CMEDS

Civil & Mineral Engineering
UNIVERSITY OF TORONTO

CME538 Introduction to Data Science

Week 2 | Lecture 3 (2.3)

Pandas III.

Pandas III

- Lambda Functions
- Iterating
- Merging

Pandas III

- **Lambda Functions**
- Iterating
- Merging

Lambda Functions

- As we learned in **Week 1 - Lecture 1.3**, you can write your very own Python functions using the def keyword.
- However, for simpler function definitions (raise_to_power) they can be converted to a lambda function.
- A lambda function is a small anonymous function that can take any number of arguments, but can only have one expression.
- The benefits of using lambda functions are:
  - You will write fewer lines of code.
  - You can create functions on the fly without assigning them a name.

```python
def raise_to_power(number, power):
    return number ** power

# Raise the number 2 to the power of 5
raise_to_power(number=2, power=5)
```

```
32
```

```python
raise_to_power = lambda number, power: number ** power

raise_to_power(number=2, power=5)
```

```
32
```

The structure of a `lambda` function is:

```python
lambda arguments : expression
```

For example,

```python
lambda argument1, argument2, (argument1 + argu
ment2) / 2
```

Lambda Functions

- We use lambda functions all the time with Pandas.
- We'll use it with the **.apply()** method in the next section.

```
def my_func(df):
    return df[%].max() < 45
```

OR

```
lambda df: df[%].max() < 45
```

Lambda Function!

```
elections.groupby('Year').filter(lambda df: df['%'].max() < 45)
```

	Year	Candidate	Party	Popular vote	Result	%
23	1860	Abraham Lincoln	Republican	1855993	win	39.699408
24	1860	John Bell	Constitutional Union	590901	loss	12.639283
25	1860	John C. Breckinridge	Southern Democratic	848019	loss	18.138998
26	1860	Stephen A. Douglas	Northern Democratic	1380202	loss	29.522311
66	1912	Eugene V. Debs	Socialist	901551	loss	6.004354
67	1912	Eugene W. Chafin	Prohibition	208156	loss	1.386325
68	1912	Theodore Roosevelt	Progressive	4122721	loss	27.457433
69	1912	William Taft	Republican	3486242	loss	23.218466
70	1912	Woodrow Wilson	Democratic	6296284	win	41.933422
115	1968	George Wallace	American Independent	9901118	loss	13.571218
116	1968	Hubert Humphrey	Democratic	31271839	loss	42.863537
117	1968	Richard Nixon	Republican	31783783	win	43.565246
139	1992	Andre Marrou	Libertarian	290087	loss	0.278516
140	1992	Bill Clinton	Democratic	44909806	win	43.118485
141	1992	Bo Gritz	Populist	106152	loss	0.101918
142	1992	George H. W. Bush	Republican	39104550	loss	37.544784
143	1992	Ross Perot	Independent	19743821	loss	18.956298

Pandas III

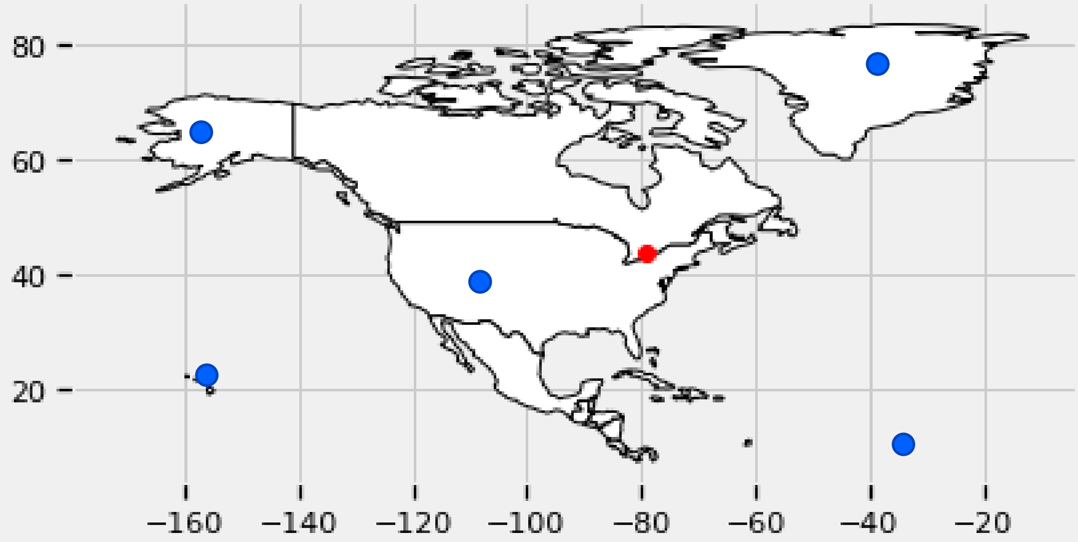
- Lambda Functions
- **Iterating**
- Merging

Iterating

- There are multiple ways to literate through DataFrames and when those DataFrame become large and the desired computation become complex, these different methods can have major impacts on compute times.
- In some cases, it could mean the different between seconds, tens of minutes and even hours.

Iterating

- Do illustrate this, let's start with a simple problem.
- Let's say we want to calculate the straight-line distance between 100,000 random geo-positions (latitude, longitude) and the city of Toronto.
  - Toronto
  - (lat: 43.651070 lon: -79.347015)



Iterating

- We'll use the Haversine (or Great Circle) distance formula, which takes the latitude and longitude of two points, adjusts for Earth's curvature, and calculates the straight-line distance between them.

```python
def haversine(lat1, lon1, lat2, lon2):
    """Defines a basic Haversine distance formula."""
    MILES = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    total_miles = MILES * c
    return total_miles
```

Iterating

- Let's create 100,000 random geo-positions.

```
num_locations = 100000
locations = pd.DataFrame({'lat': np.random.uniform(-90, 90, num_locations),
                          'lon': np.random.uniform(-180, 180, num_locations),
                          'distance': np.zeros(num_locations)})
locations.head()
```

	lat	lon	distance
0	-83.111605	-87.089523	0.0
1	-22.107472	-81.752065	0.0
2	-56.845158	-83.117859	0.0
3	-53.390859	121.957019	0.0
4	64.326892	11.298909	0.0

Iterating

- Our task is to loop through these random geo-positions and calculate the distance between them and Toronto.

Distance

```
haversine(lat1, lon1, lat2, lon2)
```

Toronto
lat: 43.651070
lon: -79.347015

Point
lat: -83.111605
lon: -87.089523

	lat	lon	distance
0	-83.111605	-87.089523	0.0
1	-22.107472	-81.752065	0.0
2	-56.845158	-83.117859	0.0
3	-53.390859	121.957019	0.0
4	64.326892	11.298909	0.0

Iterating

- Method 1: Simple **for** Loop over **range()**
- Just about every Pandas beginner I've ever worked with (Including myself) has, at some point, attempted to apply a custom function by looping over DataFrame rows one at a time.
- The advantage of this approach is that it is consistent with the way one would interact with other iterable Python objects, however, crude looping in Pandas does not take advantage of any built-in optimizations, making it extremely inefficient by comparison.

```python
def simple_for_loop_method(locations):

    distance_list = []

    # Loop through rows in locations DataFrame
    for row_index in range(locations.shape[0]):

        # Get lat and lon and row_index
        lat = locations.loc[row_index, 'lat']
        lon = locations.loc[row_index, 'lon']

        # Compute Haversine distance
        distance = haversine(lat1=toronto_lat,
                             lon1=toronto_lon,
                             lat2=lat,
                             lon2=lon)

        # Collect distance
        distance_list.append(distance)

    # Add distance values
    locations['distance'] = distance_list

    return locations
```

```python
%timeit simple_for_loop_method(locations)
```

```
3.99 s ± 143 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Iterating

- Method 1: Simple **for** Loop over **range()**
- Using a simple for loop and the range() function, it took **3.99 seconds** to iterate through 100,000 rows.

