# Why and how loops should be avoided in ETLs/ELTs

* Author: Hassan Akhlaq

When it comes to writing ETLs using loops and iterations can impact the performance and quality pretty badly. Especially in these mentioned areas:

1. **Performance and Scalability**:
   * Iterative processes can be slow, especially when dealing with large volumes of data. They may not scale efficiently to handle increasing data loads.
   * ETL jobs without iterations can often take advantage of parallel processing and distributed computing, which significantly improves performance and scalability.
2. **Data Consistency**:
   * Iterative ETL processes can lead to data inconsistencies, as different iterations may run at different times and process subsets of data independently.
   * Non-iterative ETL processes ensure data consistency by processing all data in a single pass, reducing the risk of discrepancies and errors.
3. **Reduced Complexity**:
   * Iterations increase the complexity of ETL processes, requiring more code and maintenance.
   * Non-iterative ETL processes are typically simpler to design and maintain, which reduces the chances of errors and improves long-term maintainability.
4. **Resource Consumption**:
   * Iterative processes may consume more system resources, leading to higher infrastructure costs.
   * Non-iterative ETL processes are more resource-efficient and can be optimized for cost-effectiveness.
5. **Error Handling**:
   * Iterations can complicate error handling and recovery mechanisms, making it more challenging to monitor and troubleshoot issues.
   * Non-iterative ETL processes can offer better error reporting and centralized logging, simplifying the identification and resolution of problems.
6. **Data Dependencies**:
   * In some cases, iterations may introduce data dependencies, where the output of one iteration depends on the previous one.
   * Non-iterative ETL processes eliminate these dependencies, making it easier to understand and predict data flows.
7. **Data Volume Handling**:
   * Iterations may not be well-suited for handling very large data volumes, leading to performance bottlenecks and potential processing failures.
   * Non-iterative ETL processes can be designed to efficiently handle large data volumes through optimizations and distributed processing.
8. **Maintainability and Extensibility**:
   * Non-iterative ETL processes tend to be more modular and easier to maintain.
   * They are often more extensible, allowing for the addition of new data sources and transformations without significant code rework.
9. **Real-Time and Near Real-Time Processing**:
   * Iterative processes are generally not well-suited for real-time or near real-time ETL requirements.
   * Non-iterative ETL processes can be optimized for timely data processing.

## Efficient Python Coding for Data Engineers and Scientists

### 1.1. What is meant by efficient code?

Efficient refers to code that satisfies two key concepts.

* efficient code is fast and has a small latency between execution and returning a result.
* efficient code allocates resources skillfully and isn’t subjected to unnecessary overhead

### 1.2. Python Standard Libraries

Python Standard Libraries are the Built-in components and libraries of python. These libraries come with every Python installation and are commonly cited as one of Python’s greatest strengths.

“It’s worth noting that Python’s built-ins have been optimized to work within the Python language itself. Therefore, we should default to using a built-in solution (if one exists) rather than developing our own.”

We will focus on certain built-in types, functions, and modules. The types that we will focus on are :

* Lists
* Tuples
* Sets
* Dicts

The built-in functions that we will focus on are:

* print()
* len()
* range()
* round()
* enumerate()
* map()
* zip()

### Python Functions

Let's start exploring some of the mentioned functions:

**range()**: This is a handy function whenever we want to create a sequence of numbers. Suppose we wanted to create a list of integers from zero to ten.

—--------

*# range(start,stop)*

nums = range(0,11)

nums\_list = list(nums)

print(nums\_list)

*# output [0,1,2,3,4,5,6,7,8,9,10]*

—--------

**enumerate()**: Another useful built-in function is enumerate. enumerate creates an index item pair for each item in the object provided. For example, calling enumerate on the list letters produces a sequence of indexed values.

—--------

Letters = ['a','b','c','d']

indexed\_letters = enumerate(letters)

indexed\_letters\_list = list(indexed\_letters)

print(indexed\_letters\_list)

*#output = [(0,’a’),(1,’b’),(2,’c’),(3,’d’)]*

—--------

**map()**: The last notable built-in function we’ll cover is the map() function. map applies a function to each element in an object. Notice that the map function takes two arguments; first, the function you’d like to apply, and second, the object you’d like to apply that function on.

—--------

nums = [1.5, 2.3, 3.4, 4.6, 5.0]

rnd\_nums = map(round, nums)

print(list(rnd\_nums))

*#output = [2,2,3,5,5]*

—--------

“

The map function can also be used with a lambda, or, an anonymous function. Notice here, that we can use the map function and a lambda expression to apply a function, which we’ve defined on the fly, to our original list nums. The map function provides a quick and clean way to apply a function to an object iteratively without writing a for loop.

