16.1_pandas

May 2, 2023

1 Introduction to Python for Open Source Geocomputation

- Instructor: Dr. Wei Kang
- Class Location and Time: ENV 336, Mon & Wed 12:30 pm - 1:50 pm

Content:

- what is pandas?
- data processing
- data exploration
- read and save data

2 What is Pandas?

- Pandas is a Python library for conducting data analysis.
- First release was in 2010
- The Pandas name itself is derived from panel data, an econometrics term for multidimensional structured datasets, and a play on the phrase Python data analysis.
- Pandas provides high-level data structures and functions designed to make working with structured or tabular data intuitive and flexible.
- contains data structures and data manipulation tools designed to make data cleaning and analysis fast and convenient in Python.
- Works with **structured data**:
  - Tabular or spreadsheet-like data in which each column may be a different type (string, numeric, date, or otherwise). This includes most kinds of data commonly stored in relational databases or tab- or comma-delimited text files.
  - Multidimensional arrays (matrices).

    - Multiple tables of data interrelated by key columns (what would be primary or foreign keys for a SQL user).
    - Evenly or unevenly spaced time series.

2.1 Installation of Pandas

From a terminal:

`pip install pandas`

or

`conda install pandas`

`pandas` is included in conda installation, so our working environment should already have `pandas` installed.

[1]:
```python
import pandas as pd
```

2.2 Core of Pandas: `DataFrame`

- The pandas `DataFrame` is a data structure that contains **two-dimensional** data and its corresponding row and column labels.
- Pandas blends the array-computing ideas of NumPy with the kinds of data manipulation capabilities found in spreadsheets and relational databases (such as SQL).
- `DataFrames` are widely used in data science, machine learning, scientific computing, and many other data-intensive fields.
- `DataFrames` are similar to SQL tables or the spreadsheets in Excel.
- In many cases, DataFrames are faster, easier to use, and more powerful than tables or spreadsheets because they're an integral part of the Python and NumPy ecosystems.

2.2.1 What is a Pandas `DataFrame`?

- Represents a rectangular table of data
- Contains an ordered, named collection of columns, each of which can be a different value type (numeric, string, Boolean, etc.)
- Has both a row and column index
- Can be thought of as a dictionary of `Series` all sharing the same index.

2.2.2 Creating a Pandas `DataFrame`

- Creating from a **dictionary** of **equal-length** lists or NumPy arrays
    - key is used as the column name (string)
    - value (**equal-length** lists or NumPy arrays) is used as the records
    - The resulting `DataFrame` will have its index assigned automatically
    - The columns are placed according to the order of the keys in data

  `pd.DataFrame(dict)`
- Creating from nested lists (sublists need to be **equal-length**) or a two-dimensional NumPy array
    - Column and row names can be specicified

pd.DataFrame(array/nested lists, index= list, columns=list)

[2]:
```
import numpy as np
```

[3]:
```
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]}
```

[4]:
```
data
```

[4]:
```
{'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]}
```

[5]:
```
frame = pd.DataFrame(data)
frame
```

[5]:

	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

[6]:
```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2],
        "year": [2000, 2001, 2002, 2001, 2002, 2003]
        }
frame = pd.DataFrame(data)
frame
```

[6]:

	state	pop	year
0	Ohio	1.5	2000
1	Ohio	1.7	2001
2	Ohio	3.6	2002
3	Nevada	2.4	2001
4	Nevada	2.9	2002
5	Nevada	3.2	2003

We can specify the order of the DataFrame's columns during the creation phase

[7]:
```
frame = pd.DataFrame(data, columns=["year", "state", "pop"])
frame
```

[7]:

	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
5  2003  Nevada  3.2
```

[8]:
```
frame = pd.DataFrame(data, columns=["year","state"])
frame
```

[8]:
```
   year   state
0  2000    Ohio
1  2001    Ohio
2  2002    Ohio
3  2001  Nevada
4  2002  Nevada
5  2003  Nevada
```

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

[9]:
```
frame = pd.DataFrame(data, columns=["year", "state", "pop", "poverty"])
frame
```

[9]:
```
   year   state  pop poverty
0  2000    Ohio  1.5     NaN
1  2001    Ohio  1.7     NaN
2  2002    Ohio  3.6     NaN
3  2001  Nevada  2.4     NaN
4  2002  Nevada  2.9     NaN
5  2003  Nevada  3.2     NaN
```

[10]:
```
frame.poverty
```

[10]:
```
0    NaN
1    NaN
2    NaN
3    NaN
4    NaN
5    NaN
Name: poverty, dtype: object
```

[11]:
```
frame.poverty = 0.5
```

[12]:
```
frame
```

[12]:
```
   year   state  pop  poverty
0  2000    Ohio  1.5      0.5
1  2001    Ohio  1.7      0.5
2  2002    Ohio  3.6      0.5
```