```python
def simple_for_loop_method(locations):

    distance_list = []

    # Loop through rows in locations DataFrame
    for row_index in range(locations.shape[0]):

        # Get lat and lon and row_index
        lat = locations.loc[row_index, 'lat']
        lon = locations.loc[row_index, 'lon']

        # Compute Haversine distance
        distance = haversine(lat1=toronto_lat,
                             lon1=toronto_lon,
                             lat2=lat,
                             lon2=lon)

        # Collect distance
        distance_list.append(distance)

    # Add distance values
    locations['distance'] = distance_list

    return locations
```

```python
%timeit simple_for_loop_method(locations)
```

```
3.99 s ± 143 ms per loop (mean ± std. dev.
of 7 runs, 1 loop each)
```

Iterating

- Method 2: Simple **for** loop using **.iterrows()**
- **.iterrows()** is a generator that iterates over the rows of the DataFrame and returns the index of each row, in addition to an object containing the row itself.
- **.iterrows()** is optimized to work with Pandas DataFrames, however, it's often the least efficient way to run most standard functions.

```
def iterrows_method(locations):

    distance_list = []
    # Loop through rows in locations DataFrame
    for index, row in locations.iterrows():

        # Get lat and lon and row_index
        lat = row['lat']
        lon = row['lon']

        # Compute Haversine distance
        distance = haversine(lat1=toronto_lat,
                             lon1=toronto_lon,
                             lat2=lat,
                             lon2=lon)

        # Collect distance
        distance_list.append(distance)

    # Add distance values
    locations['distance'] = distance_list

    return locations
```

```
%timeit iterrows_method(locations)
```

```
7.17 s ± 341 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Iterating

- Method 2: Simple **for** loop using **.iterrows()**
- Using the Pandas **.iterrows()** function, it took **7.17 seconds** to iterate through 100,000 rows, which is over twice as long as the simpler method.

```python
def iterrows_method(locations):

    distance_list = []
    # Loop through rows in locations DataFrame
    for index, row in locations.iterrows():

        # Get lat and lon and row_index
        lat = row['lat']
        lon = row['lon']

        # Compute Haversine distance
        distance = haversine(lat1=toronto_lat,
                             lon1=toronto_lon,
                             lat2=lat,
                             lon2=lon)

        # Collect distance
        distance_list.append(distance)

    # Add distance values
    locations['distance'] = distance_list

    return locations
```

```python
%timeit iterrows_method(locations)
```

```
7.17 s ± 341 ms per loop (mean ± std. dev.
of 7 runs, 1 loop each)
```

Iterating

- Method 3: Simple **for** loop using **.to_dict()**
- The Pandas **.to_dict()** method converts a DataFrame to a dictionary.
- We specify orient='row', which returns a list of dictionaries where each dictionary corresponds to a row.

```
def to_dict_for_loop_method(locations):

    distance_list = []
    # Loop through rows in locations DataFrame
    for row in locations.to_dict(orient='row'):

        # Get lat and lon and row_index
        lat = row['lat']
        lon = row['lon']

        # Compute Haversine distance
        distance = haversine(lat1=toronto_lat,
                             lon1=toronto_lon,
                             lat2=lat,
                             lon2=lon)

        # Collect distance
        distance_list.append(distance)

    # Add distance values
    locations['distance'] = distance_list
```

```
%timeit to_dict_for_loop_method(locations)
```

```
1.97 s ± 18.7 ms per loop (mean ± std. de
v. of 7 runs, 1 loop each)
```

Iterating

- Method 3: Simple **for** loop using **.to_dict()**
- Using the Pandas **.to_dict()** function, it took **1.97 seconds** to iterate through 100,000 rows, which is almost five times faster than **.iterrows()**.

```
def to_dict_for_loop_method(locations):

    distance_list = []
    # Loop through rows in locations DataFrame
    for row in locations.to_dict(orient='row'):

        # Get lat and lon and row_index
        lat = row['lat']
        lon = row['lon']

        # Compute Haversine distance
        distance = haversine(lat1=toronto_lat,
                             lon1=toronto_lon,
                             lat2=lat,
                             lon2=lon)

        # Collect distance
        distance_list.append(distance)

    # Add distance values
    locations['distance'] = distance_list
```

```
%timeit to_dict_for_loop_method(locations)
```

```
1.97 s ± 18.7 ms per loop (mean ± std. de
v. of 7 runs, 1 loop each)
```

Iterating

- Method 4: Using Pandas **.apply()**
- The **.apply()** method applies a function along a specific axis (meaning, either rows or columns) of a DataFrame.
- Using the Pandas **.apply()** function takes roughly **the same amount of time** as the simple loop but the code is more compact.

```python
def apply_method(locations):

    locations['distance'] = locations.apply(
        lambda row: haversine(lat1=toronto_lat,
                              lon1=toronto_lon,
                              lat2=row['lat'],
                              lon2=row['lon']),
        axis=1
    )

    return locations
```

```python
%timeit apply_method(locations)
```

```
3.21 s ± 505 ms per loop (mean ± std. dev.
of 7 runs, 1 loop each)
```

Iterating

- Method 5: Vectorization over Pandas **Series**
- Vectorization is the process of executing operations on entire arrays rather than by iterating over individual units.
- By vectorizing over Pandas Series, we see a **x165 improvement** over the **.to_dict()** method.

```
def vectorized_series_method(locations):

    locations['distance'] = haversine(lat1=toronto_lat,
                                      lon1=toronto_lon,
                                      lat2=locations['lat'],
                                      lon2=locations['lon'])

    return locations
```

```
%timeit vectorized_series_method(locations)
```

```
10.1 ms ± 109 µs per loop (mean ± std. dev. of 7 runs, 100 lo
ops each)
```

