# DataFrameDataStructure_ed

November 7, 2020

The DataFrame data structure is the heart of the Panda's library. It's a primary object that you'll be working with in data analysis and cleaning tasks.

The DataFrame is conceptually a two-dimensional series object, where there's an index and multiple columns of content, with each column having a label. In fact, the distinction between a column and a row is really only a conceptual distinction. And you can think of the DataFrame itself as simply a two-axes labeled array.

[1]:
```
# Lets start by importing our pandas library
import pandas as pd
```

[2]:
```
# I'm going to jump in with an example. Lets create three school records for␣
↪students and their
# class grades. I'll create each as a series which has a student name, the␣
↪class name, and the score.
record1 = pd.Series({'Name': 'Alice',
                        'Class': 'Physics',
                        'Score': 85})
record2 = pd.Series({'Name': 'Jack',
                        'Class': 'Chemistry',
                        'Score': 82})
record3 = pd.Series({'Name': 'Helen',
                        'Class': 'Biology',
                        'Score': 90})
```

[3]:
```
# Like a Series, the DataFrame object is index. Here I'll use a group of␣
↪series, where each series
# represents a row of data. Just like the Series function, we can pass in our␣
↪individual items
# in an array, and we can pass in our index values as a second arguments
df = pd.DataFrame([record1, record2, record3],
                  index=['school1', 'school2', 'school1'])

# And just like the Series we can use the head() function to see the first␣
↪several rows of the
# dataframe, including indices from both axes, and we can use this to verify␣
↪the columns and the rows
df.head()
```

[3]:
```
           Name      Class  Score
school1  Alice    Physics     85
school2   Jack  Chemistry     82
school1  Helen    Biology     90
```

[4]:
```
# You'll notice here that Jupyter creates a nice bit of HTML to render the␣
↪results of the
# dataframe. So we have the index, which is the leftmost column and is the␣
↪school name, and
# then we have the rows of data, where each row has a column header which was␣
↪given in our initial
# record dictionaries
```

[5]:
```
# An alternative method is that you could use a list of dictionaries, where␣
↪each dictionary
# represents a row of data.

students = [{'Name': 'Alice',
              'Class': 'Physics',
              'Score': 85},
            {'Name': 'Jack',
             'Class': 'Chemistry',
             'Score': 82},
            {'Name': 'Helen',
             'Class': 'Biology',
             'Score': 90}]

# Then we pass this list of dictionaries into the DataFrame function
df = pd.DataFrame(students, index=['school1', 'school2', 'school1'])
# And lets print the head again
df.head()
```

[5]:
```
           Name      Class  Score
school1  Alice    Physics     85
school2   Jack  Chemistry     82
school1  Helen    Biology     90
```

[6]:
```
# Similar to the series, we can extract data using the .iloc and .loc␣
↪attributes. Because the
# DataFrame is two-dimensional, passing a single value to the loc indexing␣
↪operator will return
# the series if there's only one row to return.

# For instance, if we wanted to select data associated with school2, we would␣
↪just query the
# .loc attribute with one parameter.
df.loc['school2']
```

[6]:
```
Name          Jack
Class    Chemistry
Score           82
Name: school2, dtype: object
```

[7]:
```
# You'll note that the name of the series is returned as the index value, while␣
↪the column
# name is included in the output.

# We can check the data type of the return using the python type function.
type(df.loc['school2'])
```

[7]:
```
pandas.core.series.Series
```

[8]:
```
# It's important to remember that the indices and column names along either␣
↪axes horizontal or
# vertical, could be non-unique. In this example, we see two records for␣
↪school1 as different rows.
# If we use a single value with the DataFrame lock attribute, multiple rows of␣
↪the DataFrame will
# return, not as a new series, but as a new DataFrame.

# Lets query for school1 records
df.loc['school1']
```

[8]:

| | Name | Class | Score |
|---|---|---|---|
| school1 | Alice | Physics | 85 |
| school1 | Helen | Biology | 90 |

[9]:
```
# And we can see the the type of this is different too
type(df.loc['school1'])
```

