DataAnalysisLearning-Data Loading,Storage, and File Formeats

July 5, 2019

```
In [1]: # *-* coding: utf-8 -*-
        # @author: tongzi
        # @description: Data Loading, Storage, and File Formats in pandas
        # @created date: 2019/07/02
        # @last modification: 2019/07/02
```

```
In [3]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn as sns
```

0.0.1

Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. The table below summaries some of them, though *read_csv*() and *read_table*() are likely the ones we'll use the most.

Some of these functions, like *pd.read_csv*(), perform *type inference* (), because the column data types are not part of the data format. That means we don't have to specify which columns are numeric, boolean, integer, or string.

Since this is comma-delimited, we can use *read_csv*() to read it into a DataFrame:

```
In [12]: df = pd.read_csv('./examples/ex1.csv')
```

```
In [13]: df
```

```
Out[13]:
   a   b   c   d  message
0  1   2   3   4    hello
1  5   6   7   8    world
2  9  10  11  12      foo
```

We could also have used *read_table*() and specified the delimiter:

```
In [14]: pd.read_table('./examples/ex1.csv', sep=',')
```

```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: read_table is deprecated
  """Entry point for launching an IPython kernel.
```

Out[14]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Note that *read_table*() method is deprecated, we should use *read_csv*() instead.

A file will not always have a header row. Consider this file:

To read this file, we have a couple of options, we can allow pandas to specify the default column names or we can specify the names ourself:

```
In [16]: pd.read_csv('./examples/ex2.csv', header=None)
```

Out[16]:

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [17]: pd.read_csv('./examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

Out[17]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Suppose we want the 'message' column to be the index of the returned DataFrame. We can either indicate we want the column at index 4 or 'message' using the *index_col* argument:

```
In [18]: names = ['a', 'b', 'c', 'd', 'message']
```

```
In [19]: pd.read_csv('./examples/ex2.csv', names=names, index_col=[4])
```

Out[19]:

	a	b	c	d
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

or

```
In [20]: pd.read_csv('./examples/ex2.csv', names=names, index_col='message')
```

Out[20]:

	a	b	c	d
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

In the event that we want to form a hierarchical index from multiple columns, we can pass a list of column numbers or names:

```
In [22]: parsed = pd.read_csv('./examples/ex3.csv', index_col=[0, 1])

In [23]: parsed

Out[23]:            value1  value2
         key1 key2
         one  a           1       2
              b           3       4
              c           5       6
              d           7       8
         two  a           9      10
              b          11      12
              c          13      14
              d          15      16
```

Or

```
In [24]: pd.read_csv('./examples/ex3.csv', index_col=['key1', 'key2'])

Out[24]:            value1  value2
         key1 key2
         one  a           1       2
              b           3       4
              c           5       6
              d           7       8
         two  a           9      10
              b          11      12
              c          13      14
              d          15      16

In [25]: list(open('./examples/ext.txt'))

Out[25]: [' A B   C\n',
          'aaa  -0.264438  -1.026059  -0.619500\n',
          'bbb  0.927272  0.302904  -0.032399\n',
          'ccc  -0.264273  -0.386314    -0.217601\n',
          'ddd  -0.871858  -0.348382  1.100491\n']
```

ext.txt*read_table*()*sep*'+':

```
In [28]: result = pd.read_table('./examples/ext.txt', sep='\s+')

C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: read_table is deprecated
  """Entry point for launching an IPython kernel.


In [29]: result
```

```
Out[29]:             A         B         C
         aaa -0.264438 -1.026059 -0.619500
         bbb  0.927272  0.302904 -0.032399
         ccc -0.264273 -0.386314 -0.217601
         ddd -0.871858 -0.348382  1.100491
```

skiprows

```
In [30]: list(open('./examples/ex4.csv'))

Out[30]: ['# hey!\n',
          'a,b,c,d,message\n',
          '# just wanted to make things more difficult for you\n',
          '# who reads CSV files with computers, anyway?\n',
          '1,2,3,4,hello\n',
          '5,6,7,8,world\n',
          '9,10,11,12,foo']

In [31]: pd.read_csv('./examples/ex4.csv', skiprows=[0, 2, 3])

Out[31]:    a   b   c   d message
         0  1   2   3   4   hello
         1  5   6   7   8   world
         2  9  10  11  12     foo

In [33]: pd.read_csv('./examples/ex4.csv', skiprows=[0, 2, 3])

Out[33]:    a   b   c   d message
         0  1   2   3   4   hello
         1  5   6   7   8   world
         2  9  10  11  12     foo
```