—--------

*# map() with lambda*

nums = [1, 2, 3, 4, 5]

sqrd\_nums = map(lambda x: x \*\* 2, nums)

print(list(sqrd\_nums))

—--------

NumPy, or Numerical Python, is an invaluable Python package for Data Scientists. It is the fundamental package for scientific computing in Python and provides a number of benefits for writing efficient code.

NumPy arrays provide a fast and memory-efficient alternative to Python lists. Typically, we import NumPy as np and use np dot array to create a NumPy array.

—--------

*# python list*

nums\_list = list(range(5))

print(nums\_list)

*# using numpy alternative to python lists*

import numpy as np

nums\_np = np.array(range(5))

print(nums\_np)

*#output = [0,1,2,3,4] [0 1 2 3 4]*

—--------

NumPy arrays are homogeneous, which means that they must contain elements of the same type. We can see the type of each element using the .dtype method è

—--------

nums\_np\_ints = np.array([1, 2, 3])

print(nums\_np\_ints.dtype)

—--------

“Homogeneity allows NumPy arrays to be more memory efficient and faster than Python lists. Requiring all elements be the same type eliminates the overhead needed for data type checking.”

NumPy arrays also have a special technique called boolean indexing. Suppose we wanted to gather only positive numbers from the sequence listed here. With an array, we can create a boolean mask using a simple inequality. Indexing the array is as simple as enclosing this inequality in square brackets. However, to do this using a list, we need to write a for loop to filter the list or use a list comprehension. In either case, using a NumPy array to the index is less verbose and has a faster runtime.

—--------

nums = [-2, -1, 0, 1, 2]

nums\_np = np.array(nums)

*# Boolean indexing*

print(nums\_np[nums\_np > 0])

*# No boolean indexing for lists*

*# For loop (inefficient option)*

pos = []  
for num in nums:  
 if num > 0:

pos.append(num)

print(pos)

*# List comprehension (better option but not best)*

pos = [num for num in nums if num > 0]

print(pos)

—--------

## 1. Making Your Code Efficient

### 1.1. Efficiently Combining, Counting, and Iterating

***Combining Objects***

In this subsection, we’ll cover combining, counting, and iterating over objects efficiently in python. Suppose we have two lists: one of the names and the other is the age for each of them. We want to combine these lists so that each name is stored next to its age. We can iterate over the names list using enumerate and grab each name's corresponding age using the index variable.

—--------

*# combining objects*names = ['Ahmed', 'Youssef', 'Mohammed']  
age = [25, 27, 40]  
combined = []  
for i,name in enumerate(names):  
 combined.append((name, age[i]))

print(combined)

—--------

But Python’s built-in function ***zip*** provides a more elegant solution. The name “zip” describes how this function combines objects like a zipper on a jacket (making two separate things become one). zip returns a zip object that must be unpacked into a list and printed to see the contents. Each item is a tuple of elements from the original lists.

—--------

*# Combining objects with zip*  
combined\_zip = zip(names, age)  
print(type(combined\_zip))  
combined\_zip\_list = [\*combined\_zip]  
print(combined\_zip\_list)

—--------

### 1.2. Introduction to The Set Theory

Often, we’d like to compare two objects to observe similarities and differences between their contents. When doing this type of comparison, it’s best to leverage a branch of mathematics called the set theory. As you know, Python comes with a built-in ***set*** data type. Sets come with some handy methods we can use for comparing such as:

* · **intersection()** : all elements that are in both sets
* · **difference():** all elements in one set but not the other
* · **symmetric\_difference():** all elements in exactly one set
* · **union():** all elements that are in either set

When we’d like to compare objects multiple times and in different ways, we should consider storing our data in sets to use these efficient methods. Another nice feature of Python sets is their ability to quickly check if a value exists within its members. We call this membership testing using the **in** operator. We will go through how using the **in** operator with a set is much faster than using it with a list or tuple.

Suppose we had two lists of Pokémon names: list\_a and list\_b and we would like to compare these lists to see which Pokémon appear in both lists. We could first use a nested for loop to compare each item in list\_a to each item in list\_b and collect only those items that appear in both lists. But, iterating over each item in both lists is extremely inefficient.

—--------

#Inefficient Approach

# Comparing objects with loops  
list\_a = ['Bulbasaur', 'Charmander', 'Squirtle']  
list\_b = ['Caterpie', 'Pidgey', 'Squirtle']

in\_common = []  
for pokemon\_a in list\_a:  
 for pokemon\_b in list\_b:  
 if pokemon\_a == pokemon\_b:  
 in\_common.append(pokemon\_a)

print(in\_common)

—--------

However, a better way is to use Python’s **set** data type to compare these lists. By converting each list into a set, we can use the dot-intersection method to collect the Pokémon shared between the two sets. One simple line of code and no need for a loop!