Iterating

- Method 6: Vectorization over NumPy **Array**
- Recall that the fundamental units of Pandas, DataFrames and Series, are both based on NumPy arrays.
- By vectorizing over NumPy Arrays, we se a **x252 improvement** over the **.to_dict()** method.

```
def vectorized_array_method(locations):

    locations['distance'] = haversine(lat1=toronto_lat,
                                      lon1=toronto_lon,
                                      lat2=locations['lat'].to_numpy(),
                                      lon2=locations['lon'].to_numpy())

    return locations
```

```
%timeit vectorized_array_method(locations)
```

```
8.56 ms ± 39.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Iterating

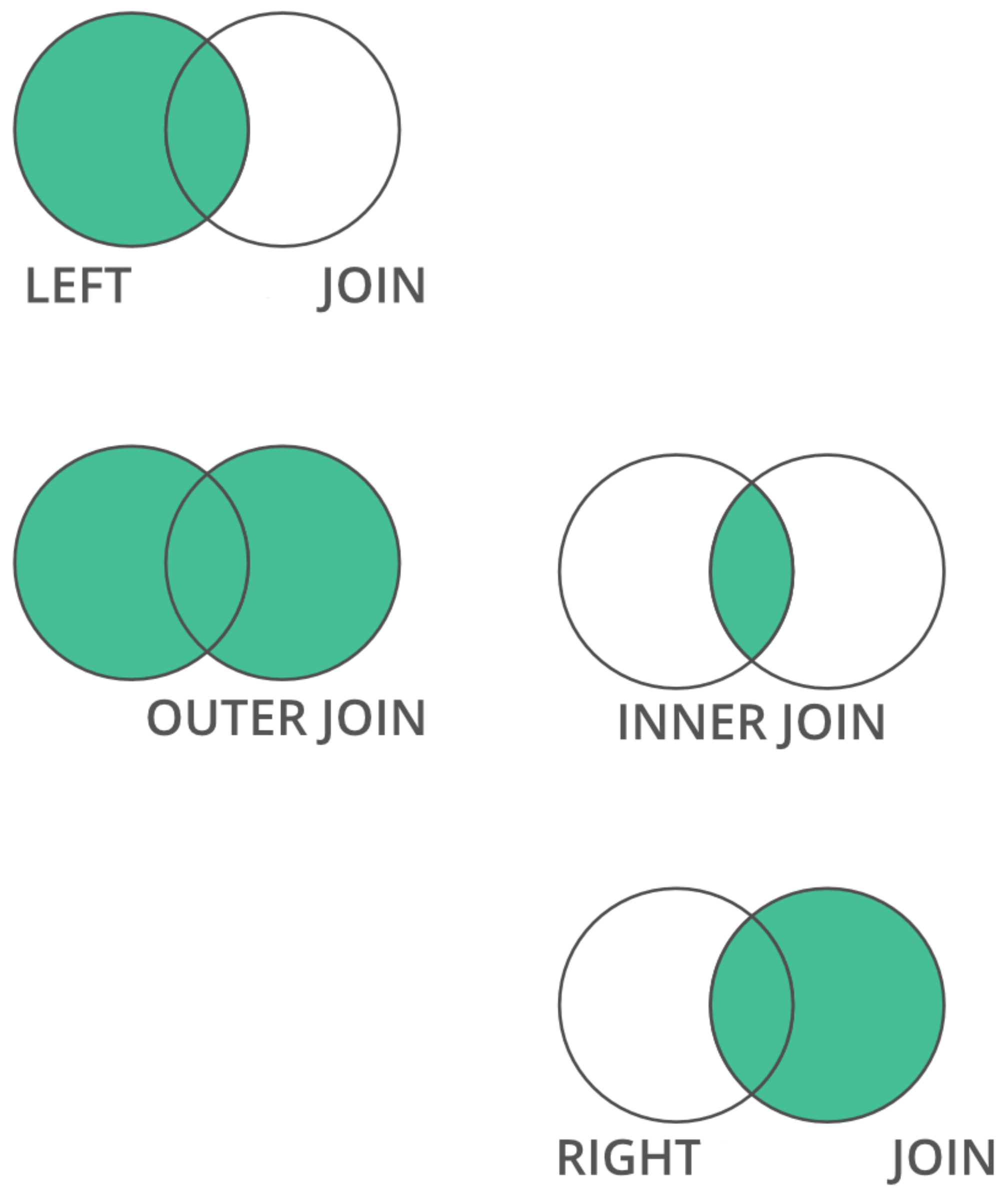
- From this quick exercise, you should know that there are **many different ways** to iterate through a DataFrame and that the different methods have **very** different performance considerations.
- Every application has different performance requirements.
- When writing code, you want it to be:
  - **Performant**
  - Modular
  - Easy to understand

Pandas III

- Lambda Functions
- Iterating
- **Merging**

Merging

- When conducting exploratory data analysis (EDA), its common that the data we want to use comes in multiple files/sources and will need to be combined.
- In assignment 4, you'll have to combine bike share ridership data with City of Toronto weather data.



Merging

- Let's start with an example.
- In the Lecture 6 folder, there are six .csv files from Uber showing monthly ridership numbers from April 2014 to September 2014.

```
os.listdir()
```

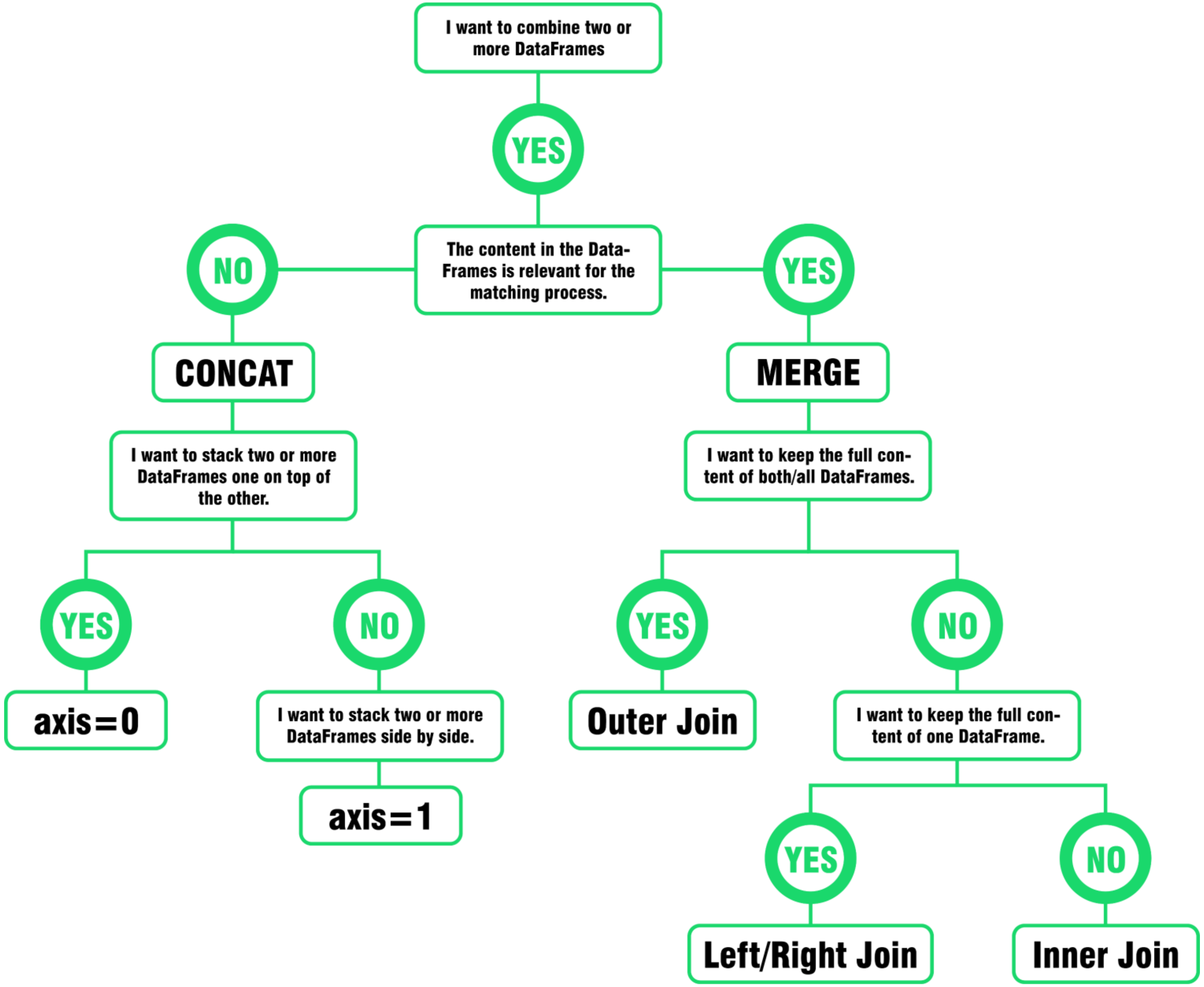
```
['.ipynb_checkpoints',
 'images',
 'lecture6.ipynb',
 'uber-raw-data-apr14.csv',
 'uber-raw-data-aug14.csv',
 'uber-raw-data-jul14.csv',
 'uber-raw-data-jun14.csv',
 'uber-raw-data-may14.csv',
 'uber-raw-data-sep14.csv']
```

Given what we've learned aleady in Lectures 4 and 5, we know how to import these `.csv` files to **Pandas** DataFrames. Lets try that.