Missing data is usually eithor not present (empty string) or marked by some sentinel value (). By default, pandas uses a set of commonly occurring sentinels, such as NA and NULL:

```
In [34]: list(open('./examples/ex5.csv'))

Out[34]: ['something,a,b,c,d,message\n',
          'one,1,2,3,4,NA\n',
          'two,5,6,,8,world\n',
          'three,9,10,11,12,foo']

In [35]: result = pd.read_csv('./examples/ex5.csv')

In [36]: result

Out[36]:   something  a   b     c   d message
         0       one  1   2   3.0   4     NaN
         1       two  5   6   NaN   8   world
         2     three  9  10  11.0  12     foo
```

```
In [37]: pd.isnull(result)

Out[37]:    something      a      b      c      d  message
         0      False  False  False  False  False     True
         1      False  False  False   True  False    False
         2      False  False  False  False  False    False
```

read_csv()na_values

```
In [38]: result = pd.read_csv('./examples/ex5.csv', na_values=['NULL'])

In [39]: result

Out[39]:   something  a   b     c   d message
         0       one  1   2   3.0   4     NaN
         1       two  5   6   NaN   8   world
         2     three  9  10  11.0  12     foo
```

>Different NA sentinels can be specified for each column in a dict.

```
In [40]: # 'message''foo''NA';
         # 'something''two'
         sentinels = {"message":['foo', 'NA'], 'something':['two']}

In [41]: pd.read_csv('./examples/ex5.csv', na_values=sentinels)

Out[41]:   something  a   b     c   d message
         0       one  1   2   3.0   4     NaN
         1       NaN  5   6   NaN   8   world
         2     three  9  10  11.0  12     NaN
```

'message''foo''NA';
'something''two'

read_csv()read_table()

0.0.2

Reading Text Files in Pieces

Before we look at a large file, we make the pandas display setting more compact ():

```
In [43]: pd.options.display.max_rows = 10 # 10

In [52]: result = pd.read_csv('./examples/ex6.csv')

In [53]: result
```

```
Out[53]:            one       two     three      four      five key
         0    -0.311751 -0.664187 -0.625109 -0.557133  0.776716   Q
         1    -0.791881  1.421330 -0.255119  1.623128  0.156707   A
         2     1.007681  0.666359  0.292274  0.541123  0.477363   F
         3    -0.918624  0.168383 -0.396418 -1.438362  1.613293   T
         4    -0.071876 -0.325441  0.165436  1.584977  1.024566   E
         ...        ...       ...       ...       ...       ...  ..
         9995 -1.331803  0.740650 -0.376655 -0.893378 -0.264872   W
         9996 -0.138084 -0.398566 -2.139323 -0.067138  0.530703   T
         9997  0.380273  0.775894  0.474533 -0.693687  0.923493   I
         9998 -1.299136 -0.030664 -0.756900 -1.399551  1.181465   V
         9999 -1.219521 -0.920930  0.032399 -0.650949  0.587994   V

         [10000 rows x 6 columns]
```

If we want only read a small number of rows (avoiding reading the entire file), specify that with *nrows*:

```
In [54]: # ex6.csv5
         pd.read_csv('./examples/ex6.csv', nrows=5)

Out[54]:         one       two     three      four      five key
         0 -0.311751 -0.664187 -0.625109 -0.557133  0.776716   Q
         1 -0.791881  1.421330 -0.255119  1.623128  0.156707   A
         2  1.007681  0.666359  0.292274  0.541123  0.477363   F
         3 -0.918624  0.168383 -0.396418 -1.438362  1.613293   T
         4 -0.071876 -0.325441  0.165436  1.584977  1.024566   E
```

To read a file in pieces, we can specify the *chunksize* options as a number of rows:

```
In [55]: # chunksize=1000read_csv()
         # 1000
         chunker = pd.read_csv('./examples/ex6.csv', chunksize=1000)

In [56]: chunker

Out[56]: <pandas.io.parsers.TextFileReader at 0x24ba382bb38>
```

chunksizeread_csv()TextParser*chunksize*chunker*key*

```
In [57]: tot = pd.Series([])
         for piece in chunker:
             tot = tot.add(piece['key'].value_counts(), fill_value=0)

In [59]: tot = tot.sort_values(ascending=False)

In [60]: tot[:10]
```

```
Out[60]: T    423.0
         A    414.0
         H    409.0
         K    396.0
         B    395.0
         O    394.0
         I    392.0
         G    390.0
         R    389.0
         W    388.0
         dtype: float64
```