[9]:
```
pandas.core.frame.DataFrame
```

[10]:
```
# One of the powers of the Panda's DataFrame is that you can quickly select␣
↪data based on multiple axes.
# For instance, if you wanted to just list the student names for school1, you␣
↪would supply two
# parameters to .loc, one being the row index and the other being the column␣
↪name.

# For instance, if we are only interested in school1's student names
df.loc['school1', 'Name']
```

[10]:
```
school1    Alice
school1    Helen
Name: Name, dtype: object
```

[11]:
```
# Remember, just like the Series, the pandas developers have implemented this␣
↪using the indexing
# operator and not as parameters to a function.
```

```
# What would we do if we just wanted to select a single column though? Well,␣
↪there are a few
# mechanisms. Firstly, we could transpose the matrix. This pivots all of the␣
↪rows into columns
# and all of the columns into rows, and is done with the T attribute
df.T
```

[11]:

```
       school1    school2  school1
Name     Alice       Jack    Helen
Class  Physics  Chemistry  Biology
Score       85         82       90
```

[12]:

```
# Then we can call .loc on the transpose to get the student names only
df.T.loc['Name']
```

[12]:

```
school1    Alice
school2     Jack
school1    Helen
Name: Name, dtype: object
```

[13]:

```
# However, since iloc and loc are used for row selection, Panda reserves the␣
↪indexing operator
# directly on the DataFrame for column selection. In a Panda's DataFrame,␣
↪columns always have a name.
# So this selection is always label based, and is not as confusing as it was␣
↪when using the square
# bracket operator on the series objects. For those familiar with relational␣
↪databases, this operator
# is analogous to column projection.
df['Name']
```

[13]:

```
school1    Alice
school2     Jack
school1    Helen
Name: Name, dtype: object
```

[14]:

```
# In practice, this works really well since you're often trying to add or drop␣
↪new columns. However,
# this also means that you get a key error if you try and use .loc with a␣
↪column name
df.loc['Name']
```

```
        ␣
↪---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call␣
↪last)
```

```
    /opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py in␣
↪get_loc(self, key, method, tolerance)
   2889             try:
-> 2890                 return self._engine.get_loc(key)
   2891             except KeyError:


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_get_loc_duplicates()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_maybe_get_bool_indexer()


    KeyError: 'Name'


During handling of the above exception, another exception occurred:


    KeyError                                  Traceback (most recent call␣
↪last)

    <ipython-input-14-b44ae13e0b9f> in <module>
      1 # In practice, this works really well since you're often trying to␣
↪add or drop new columns. However,
      2 # this also means that you get a key error if you try and use .loc␣
↪with a column name
----> 3 df.loc['Name']


    /opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪__getitem__(self, key)
   1408
   1409             maybe_callable = com.apply_if_callable(key, self.obj)
-> 1410             return self._getitem_axis(maybe_callable, axis=axis)
   1411
   1412     def _is_scalar_access(self, key: Tuple):
```

```
    /opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪_getitem_axis(self, key, axis)
   1823         # fall thru to straight lookup
   1824         self._validate_key(key, axis)
-> 1825         return self._get_label(key, axis=axis)
   1826
   1827


    /opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪_get_label(self, label, axis)
    155             raise IndexingError("no slices here, handle elsewhere")
    156
--> 157         return self.obj._xs(label, axis=axis)
    158
    159     def _get_loc(self, key: int, axis: int):


    /opt/conda/lib/python3.7/site-packages/pandas/core/generic.py in␣
↪xs(self, key, axis, level, drop_level)
   3736             loc, new_index = self.index.get_loc_level(key,␣
↪drop_level=drop_level)
   3737         else:
-> 3738             loc = self.index.get_loc(key)
   3739
   3740             if isinstance(loc, np.ndarray):


    /opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py in␣
↪get_loc(self, key, method, tolerance)
   2890                 return self._engine.get_loc(key)
   2891             except KeyError:
-> 2892                 return self._engine.get_loc(self.
↪_maybe_cast_indexer(key))
   2893         indexer = self.get_indexer([key], method=method,␣
↪tolerance=tolerance)
   2894         if indexer.ndim > 1 or indexer.size > 1:


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_get_loc_duplicates()
```

```
    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_maybe_get_bool_indexer()


    KeyError: 'Name'
```

[ ]:
```
# Note too that the result of a single column projection is a Series object
type(df['Name'])
```

