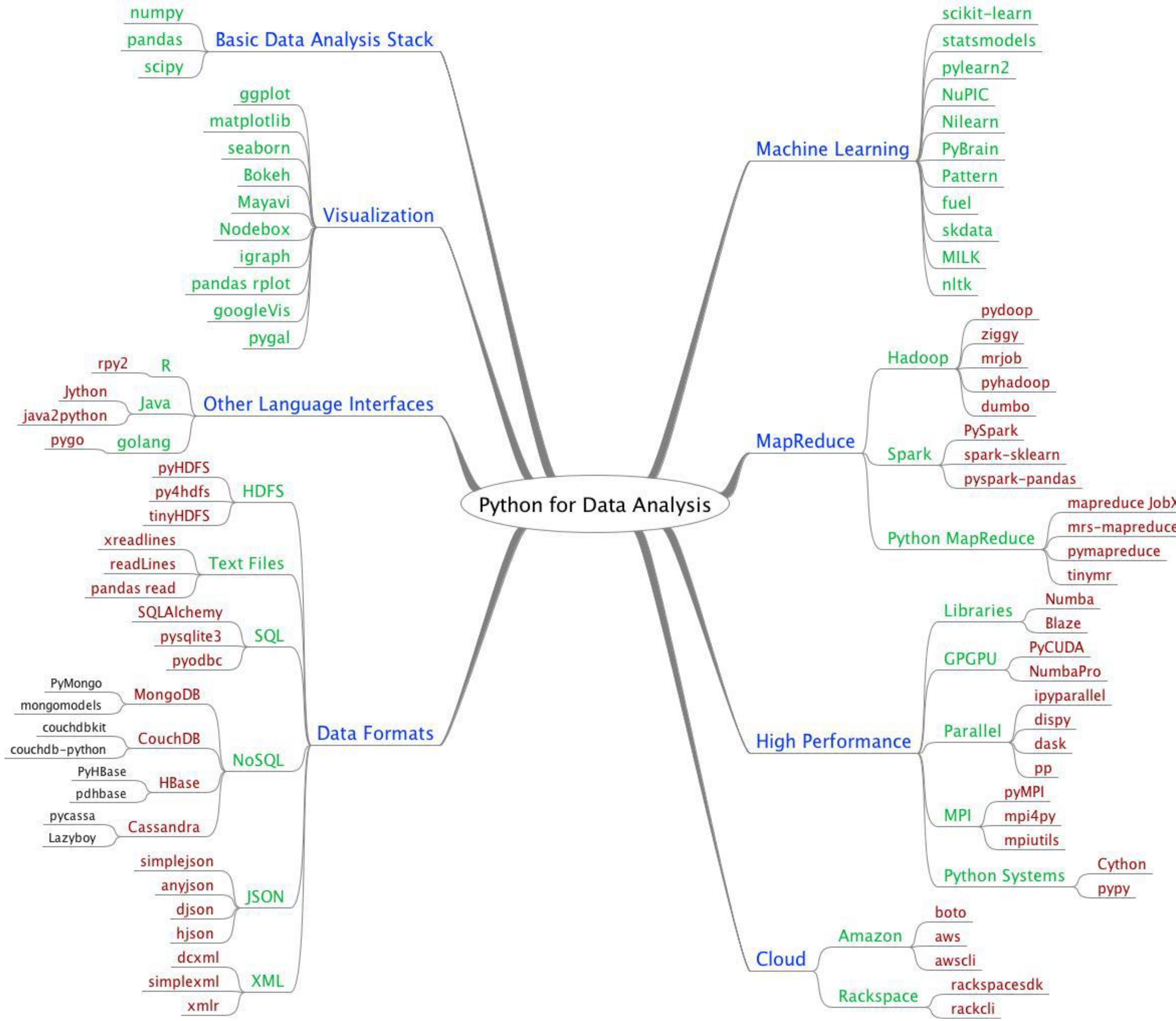
SI100B Introduction to Information Science and Technology
Python Programming

张海鹏 Haipeng Zhang
School of Information Science and Technology
ShanghaiTech University