TextParser*get_chunk*()

0.0.3

Writing Data to Text Format

Data can be exported to a delimited format. Let's consider one of the CSV file read before:

```
In [61]: data = pd.read_csv('./examples/ex5.csv')

In [62]: data

Out[62]:   something  a   b     c   d message
         0       one  1   2   3.0   4     NaN
         1       two  5   6   NaN   8   world
         2     three  9  10  11.0  12     foo
```

Using DataFrame's *to_csv*() method, we can write the data out to a comma/separated file:

```
In [63]: data.to_csv('./examples/out.csv')

In [64]: list(open('./examples/out.csv'))

Out[64]: [',something,a,b,c,d,message\n',
          '0,one,1,2,3.0,4,\n',
          '1,two,5,6,,8,world\n',
          '2,three,9,10,11.0,12,foo\n']
```

Other delimiters can be used, of course (writing to sys.stdout so it print the result to the console):

```
In [65]: import sys

In [66]: data.to_csv(sys.stdout, sep="|")

|something|a|b|c|d|message

0|one|1|2|3.0|4|
```

```
1|two|5|6||8|world

2|three|9|10|11.0|12|foo
```

As we can see above, missing values appear as empty strings in the output. We might want to denote them as some other sentinel values:

```
In [67]: # na_rep means NA replace
         data.to_csv(sys.stdout, na_rep='Null')

,something,a,b,c,d,message

0,one,1,2,3.0,4,Null

1,two,5,6,Null,8,world

2,three,9,10,11.0,12,foo
```

As we can see above, if with no other options specified, both the row and column labels are written. Both of these can be disabled:

```
In [68]: data.to_csv(sys.stdout, index=False, header=False)

one,1,2,3.0,4,

two,5,6,,8,world

three,9,10,11.0,12,foo
```

We can also write only a subset of the columns, and in an order of our choosing:

```
In [69]: data.to_csv(sys.stdout, index=False, columns=['d', 'a', 'b'])

d,a,b

4,1,2

8,5,6

12,9,10
```

Series also has a *to_csv*() method:

```
In [70]: dates = pd.date_range('7/3/2019', periods=10)

In [71]: dates

Out[71]: DatetimeIndex(['2019-07-03', '2019-07-04', '2019-07-05', '2019-07-06',
                        '2019-07-07', '2019-07-08', '2019-07-09', '2019-07-10',
                        '2019-07-11', '2019-07-12'],
                       dtype='datetime64[ns]', freq='D')

In [72]: ts = pd.Series(np.arange(10), index=dates)

In [73]: ts

Out[73]: 2019-07-03    0
         2019-07-04    1
         2019-07-05    2
         2019-07-06    3
         2019-07-07    4
         2019-07-08    5
         2019-07-09    6
         2019-07-10    7
         2019-07-11    8
         2019-07-12    9
         Freq: D, dtype: int32

In [79]: # header
         ts.to_csv('./examples/tseries.csv', header=False)

In [80]: list(open('./examples/tseries.csv'))

Out[80]: ['2019-07-03,0\n',
          '2019-07-04,1\n',
          '2019-07-05,2\n',
          '2019-07-06,3\n',
          '2019-07-07,4\n',
          '2019-07-08,5\n',
          '2019-07-09,6\n',
          '2019-07-10,7\n',
          '2019-07-11,8\n',
          '2019-07-12,9\n']
```

0.0.4 Working with Delimited Formats

```
In [81]: list(open('./examples/ex7.csv'))

Out[81]: ['"a","b","c"\n', '"1","2","3"\n', '"1","2","3"']

In [83]: pd.read_csv('./examples/ex7.csv', header=None)
```

```
Out[83]:    0  1  2
         0  a  b  c
         1  1  2  3
         2  1  2  3
```

pd.read_csv()Pythoncsv

```
In [84]: import csv

In [85]: f = open('./examples/ex7.csv')

In [86]: reader = csv.reader(f)

In [87]: for line in reader:
             print(line)

['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

From there, it is up to you to do the wrangling necessary to put the data in the form that you need. Let's take this step by step. First, we read the file into a list of lines:

```
In [88]: with open('./examples/ex7.csv') as f:
             lines = list(csv.reader(f))
```

Then, split the lines into the header line and the data lines:

```
In [89]: header, values = lines[0], lines[1:]

In [90]: header

Out[90]: ['a', 'b', 'c']

In [91]: values

Out[91]: [['1', '2', '3'], ['1', '2', '3']]

In [93]: list(zip(*values))

Out[93]: [('1', '1'), ('2', '2'), ('3', '3')]
```

Then create a dictionary of data columns using a dictionary conprehension and the expression zip(*values), which transpose the rows to columns:

```
In [94]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [95]: data_dict

Out[95]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSVcsv.DialectCSV

```
In [96]: class my_dialect(csv.Dialect):
             lineterminator = '\n'  #
             delimiter = ';'        #
             quotechar = '"'        #
             quoting = csv.QUOTE_MINIMAL
```

```
In [ ]: #
        reader = csv.reader(f, dialect=my_dialect)
```

csv.Dialect

```
In [ ]: #
        reader = csv.reader(f, delimiter='|')
```

csv.Dialect

*csv*string*split*()*re.split()*.

csv.writer()*csv.reader*()*dialect*

```
In [100]: with open('./examples/mydata.csv', 'w') as f:
              writer = csv.writer(f, dialect=my_dialect)
              writer.writerow(('one', 'two', 'three'))
              writer.writerow(('1', '2', '3'))
              writer.writerow(('4', '5', '6'))
              writer.writerow(('71', '82', '93'))
```

```
In [101]: list(open('./examples/mydata.csv'))
```

```
Out[101]: ['one;two;three\n', '1;2;3\n', '4;5;6\n', '71;82;93\n']
```

0.0.5 JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is much more free-from data format than a tabular text form like CSV.

Here is am example:

```
In [102]: obj = """
          {"name": "Wes",
          "places_lived": ["United States", "Spain", "Germany"],
          "pet": null,
          "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
          {"name": "Katie", "age": 38,
          "pets": ["Sixes", "Stache", "Cisco"]}]
          }
          """
```

JSON(dict)(list) >JSON!!!!

```
In [103]: # PythonjsonJSON
          import json
```

To convert a json string to Python form, use *json.loads*().

```
In [109]: result = json.loads(obj)

In [110]: result

Out[110]: {'name': 'Wes',
           'places_lived': ['United States', 'Spain', 'Germany'],
           'pet': None,
           'siblings': [{'name': 'Scott', 'age': 30, 'pets': ['Zeus', 'Zuko']},
            {'name': 'Katie', 'age': 38, 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

json.dumps(), on the other hand, converts a Python object back to JSON:

```
In [111]: asjson = json.dumps(result)

In [112]: asjson

Out[112]: '{"name": "Wes", "places_lived": ["United States", "Spain", "Germany"], "pet": null,

In [113]: type(result)

Out[113]: dict

In [115]: siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])

In [116]: siblings

Out[116]:     name  age
          0  Scott   30
          1  Katie   38
```

pandas.read_json()JSONSeriesDataFrame

```
In [117]: list(open('./examples/example.json'))

Out[117]: ['[{"a": 1, "b": 2, "c": 3},\n',
           '{"a": 4, "b": 5, "c": 6},\n',
           '{"a": 7, "b": 8, "c": 9}]']
```

pandas.read_json()JSON

```
In [118]: data = pd.read_json('./examples/example.json')

In [119]: data
```

```
Out[119]:    a  b  c
          0  1  2  3
          1  4  5  6
          2  7  8  9

In [120]: print(data.to_json())

{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}


In [121]: print(data.to_json(orient='records'))

[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

0.0.6 XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML files. Examples include **lxml**, **Beautiful Soup**, and **html5lib**. While **lxml** is comparatively much faster in general, the other libraries can better handle malformed HTML or XML files.