```
3  2001  Nevada  2.4      0.5
4  2002  Nevada  2.9      0.5
5  2003  Nevada  3.2      0.5
```

[13]:
```
type(frame)
```

[13]:
```
pandas.core.frame.DataFrame
```

Group exercise Create a pandas `DataFrame` using the four `array` variables. The `DataFrame` will have four columns with names `population`, `ward`, `year` and `poverty`:

```
ward = np.tile([1,2,3,4,5], 5)
year = np.array([2000] * 5 +[2001] * 5 + [2002] * 5 + [2003] * 5 + [2004] * 5)
population = np.random.randint(5000, size=(25,))
poverty = np.random.random(size=(25,))
```

Raise your hand when you are done!

[14]:
```
ward = np.tile([1,2,3,4,5], 5)
year = np.array([2000] * 5 +[2001] * 5 + [2002] * 5 + [2003] * 5 + [2004] * 5)
population = np.random.randint(5000, size=(25,))
poverty = np.random.random(size=(25,))
```

[15]:
```
dict_data = {"ward":ward , "year":year, "population": population,
             "poverty": poverty}
df = pd.DataFrame(dict_data)
df
```

[15]:

	ward	year	population	poverty
0	1	2000	1990	0.837070
1	2	2000	4139	0.763842
2	3	2000	4055	0.592783
3	4	2000	2886	0.896657
4	5	2000	2410	0.905181
5	1	2001	265	0.784550
6	2	2001	3450	0.906884
7	3	2001	3721	0.927134
8	4	2001	1247	0.001118
9	5	2001	1250	0.603424
10	1	2002	1101	0.543543
11	2	2002	3263	0.241741
12	3	2002	911	0.441993
13	4	2002	184	0.999512
14	5	2002	558	0.568797
15	1	2003	1495	0.282474
16	2	2003	2025	0.557792
17	3	2003	1208	0.466008
18	4	2003	392	0.463173

```
19     5  2003        2772  0.119412
20     1  2004        2453  0.912898
21     2  2004        4913  0.627135
22     3  2004         638  0.597489
23     4  2004         527  0.309552
24     5  2004        3057  0.026273
```

[16]:
```
ward = np.tile([1,2,3,4,5], 5) # 5 wards repeat 5 times
```

[17]:
```
ward
```

[17]:
```
array([1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2,
       3, 4, 5])
```

[18]:
```
[1,2,3,4,5]*5
```

[18]:
```
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

[19]:
```
np.array([1,2,3,4,5]*5)
```

[19]:
```
array([1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2,
       3, 4, 5])
```

[20]:
```
year = np.array([2000] * 5 +[2001] * 5 + [2002] * 5 + [2003] * 5 + [2004] * 5)␣
 ↪# 2000, 2001, 2002, 2003, 2004 each for 5 times (5 wards)
year
```

[20]:
```
array([2000, 2000, 2000, 2000, 2000, 2001, 2001, 2001, 2001, 2001, 2002,
       2002, 2002, 2002, 2002, 2003, 2003, 2003, 2003, 2003, 2004, 2004,
       2004, 2004, 2004])
```

[21]:
```
np.random.randint?
```

[22]:
```
population = np.random.randint(5000, size=(25,))
```

[23]:
```
population
```

[23]:
```
array([2724, 4117, 4653, 4770, 2450, 1257, 2199, 2674, 3612, 3870, 1396,
       3380, 4194, 3948, 2879, 4336,  136,  221, 3441,   28,  432,  250,
       4097, 2418,  504])
```

[24]:
```
poverty = np.random.random(size=(25,))
```

[25]:
```
poverty
```

[25]:
```
array([0.21585982, 0.67015997, 0.95536642, 0.96728885, 0.2449112 ,
       0.47637213, 0.59569255, 0.93357268, 0.26203938, 0.34815188,
```

```
        0.15777429, 0.05664675, 0.66245006, 0.21250013, 0.02524779,
        0.19513812, 0.27977169, 0.79045387, 0.24273107, 0.80645431,
        0.97210768, 0.01545941, 0.27633935, 0.25991025, 0.67491649])
```

[26]:
```
df_ward = pd.DataFrame({'population': population,
                        'ward': ward,
                        'poverty': poverty,
                         'year': year})
df_ward
```

[26]:

	population	ward	poverty	year
0	2724	1	0.215860	2000
1	4117	2	0.670160	2000
2	4653	3	0.955366	2000
3	4770	4	0.967289	2000
4	2450	5	0.244911	2000
5	1257	1	0.476372	2001
6	2199	2	0.595693	2001
7	2674	3	0.933573	2001
8	3612	4	0.262039	2001
9	3870	5	0.348152	2001
10	1396	1	0.157774	2002
11	3380	2	0.056647	2002
12	4194	3	0.662450	2002
13	3948	4	0.212500	2002
14	2879	5	0.025248	2002
15	4336	1	0.195138	2003
16	136	2	0.279772	2003
17	221	3	0.790454	2003
18	3441	4	0.242731	2003
19	28	5	0.806454	2003
20	432	1	0.972108	2004
21	250	2	0.015459	2004
22	4097	3	0.276339	2004
23	2418	4	0.259910	2004
24	504	5	0.674916	2004

