week4_demo

Vectorize or Apply: Is the first truly quicker?

If you want some more detail about speed related items:

top-5-tips-to-make-your-pandas-code-absurdly-fast

Apply on a Series (Single Col) vs. Vectorize

```
import pandas as pd
import numpy as np

# Example DataFrame
data = {
    "name": ["Alice", "Bob", "Charlie", "Diana"],
    "age": [25, 32, 41, 29],
    "height_cm": [165, 175, 180, 158]
}
df = pd.DataFrame(data)
print(df)
```

```
      name  age  height_cm
0    Alice   25        165
1      Bob   32        175
2  Charlie   41        180
3    Diana   29        158
```

Test 1: Apply a very simple calculation to a column.

Here we'll convert meters to cm.

Here I was seeing ~1 second difference between 100,000 iterations.

```
#simple function converting cm to m
def height_in_meters(cm):
  return cm/100.0

#vectorize the simple equation
height_in_meters_vec = np.vectorize(height_in_meters)


NUM_ITERATIONS = 10000  #how many times we'll use the vectorized equation
                        #in one test
NUM_TESTS = 10 #how many tests

import time #package for starting a timer
time_results_vec = np.array([]) #to record the times

for i in range(NUM_TESTS):    #we'll try the test a few times
  start = time.perf_counter() #start the timer

  #try vectorizing a bunch to convert cm -> m to a column
  for i in range(NUM_ITERATIONS):
    df['height_m'] = height_in_meters_vec(df["height_cm"])

  end = time.perf_counter()  #end time
  time_vec = end-start   #find how long it took to vectorize a bunch
  time_results_vec = np.append(time_results_vec, time_vec) #record the results

time_results_vec  #show the recorded times
```

```
array([1.20139006, 1.18420832, 1.1931015 , 1.17429117, 1.19634579,
       1.21232161, 1.19962644, 1.18801525, 1.19137661, 1.12591848])
```

```
time_results_apply = np.array([])

for i in range(NUM_TESTS): #we'll try the test a few times
  start = time.perf_counter() #start the timer

  #try applying the cm-> m function a bunch
  for i in range(NUM_ITERATIONS):
    df['height_m_apply'] = df['height_cm'].apply(height_in_meters)

  end = time.perf_counter()
  time_apply = end-start
  time_results_apply = np.append(time_results_apply, time_apply)

time_results_apply #show the recorded times
```

```
array([1.33103329, 1.31540052, 1.29457392, 1.28832367, 1.28829202,
       1.29098572, 1.30385756, 1.32198574, 1.30753461, 1.29664358])
```

Mean difference in times between apply on 1 col and vectorizing a function of 1 var:

```
 #on average, how much longer was apply than vec
(time_results_apply - time_results_vec).mean()

 #on average, how many times quicker was apply than vec
(time_results_apply / time_results_vec).mean()
```

```
1.099138107697461
```

Test 2: Applying to a dataframe vs. Vectorize

We'll use a BMI eqtn for this example.

This calculation requires 2 cols of data.

```
import pandas as pd

df = pd.DataFrame({
    'weight_kg': [50, 65, 80, 95],
    'height_cm': [160, 170, 180, 190]
})

df
```

	weight_kg	height_cm
0	50	160
1	65	170
2	80	180
3	95	190

```
# BMI method
def BMI(weight_kg, height_cm):
  return weight_kg / ((height_cm/100)**2)

#vectorize it
BMI_vec = np.vectorize(BMI)

time_results_vec2 = np.array([])  #record the results

for i in range(NUM_TESTS):         #we'll try the test a few times
  start = time.perf_counter()      #start the timer!

  #try using the vectorized function a bunch (now with 2 cols)
  for i in range(NUM_ITERATIONS):
    df['BMI'] = BMI_vec(df['weight_kg'], df['height_cm'])

  end = time.perf_counter()        #end the timer
  time_vec2 = end-start            #find how long it took to do a bunch
  time_results_vec2 = np.append(time_results_vec2, time_vec2) #record how long
                                                              #it took to do
                                                              #a bunch

time_results_vec2 #show the recorded times
```

```
array([1.73499643, 1.7210667 , 1.72026874, 1.77625238, 1.7291992 ,
       1.71914748, 1.75473658, 1.73634062, 1.81328119, 1.90404678])
```

```
#BMI method set up to use apply
def BMI2(row):
  return row['weight_kg'] / ((row['height_cm']/100.0)**2)

time_results_apply2 = np.array([])    #for the recorded results

for i in range(NUM_TESTS):         #we'll try the test a few times
  start = time.perf_counter()      #start the timer!

  #try applying a function a bunch (now depending on 2 cols)
  for i in range(NUM_ITERATIONS):
      df['BMI_apply'] = df.apply(BMI2, axis=1)

  end = time.perf_counter()        #end the timer
  time_apply2 = end-start          #find the time difference
  time_results_apply2 = np.append(time_results_apply2, time_apply2) #record the
                                                                    #time diff

time_results_apply2 #show the recorded times
```

```
array([3.85805458, 4.02059409, 3.6474799 , 3.77169132, 3.57106233,
       3.5833995 , 3.57658755, 3.66835815, 3.62914064, 3.56610792])
```

Mean difference in times between apply on 2 cols and vectorizing using a function of 2 vars:

```
 #on average, how much longer was apply than vec
(time_results_apply2 - time_results_vec2).mean()

 #on average, how many times quicker was apply than vec
(time_results_apply2 / time_results_vec2).mean()
```

```
2.097830099177019
```

List Comprehensions versus for loops

While we're on speed, for Loops are generally considered SLOW.

