NumPy and Pandas II

2023-10-23

Today's agenda

- We had a quick look into NumPy/Pandas fundamentals
- We will look at key data manipulation/wrangling in Pandas
- Join, concatenate, reshape, aggregate, etc.

Joining data

- Data may be spread across multiple files or databases
- We often want to combine data sets by linking rows
- We use column(s) to join data sets

Joining data

- Let's create two example data sets

```
import pandas as pd
df1 = pd.DataFrame({
  'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
  'data1': pd.Series(range(7))})
df2 = pd.DataFrame({
  'key': ['a', 'b', 'd'],
  'data2': pd.Series([False, True, True])})
```

Joining data

- Let's create two example data sets

```
print(df1)
print(df2)
```

```
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   a      5
6   b      6
  key  data2
0   a  False
1   b   True
2   d   True
```

Joinig data

- Use `pd.merge` to join the two dataframes
- Join based on the `key` column using the `on` argument
- In this case, we are doing a many-to-one join

```
df_join = pd.merge(df1, df2, on = 'key')
print(df_join)
```

```
  key  data1  data2
0   b      0   True
1   b      1   True
2   b      6   True
3   a      2  False
4   a      4  False
5   a      5  False
```

Joining data

- Without column(s) to join on specified, `pd.merge` uses overlapping column(s) automatically

```
df_join2 = pd.merge(df1, df2)
print(df_join2)
```

```
  key  data1  data2
0   b      0   True
1   b      1   True
2   b      6   True
3   a      2  False
4   a      4  False
5   a      5  False
```

Joining data

- If the column names are different in each DataFrame, specify them separately (or you can change the name(s) of the column(s))

```
df3 = pd.DataFrame({
  'key_l': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
  'data1': pd.Series(range(7))})
df4 = pd.DataFrame({
  'key_r': ['a', 'b', 'd'],
  'data2': pd.Series(range(3))})
```

Joining data

- Use the `left_on` and `right_on` arguments

```
df_join = pd.merge(df3, df4,
                   left_on = 'key_l', right_on = 'key_r')
print(df_join)
```

```
  key_l  data1 key_r  data2
0     b      0     b      1
1     b      1     b      1
2     b      6     b      1
3     a      2     a      0
4     a      4     a      0
5     a      5     a      0
```

Joining data

- But, where have `'c'`s and `'d'`s gone?
- This is because the argument `'how'` is set as `'inner'`
- With `'inner'` join, the result only contains rows whose keys find a match (`'a'`, `'b'`)

```
print(df1)
print(df2)
```

```
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   a      5
6   b      6
  key  data2
0   a  False
1   b   True
2   d   True
```

Joining data

- `outer` join takes the union of the keys
- Rows that do not match on keys in the other DataFrame will get missing values

```
df_join_outer = pd.merge(df3, df4, how = 'outer',
                         left_on = 'key_l', right_on = 'key_r')
print(df_join_outer)
```

```
  key_l  data1 key_r  data2
0     b    0.0     b    1.0
1     b    1.0     b    1.0
2     b    6.0     b    1.0
3     a    2.0     a    0.0
4     a    4.0     a    0.0
5     a    5.0     a    0.0
6     c    3.0   NaN    NaN
7   NaN    NaN     d    2.0
```

Joining data

- `left` join keeps the data frame on the left
- If there's no match for a particular row in the DataFrame on the right, it will be filled with missing values

```
df_join_left = pd.merge(df1, df2, how = 'left',
                        on = 'key')
print(df_join_left)
```

```
  key  data1  data2
0   b      0   True
1   b      1   True
2   a      2  False
3   c      3    NaN
4   a      4  False
5   a      5  False
6   b      6   True
```

Joining data

- Vice versa for `right` join

```
df_join_right = pd.merge(df1, df2, how = 'right',
                         on = 'key')
print(df_join_right)
```

```
  key  data1  data2
0   a    2.0  False
1   a    4.0  False
2   a    5.0  False
3   b    0.0   True
4   b    1.0   True
5   b    6.0   True
6   d    NaN   True
```