—--------

#Efficient Approach

# Comparing objects using intersection()

list\_a = ['Bulbasaur', 'Charmander', 'Squirtle']  
list\_b = ['Caterpie', 'Pidgey', 'Squirtle']

set\_a = set(list\_a)  
set\_b = set(list\_b)

set\_a.intersection(set\_b)

—--------

We can see that using sets is much faster than using for loops. We can also use a set method to see Pokémon that exist in one set but not in another. To gather Pokémon that exist in set\_a but not in set\_b, use set\_a.difference(set\_b).

—--------

set\_a = {'Bulbasaur', 'Charmander', 'Squirtle'}  
set\_b = {'Caterpie', 'Pidgey', 'Squirtle'}

set\_a.difference(set\_b)

—--------

To collect Pokémon that exist in exactly one of the sets (but not both), we can use a method called the symmetric difference.

—--------

set\_a = {'Bulbasaur', 'Charmander', 'Squirtle'}

set\_b = {'Caterpie', 'Pidgey', 'Squirtle'}

set\_a.symmetric\_difference(set\_b)

—--------

Finally, we can combine these sets using the **.union** method. This collects all of the unique Pokémon that appear in either or both sets.

—--------

set\_a = {'Bulbasaur', 'Charmander', 'Squirtle'}

set\_b = {'Caterpie', 'Pidgey', 'Squirtle'}

set\_a.union(set\_b)

—--------

## How to Eliminate Loops from Your Python Code

Steps:

1. Git Rid of the Loop
   * Eliminate Loops with List Comprehension, Map Function, & itertools
   * Eliminate Loops with NumPy
2. Writing Better Loops
   * Moving calculations above a loop
   * Holistic Conversions

### 1.1. Eliminate Loops with List Comprehension, Map Function, & itertools

Suppose we have a list of lists, called poke\_stats, that contains statistical values for each Pokémon.

“”# List of HP, Attack, Defense, Speed

poke\_stats = [

[90, 92, 75, 60],

[25, 20, 15, 90],

[65, 130, 60, 75],

]””

Now we will iterate through the elements of this array using for loop and print the timing:

totals = []

for row in poke\_stats:  
 totals.append(sum(row)) it takes 2.92us

We will do the same thing but using list comprehension:

totals\_comp = [sum(row) for row in poke\_stats] 2.8us

Finally, we will iterate through the elements using the **map** function:

totals\_map = [\*map(sum, poke\_stats)] 2.1us

### 1.2. Eliminate Loops with NumPy

Another powerful technique for eliminating loops is to use the NumPy package. Suppose we had the same collection of statistics we used in a previous example but stored in a NumPy array instead of a list of lists.

We’d like to collect the average stat value for each Pokémon (or row) in our array. We could use a loop to iterate over the array and collect the row averages.

—--------

avgs = []

for row in poke\_stats:  
 avg = np.mean(row)  
 avgs.append(avg) 6.42us

—--------

But, NumPy arrays allow us to perform calculations on entire arrays all at once. Here, we use the **.mean** method and specify an axis equal to 1 to calculate the mean for each row (meaning we calculate an average across the column values). This eliminates the need for a loop and is much more efficient.

—--------

avgs = []

for row in poke\_stats:  
 avg = np.mean(row)  
 avgs.append(avg) 3ns

—--------

### 2. Writing Better Loops (When unavoidable)

We’ve discussed how loops can be costly and inefficient. But, sometimes you can’t eliminate a loop. In this section, we’ll explore how to make loops more efficient when looping is unavoidable. The best way to make a loop more efficient is to analyze what’s being done within the loop. ***We want to make sure that we aren’t doing unnecessary work in each iteration***. If a calculation is performed for each iteration of a loop, but its value doesn’t change with each iteration, ***it’s best to move this calculation outside (or above) the loop***. If a loop is converting data types with each iteration, it’s possible that this conversion can be done outside (or below) the loop using a map function. Anything that can be done once should be moved outside of a loop

### 2.1. Moving calculations above a loop

| *# Calculate total average inside the loop*  for pokemon,attack in zip(names, attacks):  total\_attack\_avg = attacks.mean()   if attack > total\_attack\_avg:  print("{}'s attack: {} > average: {}!".format(pokemon, attack, total\_attack\_avg)) | *# Calculate total average once (outside the loop)*  total\_attack\_avg = attacks.mean()  for pokemon,attack in zip(names, attacks):  if attack > total\_attack\_avg:  print(  "{}'s attack: {} > average: {}!"  .format(pokemon, attack, total\_attack\_avg)) |
| --- | --- |

### 

### 2.2. Holistic Conversions

Another way ***to make loops more efficient is to use holistic conversions outside (or below) the loop***. In the example below we have three lists from the 720 Pokémon dataset: a list of each Pokémon’s name, a list corresponding to whether or not a Pokémon has a legendary status, and a list of each Pokémon’s generation.