List comprehensions are quicker.

List comprehensions allow for quick, relative basic processing on each item in a collection

In the article I shared above: `vectorize < list comps < apply < for loops`

```
#Get 100 random numbers between 0-100
percentages = np.random.random(100) * 100
percentages
```

```
array([ 1.50707391, 69.22371145, 38.09300357, 63.87666404, 38.57687964,
        1.87092028, 33.83146194,  6.60146585, 19.44311432, 81.14600576,
       11.6252723 , 54.73928633, 72.8930286 , 13.62377   , 97.69623781,
       26.42791656, 49.10281831,  6.91221982, 48.92770175, 14.28424558,
       94.45638799, 65.94865356, 16.35748282,  4.36764134, 17.40431105,
       30.83432435,  1.4393946 , 94.13652321, 76.85449275, 40.02567905,
       16.40066602, 48.72503164, 31.66656314, 99.34646616, 87.13686932,
       16.0302097 ,  8.08547591, 53.41144052, 17.34111367, 59.32759506,
       18.53399741,  3.23288124, 23.79204036, 23.61778905, 90.6009346 ,
       46.11233491, 82.28168932, 58.11491292, 50.26552185, 18.69900571,
       76.0214583 , 13.35883569, 58.50686092, 88.0031286 , 14.58559135,
       28.77625262, 10.89245766, 38.64440602, 89.71668489, 38.79378465,
       76.44992377,  3.55317891, 84.58680783, 60.48710892, 28.58119447,
       39.40623383, 90.90059258, 34.64889304, 45.41290266, 29.48966867,
       17.68662928,  5.02640387, 56.38868231, 62.22083245, 37.48615111,
       77.26153985, 16.62075232, 11.77428528,  0.37377281, 28.97357548,
       23.9631285 , 30.28674664, 40.07829004, 46.00129867, 75.86291109,
        9.43149378, 34.47085433, 76.74908853, 34.28460023, 37.67103242,
       72.65578711,  8.84544489, 38.86210522, 35.83674941, 22.96356455,
       86.49094599, 19.41772496, 76.39207056, 70.66943501, 75.73613907])
```

```
#Here we use a for loop to convert each to a P or F
#We time how long this takes with a loop
p_or_f_results = np.array([""] * 100)

start_for_loop = time.perf_counter()

for i, grade in enumerate(percentages):
  if grade < 70:
    p_or_f_results[i] = "F"
  else:
    p_or_f_results[i] = "P"


end_for_loop = time.perf_counter()

p_or_f_time_loop = end_for_loop - start_for_loop

p_or_f_time_loop
p_or_f_results
```

```
array(['F', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'P', 'F', 'F', 'P',
       'F', 'P', 'F', 'F', 'F', 'F', 'F', 'P', 'F', 'F', 'F', 'F', 'F',
       'F', 'P', 'P', 'F', 'F', 'F', 'F', 'P', 'P', 'F', 'F', 'F', 'F',
       'F', 'F', 'F', 'F', 'F', 'P', 'F', 'P', 'F', 'F', 'F', 'P', 'F',
       'F', 'P', 'F', 'F', 'F', 'F', 'P', 'F', 'P', 'F', 'P', 'F', 'F',
       'F', 'P', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'P', 'F', 'F',
       'F', 'F', 'F', 'F', 'F', 'F', 'P', 'F', 'F', 'P', 'F', 'F', 'P',
       'F', 'F', 'F', 'F', 'P', 'F', 'P', 'P', 'P'], dtype='<U1')
```

```
#Here we use a for loop to convert each to a P or F
#We time how long this takes with a loop

start_list_comp = time.perf_counter()

p_or_f_results_list_comp  = ["P" if grade >= 70 else "F" for grade in percentages]

end_list_comp = time.perf_counter()

p_or_f_time_list_comp  = end_list_comp  - start_list_comp

p_or_f_time_list_comp
p_or_f_results_list_comp
```

```
['F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'P',
 'F',
 'F',
 'P',
 'F',
 'P',
 'F',
 'F',
 'F',
 'F',
 'F',
 'P',
 'F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'P',
        'P',
        'F',
        'F',
        'F',
        'F',
        'P',
        'P',
        'F',
        'F',
        'F',
        'F',
        'F',
        'F',
        'F',
        'F',
        'F',
        'P',
        'F',
        'P',
        'F',
        'F',
        'F',
        'P',
        'F',
        'F',
        'P',
        'F',
        'F',
        'F',
        'F',
        'P',
        'F',
        'P',
        'F',
        'P',
        'F',
        'F',
        'F',
        'P',
        'F',
        'F',
        'F',
        'F',
 'F',
 'F',
 'F',
 'F',
 'P',
 'F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'F',
 'P',
 'F',
 'F',
 'P',
 'F',
 'F',
 'P',
 'F',
 'F',
 'F',
 'F',
 'P',
 'F',
 'P',
 'P',
 'P']
```

```
p_or_f_time_list_comp / p_or_f_time_loop
```

```
0.8175396610954568
```