```
april_data = pd.read_csv('uber-raw-data-apr14.csv')
may_data = pd.read_csv('uber-raw-data-may14.csv')
june_data = pd.read_csv('uber-raw-data-jun14.csv')
july_data = pd.read_csv('uber-raw-data-jul14.csv')
aug_data = pd.read_csv('uber-raw-data-aug14.csv')
sept_data = pd.read_csv('uber-raw-data-sep14.csv')
```

Merging

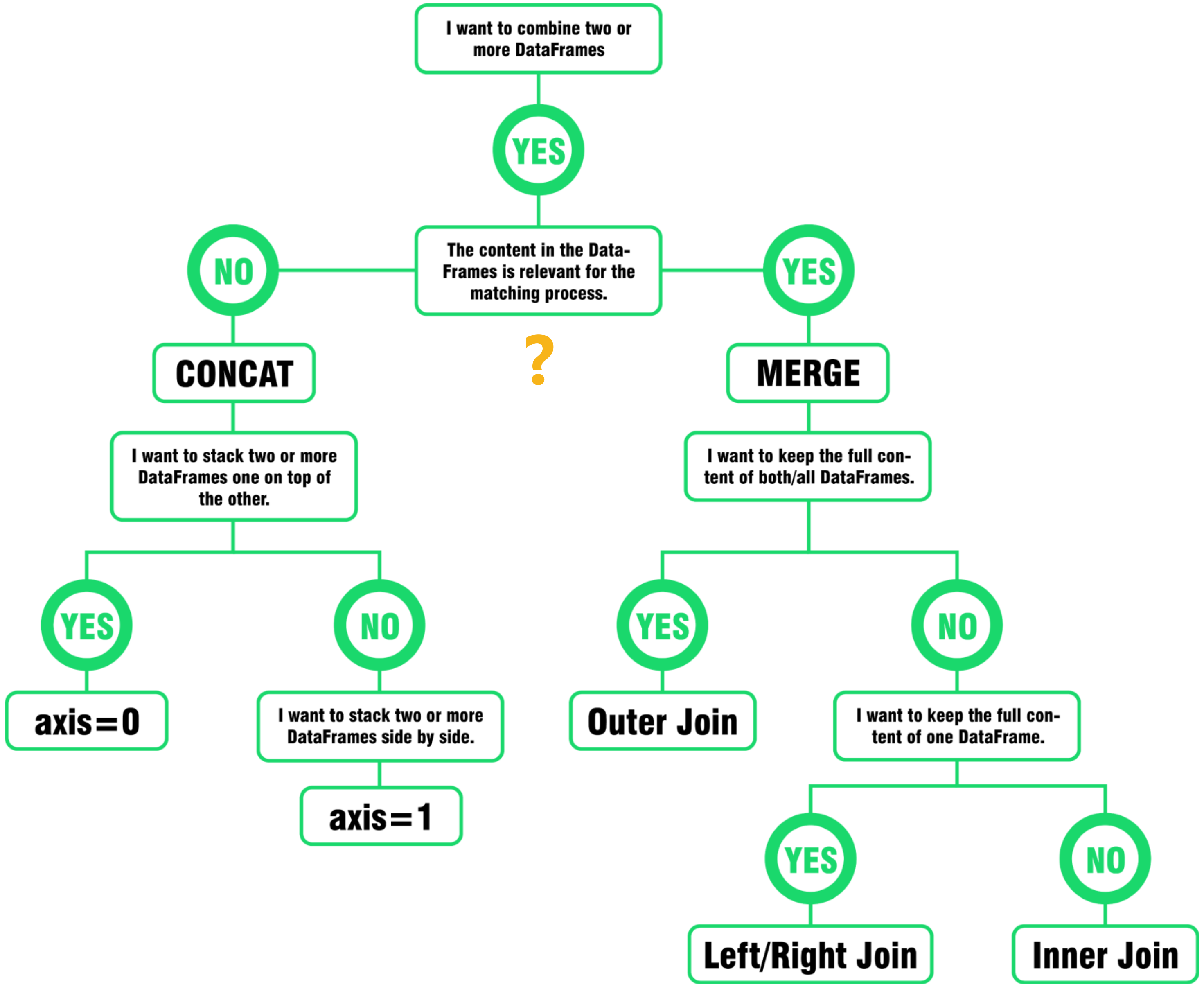
- In this DataFrame, each row is an Uber trip.
- Suppose we're asked to plot the number of trips per hour from April 2014 to September 2014.
- To tackle this problem, it would be much easier if all the data was in one DataFrame.
- There are two Pandas methods for combining DataFrames: **.concat()** and **.merge()**.



Civil & Mineral Engineering
UNIVERSITY OF TORONTO

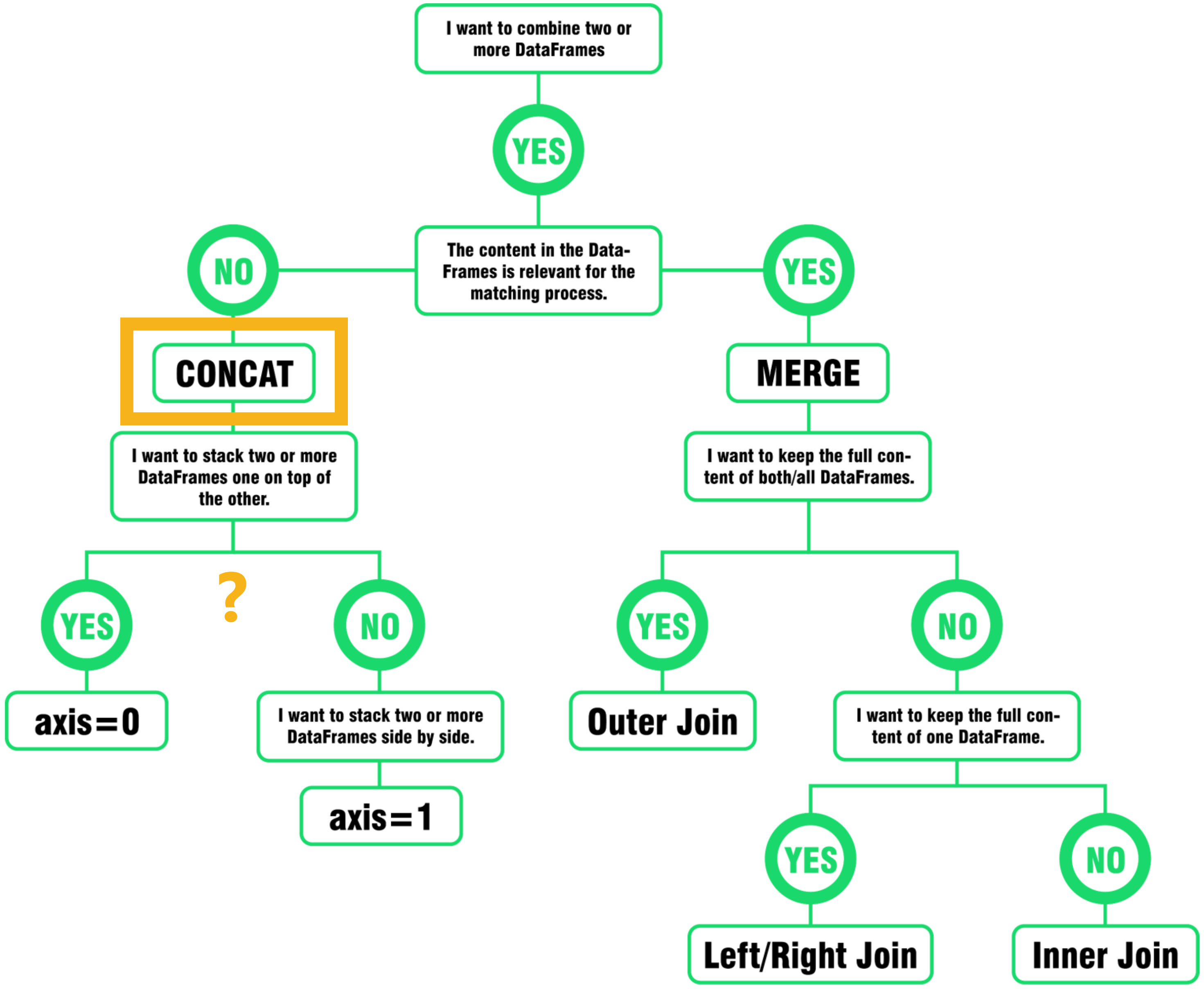
Merging

- Is the content in the DataFrame relevant for the matching process?



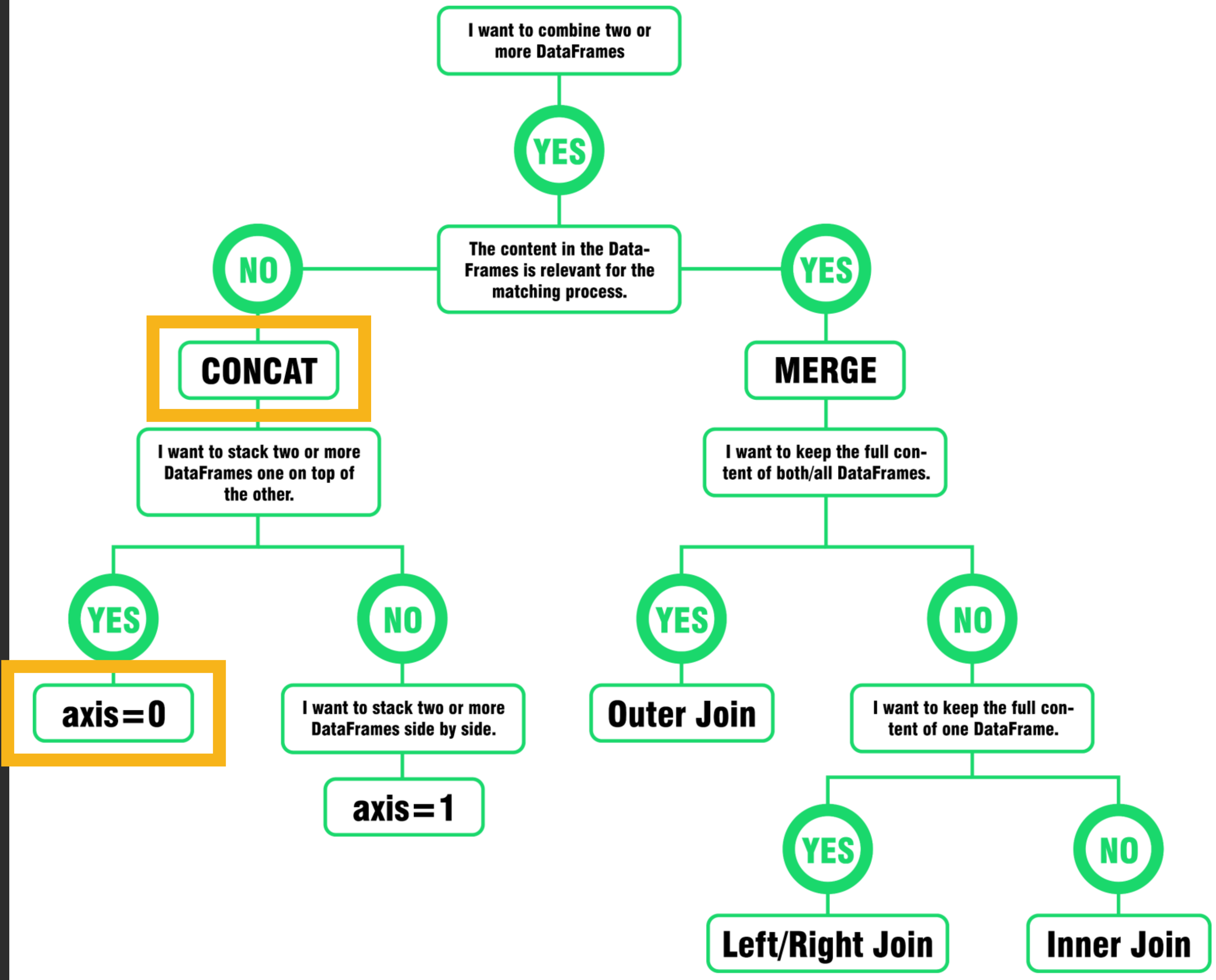
Merging

- Is the content in the DataFrame relevant for the matching process?
- **NO**