Joining data

- Joining on multiple columns

```
students = pd.DataFrame({
    'student_id': [1, 2, 3, 4],
    'name': ['Alice', 'Bob', 'Charlie', 'David'],
    'class': ['Math', 'History', 'Math', 'Science']})
grades = pd.DataFrame({
    'student_id': [1, 2, 3],
    'class': ['Math', 'History', 'Science'],
    'grade': ['A', 'B', 'A']})
print(students)
print(grades)
```

```
   student_id     name    class
0           1    Alice     Math
1           2      Bob  History
2           3  Charlie     Math
3           4    David  Science
   student_id    class grade
0           1     Math     A
1           2  History     B
2           3  Science     A
```

Joining data

- Simply provide a list of columns we wnat to join on

```
df_join_mult = pd.merge(students, grades,
                        on = ['student_id', 'class'],
                        how = 'left')
print(df_join_mult)
```

```
   student_id     name    class grade
0           1    Alice     Math     A
1           2      Bob  History     B
2           3  Charlie     Math   NaN
3           4    David  Science   NaN
```

Concatenating data

- By concatenating, we put chunks of data together row-wise or column-wise
- Note that, unlike `pd.merge`, we don't match on values
- With `axis` taking 0, we are concatenating row-wise (vertically)

```
survey_2022 = pd.DataFrame({
    'respondent_id': [1, 2, 3],
    'opinion': ['agree', 'disagree', 'neutral']})
print(survey_2022)

survey_2023 = pd.DataFrame({
    'respondent_id': [4, 5, 6],
    'opinion': ['neutral', 'agree', 'disagree']})
print(survey_2023)
```

```
   respondent_id   opinion
0              1     agree
1              2  disagree
2              3   neutral
   respondent_id   opinion
0              4   neutral
1              5     agree
2              6  disagree
```

Concatenating data

- With `axis` taking 0, we are concatenating row-wise (vertically)
- With the `ignore_index` argument set `True`, the index is reset

```
survey_concate_row = pd.concat(
  [survey_2022, survey_2023],
  axis = 0,
  ignore_index = True)
print(survey_concate_row)
```

```
   respondent_id   opinion
0              1     agree
1              2  disagree
2              3   neutral
3              4   neutral
4              5     agree
5              6  disagree
```

Concatenating data

- With `axis` taking 1, we are concatenating column-wise (horizontally)

```
demographics = pd.DataFrame({
    'age': [25, 30, 35, 40],
    'gender': ['M', 'F', 'F', 'M']},
    index=[1, 2, 3, 4])  # index is respondent_id
print(demographics)

responses = pd.DataFrame({
    'opinion': ['agree', 'disagree', 'neutral', 'agree']},
    index=[1, 2, 3, 4])  # index is respondent_id
print(responses)
```

```
   age gender
1   25      M
2   30      F
3   35      F
4   40      M
    opinion
1     agree
2  disagree
3   neutral
4     agree
```

Concatenating data

- With `axis` taking 1, we are concatenating column-wise (horizontally)

```
survey_concat_column = pd.concat(
  [demographics, responses],
  axis = 1)
print(survey_concat_column)
```

```
   age gender   opinion
1   25      M     agree
2   30      F  disagree
3   35      F   neutral
4   40      M     agree
```

Concatenating data

- What if the indices on two data source don't match?

```
demographics = pd.DataFrame({
    'age': [25, 30, 35, 40],
    'gender': ['M', 'F', 'F', 'M']},
     index=[1, 2, 6, 8])  # index is respondent_id
print(demographics)

responses = pd.DataFrame({
    'opinion': ['agree', 'disagree', 'neutral', 'agree']},
     index=[0, 1, 2, 3])  # index is respondent_id
print(responses)
```

```
   age gender
1   25      M
2   30      F
6   35      F
8   40      M
    opinion
0     agree
1  disagree
2   neutral
3     agree
```

Concatenating data

- What if the indices on two data sources don't match?

```
survey_concat_column = pd.concat(
  [demographics, responses],
  axis = 1)
print(survey_concat_column)
```

```
    age gender   opinion
1  25.0      M  disagree
2  30.0      F   neutral
6  35.0      F       NaN
8  40.0      M       NaN
0   NaN    NaN     agree
3   NaN    NaN     agree
```