We want to combine these objects so that each name, status, and generation is stored in an individual list. To do this, we’ll use a loop that iterates over the output of the zip function. Remember, zip returns a collection of tuples, so we need to convert each tuple into a list since we want to create a list of lists as our output. Then, we append each individual poke\_list to our poke\_data output variable. By printing the result, we see our desired list of lists.

However, ***converting each tuple to a list within the loop is not very efficient***. Instead, we should ***collect all of our poke\_tuples together, and use the map function to convert each tuple to a list***. The loop no longer converts tuples to lists with each iteration. Instead, we moved this tuple to list conversion outside (or below) the loop. That way, we convert data types all at once (or holistically) rather than converting in each iteration.

| # Conversion inside the loop  import pandas as pd  pokemon = pd.read\_csv('pokemon.csv')  names\_list = pokemon['Name']  legend\_status\_list = pokemon['Legendary']  generations\_list = pokemon['Generation']  Now let's run the loop and convert the tuples inside the loop and see the timing:  poke\_data = []  for poke\_tuple in zip(names\_list, legend\_status\_list, generations\_list):  poke\_list = list(poke\_tuple)  poke\_data.append(poke\_list) | # Conversion outside the loop  poke\_data\_tuples = []  for poke\_tuple in zip(names\_list, legend\_status\_list, generations\_list):  poke\_data\_tuples.append(poke\_tuple)    poke\_data = [\*map(list,poke\_data\_tuples)] |
| --- | --- |

Runtimes show that converting each tuple to a list outside of the loop is more efficient.

## Best Practices To Use Pandas Efficiently

* Selecting & Replacing Values Effectively
* Selecting Rows & Columns Efficiently using .iloc[] & .loc[]
* Replacing Values in a DataFrame Effectively
* Summary of best practices for selecting and replacing values
* Iterate Effectively Through Pandas DataFrame
* Looping effectively using the .iterrows()
* Looping effectively using .apply()
* Looping effectively using vectorization
* Summary of best practices for looping through DataFrame
* Transforming Data Effectively With .groupby()
* Common functions used with .groupby()
* Missing value imputation using .groupby() & .transform()
* Data filtration using the .groupby() & .filter()
* Summary of Best Practices

### 2. Selecting & Replacing Values Effectively

Let's first start with two of the most common tasks that you will commonly do on your DataFrame, especially in the data manipulation phase of a data science project. These two tasks are selecting specific and random rows and columns efficiently and the usage of the **replace()** function for replacing one or multiple values using lists and dictionaries

#### 2.1. Selecting Rows & Columns Efficiently using .iloc[] & .loc[]

In this subsection, we will introduce how to locate and select rows efficiently from dataframes using **.iloc[]** & **.loc[]** pandas functions. We will use **iloc[]** for the index number locator and **loc[]** for the index name locator.

In the example below we will select the first 500 rows of the poker dataset. Firstly by using the **.loc[]** function, and then by using the **.iloc[]** function.

—--------  
*# Specify the range of rows to select*rows = range(0, 500)  
# Time selecting rows using .loc[]   
loc\_start\_time = time.time()   
poker\_data.loc[rows]   
loc\_end\_time = time.time()   
print("Time using .loc[] : {} sec".format(loc\_end\_time - loc\_start\_time))

0.002455711 sec  
—--------  
*# Specify the range of rows to select*rows = range(0, 500)  
*# Time selecting rows using .iloc[]*iloc\_start\_time = time.time()  
Poker\_data.iloc[rows]  
iloc\_end\_time = time.time()  
print("Time using .iloc[]: {} sec".format(iloc\_end\_time - iloc\_start\_time))  
0.000738859 sec

—--------

While these two methods have the same syntax, **iloc[]** performs almost 70% faster than **loc[]**. The **.iloc[] function** takes advantage of the order of the indices, which are already sorted, and is therefore faster.

#### 2.2. Replacing Values in a DataFrame Effectively

Replacing values in a DataFrame is a very important task, especially in the data cleaning phase. Since you will have to keep the whole values that represent the same object the same.