Creating a pandas dataframe from a matrix/two-dimensional array

[27]:
```
data = np.arange(16).reshape((4, 4))
data
```

[27]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[28]:
```python
df_state = pd.DataFrame(data,
                        index=["Ohio", "Colorado", "Utah", "New York"],
                        columns=["one", "two", "three", "four"])
```

[29]:
```python
df_state
```

[29]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

2.3 Exploring data with Pandas

[30]:
```python
df_ward
```

[30]:

	population	ward	poverty	year
0	2724	1	0.215860	2000
1	4117	2	0.670160	2000
2	4653	3	0.955366	2000
3	4770	4	0.967289	2000
4	2450	5	0.244911	2000
5	1257	1	0.476372	2001
6	2199	2	0.595693	2001
7	2674	3	0.933573	2001
8	3612	4	0.262039	2001
9	3870	5	0.348152	2001
10	1396	1	0.157774	2002
11	3380	2	0.056647	2002
12	4194	3	0.662450	2002
13	3948	4	0.212500	2002
14	2879	5	0.025248	2002
15	4336	1	0.195138	2003
16	136	2	0.279772	2003
17	221	3	0.790454	2003
18	3441	4	0.242731	2003
19	28	5	0.806454	2003
20	432	1	0.972108	2004
21	250	2	0.015459	2004
22	4097	3	0.276339	2004
23	2418	4	0.259910	2004
24	504	5	0.674916	2004

[31]:
```python
df_ward.head() # first 5 rows
```

[31]:

	population	ward	poverty	year
0	2724	1	0.215860	2000
1	4117	2	0.670160	2000
2	4653	3	0.955366	2000
3	4770	4	0.967289	2000
4	2450	5	0.244911	2000

[32]:
```
df_ward.head(2) # first 2 rows
```

[32]:

	population	ward	poverty	year
0	2724	1	0.21586	2000
1	4117	2	0.67016	2000

[33]:
```
df_ward.tail() # last 5 rows
```

[33]:

	population	ward	poverty	year
20	432	1	0.972108	2004
21	250	2	0.015459	2004
22	4097	3	0.276339	2004
23	2418	4	0.259910	2004
24	504	5	0.674916	2004

[34]:
```
df_ward.tail(2) # last 2 rows
```

[34]:

	population	ward	poverty	year
23	2418	4	0.259910	2004
24	504	5	0.674916	2004

[35]:
```
df_ward.columns
```

[35]: Index(['population', 'ward', 'poverty', 'year'], dtype='object')

[36]:
```
df_ward.shape
```

[36]: (25, 4)

[37]:
```
len(df_ward)
```

[37]: 25

[38]:
```
df_ward.shape[0]
```

[38]: 25

[39]:
```
df_ward.shape[1]
```

[39]: 4

2.4 Indexing DataFrame

- indexing columns
- indexing rows
  - works analogously to NumPy array indexing (integer indexing)
    - `iloc`: integer-based indexing.
  - you can use the index values instead of only integers
    - `loc`: label-based indexing

[40]:
```
df_state = pd.DataFrame(data,
                        index=["Ohio", "Colorado", "Utah", "New York"],
                        columns=["one", "two", "three", "four"])
df_state
```

[40]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

[41]:
```
df_state[["three", "one"]]
```

[41]:
```
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

[42]:
```
df_state[["two"]]
```

[42]:
```
          two
Ohio        1
Colorado    5
Utah        9
New York   13
```

[43]:
```
df_state["two"]
```

[43]:
```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
```

[44]:
```
df_state.two
```

[44]:
```
Ohio         1
Colorado     5
```

```
Utah         9
New York    13
Name: two, dtype: int64
```

[45]:
```python
df_state[["three", "one"]]
```

[45]:
```
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

[46]:
```python
df_state
```

[46]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

[47]:
```python
df_state[:2]
```

[47]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
```

The row selection syntax `df_state[:2]` is provided as a convenience. Passing a single element or a list to the `[]` operator selects columns.

[48]:
```python
df_state
```

[48]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

[49]:
```python
df_state[2]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/indexes/base.py:
↪3802, in Index.get_loc(self, key, method, tolerance)
   3801 try:
-> 3802     return self._engine.get_loc(casted_key)
   3803 except KeyError as err:
```

```
File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/_libs/index.pyx:138, in
↪pandas._libs.index.IndexEngine.get_loc()

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/_libs/index.pyx:165, in
↪pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:5745, in pandas._libs.hashtable.
↪PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:5753, in pandas._libs.hashtable.
↪PyObjectHashTable.get_item()

KeyError: 2

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[49], line 1
----> 1 df_state[2]

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/frame.py:3807, in
↪DataFrame.__getitem__(self, key)
   3805 if self.columns.nlevels > 1:
   3806     return self._getitem_multilevel(key)
-> 3807 indexer = self.columns.get_loc(key)
   3808 if is_integer(indexer):
   3809     indexer = [indexer]