[ ]:
```
# Since the result of using the indexing operator is either a DataFrame or␣
↪Series, you can chain
# operations together. For instance, we can select all of the rows which␣
↪related to school1 using
# .loc, then project the name column from just those rows
df.loc['school1']['Name']
```

[ ]:
```
# If you get confused, use type to check the responses from resulting␣
↪operations
print(type(df.loc['school1'])) #should be a DataFrame
print(type(df.loc['school1']['Name'])) #should be a Series
```

[ ]:
```
# Chaining, by indexing on the return type of another index, can come with some␣
↪costs and is
# best avoided if you can use another approach. In particular, chaining tends␣
↪to cause Pandas
# to return a copy of the DataFrame instead of a view on the DataFrame.
# For selecting data, this is not a big deal, though it might be slower than␣
↪necessary.
# If you are changing data though this is an important distinction and can be a␣
↪source of error.
```

[ ]:
```
# Here's another approach. As we saw, .loc does row selection, and it can take␣
↪two parameters,
# the row index and the list of column names. The .loc attribute also supports␣
↪slicing.

# If we wanted to select all rows, we can use a colon to indicate a full slice␣
↪from beginning to end.
# This is just like slicing characters in a list in python. Then we can add the␣
↪column name as the
# second parameter as a string. If we wanted to include multiple columns, we␣
↪could do so in a list.
# and Pandas will bring back only the columns we have asked for.
```

```
# Here's an example, where we ask for all the names and scores for all schools␣
↪using the .loc operator.
df.loc[:,['Name', 'Score']]
```

[ ]:
```
# Take a look at that again. The colon means that we want to get all of the␣
↪rows, and the list
# in the second argument position is the list of columns we want to get back
```

[ ]:
```
# That's selecting and projecting data from a DataFrame based on row and column␣
↪labels. The key
# concepts to remember are that the rows and columns are really just for our␣
↪benefit. Underneath
# this is just a two axes labeled array, and transposing the columns is easy.␣
↪Also, consider the
# issue of chaining carefully, and try to avoid it, as it can cause␣
↪unpredictable results, where
# your intent was to obtain a view of the data, but instead Pandas returns to␣
↪you a copy.
```

[ ]:
```
# Before we leave the discussion of accessing data in DataFrames, lets talk␣
↪about dropping data.
# It's easy to delete data in Series and DataFrames, and we can use the drop␣
↪function to do so.
# This function takes a single parameter, which is the index or row label, to␣
↪drop. This is another
# tricky place for new users -- the drop function doesn't change the DataFrame␣
↪by default! Instead,
# the drop function returns to you a copy of the DataFrame with the given rows␣
↪removed.

df.drop('school1')
```

[ ]:
```
# But if we look at our original DataFrame we see the data is still intact.
df
```

[ ]:
```
# Drop has two interesting optional parameters. The first is called inplace,␣
↪and if it's
# set to true, the DataFrame will be updated in place, instead of a copy being␣
↪returned.
# The second parameter is the axes, which should be dropped. By default, this␣
↪value is 0,
# indicating the row axis. But you could change it to 1 if you want to drop a␣
↪column.

# For example, lets make a copy of a DataFrame using .copy()
copy_df = df.copy()
# Now lets drop the name column in this copy
copy_df.drop("Name", inplace=True, axis=1)
```

```
copy_df
```

```
[ ]: # There is a second way to drop a column, and that's directly through the use␣
     ↪of the indexing
     # operator, using the del keyword. This way of dropping data, however, takes␣
     ↪immediate effect
     # on the DataFrame and does not return a view.
     del copy_df['Class']
     copy_df
```

```
[ ]: # Finally, adding a new column to the DataFrame is as easy as assigning it to␣
     ↪some value using
     # the indexing operator. For instance, if we wanted to add a class ranking␣
     ↪column with default
     # value of None, we could do so by using the assignment operator after the␣
     ↪square brackets.
     # This broadcasts the default value to the new column immediately.

     df['ClassRanking'] = None
     df
```

In this lecture you've learned about the data structure you'll use the most in pandas, the DataFrame. The dataframe is indexed both by row and column, and you can easily select individual rows and project the columns you're interested in using the familiar indexing methods from the Series class. You'll be gaining a lot of experience with the DataFrame in the content to come.