Intro to pandas DataFrame iteration

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

pandas recap

- See pandas overview in **Intermediate Python**
- Library used for data analysis
- Main data structure is the DataFrame
    - Tabular data with labeled rows and columns
    - Built on top of the NumPy array structure
- Chapter Objective:
    - Best practice for iterating over a pandas DataFrame

Baseball stats

```
import pandas as pd


baseball_df = pd.read_csv('baseball_stats.csv')
print(baseball_df.head())
```

```
  Team League  Year   RS   RA   W    G  Playoffs
0  ARI     NL  2012  734  688  81  162         0
1  ATL     NL  2012  700  600  94  162         1
2  BAL     AL  2012  712  705  93  162         1
3  BOS     AL  2012  734  806  69  162         0
4  CHC     NL  2012  613  759  61  162         0
```

Baseball stats

```
  Team
0  ARI
1  ATL
2  BAL
3  BOS
4  CHC
```

Arizona Diamondbacks (ARI) *Atlanta Braves (ATL)* *Baltimore Orioles (BAL)*

Boston Red Sox (BOS) *Chicago Cubs (CHC)*

Baseball stats

```
  Team League  Year   RS   RA   W    G  Playoffs
0  ARI     NL  2012  734  688  81  162         0
1  ATL     NL  2012  700  600  94  162         1
2  BAL     AL  2012  712  705  93  162         1
3  BOS     AL  2012  734  806  69  162         0
4  CHC     NL  2012  613  759  61  162         0
```

Calculating win percentage

```
import numpy as np

def calc_win_perc(wins, games_played):

    win_perc = wins / games_played

    return np.round(win_perc,2)
```

```
win_perc = calc_win_perc(50, 100)
print(win_perc)
```

```
0.5
```

Adding win percentage to DataFrame

```
win_perc_list = []
for i in range(len(baseball_df)):
    row = baseball_df.iloc[i]
    wins = row['W']
    games_played = row['G']
    win_perc = calc_win_perc(wins, games_played)
    win_perc_list.append(win_perc)
baseball_df['WP'] = win_perc_list
```

7. Adding win percentage to DataFrame

We'd like to create a new column in our baseball_df DataFrame that stores each team's win percentage for a season. To do this, we'll need to iterate over the DataFrame's rows and apply our calc_win_perc function. First, we create an empty win_perc_list to store all the win percentages we'll calculate. Then, we write a loop that will iterate over each row of the DataFrame. Notice that we are using an index variable (i) that ranges from zero to the number of rows that exist within the DataFrame. We then use the dot-iloc method to lookup each individual row within the DataFrame using the index variable. Now, we grab each team's wins and games played by referencing the W and G columns. Next, we pass the team's wins and games played to calc_win_perc to calculate the win percentages. Finally, we append win_perc to win_perc_list and continue the loop. We create our desired column in the DataFrame, called WP, by setting the column value equal to the win_perc_list.

Adding win percentage to DataFrame

```
print(baseball_df.head())
```

```
  Team League  Year   RS   RA   W    G  Playoffs    WP
0  ARI     NL  2012  734  688  81  162         0  0.50
1  ATL     NL  2012  700  600  94  162         1  0.58
2  BAL     AL  2012  712  705  93  162         1  0.57
3  BOS     AL  2012  734  806  69  162         0  0.43
4  CHC     NL  2012  613  759  61  162         0  0.38
```

Iterating with .iloc

```
%%timeit
win_perc_list = []

for i in range(len(baseball_df)):
    row = baseball_df.iloc[i]

    wins = row['W']
    games_played = row['G']

    win_perc = calc_win_perc(wins, games_played)
    win_perc_list.append(win_perc)

baseball_df['WP'] = win_perc_list
```

9. Iterating with .iloc
Looping over the DataFrame with dot-iloc gave us our desired output, but is it efficient? When estimating the runtime, the dot-iloc approach took 183 milliseconds, **which is pretty inefficient.**

```
183 ms ± 1.73 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Iterating with .iterrows()

```
win_perc_list = []

for i,row in baseball_df.iterrows():
    wins = row['W']
    games_played = row['G']

    win_perc = calc_win_perc(wins, games_played)

    win_perc_list.append(win_perc)

baseball_df['WP'] = win_perc_list
```

10. Iterating with .iterrows()
pandas comes with a few efficient methods for looping over a DataFrame. The first method we'll cover is the dot-iterrows method. This is similar to the dot-iloc method, but dot-iterrows returns each DataFrame row as a tuple of (index, pandas Series) pairs. This means each object returned from dot-iterrows contains the index of each row as the first element and the data in each row as a pandas Series as the second element. Notice that we still create the empty win_perc_list, but now we don't have to create an index variable to look up each row within the DataFrame. dot-iterrows handles the indexing for us! The remainder of the for loop stays the same to create a new win percentage column within our baseball_df DataFrame.

Iterating with .iterrows()