- Do we want to stack two or more DataFrames one on top of the other?



Merging

- Is the content in the DataFrame relevant for the matching process?
- **NO**
- Do we want to stack two or more DataFrames one on top of the other?
- **YES**
- **.concat(axis=0)**



Merging

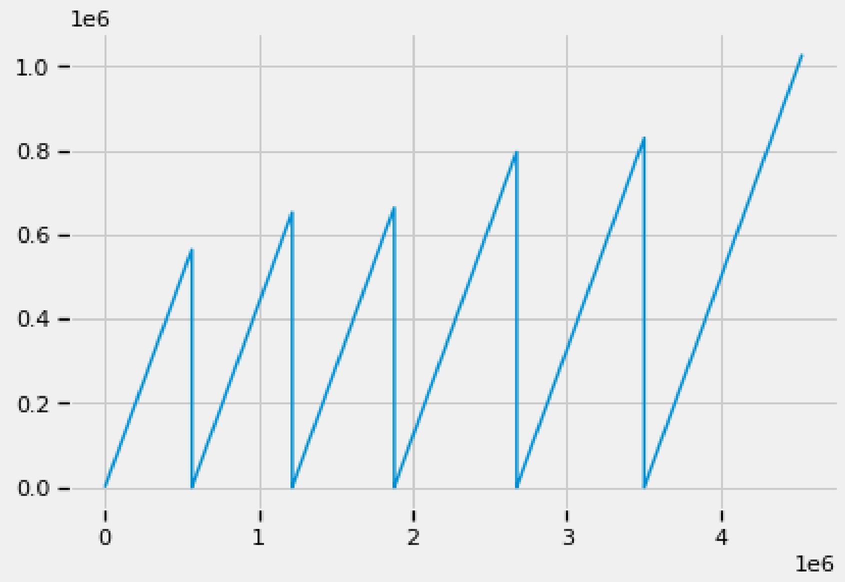
- **Concatenate**
- We use the **.concat()** function to append either columns or rows from one DataFrame to another. This happens to be the functionality we need to handle the Uber data we import above.
- **pd.concat()** has many features, which you're encouraged to explore, but the basic function is demonstrated below.
- If we want to stack multiple DataFrames side-by-side, then we set axis=1.

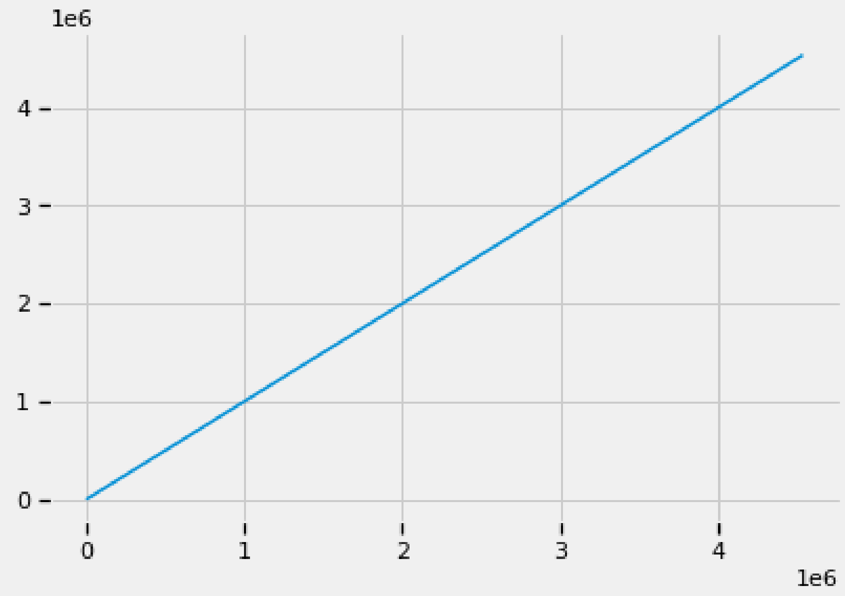
```
# Stack the DataFrames on top of each other
uber_data = pd.concat([april_data,
                       may_data,
                       june_data,
                       july_data,
                       aug_data,
                       sept_data], axis=0)