There are two ways to do it; the first one is simply defining which values we want to replace, and then what we want to replace them with. This is shown in the code below:

—--------  
start\_time = time.time()

names['Gender'].loc[names.Gender=='female'] = 'FEMALE'

end\_time = time.time()

pandas\_time = end\_time - start\_time

print("Replace values using .loc[]: {} sec".format(pandas\_time))

0.0054879 sec  
—--------

The second method is to use the panda's built-in function **.replace()** as shown in the code below:

—--------  
start\_time = time.time()  
names['Gender'].replace('female', 'FEMALE', inplace=True)   
end\_time = time.time()

replace\_time = end\_time - start\_time

print("Time using replace(): {} sec".format(replace\_time))

0.002129 sec  
—--------

We can see that there is a difference in time complexity with the built-in function **157%** faster than using the **.loc()** method to find the rows and columns index of the values and replace it.

Finally, we can also use **dictionaries** to replace both single and multiple values in your DataFrame. This will be very helpful if you would like to multiple replacing functions in one command.

We’re going to use dictionaries to replace every male’s gender with BOY and every female’s gender with GIRL.

—--------  
names = pd.read\_csv('Popular\_Baby\_Names.csv')

start\_time = time.time()

names['Gender'].replace({'MALE':'BOY', 'FEMALE':'GIRL', 'female': 'girl'}, inplace=True)

end\_time = time.time()

dict\_time = end\_time - start\_time

print("Time using .replace() with dictionary: {} sec".format(dict\_time))  
—--------

### 3. Iterate Effectively Through Pandas DataFrame

As a data scientist, you will need to iterate through your dataframe extensively, especially in the data preparation and exploration phase, so it is important to be able to do this efficiently, as it will save you much time and give space for more important work. We will walk through three methods to make your loops much faster and more efficient:

· Looping using the **.iterrows()** function

· Looping using the **.apply()** function

· Vectorization

#### **3.1. Looping effectively using** .iterrows()

Before we talk about how to use the **.iterrows()** function to improve the looping process, let’s refresh the notion of a generator function.

**Generators** are a simple tool to create iterators. Inside the body of a generator, instead of return statements, you will find only **yield()** statements. There can be just one, or several **yield()** statements. Here, we can see a generator, **city\_name\_generator()**, that produces four city names. We assign the generator to the variable **city\_names** for simplicity.

—--------  
def city\_name\_generator():

yield('New York')

yield('London')

yield('Tokyo')

yield('Sao Paolo')

city\_names = city\_name\_generator()   
—--------

To access the elements that the generator yields we can use Python’s **next()** function. Each time the **next()** command is used, the generator will produce the next value to yield, until there are no more values to yield. We here have 4 cities. Let’s run the next command four times and see what it returns:

next(city\_names)

As we can see that every time we run the **next()** function it will print a new city name.

Let's go back to the .**iterrows()** function. The **.iterrows()** function is a property of every pandas DataFrame. When called, it produces a list with two elements. We will use this generator to iterate through each line of our **poker** DataFrame. The first element is the index of the row, while the second element contains a pandas Series of each feature of the row: the Symbol and the Rank for each of the five cards. It is very similar to the notion of the **enumerate()** function, which when applied to a list, returns each element along with its index.

The most intuitive way to iterate through a Pandas DataFrame is to use the **range()** function, which is often called crude looping. This is shown with the code below:

—--------  
start\_time = time.time()

for index in range(poker\_data.shape[0]):

next

print("Time using range(): {} sec".format(time.time() - start\_time))

0.0036385 sec  
—--------

One smarter way to iterate through a pandas DataFrame is to use the **.iterrows()** function, which is optimized for this task. We simply define the ‘**for**’ loop with two iterators, one for the number of each row and the other for all the values.

Inside the loop, the **next()** command indicates that the loop moves to the next value of the iterator, without actually doing something.

—--------  
data\_generator = poker\_data.iterrows()  
start\_time = time.time()

for index, values in data\_generator:   
 next

print("Time using .iterrows(): {} sec".format(time.time() - start\_time))

1.258336 sec  
—--------

Comparing the two computational times we can also notice that the use of **.iterrows()** does not improve the speed of iterating through pandas DataFrame. It is very useful though when we need a cleaner way to use the values of each row while iterating through the dataset.

#### **3.2. Looping effectively using** .apply()

Now we will use the **.apply()** function to be able to perform a specific task while iterating through a pandas DataFrame. The **.apply()** function does exactly what it says; it applies another function to the whole DataFrame.

The syntax of the **.apply()** function is simple: we create a mapping, using a lambda function in this case, and then declare the function we want to apply to every cell. Here, we’re applying the square root function to every cell of the DataFrame. In terms of speed, it matches the speed of just using the NumPy **sqrt()** function over the whole DataFrame.

what if we want to calculate the sum of the rank of all the cards in each hand? In this case, we will use the .apply() function the same way as we did before, but we need to add **‘axis=1’** at the end of the line to specify we’re applying the function to each row.