Concatenating data

- Use `reset_index` to assign a new default index

```
survey_concat_column = pd.concat(
  [demographics, responses], axis = 1).reset_index(drop = True)
print(survey_concat_column)
```

```
    age gender   opinion
0  25.0      M  disagree
1  30.0      F   neutral
2  35.0      F       NaN
3  40.0      M       NaN
4   NaN    NaN     agree
5   NaN    NaN     agree
```

Concatenating data

- If necessary, use the `reset_index(drop = True)` before concatenating

```
survey_concat_column = pd.concat([
  demographics.reset_index(drop = True),
  responses.reset_index(drop = True)],
  axis = 1)
print(survey_concat_column)
```

```
   age gender   opinion
0   25      M     agree
1   30      F  disagree
2   35      F   neutral
3   40      M     agree
```

Concatenating data

- Put `'inner'` for the `join` argument to focus on rows without a missing value

```
survey_concat_column_inner = pd.concat(
  [demographics, responses],
  axis = 1,
  join = 'inner')
print(survey_concat_column_inner)
```

```
   age gender   opinion
1   25      M  disagree
2   30      F   neutral
```

Concatenating data

- Use `key` argument to create a hierarchical index
- This is useful when you want to keep track of the original data sources

```
manuscript_1 = pd.DataFrame(
  {'text': ['text1a', 'text1b']}
  )
manuscript_2 = pd.DataFrame(
  {'text': ['text2a', 'text2b']}
  )
print(manuscript_1)
print(manuscript_2)
```

```
     text
0  text1a
1  text1b
     text
0  text2a
1  text2b
```

Concatenating data

- Use key argument to create a hierarchical index
- This is useful when you want to keep track of the original data sources

```
combined_texts = pd.concat([manuscript_1, manuscript_2],
                           keys = ['m1', 'm2'])
print(combined_texts)
```

```
        text
m1 0  text1a
   1  text1b
m2 0  text2a
   1  text2b
```

Reshaping

- `stack` rotates from the columns to the rows

```
trade_data = pd.DataFrame({
    '2019': [100, 80, 120],
    '2020': [105, 78, 130],
    '2021': [110, 82, 135]},
     index = ['USA', 'UK', 'China']
    )
print(trade_data)
```

```
       2019  2020  2021
USA     100   105   110
UK       80    78    82
China   120   130   135
```

Reshaping

- For each entry in the index, the columns were used to generate the rows
- The column names now work as the inner index

```
trade_data_s = trade_data.stack()
print(trade_data_s)
print(type(trade_data_s))
```

```
USA    2019    100
       2020    105
       2021    110
UK     2019     80
       2020     78
       2021     82
China  2019    120
       2020    130
       2021    135
dtype: int64
<class 'pandas.core.series.Series'>
```

Reshaping

- `unstack` does the opposite
- Pick the index level to be used as column names
- The innermost index works as column names (by default)

```
trade_data = trade_data_s.unstack()
print(trade_data)
```

```
       2019  2020  2021
USA     100   105   110
UK       80    78    82
China   120   130   135
```

Reshaping

- Use of `unstack` with `level` argument (default: `-1`)
- The outer index works as column names

```
print(trade_data_s.unstack(level = 0))
```

```
      USA  UK  China
2019  100  80    120
2020  105  78    130
2021  110  82    135
```

Reshaping

- `reset_index()` can transform the returned Series into a DataFrame

```
print(trade_data_s)
```

```
USA    2019    100
       2020    105
       2021    110
UK     2019     80
       2020     78
       2021     82
China  2019    120
       2020    130
       2021    135
dtype: int64
```

Reshaping

- `reset_index()` can transform the returned Series into a dataframe

```
df_trade_data_s = trade_data_s.reset_index()
df_trade_data_s.columns = ['partner', 'year', 'volume']
print(df_trade_data_s)
```

```
  partner  year  volume
0     USA  2019     100
1     USA  2020     105
2     USA  2021     110
3      UK  2019      80
4      UK  2020      78
5      UK  2021      82
6   China  2019     120
7   China  2020     130
8   China  2021     135
```