```
%%timeit
win_perc_list = []

for i,row in baseball_df.iterrows():

    wins = row['W']
    games_played = row['G']

    win_perc = calc_win_perc(wins, games_played)
    win_perc_list.append(win_perc)

baseball_df['WP'] = win_perc_list
```

11. Iterating with .iterrows()
Using dot-iterrows takes roughly half the time dot-iloc takes to iterate over our DataFrame. We'll explore more efficient ways to loop over a DataFrame later on in the chapter. But for now, we know that using dot-iloc is not efficient and shouldn't be used to iterate over a DataFrame.

```
95.3 ms ± 3.57 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Practice DataFrame iterating with .iterrows()

WRITING EFFICIENT PYTHON CODE

Another iterator method: .itertuples()

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

Team wins data

```
print(team_wins_df)
```

```
     Team  Year    W
0     ARI  2012   81
1     ATL  2012   94
2     BAL  2012   93
3     BOS  2012   69
4     CHC  2012   61
...
```

```python
for row_tuple in team_wins_df.iterrows():
    print(row_tuple)
    print(type(row_tuple[1]))
```

```
(0, Team      ARI
Year     2012
W          81
Name: 0, dtype: object)
<class 'pandas.core.series.Series'>

(1, Team      ATL
Year     2012
W          94
Name: 1, dtype: object)
<class 'pandas.core.series.Series'>
...
```

3. Iterating with .iterrows()
If we use dot-iterrows to loop over our team_wins_df DataFrame and print each row's tuple, we see that each row's values are stored as a pandas Series. Remember, dot-iterrows returns each DataFrame row as a tuple of (index, pandas Series) pairs, so we have to access the row's values with square bracket indexing.

Iterating with .itertuples()

```
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple)
```

```
Pandas(Index=0, Team='ARI', Year=2012, W=81)
Pandas(Index=1, Team='ATL', Year=2012, W=94)
...
```

```
print(row_namedtuple.Index)
```

```
1
```

```
print(row_namedtuple.Team)
```

```
ATL
```

4. Iterating with .itertuples()
But, we could use dot-itertuples to loop over our DataFrame rows instead. The dot-itertuples method returns each DataFrame row as a special data type called a namedtuple. These data types behave just like a Python tuple but have fields accessible using attribute lookup. What does this mean? Notice in the output that each printed row_namedtuple has an Index attribute and each column in our team_wins_df as an attribute. That means we can access each of these attributes with a lookup using a dot method. Here, we can print the last row_namedtuple's Index using row_namedtuple-dot-Index. We can print this row_namedtuple's Team with row_namedtuple-dot-Team, Year with row_namedtuple-dot-Year and so on.

Comparing methods

5. Comparing methods
When we compare dot-iterrows to dot-itertuples, we see that there is quite a bit of improvement! The reason dot-itertuples is more efficient than dot-iterrows is due to the way each method stores its output. Since dot-iterrows returns each row's values as a pandas Series, there is a bit more overhead.

```
%%timeit
for row_tuple in team_wins_df.iterrows():
    print(row_tuple)
```

```
527 ms ± 41.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple)
```

```
7.48 ms ± 243 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
for row_tuple in team_wins_df.iterrows():
    print(row_tuple[1]['Team'])
```

```
ARI
ATL
...
```

```
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple['Team'])
```

```
TypeError: tuple indices must be integers or slices, not str
```

```
for row_namedtuple in team_wins_df.itertuples():
    print(row_namedtuple.Team)
```

```
ARI
ATL
...
```

Let's keep iterating!

WRITING EFFICIENT PYTHON CODE

pandas alternative to looping

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

1. pandas alternative to looping

We've been looping over DataFrames row-by-row with ease in the past two lessons. But remember, in order to write efficient code, we want to avoid looping when possible. In this lesson, we'll explore an alternative to using dot-iterrows and dot-itertuples to perform calculations on a DataFrame.

```
print(baseball_df.head())
```

```
  Team League  Year   RS   RA   W    G  Playoffs
0  ARI     NL  2012  734  688  81  162         0
1  ATL     NL  2012  700  600  94  162         1
2  BAL     AL  2012  712  705  93  162         1
3  BOS     AL  2012  734  806  69  162         0
4  CHC     NL  2012  613  759  61  162         0
```

```
def calc_run_diff(runs_scored, runs_allowed):

    run_diff = runs_scored - runs_allowed

    return run_diff
```