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/indexes/base.py:
↪3804, in Index.get_loc(self, key, method, tolerance)
   3802     return self._engine.get_loc(casted_key)
   3803 except KeyError as err:
-> 3804     raise KeyError(key) from err
   3805 except TypeError:
   3806     # If we have a listlike key, _check_indexing_error will raise
   3807     #  InvalidIndexError. Otherwise we fall through and re-raise
   3808     #  the TypeError.
   3809     self._check_indexing_error(key)

KeyError: 2
```

[50]:
```
df_state[:2]
```

[50]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

[51]:
```
df_state
```

[51]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

[52]:
```
df_state[1:3]
```

[52]:
```
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
```

[53]:
```
df_state[-2:]
```

[53]:
```
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

2.4.1 "Row" selection on DataFrame with loc and iloc

- `loc`: label-based indexing
- `iloc`: integer-based indexing.

[54]:
```
df_state
```

[54]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

[55]:
```
df_state.loc["Colorado"]
```

[55]:
```
one      4
two      5
three    6
four     7
Name: Colorado, dtype: int64
```

[56]:
```
df_state.loc["Utah"]
```

[56]:
```
one       8
two       9
three    10
four     11
```

```
Name: Utah, dtype: int64
```

[57]:
```
df_state.iloc[1]
```

[57]:
```
one      4
two      5
three    6
four     7
Name: Colorado, dtype: int64
```

[58]:
```
df_state
```

[58]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

[59]:
```
df_state.loc[["Utah","Ohio"]]
```

[59]:
```
      one  two  three  four
Utah    8    9     10    11
Ohio    0    1      2     3
```

[60]:
```
df_state.iloc[[2,0]]
```

[60]:
```
      one  two  three  four
Utah    8    9     10    11
Ohio    0    1      2     3
```

Filter data with conditions

[61]:
```
df_state
```

[61]:
```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

[62]:
```
df_state < 9
```

[62]:
```
            one    two  three   four
Ohio       True   True   True   True
Colorado   True   True   True   True
Utah       True  False  False  False
New York  False  False  False  False
```

```
[63]: df_state[df_state < 9]
```

```
[63]:           one  two  three  four
      Ohio      0.0  1.0    2.0   3.0
      Colorado  4.0  5.0    6.0   7.0
      Utah      8.0  NaN    NaN   NaN
      New York  NaN  NaN    NaN   NaN
```

```
[64]: df_state[df_state < 9] = 9
```

```
[65]: df_state
```

```
[65]:           one  two  three  four
      Ohio        9    9      9     9
      Colorado    9    9      9     9
      Utah        9    9     10    11
      New York   12   13     14    15
```

```
[66]: df_state
```

```
[66]:           one  two  three  four
      Ohio        9    9      9     9
      Colorado    9    9      9     9
      Utah        9    9     10    11
      New York   12   13     14    15
```

```
[67]: df_state.three== 10
```

```
[67]: Ohio        False
      Colorado    False
      Utah         True
      New York    False
      Name: three, dtype: bool
```

```
[68]: df_state[df_state.three==10]
```

```
[68]:       one  two  three  four
      Utah    9    9     10    11
```

try on the other DataFrame

```
[69]: df_ward.head(2)
```

```
[69]:    population  ward  poverty  year
      0        2724     1  0.21586  2000
      1        4117     2  0.67016  2000
```

```python
[70]: df_ward['population']
```

```
[70]: 0     2724
      1     4117
      2     4653
      3     4770
      4     2450
      5     1257
      6     2199
      7     2674
      8     3612
      9     3870
      10    1396
      11    3380
      12    4194
      13    3948
      14    2879
      15    4336
      16     136
      17     221
      18    3441
      19      28
      20     432
      21     250
      22    4097
      23    2418
      24     504
      Name: population, dtype: int64
```

```python
[71]: df_ward.population
```

```
[71]: 0     2724
      1     4117
      2     4653
      3     4770
      4     2450
      5     1257
      6     2199
      7     2674
      8     3612
      9     3870
      10    1396
      11    3380
      12    4194
      13    3948
      14    2879
      15    4336
```

```
16      136
17      221
18     3441
19       28
20      432
21      250
22     4097
23     2418
24      504
Name: population, dtype: int64
```

[72]:
```
df_ward.head(2)
```

[72]:

	population	ward	poverty	year
0	2724	1	0.21586	2000
1	4117	2	0.67016	2000

[73]:
```
df_ward[0:4]
```

[73]:

	population	ward	poverty	year
0	2724	1	0.215860	2000
1	4117	2	0.670160	2000
2	4653	3	0.955366	2000
3	4770	4	0.967289	2000

[74]:
```
df_ward[-4:]
```

[74]:

	population	ward	poverty	year
21	250	2	0.015459	2004
22	4097	3	0.276339	2004
23	2418	4	0.259910	2004
24	504	5	0.674916	2004

[75]:
```
df_ward[df_ward.ward==2]
```

[75]:

	population	ward	poverty	year
1	4117	2	0.670160	2000
6	2199	2	0.595693	2001
11	3380	2	0.056647	2002
16	136	2	0.279772	2003
21	250	2	0.015459	2004

[76]:
```
df_ward[df_ward.population<1000]
```

[76]:

	population	ward	poverty	year
16	136	2	0.279772	2003
17	221	3	0.790454	2003