Wide/long data format

- Wide data format provides a quick overview of all variables at a glance

```
data_wide = {'Jan': [30, 25],
             'Feb': [31, 24],
             'Mar': [29, 26]}
df_wide = pd.DataFrame(data_wide,
                       pd.Index(['CA', 'NY'],
                       name = 'state'))
df_wide.columns.name = 'month'
print(df_wide)
```

```
month  Jan  Feb  Mar
state
CA      30   31   29
NY      25   24   26
```

Wide/long data format

- Long data format is ideal for time-series data or repeated measures

```
data_long = {
  'state': ['CA', 'CA', 'CA', 'NY', 'NY', 'NY'],
  'month': ['Jan', 'Feb', 'Mar', 'Jan', 'Feb', 'Mar'],
  'usage': [30, 31, 29, 25, 24, 26]
}
df_long = pd.DataFrame(data_long)
print(df_long)
```

```
  state month  usage
0    CA   Jan     30
1    CA   Feb     31
2    CA   Mar     29
3    NY   Jan     25
4    NY   Feb     24
5    NY   Mar     26
```

From wide to long

- Example wide-format data

```
coffee_data = pd.DataFrame({
    'department': ['biology', 'history', 'physics'],
    '0': [5, 2, 10], # not at all
    '1': [7, 3, 16], # sometimes
    '2': [15, 9, 20], # always
    })
print(coffee_data)
```

```
  department   0   1   2
0    biology   5   7  15
1    history   2   3   9
2    physics  10  16  20
```

From wide to long

- Use `melt` method
- Specify `id_vars`
- (Optionally) `var_name` and `value_name` as well

```
long_format_coffee_data = coffee_data.melt(
  id_vars = ['department'],
  var_name = 'consumption',
  value_name = 'num_students'
  )
```

From wide to long

```
print(long_format_coffee_data)
```

```
  department consumption  num_students
0    biology           0             5
1    history           0             2
2    physics           0            10
3    biology           1             7
4    history           1             3
5    physics           1            16
6    biology           2            15
7    history           2             9
8    physics           2            20
```

From wide to long

- This is equivalent to creating an index using `set_index` and then running `stack`

```
long_format = coffee_data.set_index('department').stack()
print(long_format)
```

```
department
biology     0     5
            1     7
            2    15
history     0     2
            1     3
            2     9
physics     0    10
            1    16
            2    20
dtype: int64
```

From wide to long

```
print(long_format)
```

```
department
biology     0     5
            1     7
            2    15
history     0     2
            1     3
            2     9
physics     0    10
            1    16
            2    20
dtype: int64
```

From long to wide

```
sales_dict = {'product': ['A', 'A', 'B', 'B'],
              'month': ['Jan', 'Feb', 'Jan', 'Feb'],
              'sales': [100, 110, 90, 95],
              'stock': [32, 53, 23, 44]}
sales_df = pd.DataFrame(sales_dict)
print(sales_df)
```

```
  product month  sales  stock
0       A   Jan    100     32
1       A   Feb    110     53
2       B   Jan     90     23
3       B   Feb     95     44
```

From long to wide

- Use the `pivot` method
- Specify `index`, `columns`, and `values`

```
p_sales1 = sales_df.pivot(index = 'month',
                          columns = 'product',
                          values = 'sales')
print(p_sales1)
```

```
product    A   B
month
Feb      110  95
Jan      100  90
```

From long to wide

- Multiple `values`

```
p_sales2 = sales_df.pivot(index = 'month',
                          columns = 'product',
                          values = ['sales', 'stock'])
print(p_sales2)
print(type(p_sales2))
```

```
        sales     stock
product     A   B     A   B
month
Feb       110  95    53  44
Jan       100  90    32  23
<class 'pandas.core.frame.DataFrame'>
```

From long to wide

- This is equivalent to creating a hierarchical index using `set_index` and then running `unstack`

```
print(sales_df.set_index(['product', 'month']))
```

```
               sales  stock
product month
A       Jan      100     32
        Feb      110     53
B       Jan       90     23
        Feb       95     44
```

```
print(sales_df.set_index(['product', 'month']).unstack(level = 0))
```

```
        sales     stock
product     A   B     A   B
month
Feb       110  95    53  44
Jan       100  90    32  23
```