Run differentials with a loop

```
run_diffs_iterrows = []

for i,row in baseball_df.iterrows():
    run_diff = calc_run_diff(row['RS'], row['RA'])
    run_diffs_iterrows.append(run_diff)

baseball_df['RD'] = run_diffs_iterrows
print(baseball_df)
```

```
     Team League  Year   RS   RA   W    G  Playoffs   RD
0     ARI     NL  2012  734  688  81  162         0   46
1     ATL     NL  2012  700  600  94  162         1  100
2     BAL     AL  2012  712  705  93  162         1    7
...
```

pandas .apply() method

- Takes a function and applies it to a DataFrame
  - Must specify an axis to apply (`0` for columns; `1` for rows)
- Can be used with anonymous functions (`lambda` functions)
- Example:

```
baseball_df.apply(
    lambda row: calc_run_diff(row['RS'], row['RA']),
    axis=1
)
```

7. Comparing approaches
But, using the dot-apply method took only 30 milliseconds. A definite improvement!

Run differentials with .apply()

```
run_diffs_apply = baseball_df.apply(
         lambda row: calc_run_diff(row['RS'], row['RA']),
         axis=1)
baseball_df['RD'] = run_diffs_apply
print(baseball_df)
```

```
     Team League  Year   RS   RA   W    G  Playoffs   RD
0     ARI     NL  2012  734  688  81  162         0   46
1     ATL     NL  2012  700  600  94  162         1  100
2     BAL     AL  2012  712  705  93  162         1    7
...
```

Comparing approaches

```
%%timeit
run_diffs_iterrows = []

for i,row in baseball_df.iterrows():
    run_diff = calc_run_diff(row['RS'], row['RA'])
    run_diffs_iterrows.append(run_diff)

baseball_df['RD'] = run_diffs_iterrows
```

```
86.8 ms ± 3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Comparing approaches

```
%%timeit
run_diffs_apply = baseball_df.apply(
          lambda row: calc_run_diff(row['RS'], row['RA']),
          axis=1)

baseball_df['RD'] = run_diffs_apply
```

```
30.1 ms ± 1.75 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

7. Comparing approaches
But, using the dot-apply method took only 30 milliseconds. A definite improvement!

Let's practice using pandas .apply() method!