—--------  
apply\_start\_time = time.time()

poker\_data[['R1', 'R2', 'R3', 'R4', 'R5']].apply(lambda x: sum(x), axis=1)

apply\_end\_time = time.time()

apply\_time = apply\_end\_time - apply\_start\_time

print("Time using .apply(): {} sec".format(time.time() - apply\_start\_time))

0.221885 sec  
—--------

Then, we will use the **.iterrows()** function we saw previously, and compare their efficiency.

—--------  
for\_loop\_start\_time = time.time()

for ind, value in poker\_data.iterrows():  
 sum([value[1], value[3], value[5], value[7], value[9]])

for\_loop\_end\_time = time.time()

for\_loop\_time = for\_loop\_end\_time - for\_loop\_start\_time

print("Time using .iterrows(): {} sec".format(for\_loop\_time))

1.14859 sec  
—--------

Using the **.apply()** function is significantly faster than the **.iterrows()** function, with a magnitude of around 400 percent, which is a massive improvement!

By comparing the .**apply()** function with the native panda's function for summing over rows, we can see that pandas’ native **.sum()** functions perform the same operation faster.

—--------  
pandas\_start\_time = time.time()

poker\_data[['R1', 'R1', 'R3', 'R4', 'R5']].sum(axis=0)

pandas\_end\_time = time.time()

pandas\_time = pandas\_end\_time - pandas\_start\_time

print("Time using pandas: {} sec".format(pandas\_time))

0.003397515 sec  
—--------

#### 3.3. Looping effectively using vectorization

To understand how we can reduce the amount of iteration performed by the function, recall that the fundamental units of Pandas, DataFrames, and Series, are both based on arrays. Pandas perform more efficiently when an operation is performed to a whole array than to each value separately or sequentially. This can be achieved through **vectorization**. ***Vectorization is the process of executing operations on entire arrays.***

In the code below we want to calculate the sum of the ranks of all the cards in each hand. In order to do that, we slice the poker dataset keeping only the columns that contain the ranks of each card. Then, we call the built-in **.sum()** property of the DataFrame, using the parameter axis = 1 to denote that we want the sum for each row. In the end, we print the sum of the first five rows of the data.

—--------  
start\_time\_vectorization = time.time()

poker\_data[['R1', 'R2', 'R3', 'R4', 'R5']].sum(axis=1)

end\_time\_vectorization = time.time()

vectorization\_time = end\_time\_vectorization - start\_time\_vectorization

print("Time using pandas vectorization: {} sec".format(vectorization\_time))

0.009327 sec  
—--------

We can also use another vectorization method to effectively iterate through the DataFrame which is using Numpy arrays to vectorize the DataFrame.

The NumPy library, which defines itself as a “fundamental package for scientific computing in Python”, ***performs operations under the hood in optimized, pre-compiled C code***. Similar to pandas working with arrays, NumPy operates on arrays called **ndarrays**. A major difference between Series and ndarrays is that ndarrays leave out many operations such as indexing, data type checking, etc. As a result, operations on NumPy arrays can be significantly faster than operations on pandas Series. NumPy arrays can be used in place of the pandas Series when the additional functionality offered by the pandas Series isn’t critical.

For the problems we explore in this article, we could use NumPy ndarrays instead of the pandas series. The question at stake is whether this would be more efficient or not.

Again, we will calculate the sum of the ranks of all the cards in each hand. We convert our rank arrays from pandas Series to NumPy arrays simply by using the **.values** method of pandas Series, which returns a pandas Series as a NumPy **ndarray**. As with vectorization on the series, passing the NumPy array directly into the function will lead pandas to apply the function to the entire vector.

—--------  
start\_time = time.time()

poker\_data[['R1', 'R2', 'R3', 'R4', 'R5']].values.sum(axis=1)

print("Time using NumPy vectorization: {} sec" .format(time.time() - start\_time))

0.00174546 sec  
—--------

*“ At this point, we can see that vectorizing over the pandas series achieves the overwhelming majority of optimization needs for everyday calculations. However, if speed is of the highest priority, we can call in reinforcements in the form of the* ***NumPy*** *Python library. Compared to the previous state-of-the-art method, the panda's optimization, we still get an improvement in the operating time. ”*

### 4. Transforming Data Effectively With .groupby()

In this last section of the article, we will use how to use the **.groupby()** function effectively to group the entries of a DataFrame according to the values of a specific feature. The **.groupby()** method is applied to a DataFrame and groups it according to a feature. Then, we can apply some simple or more complicated functions on that grouped object. This is a very important tool for every data scientist working on tabular or structured data as it will help you to manipulate data easily and in a more effective way.