From long to wide

- `pivot` will raise an error if the specified index and columns pair is not unique
- `pivot_table` allows you to reshape data and also aggregate it
- Useful when there are duplicates that need to be aggregated

```
sales_dict2 = {'product': ['A', 'A', 'B', 'B', 'A'],
        'month': ['Jan', 'Feb', 'Jan', 'Feb', 'Jan'],
        'sales': [100, 110, 90, 95, 105]}
sales_df2 = pd.DataFrame(sales_dict2)
print(sales_df2)
```

```
  product month  sales
0       A   Jan    100
1       A   Feb    110
2       B   Jan     90
3       B   Feb     95
4       A   Jan    105
```

From long to wide

- aggfunccan take 'sum', 'mean', 'max', etc., or even a custom function
- Also a list of these (e.g., ['min', 'max'])

```
pt_sales = sales_df2.pivot_table(
  index = 'month', columns = 'product', values = 'sales',
  aggfunc = 'mean')
print(pt_sales)
```

```
product      A     B
month
Feb      110.0  95.0
Jan      102.5  90.0
```

Group operations and aggregation

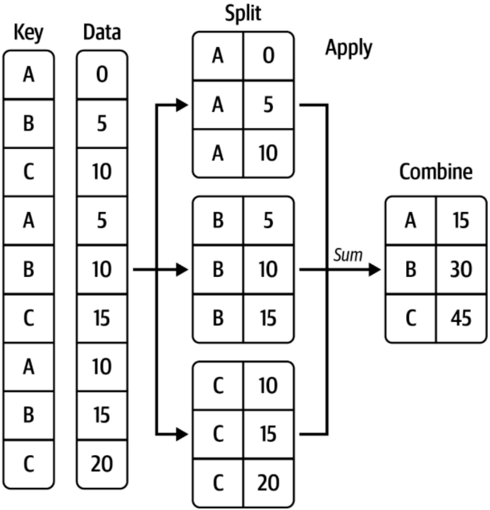

Figure 1: Split-Apply-Combine framework

Grouping

- Let's create an example data

```
import numpy as np
df = pd.DataFrame({
  'key1' : ['a', 'a', 'c', 'b', 'b', 'a', 'a'],
  'key2' : pd.Series([2, 2, 1, 2, 1, None, 1]),
  'data1' : np.random.standard_normal(7),
  'data2' : np.random.standard_normal(7)})
print(df)
```

```
  key1  key2     data1     data2
0    a   2.0 -0.480759  0.258820
1    a   2.0 -0.212584  0.576520
2    c   1.0  0.871836 -0.172507
3    b   2.0 -0.748853  1.415610
4    b   1.0  0.032099 -1.074665
5    a   NaN  1.026870 -0.013686
6    a   1.0 -0.123515  2.074180
```

Grouping

- Generate a `GroupBy` object

```
grouped = df.groupby(df['key1'])
print(type(grouped))
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

Grouping

- We can actually see the keys and groups in a `GroupBy` object

```
for x, y in df.groupby(['key1']):
  print(x)
  print(y)
```

```
('a',)
  key1  key2     data1     data2
0    a   2.0 -0.480759  0.258820
1    a   2.0 -0.212584  0.576520
5    a   NaN  1.026870 -0.013686
6    a   1.0 -0.123515  2.074180
('b',)
  key1  key2     data1     data2
3    b   2.0 -0.748853  1.415610
4    b   1.0  0.032099 -1.074665
('c',)
  key1  key2     data1     data2
2    c   1.0  0.871836 -0.172507
```

Grouping

- Subset a `GroupBy` object and do some simple aggregation

```
print(grouped['data1'].mean())
print(grouped[['data1','data2']].mean())
```

```
key1
a    0.052503
b   -0.358377
c    0.871836
Name: data1, dtype: float64
         data1     data2
key1
a     0.052503  0.723958
b    -0.358377  0.170472
c     0.871836 -0.172507
```

- Use the dropna argument to keep missing values

