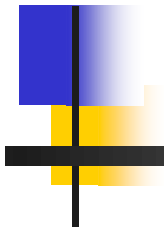
Intro to Pandas

IOE 373 Lecture 18



Topics

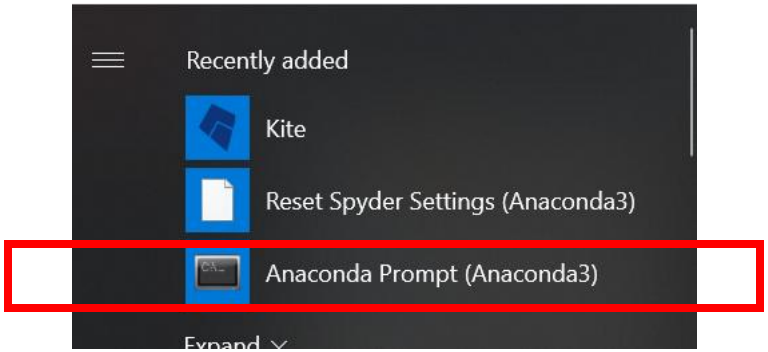
- What is “Pandas”?
  - Installation
- Series
- Data Frames
- Operations

What is “Pandas”?

- Pandas is an open source library built on NumPy
- Allows data cleaning, preparation and fast analysis
- It has built-in visualization features
- Can work with data from a variety of sources

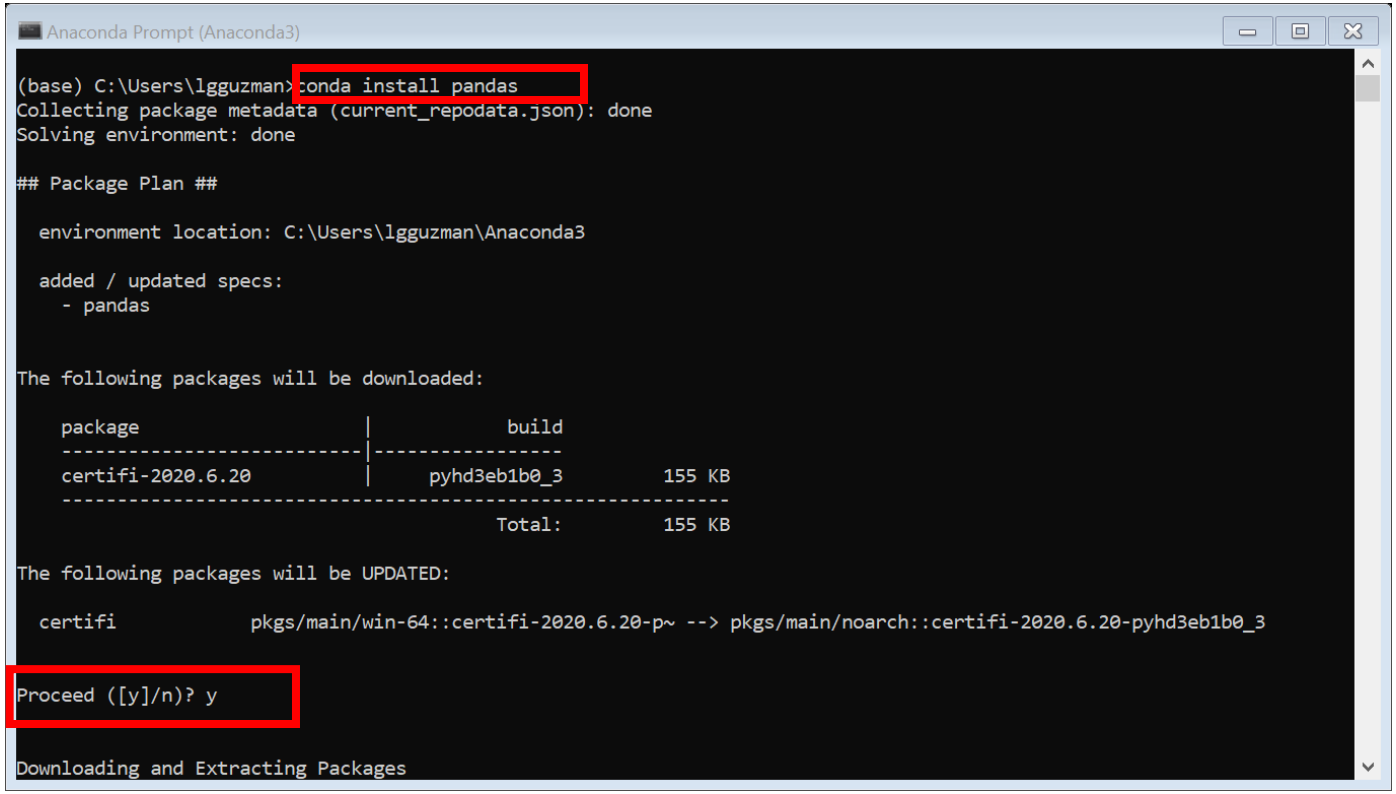
Install Pandas

- If Pandas is not installed, run Anaconda Prompt (should be under the Anaconda folder).
  - If you're using CAEN computers, Pandas should be installed already



Install Pandas

- Type: “conda install pandas”
- Type “y” when prompted “Proceed (y/n)?”