#### 4.1. Common functions used with .groupby()

One of the simplest methods to apply to an aggregated group is the **.count().** In the example below we will apply this to the restaurant dataset. At first, we group the restaurant data according to whether the customer was a smoker or not. Then, we apply the **.count()** method. We obtain the count of smokers and non-smokers.

—--------  
restaurant = pd.read\_csv('restaurant\_data.csv')

restaurant\_grouped = restaurant.groupby('smoker')

print(restaurant\_grouped.count())  
—--------

It is no surprise that we get the same results for all the features, as the **.count()** method counts the number of occurrences of each group in each feature. As there are no missing values in our data, the results should be the same in all columns.

After grouping the entries of the DataFrame according to the values of a specific feature, we can apply any kind of **transformation** we are interested in. Here, we are going to apply the z score, a normalization transformation, which is the distance between each value and the mean, divided by the standard deviation. This is a very useful transformation in statistics, often used with the z-test in standardized testing. To apply this transformation to the grouped object, we just need to call the .transform() method containing the lambda transformation we defined.

This time, we will group according to the type of **meal**: was it a dinner or a lunch? As the z-score transformation is a group-related transformation, the resulting table is just the original table. For each element, we subtract the mean and divide it by the standard deviation of the group it belongs to. We can also see that numerical transformation are applied only to numerical features of the DataFrame.

—--------  
zscore = lambda x: (x - x.mean() ) / x.std()

restaurant\_grouped = restaurant.groupby('time')

restaurant\_transformed = restaurant\_grouped.transform(zscore)

restaurant\_transformed.head()  
—--------

While the **transform()** method simplifies things a lot, is it actually more efficient than using native Python code? As we did before, we first group our data, this time according to **sex**. Then we apply the z-score transformation we applied before, measuring its efficiency. We omit the code for measuring the time of each operation here, as you are already familiar with this. We can see that with the use of the transform() function, we achieve a massive speed improvement. On top of that, we’re only using one line to perform the operation of interest.

—--------  
restaurant.groupby('sex').transform(zscore)

mean\_female = restaurant.groupby('sex').mean()['total\_bill']['Female']

mean\_male = restaurant.groupby('sex').mean()['total\_bill']['Male']

std\_female = restaurant.groupby('sex').std()['total\_bill']['Female']

std\_male = restaurant.groupby('sex').std()['total\_bill']['Male']

for i in range(len(restaurant)):

if restaurant.iloc[i][2] == 'Female':  
  
 restaurant.iloc[i][0] = (restaurant.iloc[i][0] - mean\_female)/std\_female

else:

restaurant.iloc[i][0] = (restaurant.iloc[i][0] - mean\_male)/std\_male  
—--------

#### 4.2. Missing value imputation using .groupby() & .transform()

Now that we have seen why and how to use the **transform()** function on a grouped pandas object, we will address a very specific task that is imputing missing value.

Before we actually see how we can use the **transform()** function for missing value imputation, we will see how many missing values there are in our variable of interest in each of the groups. We can see below the number of data points in each of the “**time”** feature and they are 176+68 = 244.  
—--------  
prior\_counts = restaurant.groupby('time')  
prior\_counts['total\_bill'].count()   
—--------

Next, we will create a **restaurant\_nan** dataset, in which the total bill of 10% random observations was set to **NaN** using the code below:

—--------  
import pandas as pd

import numpy as np

p = 0.1 #percentage missing data required

mask = np.random.choice([np.nan,1], size=len(restaurant), p=[p,1-p])

restaurant\_nan = restaurant.copy()

restaurant\_nan['total\_bill'] = restaurant\_nan['total\_bill'] \* mask  
—--------

Now, let's print the number of data points in each of the “**time”** feature and we can see that they are now 155+62 = 217. Since the total data points we have are 244 then the missing data points are 24 which is equal to 10%.

—--------  
prior\_counts = restaurant.groupby('time')

prior\_counts['total\_bill'].count()  
—--------

After counting the number of missing values in our data, we will show how to fill these missing values with a group-specific function. The most common choices are the mean and the median, and the selection has to do with the skewness of the data. As we did before, we define a **lambda** transformation using the **fillna**() function to replace every missing value with its group average. As before, we group our data according to the time of the meal and then replace the missing values by applying the pre-defined transformation.

—--------  
*# Missing value imputation*

missing\_trans = lambda x: x.fillna(x.mean())

restaurant\_nan\_grouped = restaurant\_nan.groupby('time')['total\_bill']

restaurant\_nan\_grouped.transform(missing\_trans)  
—--------

As we can see, the observations at index 0 and index 4 are exactly the same, which means that their missing value has been replaced by their group’s mean.

Also, we can see the computation time using this method is 0.007 seconds.