# View combined DataFrame
uber_data.head()
```

	Date/Time	Lat	Lon	Base
0	4/1/2014 0:11:00	40.7690	-73.9549	B02512
1	4/1/2014 0:17:00	40.7267	-74.0345	B02512
2	4/1/2014 0:21:00	40.7316	-73.9873	B02512
3	4/1/2014 0:28:00	40.7588	-73.9776	B02512
4	4/1/2014 0:33:00	40.7594	-73.9722	B02512

Merging

- **Concatenate**
- If we plot the index, we can clearly see from the plot that when concatenating the DataFrames, the original indexes have been preserved, meaning that we have duplicates, which will be an issue moving forward.
- To adjust the row index automatically, we can set the argument **ignore_index=True** while calling the **.concat()** function.





Merging

- Let's try another example.
- We have two DataFrames.
  - The first and last name of test participants.
  - The test score for participants.
- The two DataFrames have a common column, **participant_id**.

Left

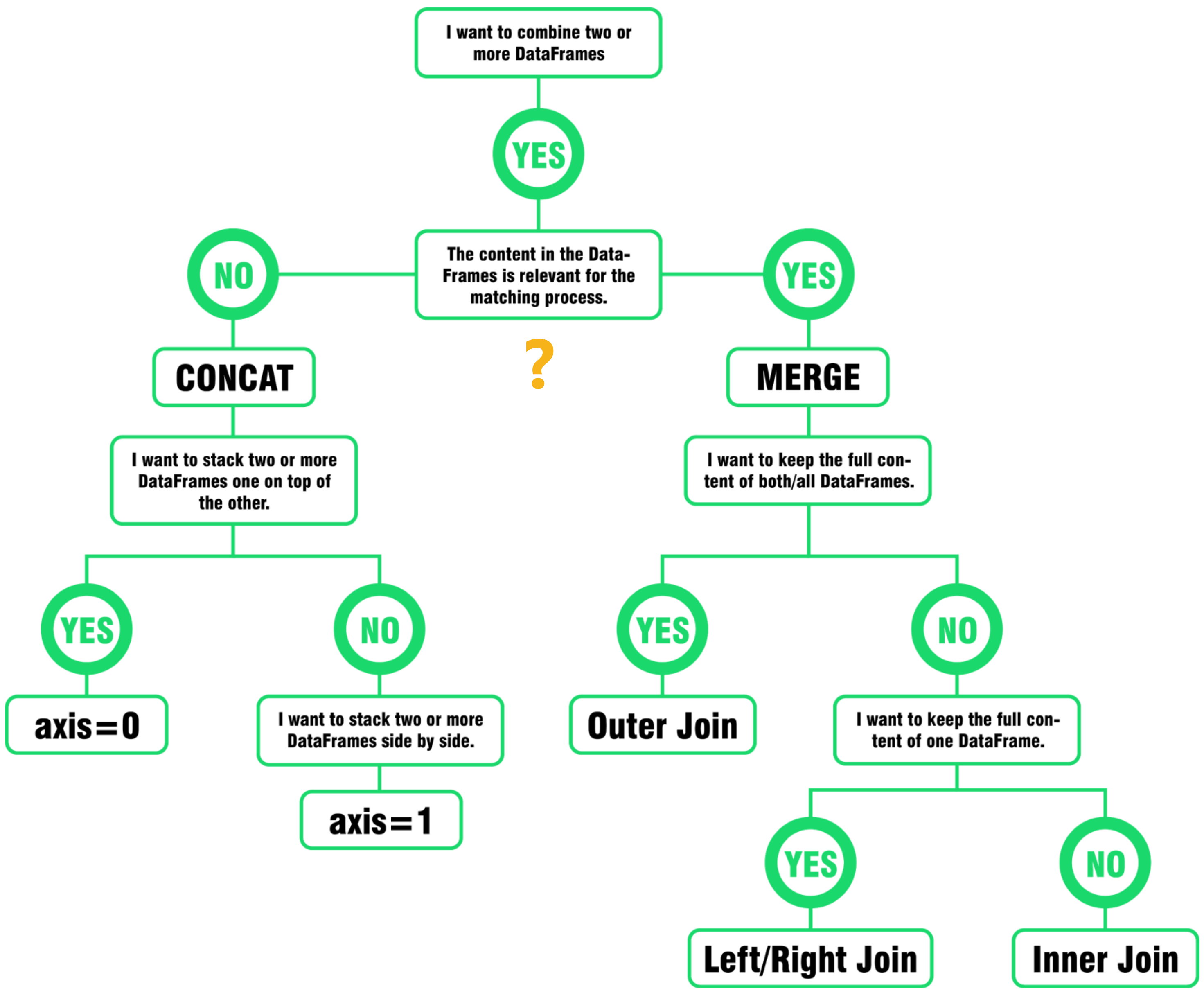
	participant_id	first_name	last_name
0	1	Shoshanna	Saxe
1	6	Marianne	Touchie
2	33	Karl	Peterson