Python for Data Analysis
Basic Data Analysis Stack
numpy
pandas
scipy
Visualization
ggplot
matplotlib
seaborn
Bokeh
Mayavi
Nodebox
igraph
pandas rplot
googleVis
pygal
Other Language Interfaces
R
rpy2
Java
Jython
java2python
golang
pygo
Data Formats
HDFS
pyHDFS
py4hdfs
tinyHDFS
Text Files
xreadlines
readLines
pandas read
SQL
SQLAlchemy
pysqlite3
pyodbc
NoSQL
MongoDB
PyMongo
mongomodels
CouchDB
couchdbkit
couchdb-python
HBase
PyHBase
pdhbase
Cassandra
pycassa
Lazyboy
JSON
simplejson
anyjson
djson
hjson
XML
dcxml
simplexml
xmlr
Machine Learning
scikit-learn
statsmodels
pylearn2
NuPIC
Nilearn
PyBrain
Pattern
fuel
skdata
MILK
nltk
MapReduce
Hadoop
pydoop
ziggy
mrjob
pyhadoop
dumbo
Spark
PySpark
spark-sklearn
pyspark-pandas
Python MapReduce
mapreduce JobX
mrs-mapreduce
pymapreduce
tinymr
High Performance
Libraries
Numba
Blaze
GPGPU
PyCUDA
NumbaPro
Parallel
ipyparallel
dispy
dask
pp
MPI
pyMPI
mpi4py
mpiutils
Python Systems
Cython
pypy
Cloud
Amazon
boto
aws
awscli
Rackspace
rackspacesdk
rackcli

Learning Objectives

- Understand and use
  - NumPy
  - Pandas

NumPy and Pandas

- NumPy: a general-purpose library that provides numerical arrays, and functions to manipulate the arrays efficiently
- Pandas: a data-manipulation library that provides data structures and operations for manipulating tables and time series data

NumPy

- NumPy is the fundamental package for scientific computing with Python
- It contains:
  1. a powerful N-dimensional array object and related functions for manipulating arrays
  2. useful linear algebra, Fourier transform, and random number capabilities
  3. reading data from and writing data to files
  4. vectorized computation
- How to install NumPy

```
pip3 install numpy
```

NumPy

- NumPy is the fundamental package for scientific computing with Python
- It contains:
  1. a powerful N-dimensional array object and related functions for manipulating arrays
  2. useful linear algebra, Fourier transform, and random number capabilities
  3. reading data from and writing data to files
  4. vectorized computation
- How to install NumPy

```
pip3 install numpy
```

The NumPy array object

- NumPy provides a multidimensional array object called ndarray
- NumPy arrays are typed arrays of fixed-size items.
- NumPy arrays are homogenous and can contain objects of only one type
- An ndarray consists of two parts:
    1. The actual data that is stored in a contiguous block of memory
    2. The metadata describing the actual data

Advantages of NumPy arrays

- NumPy arrays
  - takes less space, faster
  - homogenous
  - optimized functions built in such as linear algebra operations (numpy.linalg)
  - utilizes an optimized C API to make the array operations particularly quick
  - Vectorized operations
  - good at large data analysis
- Lists
  - takes more space, slower
  - heterogeneous
  - have to loop through the list

Create an ndarray object

```
>>> import numpy as np
>>> data = [[1, 2, 3, 4], [5, 6, 7, 8]]
>>> a = np.array(data) #create an array from a list
>>> a
array([[1, 2, 3, 4],
[5, 6, 7, 8]])
  # create an 0-array with shape given by a tuple
>>> np.zeros((3,6))
array([[0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0.]])
>>> np.array([np.arange(2,5),np.arange(4,7)])
array([[2, 3, 4],                     # from arrange ~range
[4, 5, 6]])
```

Some metadata

```
>>> import numpy as np
>>> data = [[1, 2, 3, 4], [5, 6, 7, 8]]
>>> a = np.array(data) # create an array from list
>>> a
array([[1, 2, 3, 4],
[5, 6, 7, 8]])
>>> a.ndim                 # metadata ndim = dimension
2
>>> a.shape                           # metadata shape
(2, 4)
>>> a.dtype
dtype('int32')
```