WRITING EFFICIENT PYTHON CODE

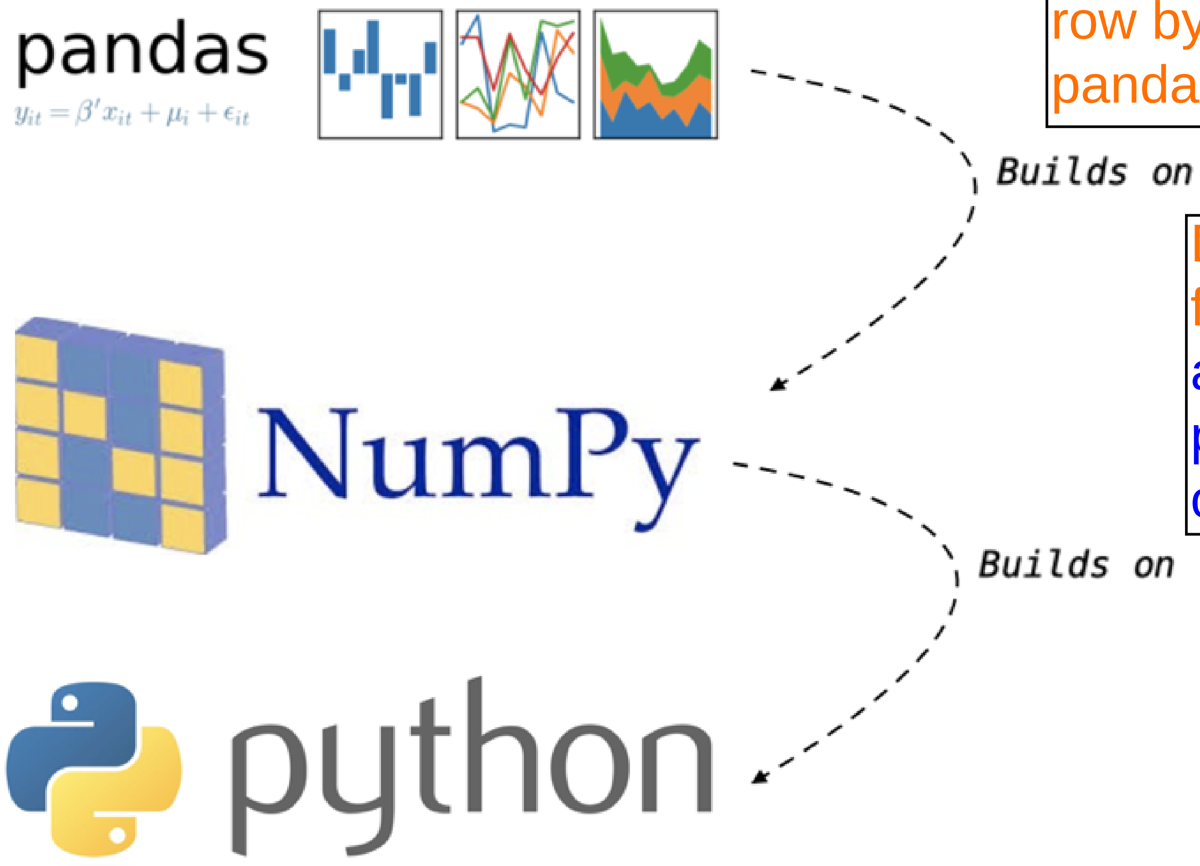
Optimal pandas iterating

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

pandas internals

- Eliminating loops applies to using pandas as well
- pandas is built on NumPy
  - Take advantage of NumPy array efficiencies



2. pandas internals
As you know, we should try to stay away from loops when writing Python code - and working with pandas is no exception. In the previous lessons, we were iterating over a DataFrame row by row in order to perform a calculation. pandas is a library that is built on NumPy.

Do you remember an array's broadcasting functionality? Broadcasting allows NumPy arrays to vectorize operations, so they are performed on all elements of an object at once.

```
print(baseball_df)
```

```
  Team League  Year   RS   RA   W    G  Playoffs
0  ARI     NL  2012  734  688  81  162         0
1  ATL     NL  2012  700  600  94  162         1
2  BAL     AL  2012  712  705  93  162         1
...
```

```
wins_np = baseball_df['W'].values
print(type(wins_np))
```

```
<class 'numpy.ndarray'>
```

```
print(wins_np)
```

```
[ 81  94  93 ...]
```

Power of vectorization

- Broadcasting (vectorizing) is extremely efficient!

```
baseball_df['RS'].values - baseball_df['RA'].values
```

```
array([  46,  100,    7, ...,  188,  110, -117])
```

4. Power of vectorization

The beauty of knowing that pandas is built on NumPy can be seen when taking advantage of a NumPy array's broadcasting abilities. Remember, this means we can vectorize our calculations, and perform them on entire arrays all at once! Instead of looping over a DataFrame, and treating each row independently, like we've done with dot-iterrows, dot-itertuples, and dot-apply, we can perform calculations on the underlying NumPy arrays of our baseball_df DataFrame. Here, we gather the RS and RA columns in our DataFrame as NumPy arrays, and use broadcasting to calculate run differentials all at once!

Run differentials with arrays

```
run_diffs_np = baseball_df['RS'].values - baseball_df['RA'].values
baseball_df['RD'] = run_diffs_np
print(baseball_df)
```

```
     Team League  Year   RS   RA   W    G  Playoffs   RD
0     ARI     NL  2012  734  688  81  162         0   46
1     ATL     NL  2012  700  600  94  162         1  100
2     BAL     AL  2012  712  705  93  162         1    7
3     BOS     AL  2012  734  806  69  162         0  -72
4     CHC     NL  2012  613  759  61  162         0 -146
...
```

Comparing approaches

```
%%timeit
run_diffs_np = baseball_df['RS'].values - baseball_df['RA'].values

baseball_df['RD'] = run_diffs_np
```

```
124 µs ± 1.47 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

6. Comparing approaches
When we time our NumPy arrays approach, we see that our run differential calculations take microseconds! All other approaches were reported in milliseconds. Our array approach is orders of magnitude faster than all previous approaches!

Let's put our skills into practice!

WRITING EFFICIENT PYTHON CODE

Congratulations!

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

What you have learned

- The definition of **efficient** and **Pythonic** code
- How to use Python's powerful built-in library
- The advantages of NumPy arrays
- Some handy magic commands to profile code
- How to deploy efficient solutions with `zip()`, `itertools`, `collections`, and set theory
- The cost of looping and how to eliminate loops
- Best practices for iterating with pandas DataFrames

Well done!

WRITING EFFICIENT PYTHON CODE