Series

- A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data:

```
In [3]: import pandas as pd

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

In [5]: obj
Out[5]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series

- Often it will be desirable to create a Series with an index identifying each data point:

```
In [6]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [7]: obj2
Out[7]:
d    4
b    7
a   -5
c    3
dtype: int64

In [8]: obj2.index
Out[8]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Series – NumPy Array

- Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values:

```
In [9]: obj2['a']
Out[9]: -5

In [10]: obj2['d'] = 6

In [11]: obj2[['c', 'a', 'd']]
Out[11]:
c    3
a   -5
d    6
dtype: int64
```

NumPy Operations

- NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [14]: obj2[obj2 > 0]
Out[14]:
d    6
b    7
c    3
dtype: int64
```

```
In [15]: obj2 * 2
Out[15]:
d    12
b    14
a   -10
c     6
dtype: int64
```

```
In [16]: np.exp(obj2)
Out[16]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Series and Dictionaries

- Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values.
- If you have data contained in a Python dict, you can create a Series from it by passing the dictionary
  - Note that the index in the resulting Series will have the keys in sorted order

```
In [15]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000,
'Utah': 5000}

In [16]: obj3 = pd.Series(sdata)

In [17]: obj3
Out[17]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Operations

- A critical Series feature for many applications is that it automatically aligns differently-indexed data in arithmetic operations:

```
In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
In [29]: obj4
Out[29]:
California      NaN
Ohio          35000
Oregon        16000
Texas         71000
dtype: float64
```

```
In [30]: obj3 + obj4
Out[30]:
California       NaN
Ohio           70000
Oregon         32000
Texas         142000
Utah             NaN
dtype: float64
```

Data Frames

- DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns
  - Each can be a different value type (numeric, string, boolean, etc.).
- DataFrame has both a row and column index
- Numerous ways to construct a DataFrame,
  - Most common is from a dictionary of equal-length lists or NumPy arrays

```
In [18]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada',
'Nevada'],'year': [2000, 2001, 2002, 2001, 2002],'pop': [1.5, 1.7,
3.6, 2.4, 2.9]}

In [19]: frame = pd.DataFrame(data)
```

Data Frames

- The resulting DataFrame will have its index assigned automatically as with Series (starting at index 0), and the columns are placed in sorted order:

```
In [20]: frame
Out[20]:
    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
```

Data Frames

- You can specify the sequence of columns
  - The DataFrame's columns will be exactly what you pass:

```
In [22]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[22]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
```

Data Frames

- If you pass a column that isn't contained in data, it will appear with NA values in the result:

```
In [23]: frame2 = pd.DataFrame(data, columns=['year', 'state',
'pop', 'debt'], index=['one', 'two', 'three', 'four', 'five'])

In [24]: frame2
Out[24]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN

In [25]: frame2.columns
Out[25]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Data Frames

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [26]: frame2.state
Out[26]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
```

```
In [27]: frame2.year
Out[27]:
one      2000
two      2001
three    2002
four     2001
five     2002
Name: year, dtype: int64
```

Data Frames

- Rows can also be retrieved by position or name by a couple of methods, such as the .loc indexing field or .iloc(for integer index)

```
In [33]: frame2.loc['three']
Out[33]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

Data Frames

- Columns can be modified by assignment.
- For example, the empty 'debt' column could be assigned a scalar value or an array of values

```
In [34]: frame2['debt'] = 16.5

In [35]: frame2
Out[35]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
```

```
In [37]: import numpy as np

In [38]: frame2['debt'] = np.arange(5.)

In [39]: frame2
Out[39]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
```

Data Frames

- When assigning lists or arrays to a column, the value's length must match the length of the DataFrame.

```
In [40]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four',
'five'])

In [41]: frame2['debt'] = val

In [42]: frame2
Out[42]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
```

It will insert missing values in any holes

- Missing 'debt' values for one and three

Data Frames

- Assigning a column that doesn't exist will create a new column.

```
In [43]: frame2['eastern'] = frame2.state == 'Ohio'

In [44]: frame2
Out[44]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
```

Data Frames

- Another common form of data is a nested dict of dicts format:

```
In [48]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},'Ohio': {2000: 1.5,
2001: 1.7, 2002: 3.6}}
```

- If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [49]: frame3 = pd.DataFrame(pop)

In [50]: frame3
Out[50]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2000     NaN   1.5
```

You can transpose with .T

```
In [51]: frame3.T
Out[51]:
        2001  2002  2000
Nevada   2.4   2.9   NaN
Ohio     1.7   3.6   1.5
```

Data Frames

- When searching for records, the keys in the inner dicts are unioned and sorted to form the index in the result.