NumPy array vs list

```
import numpy as np
import time

a = np.arange(1000000)
l = list(range(1000000))

start = time.time()
for _ in range(10):
    a2 = a * 2
print(time.time() - start)

start = time.time()
for _ in range(10):
    l2 = [x * 2 for x in l]
print(time.time() - start)
```

Output:

0.05100297927856445
1.9661142826080322

List comprehensions: https://docs.python.org/3/tutorial/datastructures.html

Data Types for ndarrays

- Python has an integer type, a float type, and complex type; nonetheless, this is not sufficient for scientific calculations

- In practice, we still demand more data types with varying precisions and, consequently, different storage sizes of the type

NumPy numerical types:

Type	Description
bool_	Boolean (True or False) stored as a byte
int_	Platform integer (normally either int32 or int64)
intn (n=8,16,32,64)	Integer (-2^{n-1} to $2^{n-1}-1$)
uintn (n=8,16,32,64)	Unsigned integer (0 to $2^{n}-1$)
floatn (n=16,32,64)	Half/single/double precision
complexn (n=64,128)	Complex number, represented by two n/2 bit floats (real and imaginary components)

PS: float = float64

Create ndarray with specific type

```
>>> import numpy as np
>>> data = [[1, 2, 3, 4], [5, 6, 7, 8]]
>>> a = np.array(data) # create an array from list
>>> a
array([[1, 2, 3, 4],
[5, 6, 7, 8]])
>>> a.dtype
dtype('int32')
>>> b = np.array(data, dtype=np.int8)
>>> b
array([[1, 2, 3, 4],
[5, 6, 7, 8]], dtype=int8)
>>> b.dtype
dtype('int8')
```

ndarray Type casting

```
>>> import numpy as np
>>> data = [[1, 2, 3, 4], [5, 6, 7, 8]]
>>> a = np.array(data) # create an array from list
>>> a
array([[1, 2, 3, 4],
[5, 6, 7, 8]])
>>> a.dtype
dtype('int32')
>>> b = a.astype(np.float32) #astype: type casting
>>> b
array([[1., 2., 3., 4.],
[5., 6., 7., 8.]], dtype=float32)
>>> b.dtype
dtype('float32')
```

Indexing

- indexing is similar to list
  - indexing: $a[i_1]...a[i_k]$
  - assignment via indexing:

    $a[i_1]...a[i_k]$
- Efficient indexing for array (not work for list),
  - Tuple indexing: $a[(i_1,...,i_k)]$ or $a[i_1,...,i_k]$
  - assignment via tuple indexing:

    $a[(i_1,...,i_k)]$ or $a[i_1,...,i_k]$
- All of these indexing return a new view of original data (pointer to the original data), it does not copy items in array

```
>>> l = [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11,
12]]]
>>> a = np.array(l)
>>> l[0][1]
[4, 5, 6]
>>> a[0][1]
array([4, 5, 6])
>>> a[0,1]     # [0,1] => [(0,1)] (0,1) is a tuple
array([4, 5, 6])
>>> l[0,1]
Traceback (most recent call last):…TypeError: list
indices must be integers or slices, not tuple
>>> a[0][1] = [1,2,3]
>>> a
array([[[ 1, 2, 3],
[ 1, 2, 3]],
[[ 7, 8, 9],
[10, 11, 12]]])
```

Array Indexing

- Array indexing (or any sequence-like object that can be converted to an array, with the exception of tuples)

$$a[\ [i_1,\ldots,i_k]\ ]$$

  - i_j indicates which value in array to use in place of the index
  - what is returned is a copy of the original data, not a view as one gets for other indexing