pandas.read_html()html

DataFrame:

```
In [122]: tables = pd.read_html('./examples/fdic_failed_bank_list.html')

In [123]: len(tables)

Out[123]: 1

In [124]: failures =  tables[0]

In [125]: failures.head()

Out[125]:                                           Bank Name        City  ST   CERT  \
          0                            The Enloe State Bank      Cooper  TX  10716
          1               Washington Federal Bank for Savings     Chicago  IL  30570
          2   The Farmers and Merchants State Bank of Argonia     Argonia  KS  17719
          3                              Fayette County Bank  Saint Elmo  IL   1802
          4  Guaranty Bank, (d/b/a BestBank in Georgia & Mi...   Milwaukee  WI  30003

                        Acquiring Institution       Closing Date       Updated Date
          0                  Legend Bank, N. A.       May 31, 2019      June 18, 2019
          1                  Royal Savings Bank  December 15, 2017   February 1, 2019
          2                        Conway Bank   October 13, 2017  February 21, 2018
          3           United Fidelity Bank, fsb       May 26, 2017   January 29, 2019
          4  First-Citizens Bank & Trust Company        May 5, 2017     March 22, 2018
```

From here, we can do proceed to do some data cleaning and analysis, like computing the number of bank failures by year:

```
In [126]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [128]: close_timestamps

Out[128]: 0    2019-05-31
          1    2017-12-15
          2    2017-10-13
          3    2017-05-26
          4    2017-05-05
                  ...
          20   2015-01-23
          21   2015-01-16
          22   2014-12-19
          23   2014-11-07
          24   2014-10-24
          Name: Closing Date, Length: 25, dtype: datetime64[ns]

In [127]: close_timestamps.dt.year.value_counts()

Out[127]: 2015    8
          2017    8
          2016    5
          2014    3
          2019    1
          Name: Closing Date, dtype: int64
```

0.0.7 Parsing XML with lxml.objectify

```
In [130]: from lxml import objectify

In [131]: path = './examples/performance_mnr.xml'

In [132]: parsed = objectify.parse(open(path))

In [133]: root = parsed.getroot() #

In [136]: from io import StringIO

In [137]: tag = '<a href="https://www.biying.org">biying</a>'

In [138]: root = objectify.parse(StringIO(tag)).getroot()

In [140]: root

Out[140]: <Element a at 0x24ba65247c8>

In [141]: root.get('href')

Out[141]: 'https://www.biying.org'

In [142]: root.text

Out[142]: 'biying'
```

0.0.8 Binary Data Formats

One of the easiest way to store data (also known as serialization) in binary format is using Python built-in **pickle** serialization. pandas objects all have a *to_pickle*() method that writes the data to disk in pickle format:

```
In [143]: frame = pd.read_csv('./examples/ex1.csv')

In [144]: frame

Out[144]:    a   b   c   d  message
          0  1   2   3   4    hello
          1  5   6   7   8    world
          2  9  10  11  12      foo

In [145]: frame.to_pickle('./examples/frame_pickle')
```

We can read any 'pickle' object stored in file by using the built-in pickle directly, or even more conveniently using *pandas.read_pickle*():

```
In [146]: pd.read_pickle('./examples/frame_pickle')

Out[146]:    a   b   c   d  message
          0  1   2   3   4    hello
          1  5   6   7   8    world
          2  9  10  11  12      foo
```

picklepicklebackward compatibility

0.0.9 Using HDF5 Format

HDF5 is a well-regarded file format intenden for storing large quantities of scientific array data. The "HDF" stands for *hierarchical data format*. HDF5 can be a good choice for working with very large datasets that don't fit into memory, as we can efficiently read and write small section of much larger arrays.

PyTablesh5pyHDF5**pandas**SeriesDataFrame*pandas.HDFStore*()(dict)

```
In [148]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [149]: store = pd.HDFStore('./examples/mydata.h5')

C:\Anaconda3\lib\importlib\_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may inc
  return f(*args, **kwds)


In [150]: store['obj1'] = frame

In [151]: store['obj1_col'] = frame['a']

In [152]: store
```

```
Out[152]: <class 'pandas.io.pytables.HDFStore'>
          File path: ./examples/mydata.h5

In [153]: store['obj1']

Out[153]:            a
          0  -0.548035
          1  -0.710008
          2  -1.331906
          3   1.015062
          4  -0.355029
          ..       ...
          95 -1.084802
          96 -0.092428
          97 -1.775375
          98  2.008725
          99 -0.257653

          [100 rows x 1 columns]
```