3	42	Brent	Sleep
4	65	John	Harrison
5	8	Marcus	Aurelius
6	20	Bruce	Wayne
7	13	Judi	Dench
8	14	Denzel	Washington

Right

	participant_id	score
0	22	80
1	98	76
2	71	72
3	33	66
4	42	77
5	65	64
6	8	59
7	20	60
8	13	62
9	14	89
10	34	67
11	54	58

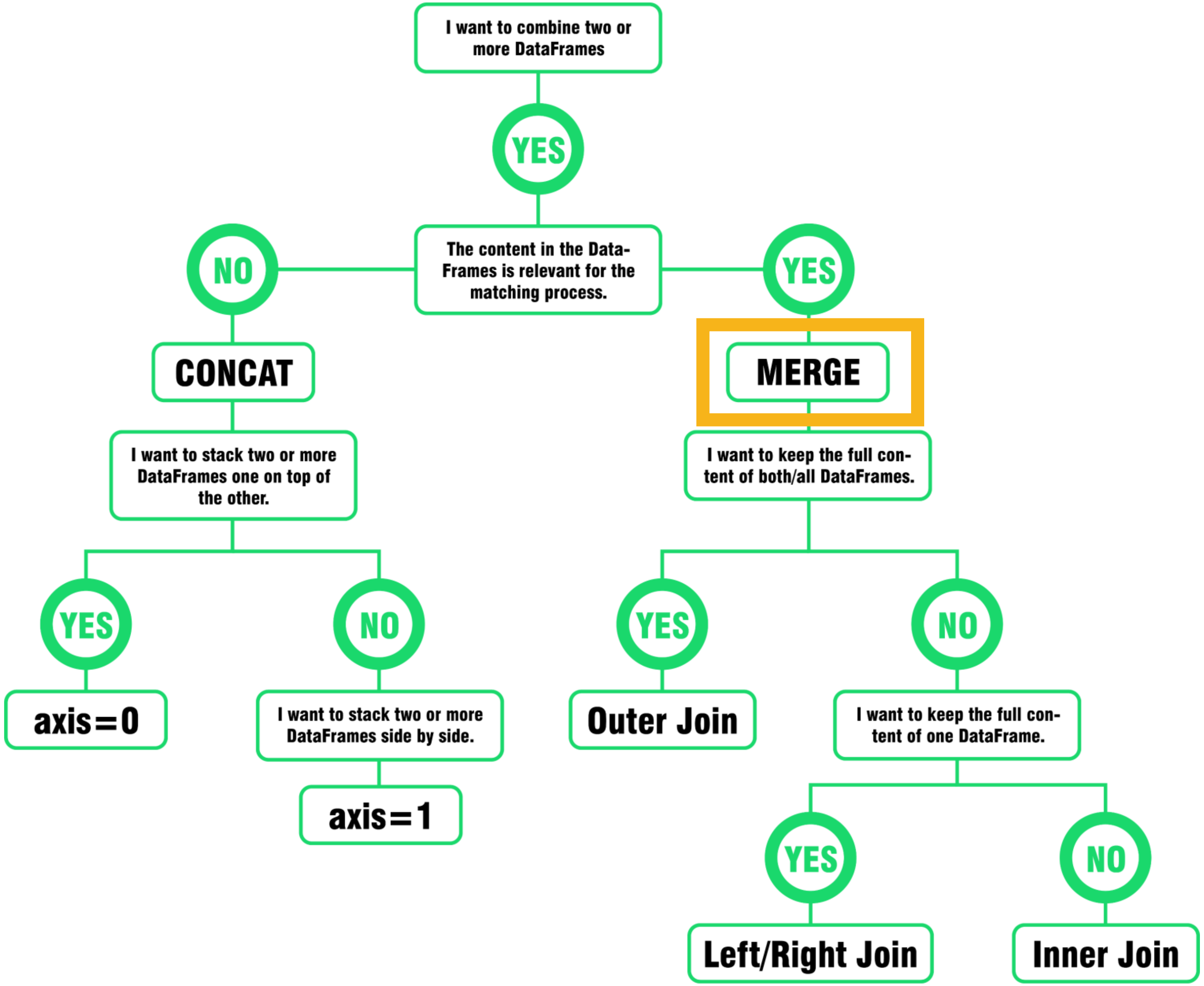
Merging

- Is the content in the DataFrame relevant for the matching process?



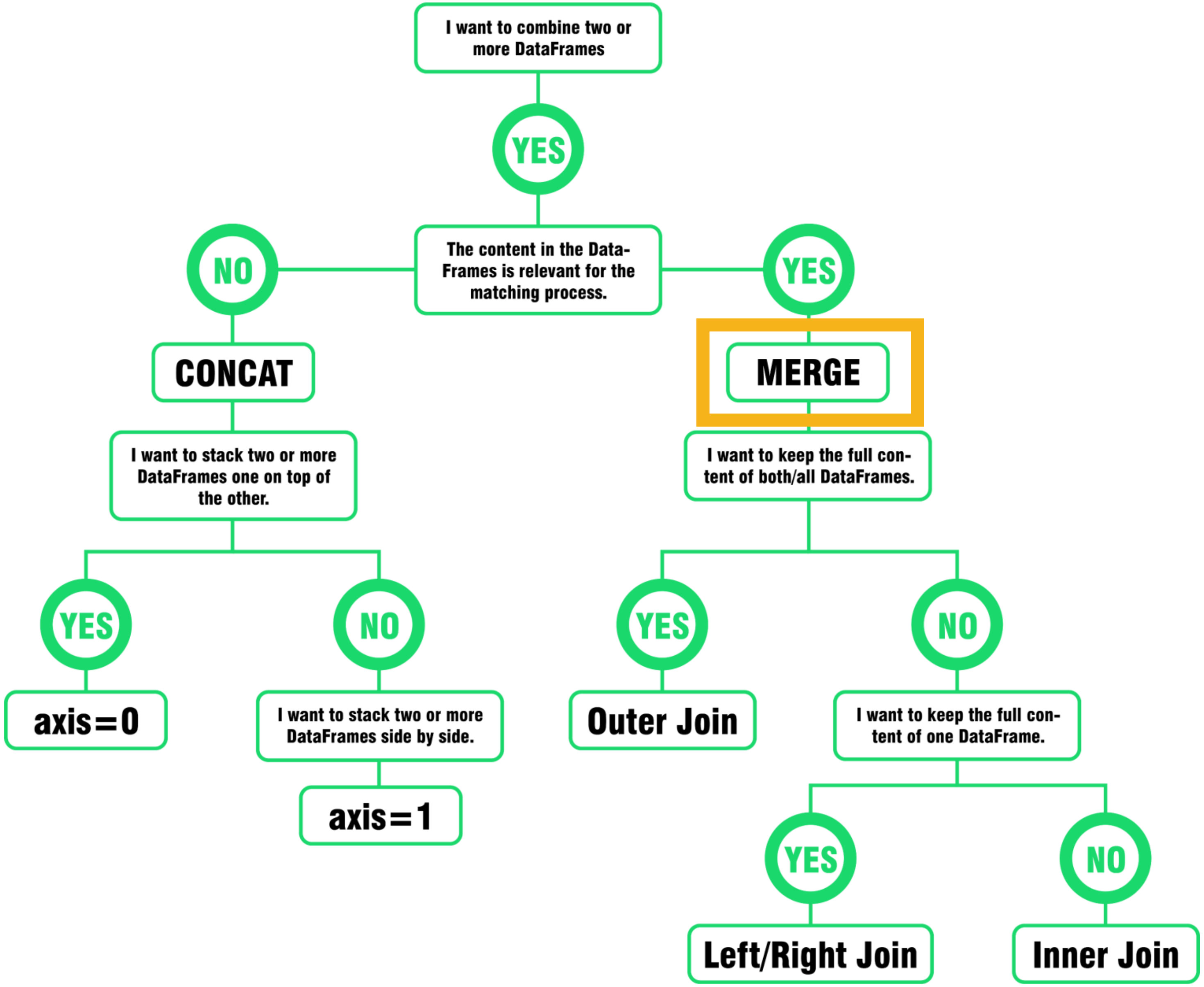
Merging

- Is the content in the DataFrame relevant for the matching process?
- **YES**



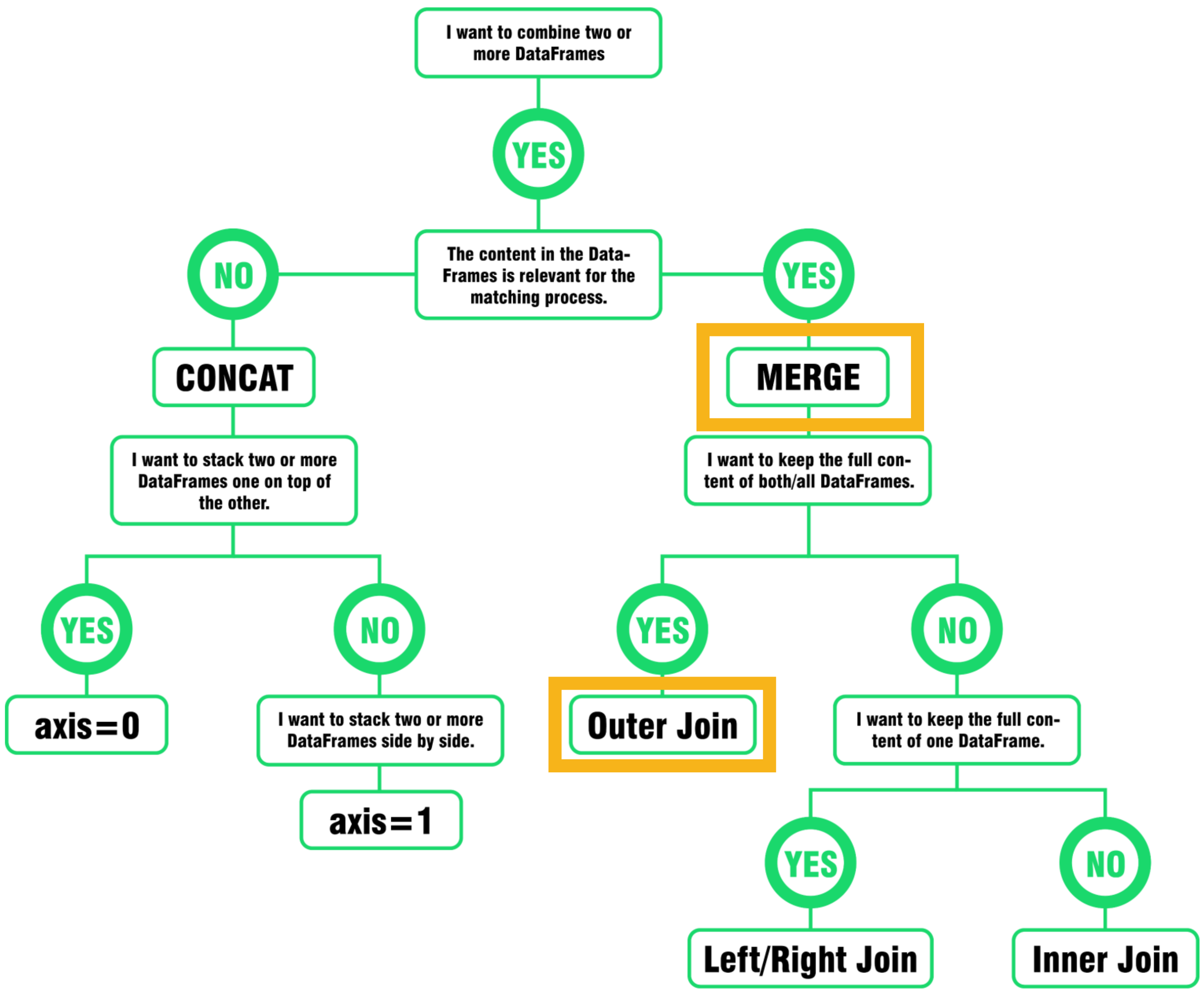
Merging

- **Merge**
- **.merge()** allows you to execute the following join operations: inner join, full outer join, left outer join, right outer join (see Figure above).
- Let's work through the four paths outlined in the figure to the right.
- We're using **.merge()** because the contents of the DataFrame are required for combining the DataFrames.
- The common column participant_id will be used to merge df1 and df2.



Merging

- **Merge** **Outer Join**
- I want to keep the full content of both DataFrames.



