.852480
         lng        89.900018
         pop    269210.000000
         Name: 0.8, dtype: float64
```

```
In [76]: city_data[columns_numeric].sum(axis = 1).head()

Out[76]: 0      3095.116300
         1     15099.766702
         2    201641.942998
         3     49943.998999
         4     10098.499997
         dtype: float64
```

```
In [77]: city_data[columns_numeric].describe()
```

Out[77]:

	lat	lng	pop
count	7322.000000	7322.000000	7.322000e+03
mean	20.662876	10.711914	2.654631e+05
std	29.134818	79.044615	8.287622e+05
min	-89.982894	-179.589979	-9.900000e+01
25%	-0.324710	-64.788472	1.734425e+04
50%	26.792730	18.617509	6.132275e+04
75%	43.575448	73.103628	2.001726e+05
max	82.483323	179.383304	2.200630e+07

Concatenating >Concatenating Using the concat Method

The first method to concatenate different dataframes in pandas is by using the concat method. The majority of the concatenation operations on dataframes will be possible by tweaking the parameters of the concat method. Let's look at a couple of examples to understand how the concat method works. The simplest scenario of concatenating is when we have more than one fragment of the same dataframe (which may happen if you are reading it from a stream or in chunks). In that case, we can just supply the constituent dataframes to the concat function as follows.

In [78]:
```
city_data1 = city_data.sample(3)
```

Concatanating data frames

In [79]:
```
city_data2 = city_data.sample(3)
city_data_combine = pd.concat([city_data1,city_data2])
city_data_combine
```

Out[79]:

	city	city_ascii	lat	lng	pop	country	iso2	iso3	province
4857	Shebekino	Shebekino	50.414350	36.894378	41301.5	Russia	RU	RUS	Belgorod
1561	Bouar	Bouar	5.950010	15.599967	31476.5	Central African Republic	CF	CAF	Nana-Mambéré
6650	Scottsbluff	Scottsbluff	41.867508	-103.660686	20172.0	United States of America	US	USA	Nebraska
964	Janauba	Janauba	-15.799618	-43.309977	38641.0	Brazil	BR	BRA	Minas Gerais
3896	Altata	Altata	24.636045	-107.916215	750.0	Mexico	MX	MEX	Sinaloa
7201	Tra Vinh	Tra Vinh	9.934002	106.334002	131360.0	Vietnam	VN	VNM	Trà Vinh

Concatenating > Joining by columns

Another common scenario of concatenating is when we have information about the columns of same dataframe split across different dataframes. Then we can use the concat method again to combine all the dataframes. Consider the following example:

In [83]:
```
country_data = city_data[['iso3','country']].drop_duplicates()
```

In [84]:
```
country_data.shape
```

Out[84]: (223, 2)

In [85]:
```
country_data.head()
```

Out[85]:

	iso3	country
0	AFG	Afghanistan
33	ALD	Aland
34	ALB	Albania
60	DZA	Algeria
111	ASM	American Samoa

In [86]:
```
del(city_data['country'])
```

In [87]:
```
city_data.merge(country_data, 'inner').head()
```

Out[87]:

	city	city_ascii	lat	lng	pop	iso2	iso3	province	country
0	Qal eh-ye Now	Qal eh-ye	34.983000	63.133300	2997.0	AF	AFG	Badghis	Afghanistan
1	Chaghcharan	Chaghcharan	34.516701	65.250001	15000.0	AF	AFG	Ghor	Afghanistan
2	Lashkar Gah	Lashkar Gah	31.582998	64.360000	201546.0	AF	AFG	Hilmand	Afghanistan
3	Zaranj	Zaranj	31.112001	61.886998	49851.0	AF	AFG	Nimroz	Afghanistan
4	Tarin Kowt	Tarin Kowt	32.633298	65.866699	10000.0	AF	AFG	Uruzgan	Afghanistan

Challenge on Pandas

1.Use seaborn to load a simple dataset of titanic passengers:

```
import seaborn as sns
print(sns.get_dataset_names())
data = sns.load_dataset('titanic')
data.head()
```

✓ 2.6s

```
['anagrams', 'anscombe', 'attention', 'brain_networks', 'car_crashes', 'diamonds', 'dots', 'dowjones', 'exercise', 'flights', 'fm
'penguins', 'planets', 'seaice', 'taxis', 'tips', 'titanic']
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

2. Remove the rows where at least one element is missing and show few rows of new data. (hint: use .dropna())

3.Create a new dataframe containing only females

4.Print number of females that survived .(you can use .count())

5.Create dataset of females aged under 30 and still alive .

6.create two dataset of consisting only categorical columns and only numerical columns.

Hint(use .select_dtype())

References

- Dipanjan Sarkar, Raghav Bali, Tushar Sharma, Practical Machine Learning with Python- A Problem-Solver’s Guide to Building Real-World Intelligent Systems, Apress, 2018.