HDFStore supports two storage formats: 'fix' and 'table'. The latter is generally slower, but it supports query operations using a special syntax:

```
In [154]: store.put('obj2', frame, format='table')

In [155]: store.select('obj2', where=['index >= 10 and index <= 20'])

Out[155]:            a
          10 -0.000648
          11  1.253861
          12 -0.157897
          13 -1.138644
          14 -0.731594
          ..       ...
          16  1.355059
          17  0.856581
          18 -0.294098
          19  1.970789
          20  0.945458

          [11 rows x 1 columns]
```

store*put*()store['obj2']=frame

The *pandas.read_hdf*() function gives us a shortcut to these tools:

```
In [158]: frame.to_hdf("./examples/mydata2.h5", 'obj3', format='table')

In [160]: pd.read_hdf('./examples/mydata2.h5', 'obj3', where=['index <= 10'])
```

```
Out[160]:           a
          0  -0.548035
          1  -0.710008
          2  -1.331906
          3   1.015062
          4  -0.355029
          ..       ...
          6   0.132335
          7  -0.287310
          8   0.128023
          9  -0.407593
          10 -0.000648

          [11 rows x 1 columns]
```

0.0.10 Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either **ExcelFile** class or *pandas.read_excel*() function. Internally, these tools use the add-on packages **xlrd** and **openpyxl** to read XLS and XLSX files, respectively.

To use ExcelFile, create an instance by passing a path to an xls or xlsx file:

```
In [168]: path_base = './examples/'

In [163]: xlsx = pd.ExcelFile('./examples/test.xlsx')
```

Data stored in a sheet can then be read into a DataFrame with parse:

```
In [167]: pd.read_excel(xlsx, 'Sheet3', header=None)

Out[167]:     0   1          2          3          4          5          6          7   \
          0   17  53 -11.365891 -11.652242 -12.048091 -12.541557 -13.097980 -13.739284
          1   17  52 -11.364234 -11.646589 -12.052842 -12.538041 -13.098351 -13.742386
          2   17  51 -11.370131 -11.657349 -12.048091 -12.540817 -13.098720 -13.740379
          3   17  50 -11.364417 -11.651693 -12.045170 -12.525095 -13.105573 -13.739830
          4   17  49 -11.367733 -11.646589 -12.052110 -12.539152 -13.094093 -13.751884
          ..  ..  ..        ...        ...        ...        ...        ...        ...
          12  17  41 -11.365154 -11.652787 -12.041155 -12.528053 -13.097795 -13.746770
          13  17  40 -11.365707 -11.657895 -12.051013 -12.532676 -13.092058 -13.744030
          14  17  39 -11.368103 -11.656984 -12.051013 -12.531935 -13.089468 -13.741291
          15  17  38 -11.369024 -11.662276 -12.050467 -12.534155 -13.082629 -13.756273
          16  17  37 -11.365891 -11.654065 -12.047178 -12.517706 -13.096498 -13.740196

                    8          9          10         11
          0  -14.490947 -15.435276 -16.384909 -17.424860
          1  -14.485475 -15.449348 -16.386923 -17.427637
          2  -14.487480 -15.439479 -16.393703 -17.439686
          3  -14.487662 -15.442402 -16.383993 -17.419119
          4  -14.492771 -15.442766 -16.385824 -17.447666
```

```
..         ...         ...         ...         ...
12 -14.486569 -15.452276 -16.382345 -17.434866
13 -14.496239 -15.442766 -16.389854 -17.427082
14 -14.488575 -15.439479 -16.378319 -17.437090
15 -14.492042 -15.438199 -16.379051 -17.435791
16 -14.488939 -15.450629 -16.393337 -17.440985

[17 rows x 12 columns]
```