```
grouped2 = df.groupby(
  [df['key1'], df['key2']],
  dropna = False)
```

```
for (x, y), z in grouped2:
  print((x, y))
  print(y)
```

```
('a', 1.0)
1.0
('a', 2.0)
2.0
('a', nan)
nan
('b', 1.0)
1.0
('b', 2.0)
2.0
('c', 1.0)
1.0
```

Aggregation

- Aggregation refers to any data transformation that produces scalar values from arrays
- With `groupby`, there are many ways to aggregate
- `mean`, `median`, `sum`, `size`, `min`, `max`, `count`, etc.

Aggregation

- Example data on voter turnout by region and district

```
dict_to = {
  'region': ['R1', 'R1', 'R2', 'R2', 'R3', 'R3'],
  'district': ['A', 'B', 'C', 'D', 'E', 'F'],
  'turnout': [0.61, 0.46, 0.64, 0.75, 0.63, 0.55]
  }
df_to = pd.DataFrame(dict_to)
print(df_to)
```

```
  region district  turnout
0     R1        A     0.61
1     R1        B     0.46
2     R2        C     0.64
3     R2        D     0.75
4     R3        E     0.63
5     R3        F     0.55
```

Aggregation

- For instance, get the minimum and maximum values for each region

```
print(
  df_to.groupby('region')['turnout'].agg(
  ['min', 'max'])
  )
```

```
         min   max
region
R1      0.46  0.61
R2      0.64  0.75
R3      0.55  0.63
```

Aggregation

- We can also use our own aggregation functions

```
def min_max_diff(arr):
  return arr.max() - arr.min()
df_to_agg = df_to.groupby('region')['turnout'].agg(
    [min_max_diff, 'min', 'max']
    )
print(df_to_agg)
```

```
        min_max_diff   min   max
region
R1              0.15  0.46  0.61
R2              0.11  0.64  0.75
R3              0.08  0.55  0.63
```

Aggregation

- Index as a column

```
df_to_agg_index_col = df_to.groupby(
  'region', as_index = False)['turnout'].agg(
    [min_max_diff, 'min', 'max']
    )
print(df_to_agg_index_col)
```

```
  region  min_max_diff   min   max
0     R1          0.15  0.46  0.61
1     R2          0.11  0.64  0.75
2     R3          0.08  0.55  0.63
```

apply operations

- Apply a function along an axis of a DataFrame or Series

```
df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
print(df)
```

```
   A  B
0  4  9
1  4  9
2  4  9
```

`apply` operations

- Apply a function along an axis of a DataFrame or Series

```
print(df.apply(np.sqrt))
```

```
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

apply operations

- Apply a function along an axis of a DataFrame or Series

```
print(df.apply(np.sum, axis = 0))
print(df.apply(np.sum, axis = 1))
```

```
A    12
B    27
dtype: int64
0    13
1    13
2    13
dtype: int64
```

apply operations

```python
dict_sales = {
    'salesperson': ['Alice', 'Alice', 'Bob', 'Bob'],
    'month': ['Jan', 'Feb', 'Jan', 'Feb'],
    'sales': [100, 120, 110, 105]
    }
df_sales = pd.DataFrame(dict_sales)
print(df_sales)
```

```
  salesperson month  sales
0       Alice   Jan    100
1       Alice   Feb    120
2         Bob   Jan    110
3         Bob   Feb    105
```

apply operations

```
def top_month(group):
    best_row = group[group['sales'] == group['sales'].max()]
    return best_row[['month', 'sales']]
```

apply operations

```
df_sales_aply = df_sales.groupby('salesperson').apply(top_month)
print(df_sales_aply)
```

```
              month  sales
salesperson
Alice       1   Feb    120
Bob         2   Jan    110
```

apply operations

```python
students = pd.DataFrame({
    'name': ['a', 'b', 'c', 'd', 'e', 'f'],
    'score': [85, 90, 78, 88, 92, 74],
    'age': [12, 14, 16, 13, 15, 24]})
print(students)
```

```
  name  score  age
0    a     85   12
1    b     90   14
2    c     78   16
3    d     88   13
4    e     92   15
5    f     74   24
```

apply operations

```
def stage(age):
    if age < 13:
        return 'child'
    elif 13 <= age < 20:
        return 'teen'
    else:
        return 'adult'
```

apply operations

```
students.groupby(students['age'].apply(stage))['score'].agg('mean')
```

```
age
adult    74.0
child    85.0
teen     87.0
Name: score, dtype: float64
```