```
19          28     5  0.806454  2003
20         432     1  0.972108  2004
21         250     2  0.015459  2004
24         504     5  0.674916  2004
```

[77]:
```
df_ward[(df_ward.ward==2) & (df_ward.population < 1000)] # & binary operator to perform and operation on lists of boolean values
```

[77]:
```
    population  ward   poverty  year
16         136     2  0.279772  2003
21         250     2  0.015459  2004
```

[78]:
```
(df_ward.ward==2) & (df_ward.population < 1000)
```

[78]:
```
0     False
1     False
2     False
3     False
4     False
5     False
6     False
7     False
8     False
9     False
10    False
11    False
12    False
13    False
14    False
15    False
16     True
17    False
18    False
19    False
20    False
21     True
22    False
23    False
24    False
dtype: bool
```

[79]:
```
df_ward[(df_ward.ward==2) | (df_ward.population < 1000)] # | binary operator to perform or operation on lists of boolean values
```

[79]:
```
    population  ward   poverty  year
1         4117     2  0.670160  2000
6         2199     2  0.595693  2001
```

11	3380	2	0.056647	2002
16	136	2	0.279772	2003
17	221	3	0.790454	2003
19	28	5	0.806454	2003
20	432	1	0.972108	2004
21	250	2	0.015459	2004
24	504	5	0.674916	2004

[80]:
```
df_ward[(~(df_ward.ward==2)) & (df_ward.population < 1000)] # not in ward 2 and␣
↪less than 1000 population
```

[80]:

	population	ward	poverty	year
17	221	3	0.790454	2003
19	28	5	0.806454	2003
20	432	1	0.972108	2004
24	504	5	0.674916	2004

[81]:
```
df_ward[~((df_ward.ward==2) & (df_ward.population < 1000))] # not (in ward 2␣
↪and less than 1000 population)
```

[81]:

	population	ward	poverty	year
0	2724	1	0.215860	2000
1	4117	2	0.670160	2000
2	4653	3	0.955366	2000
3	4770	4	0.967289	2000
4	2450	5	0.244911	2000
5	1257	1	0.476372	2001
6	2199	2	0.595693	2001
7	2674	3	0.933573	2001
8	3612	4	0.262039	2001
9	3870	5	0.348152	2001
10	1396	1	0.157774	2002
11	3380	2	0.056647	2002
12	4194	3	0.662450	2002
13	3948	4	0.212500	2002
14	2879	5	0.025248	2002
15	4336	1	0.195138	2003
17	221	3	0.790454	2003
18	3441	4	0.242731	2003
19	28	5	0.806454	2003
20	432	1	0.972108	2004
22	4097	3	0.276339	2004
23	2418	4	0.259910	2004
24	504	5	0.674916	2004

2.4.2 Group exercise

```
ward = np.tile([1,2,3,4,5], 5)
year = np.array([2000] * 5 +[2001] * 5 + [2002] * 5 + [2003] * 5 + [2004] * 5)
population = np.random.randint(5000, size=(25,))
poverty = np.random.random(size=(25,))
df_ward = pandas.DataFrame({'population': population,
                      'ward': ward,
                      'poverty': poverty})
```

Selecting records from `df_ward` that are in ward 3, larger than 500 population, and poverty rate less than 40%

When you are done, raise your hand

[82]:
```
df_ward[(df_ward.ward==3) & (df_ward.population > 500) & (df_ward.poverty<0.4)]
```

[82]:

	population	ward	poverty	year
22	4097	3	0.276339	2004

2.5 Creating New Columns in an existing `DataFrame`

[83]:
```
df_ward.head()
```

[83]:

	population	ward	poverty	year
0	2724	1	0.215860	2000
1	4117	2	0.670160	2000
2	4653	3	0.955366	2000
3	4770	4	0.967289	2000
4	2450	5	0.244911	2000

[84]:
```
pop_pov = df_ward.population * df_ward.poverty # elementwise operation similar␣
↪to numpy array
pop_pov
```

[84]:
```
0      588.002144
1     2759.048581
2     4445.319943
3     4613.967837
4      600.032443
5      598.799771
6     1309.927918
7     2496.373335
8      946.486252
9     1347.347781
10     220.252915
11     191.466005
12    2778.315548
13     838.950514
```