If we are reading multiple sheets in a file, then it is faster to create the ExcelFile, but we can also simply pass the filename to *pandas.read_excel*():

```
In [173]: frame = pd.read_excel(path_base + 'test.xlsx',
                                'Sheet3', header=None)

In [174]: frame

Out[174]:     0   1          2          3          4          5          6          7   \
          0   17  53 -11.365891 -11.652242 -12.048091 -12.541557 -13.097980 -13.739284
          1   17  52 -11.364234 -11.646589 -12.052842 -12.538041 -13.098351 -13.742386
          2   17  51 -11.370131 -11.657349 -12.048091 -12.540817 -13.098720 -13.740379
          3   17  50 -11.364417 -11.651693 -12.045170 -12.525095 -13.105573 -13.739830
          4   17  49 -11.367733 -11.646589 -12.052110 -12.539152 -13.094093 -13.751884
          ..  ..  ..         ...         ...         ...         ...         ...         ...
          12  17  41 -11.365154 -11.652787 -12.041155 -12.528053 -13.097795 -13.746770
          13  17  40 -11.365707 -11.657895 -12.051013 -12.532676 -13.092058 -13.744030
          14  17  39 -11.368103 -11.656984 -12.051013 -12.531935 -13.089468 -13.741291
          15  17  38 -11.369024 -11.662276 -12.050467 -12.534155 -13.082629 -13.756273
          16  17  37 -11.365891 -11.654065 -12.047178 -12.517706 -13.096498 -13.740196

                     8          9          10         11
          0  -14.490947 -15.435276 -16.384909 -17.424860
          1  -14.485475 -15.449348 -16.386923 -17.427637
          2  -14.487480 -15.439479 -16.393703 -17.439686
          3  -14.487662 -15.442402 -16.383993 -17.419119
          4  -14.492771 -15.442766 -16.385824 -17.447666
          ..         ...         ...         ...         ...
          12 -14.486569 -15.452276 -16.382345 -17.434866
          13 -14.496239 -15.442766 -16.389854 -17.427082
          14 -14.488575 -15.439479 -16.378319 -17.437090
          15 -14.492042 -15.438199 -16.379051 -17.435791
          16 -14.488939 -15.450629 -16.393337 -17.440985

          [17 rows x 12 columns]
```

To write pandas data to Excel format, we must first create a ExcelWriter, then write data to it using pandas objects' *to_excel*():

```
In [175]: df = pd.DataFrame(np.random.randint(1, 100, size=(8,4)))
```

```
In [176]: df
```

```
Out[176]:     0   1   2   3
          0  68  18  79   3
          1  29  75  68  40
          2  14  84  36  69
          3  54  53  15  70
          4  44   2  10  48
          5  82  93   9  11
          6  30  39  75  19
          7  22  30  40  88
```

```
In [177]: writer = pd.ExcelWriter(path_base + 'df.xlsx')

In [178]: df.to_excel(writer, 'df')

In [179]: writer.save() # must call save() method
```

We can also pass a file path to *pandas.to_excel*() and avoid the ExcelWriter:

```
In [180]: df.to_excel(path_base + 'df2.xlsx')
```

0.0.11 Interacting with Web API

Many websites have public APIs providing data feeds via JSON or some other format. There are a number of ways to access these APIs from Python; one easy-to-use method that I recommend is the **requests** package.

```
In [181]: import requests

In [182]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'

In [183]: resp = requests.get(url)

In [185]: resp
```

```
Out[185]: <Response [200]>
```

The Response objects' *json*() method will return a dictionary containing JSON parsed into native python objects:

```
In [186]: data = resp.json()

In [189]: len(data)
```

```
Out[189]: 30
```

```
In [193]: data[0]['milestone']
```

```
Out[193]: {'url': 'https://api.github.com/repos/pandas-dev/pandas/milestones/61',
           'html_url': 'https://github.com/pandas-dev/pandas/milestone/61',
           'labels_url': 'https://api.github.com/repos/pandas-dev/pandas/milestones/61/labels'
           'id': 3759483,
           'node_id': 'MDk6TWlsZXN0b25lMzc1OTQ4Mw==',
           'number': 61,
           'title': '0.25.0',
           'description': '',
           'creator': {'login': 'jreback',
            'id': 953992,
            'node_id': 'MDQ6VXNlcjk1Mzk5Mg==',
            'avatar_url': 'https://avatars2.githubusercontent.com/u/953992?v=4',
            'gravatar_id': '',
            'url': 'https://api.github.com/users/jreback',
            'html_url': 'https://github.com/jreback',
            'followers_url': 'https://api.github.com/users/jreback/followers',
            'following_url': 'https://api.github.com/users/jreback/following{/other_user}',
            'gists_url': 'https://api.github.com/users/jreback/gists{/gist_id}',
            'starred_url': 'https://api.github.com/users/jreback/starred{/owner}{/repo}',
            'subscriptions_url': 'https://api.github.com/users/jreback/subscriptions',
            'organizations_url': 'https://api.github.com/users/jreback/orgs',
            'repos_url': 'https://api.github.com/users/jreback/repos',
            'events_url': 'https://api.github.com/users/jreback/events{/privacy}',
            'received_events_url': 'https://api.github.com/users/jreback/received_events',
            'type': 'User',
            'site_admin': False},
           'open_issues': 24,
           'closed_issues': 1131,
           'state': 'open',
           'created_at': '2018-10-23T02:34:15Z',
           'updated_at': '2019-07-03T12:14:44Z',
           'due_on': '2019-07-01T07:00:00Z',
           'closed_at': None}
```

With a bit of elbow grease (), we can create some high-level interfaces to common web APIs that return DataFrame objects for easy analysis.