```
In [53]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[53]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

Note that there're no values for 2003 in the pop dictionary (see previous slide)

Data Frames

- Like Series, the values attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [54]: frame3.values
Out[54]:
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

Operations – Unique Values

- Finding unique values in a Data Frame

```
In [58]: df=pd.DataFrame({'col1':[1,2,3,4],'col2':
[444,555,666,444],'col3':['abc','def','ghi','xyz']})

In [59]: df.head()
Out[59]:
   col1  col2 col3
0     1   444  abc
1     2   555  def
2     3   666  ghi
3     4   444  xyz
```

- Unique method **unique()** will return an array with the unique values

```
In [60]: df['col2'].unique()
Out[60]: array([444, 555, 666], dtype=int64)
```

Operations

- What if I want the number of unique values?
  - Can find out the length of the array of unique values using the **len** function:

```
In [61]: len(df['col2'].unique())
Out[61]: 3
```

  - Or the built in method **nunique()**

```
In [62]: df['col2'].nunique()
Out[62]: 3
```

Operations

- Frequency table with **value_counts()** function

```
In [63]: df['col2'].value_counts()
Out[63]:
444    2
555    1
666    1
Name: col2, dtype: int64
```

Operations – Selecting Data

- Conditional Selection, specify the criterion for selection:

```
In [64]: df[df['col1']>2]
Out[64]:
   col1  col2 col3
2     3   666  ghi
3     4   444  xyz
```

- To combine conditions, enclose in parenthesis and use '&' for and or '|' for or (| is the vertical bar not the letter I)

```
In [65]: df[(df['col1']>2)&(df['col2']==444)]
Out[65]:
   col1  col2 col3
3     4   444  xyz
```

Operations – Apply function

- Aside from standard functions such as sum(), you can apply custom functions to your data frames.

```
In [71]: df['col1'].sum()
Out[71]: 10
```

```
In [69]: def times2(x):
    ...:     return x*2
    ...:

In [70]: df['col1'].apply(times2)
Out[70]:
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

Operations – Apply Function

- We can also apply built-in functions
  - For example, say we want to know the length of the strings in one of the columns, we could apply the **len** function:

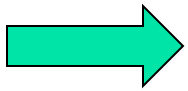
```
In [72]: df['col3']
Out[72]:
0    abc
1    def
2    ghi
3    xyz
Name: col3, dtype: object
```

```
In [73]: df['col3'].apply(len)
Out[73]:
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

Operations – Apply Lambda Expression

- Lambda expressions are little, anonymous functions, subject to a more restrictive but more concise syntax than regular Python functions that can be embedded within an apply function.

```
In [76]: df['col2']
Out[76]:
0    444
1    555
2    666
3    444
Name: col2, dtype: int64
```



```
In [75]: df['col2'].apply(lambda x:x*2)
Out[75]:
0     888
1    1110
2    1332
3     888
Name: col2, dtype: int64
```

Operations - sort_values

- Specify the criterion for sorting:

```
In [76]: df['col2']
Out[76]:
0    444
1    555
2    666
3    444
Name: col2, dtype: int64
```

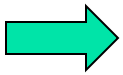
```
In [78]: df.sort_values('col2')
Out[78]:
   col1  col2 col3
0     1   444  abc
3     4   444  xyz
1     2   555  def
2     3   666  ghi
```

```
In [79]: df.sort_values(by='col2')
Out[79]:
   col1  col2 col3
0     1   444  abc
3     4   444  xyz
1     2   555  def
2     3   666  ghi
```

Operations – isnull()

- Use isnull() to check for null values

```
In [81]: df
Out[81]:
   col1  col2 col3
0     1   444  abc
1     2   555  def
2     3   666  ghi
3     4   444  xyz
```



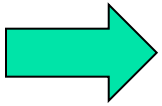
```
In [80]: df.isnull()
Out[80]:
    col1   col2   col3
0  False  False  False
1  False  False  False
2  False  False  False
3  False  False  False
```

Pivot_Table Function

- Pivot tables in data frames

```
In [84]: data={'A':['foo','foo','foo','bar','bar','bar'],'B':
['one','one','two','two','one','one'],'C':
['x','y','x','y','x','y'],'D':[1,3,2,5,4,1]}

In [85]: df=pd.DataFrame(data)
```



```
In [86]: df
Out[86]:
     A    B  C  D
0  foo  one  x  1
1  foo  one  y  3
2  foo  two  x  2
3  bar  two  y  5
4  bar  one  x  4
5  bar  one  y  1
```

```
In [87]: df.pivot_table(values='D',index=['A','B'],columns=['C'])
Out[87]:
C          x    y
A   B
bar one  4.0  1.0
    two  NaN  5.0
foo one  1.0  3.0
    two  2.0  NaN
```