- Multi-array indexing

$$a[l_1,\ldots,l_k]$$

  - $l_1,\ldots,l_k$ are sequence-like objects except tuples

For more, visit:
https://docs.scipy.org/doc/numpy/user/basics.indexing.html

```
>>> l = [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11,
12]]]
>>> a = np.array(l)
>>> a
array([[[ 1, 2, 3],
[ 4, 5, 6]],
[[ 7, 8, 9],
[10, 11, 12]]])

>>> a[ [1,1] ] # array indexing, np.array([a[1],a[1]])
array([[[ 7, 8, 9],
[10, 11, 12]],
[[ 7, 8, 9],
[10, 11, 12]]])

>>> a[[0,1,1],[0,0,1]] # Multi-array indexing
array([[1, 2, 3], # np.array([a[0,0],a[1,0],a[1,1]])
[7, 8, 9],
[10, 11, 12]])
```

Slicing

- 1-dimensional array: same as sequence-like object
  - slicing: a[start=0[:stop=-1[:step=1]]]
  - assigning: a[start=0[:stop=-1[:step=1]]] = newsubarray

- multi-dimensional array: dimensional-wise slicing

  a[
    $start_1$=0[:$stop_1$=-1[:$step_1$=1]]], # first dim
    ......,
    $start_k$=0[:$stop_1$=-1[:$step_k$=1]]], #k^{th} dim
  ]

- Only return a new view of original data

```
>>> a = np.array([[[1, 2, 3], [4, 5, 6]],
          [[7, 8, 9], [10, 11, 12]]])
>>> a[0:2,1:2] #elem0, elem1 from 1st dim, elem1 from 2nd dim
array([[[ 4, 5, 6]],
[[10, 11, 12]]])

>>> a[0:2][1:2] #c = a[0:2] c[1:2]
array([[[ 7, 8, 9],
[10, 11, 12]]])
>>> a[0:2,1:2,1:2] #... elem1 from 3rd dim
array([[[5]],
[[11]]])
>>>
```

Boolean indexing

- It returns a 1-D array containing all the elements in the indexed array corresponding to all the true elements in the boolean array

a[b]

```python
import numpy as np
arr = np.zeros((5,5))

for i in range(5):
    arr[i] = i
print(arr)

b1 = arr >=1
b2 = arr <=3
b = b1 & b2

print(b1)
print(b)
print(arr[b])
```

```
[[0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1.]
 [2. 2. 2. 2. 2.]
 [3. 3. 3. 3. 3.]
 [4. 4. 4. 4. 4.]]
[[False False False False False]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]]
[[False False False False False]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [False False False False False]]
[1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 3. 3. 3. 3. 3.]
```

Broadcasting

- NumPy attempts to execute a procedure even though the operands do not have the same shape
- For an operation op on an array object a and a scalar s

  s op a or a op s

  - the scalar s is broadened to the shape of the array a
  - then the operation is executed on two array objects in an element-by-element fashion

```
>>> a = np.array([[1,2],[3,4]])
>>> a + 2
array([[3, 4],
[5, 6]])
>>> a + np.array([[2,2],[2,2]])
array([[3, 4],
[5, 6]])
>>> a * 2
array([[2, 4],
[6, 8]])
>>> a ** 2
array([[ 1, 4],
[ 9, 16]], dtype=int32)
>>> a * a
array([[ 1, 4],
[ 9, 16]])
```

Universal Functions

- A universal function (ufunc) is a function that operates on ndarrays in an element-by-element fashion, supporting
  - array broadcasting,
  - type casting,
  - and several other standard features.
- A ufunc is a “vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs
- ufuncs are instances of the numpy.ufunc class
- Many of the built-in functions are implemented in compiled C code

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
[3, 4]])
>>> b = a**2
>>> b
array([[ 1, 4],
[ 9, 16]], dtype=int32)
>>> np.sqrt(b)
array([[1., 2.],
[3., 4.]])
>>> c = np.array([[-1,4],[-3,5]])
>>> np.maximum(a,c)
array([[1, 4],
[3, 5]])
```

File Input and Output with Arrays

- Save and load one array

save(file, arr) and load(file)

  - save an array to a binary file in ``.npy`` format
  - load an array from a binary file in ``.npy`` format
  - file : file, str, or pathlib.Path
  - arr : array data to be saved