Let's compare this with the conventional method:

—--------  
start\_time = time.time()

mean\_din = restaurant\_nan.loc[restaurant\_nan.time == "Dinner"]["total\_bill"].mean()

mean\_lun = restaurant\_nan.loc[restaurant\_nan.time == "Lunch"]["total\_bill"].mean()

for row in range(len(restaurant\_nan)):

if restaurant\_nan.iloc[row]["time"] == "Dinner":

restaurant\_nan.loc[row, "total\_time"] = mean\_din

else:

restaurant\_nan.loc[row, "total\_time"] = mean\_lun

print(

"Results from the above operation calculated in %s seconds"

% (time.time() - start\_time))

—--------

0.1056 sec

We can see that using the **.transform()** function applied on a grouped object performs faster than the native Python code for this task.

#### 4.3. Data filtration using the .groupby() & .filter()

Now we will discuss how we can use the **filter()** function on a grouped pandas object. This allows us to include only a subset of those groups, based on some specific conditions.

Often, after grouping the entries of a DataFrame according to a specific feature, we are interested in including only a subset of those groups, based on some conditions. Some examples of filtration conditions are the number of missing values, the mean of a specific feature, or the number of occurrences of the group in the dataset.

We are interested in finding the mean amount of tips given, on the days when the mean amount paid to the waiter is more than 20 USD. The **.filter()** function accepts a **lambda** function that operates on a DataFrame of each of the groups. In this example, the lambda function selects “total\_bill” and checks that the **mean()** is greater than 20. If that lambda function returns True, then the **mean()** of the tip is calculated. If we compare the total mean of the tips, we can see that there is a difference between the two values, meaning that the filtering was performed correctly.

—--------  
restaurant\_grouped = restaurant.groupby("day")  
filter\_trans = lambda x: x["total\_bill"].mean() > 20  
restaurant\_filtered = restaurant\_grouped.filter(filter\_trans)  
print(restaurant\_filtered["tip"].mean()) # 3.11522 sec  
print(restaurant["tip"].mean()) # 2.9982 sec  
—--------

If we attempt to perform this operation without using **groupby()**, we end up with this inefficient code. At first, we use a list comprehension to extract the entries of the DataFrame that refer to days that have a mean meal greater than $20 and then use a for loop to append them into a list and calculate the mean. It might seem very intuitive, but as we see, it’s also very inefficient.

—--------  
t = [restaurant.loc[restaurant["day"] == i]["tip"] for i in restaurant["day"].unique() if restaurant.loc[restaurant["day"] == i]["total\_bill"].mean() > 20 ]

restaurant\_filtered = t[0]  
for j in t[1:]:  
 restaurant\_filtered = restaurant\_filtered.append(j, ignore\_index=True)  
—--------

Important functions in Pandas

* df.describe()
* df.info()
* df.assign()
* df.query()
* df.sort\_values()
* df.sample()
* df.isnull()
* df.fillna()
* df.dropna()
* df.drop()
* pd.pivot\_table()
* df.groupby()
* df.transpose()
* df.merge()
* df.rename()

### 5. Summary of Best Practices

* *Selecting rows and columns is faster using the .iloc[] function. So it is better to use unless it is easier or more convenient to use .loc[] and the speed is not a priority or you are just doing it once.*
* *Using the built-in replace() function is much faster than just using conventional methods.*
* *Replacing multiple values using python dictionaries is faster than using lists.*
* *Using .iterrows() does not improve the speed of iterating through the DataFrame but it is more efficient.*
* *The .apply() function performs faster when we want to iterate through all the rows of a pandas DataFrame, but is slower when we perform the same operation through a column.*
* *Vectorizing over the pandas series achieves the overwhelming majority of optimization needs for everyday calculations. However, if speed is of the highest priority, we can call in reinforcements in the form of the NumPy Python library.*
* *Using .groupby() to group it according to a certain feature and then using other functions to apply it to the data is much faster than using the conventional coding method.*

## Top 10 Mistakes to Steer Clear of in Your Code

* *Having Column Names with Spaces*
* *Not Using Query Method for Filtering*
* *Not using @ Symbol when Writing Complex Queries*
* *Iterating over Dataframe instead of using Vectorization*
* *Treating Slices of Dataframe as New Dataframe*
* *Not Using Chain Commands for Multiple Transformations*
* *Not Setting Column dtypes Correctly*
* *Not Using Pandas Plotting Builtin Function*
* *Aggregation manually instead of using .groupby()*
* *Saving Large Datasets as CSV File*

### Pandas vs SQL

When you have the option of a relational database or a data warehouse, you should choose to get your transformations done through SQL rather than Pandas. <To be expanded by Hassan: Explain the reason and ideally an example. Also make a short reference to the ELT paradigm in general>