Introduction to Python

pandas for Tabular Data

Topics

1) pandas
    1) Series
    2) Dataframe

pandas

NumPy's array is optimized for homogeneous numeric data that's accessed via integer indices. For example, a 2D Numpy of floats representing grades.

Data science presents unique demands for which more customized data structures are required.

Big data applications must support mixed data types, customized indexing, missing data, data that's not structured consistently and data that needs to be manipulated into forms appropriate for the databases and data analysis packages you use.

Pandas is the most popular library for dealing with such data. It is built on top of Numpy and provides two key collections: **Series** for one-dimensional collections and **DataFrames** for two-dimensional collections.

Series

A Series is an enhanced one-dimensional array.

Whereas arrays use only zero-based integer indices, Series support custom indexing, including even non-integer indices like strings.

Series also offer additional capabilities that make them more convenient for many data-science oriented tasks. For example, Series may have missing data, and many Series operations ignore missing data by default.

Series

By default, a Series has integer indices numbered sequentially from 0. The following creates a Series of student grades from a list of integers. The initializer also may be a tuple, a dictionary, an array, another Series or a single value.

```
import pandas as pd
In[1]: grades = pd.Series([87, 100, 94])
In[2]: grades
Out[25]:
0     87
1    100
2     94
dtype: int64

In[3]: grades[0]
Out[25]: 87
```

Descriptive Statistics

```
import pandas as pd
```

```
In[2]: grades.count()
Out[2]: 3
In[2]: grades.mean()
Out[2]: 93.66666666666667
In[2]: grades.std()
Out[2]: 6.506407098647712
```

```
 In[2]: grades.describe()
Out[2]:
count       3.000000
mean       93.666667
std         6.506407
min        87.000000
25%        90.500000
50%        94.000000
75%        97.000000
max       100.000000
dtype: float64
```

Custom Indices

You can specify custom indices with the index keyword argument:

```
import pandas as pd
In[1]: grades = pd.Series([87, 100, 94],
                          index=['John','Sara','Mike'])
In[2]: grades
Out[25]:
John     87
Sara    100
Mike     94
dtype: int64
```

We can also use a dictionary to create a Series.
This is equivalent to the code above:

```
grades = pd.Series({'John': 87, 'Sara': 100, 'Mike': 94})
```

In this case, we used string indices, but you can use other immutable types, including integers not beginning at 0 and nonconsecutive integers. Again, notice how nicely and concisely pandas formats a Series for display.

Custom Indices

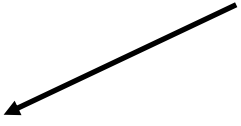
You can specify custom indices with the index keyword argument:

```
import pandas as pd
In[1]: grades = pd.Series([87, 100, 94],
                          index=['John','Sara','Mike'])
In[2]: grades['John']
Out[25]: 87


In[2]: grades.dtype
Out[25]: int64


In[2]: grades.values
Out[25]: array([ 87, 100, 94])
```

A Series underlying values is a Numpy array!



DataFrames

A DataFrame is an enhanced two-dimensional array.

Like Series, DataFrames can have custom row and column indices, and offer additional operations and capabilities that make them more convenient for many data-science oriented tasks.

DataFrames also support missing data. Each column in a DataFrame is a Series. The Series representing each column may contain different element types, as you'll soon see when we discuss loading datasets into DataFrames.

DataFrames

Pandas displays DataFrames in tabular format with the indices left aligned in the index column and the remaining columns' values right aligned.

```
import pandas as pd
In[1]:grades_dict = {'Wally': [87, 96, 70],
                     'Eva': [100, 87,90],
                     'Sam': [94, 77, 90],
                     'Katie': [100, 81, 82],
                     'Bob': [83, 65, 85]}
In[1]: grades = pd.DataFrame(grades_dict)
In[1]: grades
Out[25]:
```

	Wally	Eva	Sam	Katie	Bob
0	87	100	94	100	83
1	96	87	77	81	65
2	70	90	90	82	85

The dictionary's keys become the column names and the values associated with each key become the element values in the corresponding column.

index Attribute

Let's use the index attribute to change the DataFrame's indices from sequential integers to labels:

```
import pandas as pd
In[1]: grades.index = ['Test1', 'Test2', 'Test3']
In[1]: grades
Out[25]:
```