```
>>> x = np.array([[0,1,2],[3,4,5],[6,7,8]])
>>> np.save("x.npy",x)
>>> np.load("x.npy")
array([[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
>>>
```

File Input and Output with Arrays

- Save and load arrays

  savez(file, *args, **kwds) and load(file)

  - save arrays to a binary file in ``.npz`` format
  - load arrays from a binary file in ``.npz`` format
  - file : file, str, or pathlib.Path
  - args (optional): arrays to save to the file. The arrays will be saved with names "arr_0", "arr_1", and so on.
  - kwds (optional) : arrays to save to the file. Arrays will be saved in the file with the keyword names
  - At least one argument is given

File Input and Output with Arrays

```
>>> x = np.array([[0,1,2],[3,4,5],[6,7,8]])
>>> y = np.array([[1,2],[3,4]])
>>> np.savez("file.npz",x,y)
>>> xy = np.load("file.npz")
>>> xy['arr_0']
array([[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
>>> xy['arr_1']
array([[1, 2],
[3, 4]])
```

Pandas

- pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for Python
- The two primary data structures of pandas,
  - Series (1-dimensional)
  - DataFrame (2-dimensional)
- Handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering
- How to install pandas

  pip3 install pandas

Series

- The Pandas Series data structure is a one-dimensional, heterogeneous array with labels
- Ordered dict
- A Series data structure can be created via:
  - Using a Python dict: the sorted dict keys will become the index unless supply the index
  - Using a NumPy array: index values starting from 0
  - Using a single scalar value: must supply the index
- Index and values can be obtained via

  s.index and s.values

- Access values of specific index: $s[[i_1,\ldots,i_k]] = v$

Series from array

```
>>> import pandas as pd
>>> s1 = pd.Series([4,7,-5,3])
>>> s1
0 4            # first column is index
1 7                    # 2nd column is value
2 -5
3 3
dtype: int64
>>> s1.index
RangeIndex(start=0, stop=4, step=1)
>>> s1.values
array([ 4, 7, -5, 3], dtype=int64)
>>> s1[2]
-5
```

Series from array

```
>>> import pandas as pd
>>> s2 = pd.Series([4,7,-5,3],
                   index = ['a','b','c','d'])
>>> s2                          # specify index
a 4
b 7
c -5
d 3
dtype: int64
>>> s2.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> s2.values
array([ 4, 7, -5, 3], dtype=int64)
>>> >>> s2['b']
7
```

Series from dict

```
>>> d = {'Ohio': 35000, 'Texas': 71000,
'Oregon':16000, 'Utah': 5000}
>>> s3 = pd.Series(d)
>>> s3
Ohio 35000
Texas 71000
Oregon 16000
Utah 5000
dtype: int64
>>> s3 [["Utah","Ohio"]] # select view of some
Utah 5000
Ohio 35000
dtype: int64
```

Series from dict

- **Create a Series from dict with defined order index**
- **The 2nd argument determines the order**
- **Missing data is denoted by NaN**

(pd.isnull() and pd.notnull() to check null values)

```
>>> d = {'Ohio': 35000, 'Texas': 71000,
'Oregon':16000, 'Utah': 5000}
>>> o = ['Oregon','Utah','Texas','Shanghai']
>>> s4 = pd.Series(d,o)
>>> s4
Oregon 16000.0
Utah 5000.0
Texas 71000.0
Shanghai NaN
dtype: float64
```

Index can be renamed

- **Index of a series can be renamed via**
  s.index = newindex
- Note: can't do s.values = newvalues

```
>>> d = {'Ohio': 35000, 'Texas': 71000,
'Oregon':16000, 'Utah': 5000}
>>> s = pd.Series(d)
>>> s.index = [1,2,3,4]
>>> s
1 35000
2 71000
3 16000
4 5000
dtype: int64
```

Operations on Series

- Index-label by index-label computation
- NaN op v = NaN ; v op NaN = NaN
- Slicing via index   s[start=0:end=-1:stop=1]
- NumPy functions can operate on Series

```
>>> s3
Ohio 35000
Texas 71000
Oregon 16000
Utah 5000
dtype: int64
```

```
>>> s4
Oregon 16000.0
Utah 5000.0
Texas 71000.0
Shanghai NaN
dtype: float64
```