0.0.12 Interacting with Databases

Loading data from SQL into a DataFrame is fairly straightforward, and pandas has some functions to simplify the process. As a example, I'll create a SQLite database using Python's built-in **sqlite3** driver:

```
In [194]: import sqlite3

In [195]: query = """
          create table tb_test
            (
            a varchar(20),
```

```
            b varchar(20),
            c real,
            d integer

            );
          """
```

```
In [196]: # mydata.sqlite
          con = sqlite3.connect(path_base + 'mydata.sqlite')

In [197]: # SQL
          con.execute(query)

Out[197]: <sqlite3.Cursor at 0x24ba91df2d0>

In [198]: #
          con.commit()
```

Then insert a few rows of data:

```
In [199]: data = [('Atlanta', 'Georgia', 1.25, 6),
          .....: ('Tallahassee', 'Florida', 2.6, 3),
          .....: ('Sacramento', 'California', 1.7, 5)]

In [200]: # ?
          stmt = "insert into tb_test values(?, ?, ?, ?)"

In [201]: #
          con.executemany(stmt, data)

Out[201]: <sqlite3.Cursor at 0x24ba91df420>

In [202]: #
          con.commit()
```

Most Python SQL drivers (PyODBC, psycopg2, MySQLdb, pymssql, etc) return a list of tuples when selecting data fram table:

```
In [203]: cusor = con.execute('select * from tb_test;')

In [204]: rows = cusor.fetchall()

In [205]: rows

Out[205]: [('Atlanta', 'Georgia', 1.25, 6),
           ('Tallahassee', 'Florida', 2.6, 3),
           ('Sacramento', 'California', 1.7, 5)]

In [206]: cusor.description
```

```
Out[206]: (('a', None, None, None, None, None, None),
           ('b', None, None, None, None, None, None),
           ('c', None, None, None, None, None, None),
           ('d', None, None, None, None, None, None))
```

```
In [207]: pd.DataFrame(rows, columns = [x[0] for x in cusor.description])
```

```
Out[207]:
          a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

The **SQLAlchemy** project is a popular Python SQL toolkit that abstracts away many of the common differences between SQL databases. pandas has *read_sql*() method that enables us to read data easily from a general SQLAlchemy connection. Here, we'll connect to a MySQL database with SQLAlchemy and read data from the table created before:

```
In [208]: import sqlalchemy as sqla
```

```
In [215]: db = sqla.create_engine("mysql+pymysql://root:2015201315@localhost:3306/db_rohdeschwa
                                  echo=False)
```

create_engine("mysql+pymysql://root:2015201315@localhost:3306/db_rohdeschwarzesrp_work",echo=Tr
"mysql+pymysql://root:2015201315@localhost:3306/db_rohdeschwarzesrp_work""+://:@:/"
echo=True,SQL

```
In [216]: pd.read_sql('select * from tb_esrp_2019_03_01_15_06_27',
                      db)
```

```
Out[216]:
          ID   X   Y  Z                                                FDATA
0      34997  10   7  0  [84.242828369140625,68.662445068359375,69.3799...
1      34998  10   8  0  [103.77669525146484,67.675567626953125,72.5693...
2      34999  10   9  0  [98.51885986328125,69.135581970214844,67.95497...
3      35000  10  10  0  [99.857666015625,68.413177490234375,69.7382202...
4      35001  10  11  0  [101.33040618896484,68.914802551269531,67.6216...
...      ...  ..  .. ..                                                ...
1979   36976  73  11  0  [101.64801025390625,68.587387084960938,71.5124...
1980   36977  73  10  0  [101.52200317382812,69.431022644042969,68.9145...
1981   36978  73   9  0  [101.53518676757812,68.680587768554687,71.2132...
1982   36979  73   8  0  [101.14675903320312,69.554557800292969,68.8946...
1983   36980  73   7  0  [101.58643341064453,68.435829162597656,69.8215...

[1984 rows x 5 columns]
```

!