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85

Equivalently, we could have done this using an index keyword argument as in the Series case(see slide 26).

```
grades = pd.DataFrame(grades_dict,
                      index=['Test1', 'Test2', 'Test3'])
```

Accessing Columns

One benefit of pandas is that you can quickly and conveniently look at your data in many different ways, including selecting portions of the data. Let's start by getting Eva's grades by name, which displays her column as a Series:

```
import pandas as pd
In[1]: grades['Eva']
Out[25]:
Test1    100
Test2     87
Test3     90
Name: Eva, dtype: int64
```

Selecting Rows via loc

Though DataFrames support indexing capabilities with [], the pandas documentation recommends using the attributes loc, iloc, at and iat, which are optimized to access DataFrames and also provide additional capabilities beyond what you can do only with []. You can access a row by its label via the DataFrame's loc attribute.

```
import pandas as pd
```

```
In[1]: grades.loc['Test1']
```

```
Out[25]:
Wally     87
Eva      100
Sam       94
Katie    100
Bob       83
Name: Test1, dtype: int64
```

Selecting Rows via iloc

You also can access rows by integer zero-based indices using the iloc attribute (the i in iloc means that it's used with integer indices). The following lists all the grades in the second row:

```
import pandas as pd
In[1]: grades.iloc[2]
Out[25]:
Wally    96
Eva      87
Sam      77
Katie    81
Bob      65
Name: Test2, dtype: int64
```

Selecting Rows via Slices

The index can be a slice. When using slices containing labels with loc, the range specified includes the high index, but when using slices containing integer indices with iloc, the range you specify excludes the high index.

```
In[1]: grades.loc['Test1':'Test3']
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test1    100  100   94    100   83
Test2     96   87   77     81   65
Test3     70   90   90     82   85
```

```
In[1]: grades.iloc[0:2]
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test1    100  100   94    100   83
Test2     96   87   77     81   65
```

Note that Test3 is excluded!

Selecting Rows via Slices

To select specific rows, use a list rather than slice notation with loc or iloc:

```
In[1]: grades.loc[['Test1','Test3']]
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test1    100  100   94    100   83
Test3     70   90   90     82   85

In[1]: grades.iloc[[0, 2]]
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test1    100  100   94    100   83
Test3     70   90   90     82   85
```

Selecting Subsets of Rows and Columns

So far, we've selected only entire rows. You can focus on small subsets of a DataFrame by selecting rows and columns using two slices, two lists or a combination of slices and lists.

```
In[1]: grades.loc['Test1':'Test2', ['Eva', 'Katie']]
```

Out[25]:

	Eva	Katie
Test1	**100**	**100**
Test2	**87**	**81**

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

Out[25]:

	Wally	Eva	Sam
Test1	100	100	94
Test3	70	90	90

Boolean Indexing

One of pandas' more powerful selection capabilities is Boolean indexing. For example, let's select all the A grades—that is, those that are greater than or equal to 90:

```
In[1]: grades[grades >= 90]
```

```
Out[25]:
```

	Wally	Eva	Sam	Katie	Bob
Test1	100.0	100.0	94.0	100.0	NaN
Test2	96.0	NaN	NaN	NaN	NaN
Test3	NaN	90.0	90.0	NaN	NaN

Pandas checks every grade to determine whether its value is greater than or equal to 90 and, if so, includes it in the new DataFrame. Grades for which the condition is False are represented as NaN (not a number) in the new DataFrame. NaN is pandas' notation for missing values.

Boolean Indexing

Pandas Boolean indices combine multiple conditions with the Python operator & (bitwise AND), not the and Boolean operator. For or conditions, use | (bitwise OR).

Let's select all the B grades in the range 80–89:

```
In[1]: grades[(grades >= 80) & (grades < 90)]
Out[25]:
       Wally   Eva  Sam  Katie   Bob
Test1    NaN   NaN  NaN    NaN  83.0
Test2    NaN  87.0  NaN   81.0   NaN
Test3    NaN   NaN  NaN   82.0  85.0
```

Descriptive Statistics

Both Series and DataFrames have a describe method that calculates basic descriptive statistics for the data and returns them as a DataFrame. In a DataFrame, the statistics are calculated by column.

```
In[1]: grades.describe()
```

Out[25]:

	Wally	Eva	Sam	Katie	Bob
count	3.000000	3.000000	3.000000	3.000000	3.000000
mean	88.666667	92.333333	87.000000	87.666667	77.666667
std	16.289056	6.806859	8.888194	10.692677	11.015141
min	70.000000	87.000000	77.000000	81.000000	65.000000
25%	83.000000	88.500000	83.500000	81.500000	74.000000
50%	96.000000	90.000000	90.000000	82.000000	83.000000
75%	98.000000	95.000000	92.000000	91.000000	84.000000
max	100.000000	100.000000	94.000000	100.000000	85.000000

Descriptive Statistics

```
In[1]: grades.mean()
Out[25]:
Wally    88.666667
Eva      92.333333
Sam      87.000000
Katie    87.666667
Bob      77.666667
dtype: float64
```

Descriptive Statistics

You can quickly transpose the rows and columns—so the rows become the columns, and the columns become the rows—by using the T attribute:

```
In[1]: grades.T
```

Out[25]:

	Test1	Test2	Test3
Wally	100	96	70
Eva	100	87	90
Sam	94	77	90
Katie	100	81	82
Bob	83	65	85

Descriptive Statistics

Let's assume that rather than getting the summary statistics by student, you want to get them by test. Simply call describe on grades.T, as in:

```
In[1]: grades.T.describe()
Out[25]:
```

	Test1	Test2	Test3
count	5.000000	5.00000	5.000000
mean	95.400000	81.20000	83.400000
std	7.402702	11.54123	8.234076
min	83.000000	65.00000	70.000000
25%	94.000000	77.00000	82.000000
50%	100.000000	81.00000	85.000000
75%	100.000000	87.00000	90.000000
max	100.000000	96.00000	90.000000

Descriptive Statistics

To see the average of all the students' grades on each test, just call mean on the T attribute:

```
In[1]: grades.T.mean()
Out[25]:
Test1    95.4
Test2    81.2
Test3    83.4
dtype: float64
```

Sorting by Row Indices

You'll often sort data for easier readability. You can sort a DataFrame by its rows or columns, based on their indices or values.

Let's sort the rows by their indices in descending order using sort_index and its keyword argument ascending=False (the default is to sort in ascending order). This returns a new DataFrame containing the sorted data:

```
In[1]: grades.sort_index(ascending=False)
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test3     70   90   90     82   85
Test2     96   87   77     81   65
Test1    100  100   94    100   83
```

Sorts rows in descending order.

Sorting by Column Indices

Now let's sort the columns into ascending order (left-to-right) by their column names. Passing the axis=1 keyword argument indicates that we wish to sort the column indices, rather than the row indices—axis=0 (the default) sorts the row indices:

```
In[1]: grades.sort_index(axis=1)
Out[25]:
```

Sorts columns in ascending alphabetical order.

	Bob	Eva	Katie	Sam	Wally
Test1	83	100	100	94	100
Test2	65	87	81	77	96
Test3	85	90	82	90	70

Sorting a Column by Values

The method sort_values() can be used to sort the values of a row or column. By default, it sorts values of a column(axis=0) in ascending order.

```
In[1]: grades
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test2     96   87   77     81   65
Test3     70   90   90     82   85
```

```
In[1]: grades.sort_values(by='Wally')
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test3     70   90   90     82   85
Test2     96   87   77     81   65
Test1    100  100   94    100   83
```

Sorts values of Wally's tests in ascending order.

Sorting a Row by values

We can also sort the values of a row(axis=1).

```
In[1]: grades
Out[25]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test2     96   87   77     81   65
Test3     70   90   90     82   85

In[1]: grades.sort_values(by='Test1', axis=1)
Out[25]:
       Bob  Sam  Wally  Eva  Katie
Test1   83   94    100  100    100
Test2   65   77     96   87     81
Test3   85   90     70   90     82
```

Sorts values of Test1 in ascending order.

Sorting the Transpose.

```
In[1]: grades.T.sort_values(by='Test1', ascending=False)
Out[25]:
       Test1  Test2  Test3
Wally    100     96     70
Eva      100     87     90
Katie    100     81     82
Sam       94     77     90
Bob       83     65     85
```

Sorting a particular Series

In the previous example, since we're only sorting Test1, we might not want to see the other tests.

```
In[2]: grades.loc['Test1'].sort_values(ascending=False)
Out[65]:
Katie    100
Eva      100
Wally    100
Sam       94
Bob       83
Name: Test1, dtype: int64
```