```
>>> s3 + s4
Ohio NaN
Oregon 32000.0
Shanghai NaN
Texas 142000.0
Utah 10000.0
dtype: float64
```

DataFrames

- DataFrame is a labeled two-dimensional data structure similar to Microsoft Excel
- The columns in Pandas DataFrame can be of different types
- DataFrame can be created via:
  - Using another DataFrame or Series
  - Using 1-D NumPy array, list, dict
  - Composition of arrays that has a 2-D shape
  - Reading from a file, such as a CSV/Excel file

Create a DataFrame from Dict

```
>>> data = {'state': ['Ohio', 'Ohio', 'Ohio',
'Nevada', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002, 2003],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> frame = pd.DataFrame(data)
>>> frame
state year pop
0 Ohio 2000 1.5
1 Ohio 2001 1.7
2 Ohio 2002 3.6
3 Nevada 2001 2.4
4 Nevada 2002 2.9
5 Nevada 2003 3.2
>>>
```

Get Header (first five rows)

```
>>> data = {'state': ['Ohio', 'Ohio', 'Ohio',
'Nevada', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002, 2003],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> frame = pd.DataFrame(data)
>>> frame.head()
state year pop
0 Ohio 2000 1.5
1 Ohio 2001 1.7
2 Ohio 2002 3.6
3 Nevada 2001 2.4
4 Nevada 2002 2.9
>>>
```

Create a DataFrame from Dict

- Create a DataFrame from dict with defined order
- The 2nd argument determines the order
- Missing column is denoted by NaN

```
>>> data = {'state': ['Ohio', 'Ohio', 'Ohio'],
'year': [2000, 2001, 2002],
'pop': [1.5, 1.7, 3.6]}
>>> frame = pd.DataFrame(data,
columns=['year','state','pop','nocol'])
>>> frame
year state pop nocol
0 2000 Ohio 1.5 NaN
1 2001 Ohio 1.7 NaN
2 2002 Ohio 3.6 NaN
```

Indexing on DataFrame

- Get row index: frame.index
- Row index renaming: frame.index = newIndex
- Get column index: frame.columns
- Column index renaming: frame.columns = newColumns
- Get a specific column: frame[columnName]
- Set/add a column: frame[columnName]=Column
- Row/column reindex:
  - frame.reindex([a list of row reindex])
  - frame.reindex(columns=[a list of column reindex])
- Drop row/columns:
  - frame.drop([a list of row index])
  - frame.drop([a list of column index], axis='columns')

Indexing of DataFrame

```
>>> data = {'state': ['Ohio', 'Ohio', 'Ohio'],
'year': [2000, 2001, 2002],
'pop': [1.5, 1.7, 3.6]}
>>> frame = pd.DataFrame(data)
>>> frame.index
RangeIndex(start=0, stop=3, step=1)
>>> frame.columns
Index(['state', 'year', 'pop'], dtype='object')
>>> frame['year']
0 2000
1 2001
2 2002
Name: year, dtype: int64
```

Selection with loc and iloc

- Selects specific rows and columns
  - frame.loc([rownames],[columnnames])
  - frame.iloc([rowindex],[columnIndex])

- Set values of specific rows and columns
  - frame.loc([rownames],[columnnames]) = values
  - frame.iloc([rowindex],[columnIndex]) = values

For more, visit:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html

Selection with loc and iloc

```
>>> import numpy as np
>>> import pandas as pd
>>> frame = pd.DataFrame(np.arange(16).reshape((4,
4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns=['one', 'two', 'three', 'four'])
>>> frame.loc['Colorado', ['two', 'three']]
two 5
three 6
Name: Colorado, dtype: int32
>>> frame.iloc[1, [1, 2]]
two 5
three 6
Name: Colorado, dtype: int32
```

Recap

- Understand and use
  - NumPy
  - Pandas

Readings (recommended)

– Numpy indexing
  - https://docs.scipy.org/doc/numpy/user/basics.indexing.html
– Pandas indexing
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html