```
14      72.688373
15     846.118899
16      38.048950
17     174.690305
18     835.237618
19      22.580721
20     419.950516
21       3.864853
22    1132.162300
23     628.462986
24     340.157910
dtype: float64
```

[85]:
```
df_ward
```

[85]:

	population	ward	poverty	year
0	2724	1	0.215860	2000
1	4117	2	0.670160	2000
2	4653	3	0.955366	2000
3	4770	4	0.967289	2000
4	2450	5	0.244911	2000
5	1257	1	0.476372	2001
6	2199	2	0.595693	2001
7	2674	3	0.933573	2001
8	3612	4	0.262039	2001
9	3870	5	0.348152	2001
10	1396	1	0.157774	2002
11	3380	2	0.056647	2002
12	4194	3	0.662450	2002
13	3948	4	0.212500	2002
14	2879	5	0.025248	2002
15	4336	1	0.195138	2003
16	136	2	0.279772	2003
17	221	3	0.790454	2003
18	3441	4	0.242731	2003
19	28	5	0.806454	2003
20	432	1	0.972108	2004
21	250	2	0.015459	2004
22	4097	3	0.276339	2004
23	2418	4	0.259910	2004
24	504	5	0.674916	2004

[86]:
```
df_ward['pop_pov'] = pop_pov.astype('int')
```

[87]:
```
df_ward.head()
```

[87]:

```
   population  ward   poverty  year  pop_pov
0        2724     1  0.215860  2000      588
1        4117     2  0.670160  2000     2759
2        4653     3  0.955366  2000     4445
3        4770     4  0.967289  2000     4613
4        2450     5  0.244911  2000      600
```

2.6 Aggregation/Groupby

[88]:

```python
df_ward[df_ward.ward==1]
```

[88]:

```
    population  ward   poverty  year  pop_pov
0         2724     1  0.215860  2000      588
5         1257     1  0.476372  2001      598
10        1396     1  0.157774  2002      220
15        4336     1  0.195138  2003      846
20         432     1  0.972108  2004      419
```

[89]:

```python
df_ward.groupby(by='ward').sum()
```

[89]:

```
      population   poverty   year  pop_pov
ward
1          10145  2.017252  10010     2671
2          10082  1.617730  10010     4300
3          15839  3.618182  10010    11025
4          18189  1.944470  10010     7860
5           9731  2.099682  10010     2381
```

[90]:

```python
df_ward.groupby(by='ward').sum()[['population', 'pop_pov']]
```

[90]:

```
      population  pop_pov
ward
1          10145     2671
2          10082     4300
3          15839    11025
4          18189     7860
5           9731     2381
```

[91]:

```python
ward_df = df_ward.groupby(by='ward').sum()[['population', 'pop_pov']]
```

[92]:

```python
ward_df
```

[92]:

```
      population  pop_pov
ward
1          10145     2671
2          10082     4300
3          15839    11025
```

```
4           18189     7860
5            9731     2381
```

[93]:
```
ward_df['poverty'] = ward_df.pop_pov / ward_df.population
```

[94]:
```
ward_df
```

[94]:
```
      population  pop_pov   poverty
ward
1          10145     2671  0.263282
2          10082     4300  0.426503
3          15839    11025  0.696067
4          18189     7860  0.432129
5           9731     2381  0.244682
```

2.7 Joins/Merge

[95]:
```
ward_df
```

[95]:
```
      population  pop_pov   poverty
ward
1          10145     2671  0.263282
2          10082     4300  0.426503
3          15839    11025  0.696067
4          18189     7860  0.432129
5           9731     2381  0.244682
```

[96]:
```
df_ward
```

[96]:
```
    population  ward   poverty  year  pop_pov
0         2724     1  0.215860  2000      588
1         4117     2  0.670160  2000     2759
2         4653     3  0.955366  2000     4445
3         4770     4  0.967289  2000     4613
4         2450     5  0.244911  2000      600
5         1257     1  0.476372  2001      598
6         2199     2  0.595693  2001     1309
7         2674     3  0.933573  2001     2496
8         3612     4  0.262039  2001      946
9         3870     5  0.348152  2001     1347
10        1396     1  0.157774  2002      220
11        3380     2  0.056647  2002      191
12        4194     3  0.662450  2002     2778
13        3948     4  0.212500  2002      838
14        2879     5  0.025248  2002       72
15        4336     1  0.195138  2003      846
16         136     2  0.279772  2003       38
```

```
17         221     3  0.790454  2003      174
18        3441     4  0.242731  2003      835
19          28     5  0.806454  2003       22
20         432     1  0.972108  2004      419
21         250     2  0.015459  2004        3
22        4097     3  0.276339  2004     1132
23        2418     4  0.259910  2004      628
24         504     5  0.674916  2004      340
```

[97]:

```
df_all = df_ward.merge(ward_df,  on='ward')
```

[98]:

```
df_all
```

[98]:

```
    population_x  ward  poverty_x  year  pop_pov_x  population_y  pop_pov_y  \
0           2724     1   0.215860  2000        588         10145       2671
1           1257     1   0.476372  2001        598         10145       2671
2           1396     1   0.157774  2002        220         10145       2671
3           4336     1   0.195138  2003        846         10145       2671
4            432     1   0.972108  2004        419         10145       2671
5           4117     2   0.670160  2000       2759         10082       4300
6           2199     2   0.595693  2001       1309         10082       4300
7           3380     2   0.056647  2002        191         10082       4300
8            136     2   0.279772  2003         38         10082       4300
9            250     2   0.015459  2004          3         10082       4300
10          4653     3   0.955366  2000       4445         15839      11025
11          2674     3   0.933573  2001       2496         15839      11025
12          4194     3   0.662450  2002       2778         15839      11025
13           221     3   0.790454  2003        174         15839      11025
14          4097     3   0.276339  2004       1132         15839      11025
15          4770     4   0.967289  2000       4613         18189       7860
16          3612     4   0.262039  2001        946         18189       7860
17          3948     4   0.212500  2002        838         18189       7860
18          3441     4   0.242731  2003        835         18189       7860
19          2418     4   0.259910  2004        628         18189       7860
20          2450     5   0.244911  2000        600          9731       2381
21          3870     5   0.348152  2001       1347          9731       2381
22          2879     5   0.025248  2002         72          9731       2381
23            28     5   0.806454  2003         22          9731       2381
24           504     5   0.674916  2004        340          9731       2381

    poverty_y
0    0.263282
1    0.263282
2    0.263282
3    0.263282
4    0.263282
5    0.426503
```

```
6     0.426503
7     0.426503
8     0.426503
9     0.426503
10    0.696067
11    0.696067
12    0.696067
13    0.696067
14    0.696067
15    0.432129
16    0.432129
17    0.432129
18    0.432129
19    0.432129
20    0.244682
21    0.244682
22    0.244682
23    0.244682
24    0.244682
```

[99]:
```
df_all = df_ward.merge(ward_df, on='ward',suffixes = ('_year', '_allyears'))
```

[100]:
```
df_all
```

[100]:
```
    population_year  ward  poverty_year  year  pop_pov_year  \
0              2724     1      0.215860  2000           588
1              1257     1      0.476372  2001           598
2              1396     1      0.157774  2002           220
3              4336     1      0.195138  2003           846
4               432     1      0.972108  2004           419
5              4117     2      0.670160  2000          2759
6              2199     2      0.595693  2001          1309
7              3380     2      0.056647  2002           191
8               136     2      0.279772  2003            38
9               250     2      0.015459  2004             3
10             4653     3      0.955366  2000          4445
11             2674     3      0.933573  2001          2496
12             4194     3      0.662450  2002          2778
13              221     3      0.790454  2003           174
14             4097     3      0.276339  2004          1132
15             4770     4      0.967289  2000          4613
16             3612     4      0.262039  2001           946
17             3948     4      0.212500  2002           838
18             3441     4      0.242731  2003           835
19             2418     4      0.259910  2004           628
20             2450     5      0.244911  2000           600
21             3870     5      0.348152  2001          1347
```

22	2879	5	0.025248	2002	72
23	28	5	0.806454	2003	22
24	504	5	0.674916	2004	340

	population_allyears	pop_pov_allyears	poverty_allyears
0	10145	2671	0.263282
1	10145	2671	0.263282
2	10145	2671	0.263282
3	10145	2671	0.263282
4	10145	2671	0.263282
5	10082	4300	0.426503
6	10082	4300	0.426503
7	10082	4300	0.426503
8	10082	4300	0.426503
9	10082	4300	0.426503
10	15839	11025	0.696067
11	15839	11025	0.696067
12	15839	11025	0.696067
13	15839	11025	0.696067
14	15839	11025	0.696067
15	18189	7860	0.432129
16	18189	7860	0.432129
17	18189	7860	0.432129
18	18189	7860	0.432129
19	18189	7860	0.432129
20	9731	2381	0.244682
21	9731	2381	0.244682
22	9731	2381	0.244682
23	9731	2381	0.244682
24	9731	2381	0.244682

[101]:
```
df_all[df_all.poverty_year > df_all.poverty_allyears]
```

[101]:

	population_year	ward	poverty_year	year	pop_pov_year	\
1	1257	1	0.476372	2001	598	
4	432	1	0.972108	2004	419	
5	4117	2	0.670160	2000	2759	
6	2199	2	0.595693	2001	1309	
10	4653	3	0.955366	2000	4445	
11	2674	3	0.933573	2001	2496	
13	221	3	0.790454	2003	174	
15	4770	4	0.967289	2000	4613	
20	2450	5	0.244911	2000	600	
21	3870	5	0.348152	2001	1347	
23	28	5	0.806454	2003	22	
24	504	5	0.674916	2004	340	

```
    population_allyears  pop_pov_allyears  poverty_allyears
1                 10145              2671          0.263282
4                 10145              2671          0.263282
5                 10082              4300          0.426503
6                 10082              4300          0.426503
10                15839             11025          0.696067
11                15839             11025          0.696067
13                15839             11025          0.696067
15                18189              7860          0.432129
20                 9731              2381          0.244682
21                 9731              2381          0.244682
23                 9731              2381          0.244682
24                 9731              2381          0.244682
```

Which ward has the highest average poverty rate?

[102]:
```
df_all.poverty_allyears.idxmax()
```

[102]:
```
10
```

[103]:
```
df_all.loc[df_all['poverty_allyears'].idxmax()]
```

[103]:
```
population_year         4653.000000
ward                       3.000000
poverty_year               0.955366
year                    2000.000000
pop_pov_year            4445.000000
population_allyears    15839.000000
pop_pov_allyears       11025.000000
poverty_allyears           0.696067
Name: 10, dtype: float64
```

Which ward in which year has the lowest poverty rate?

[104]:
```
df_all.poverty_year.idxmin()
```

[104]:
```
9
```

[105]:
```
df_all.loc[df_all['poverty_year'].idxmin()]
```

[105]:
```
population_year          250.000000
ward                       2.000000
poverty_year               0.015459
year                    2004.000000
pop_pov_year               3.000000
population_allyears    10082.000000
pop_pov_allyears        4300.000000
poverty_allyears           0.426503
```

```
Name: 9, dtype: float64
```

2.8 Reading and Writing Data with Pandas

- Pandas features a number of functions for reading tabular data as a `DataFrame` object.
- Works with many different data formats
- Works with different data source:
  - reading text files and other more efficient on-disk formats
  - loading data from databases
  - interacting with network sources like web APIs

2.8.1 An example with working with csv files

- `read_csv` function: Load delimited data from a file, URL, or file-like object; use comma as default delimiter
  - A long list of optional arguments to deal with messy data in the real world
- `to_csv` method (associated with a `DataFrame` instance): Writing to a csv file

[106]:
```python
df1 = pd.read_csv("ex1.csv")
df1
```

[106]:
```
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

If only the path is supplied, the first row of the file will be used as the header (column names) of the `DataFrame` object and column names are inferred from the first line of the file.

[107]:
```python
df2 = pd.read_csv("ex1.csv", header=None)
df2
```

[107]:
```
   0   1   2   3        4
0  a   b   c   d  message
1  1   2   3   4    hello
2  5   6   7   8    world
3  9  10  11  12      foo
```

If `header=None`, integer index starting from 0 will be used as column names.

[108]:
```python
df3 = pd.read_csv("ex1.csv", names=["col1", "col2", "col3", "col4", "col5"])
df3
```

[108]:
```
  col1 col2 col3 col4     col5
0    a    b    c    d  message
1    1    2    3    4    hello
2    5    6    7    8    world
3    9   10   11   12      foo
```

We can pass a list of column names to the argument `names`

[109]:
```
df4 = pd.read_csv("ex1.csv", index_col="message")
df4
```

[109]:
```
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

We can specify the column name/index in the argument `index_col` as the row labels of the DataFrame

[110]:
```
df4 = pd.read_csv("ex1.csv", index_col=4)
df4
```

[110]:
```
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

[111]:
```
df5 = pd.read_csv("ex1.csv", skiprows=[1,2])
df5
```

[111]:
```
   a   b   c   d message
0  9  10  11  12     foo
```

Argument `skiprows`: Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

[112]:
```
df6 = pd.read_csv("ex1.csv", skiprows=2)
df6
```

[112]:
```
   5   6   7   8 world
0  9  10  11  12   foo
```

Dealing with missing values

- To control which values are parsed as missing values (which are signified by NaN), specify a string in `na_values`.
- If you specify a list of strings, then all values in it are considered to be missing values.
- If you specify a number (a float, like 5.0 or an integer like 5), the corresponding equivalent values will also imply a missing value (in this case effectively [5.0, 5] are recognized as NaN).

[113]:
```
df_ex5 = pd.read_csv("ex5.csv")
df_ex5
```

```
[113]:   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
```

```
[114]: df_ex5 = pd.read_csv("ex5.csv", na_values=["one", 1])
       df_ex5
```

```
[114]:   something    a   b     c   d message
       0       NaN  NaN   2   3.0   4     NaN
       1       two  5.0   6   NaN   8   world
       2     three  9.0  10  11.0  12     foo
```

```
[115]: df_ex5.dropna() #Drop the rows where at least one element is missing.
```

```
[115]:   something    a   b     c   d message
       2     three  9.0  10  11.0  12     foo
```

```
[116]: df_ex5.dropna(axis='columns') # Drop the columns where at least one element is␣
        ↪missing.
```

```
[116]:    b   d
       0   2   4
       1   6   8
       2  10  12
```

```
[117]: df_ex5.dropna(subset=["something"]) #Define in which columns to look for␣
        ↪missing values.
```

```
[117]:   something    a   b     c   d message
       1       two  5.0   6   NaN   8   world
       2     three  9.0  10  11.0  12     foo
```

Save a Dataframe to a csv file

```
[118]: df4.to_csv("data/output1.csv")
```

Read panda's documentation to better understand the functionality of pandas's `read_csv` function.

3 Further readings

- Python for Data Analysis, 3E, by Wes McKinney