Merging

- **Merge Outer Join**
- I want to keep the full content of both DataFrames.

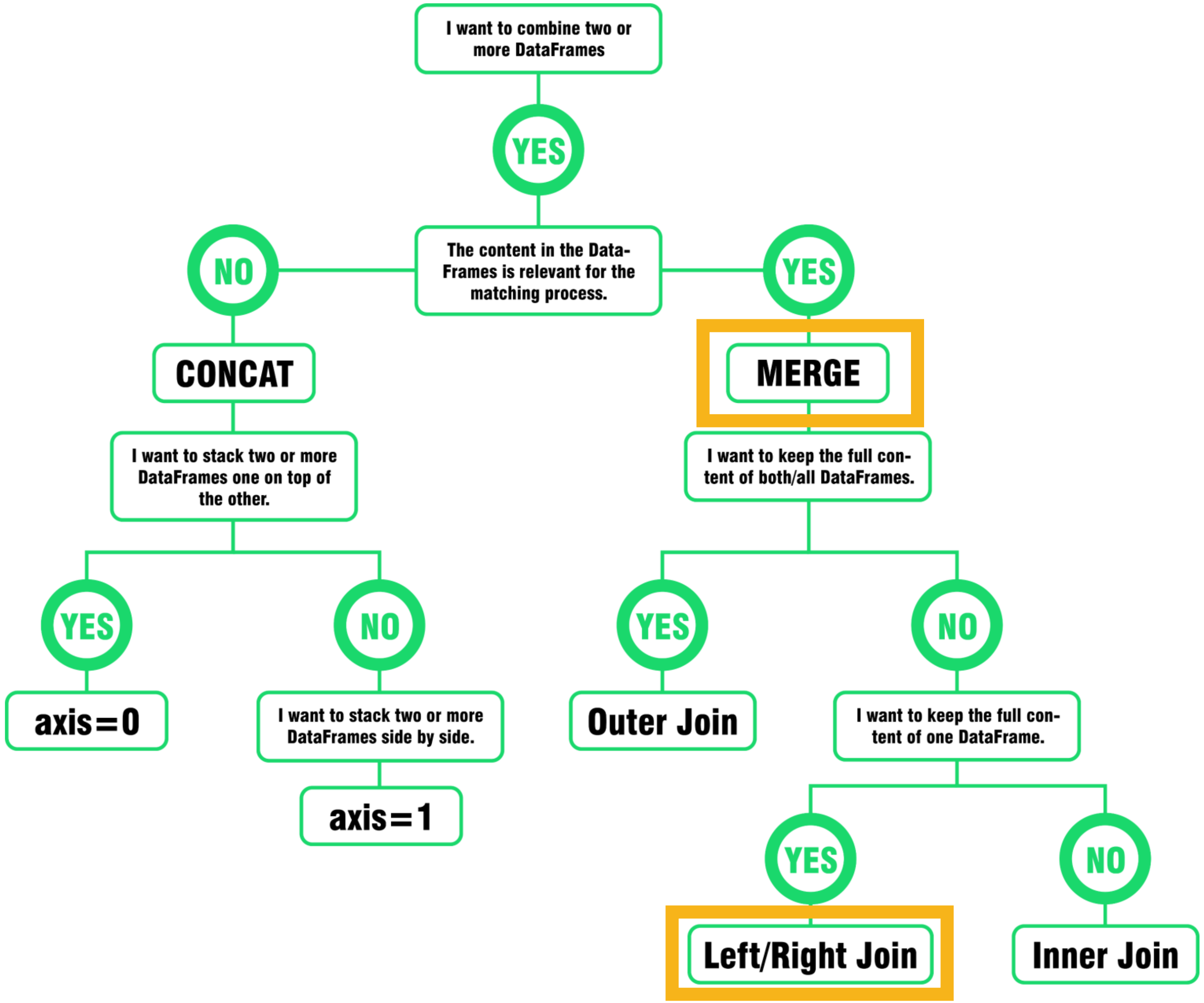
```python
df_outer_join = df1.merge(right=df2,
                          how='outer',
                          on='participant_id')
df_outer_join
```

	participant_id	first_name	last_name	score
0	1	Shoshanna	Saxe	NaN
1	6	Marianne	Touchie	NaN
2	33	Karl	Peterson	66.0
3	42	Brent	Sleep	77.0
4	65	John	Harrison	64.0
5	8	Marcus	Aurelius	59.0
6	20	Bruce	Wayne	60.0
7	13	Judi	Dench	62.0
8	14	Denzel	Washington	89.0
9	22	NaN	NaN	80.0
10	98	NaN	NaN	76.0
11	71	NaN	NaN	72.0
12	34	NaN	NaN	67.0
13	54	NaN	NaN	58.0

Values not in both DataFrames

Merging

- **Merge** **Left Outer Join**
- I want to keep the full content of the left DataFrame and merge any matching data from the right DataFrame.
- Practically, we want a DataFrame with all of the participants from **LEFT** and we want to merge their scores from **RIGHT**.



Merging

- **Merge Left Outer Join**
- I want to keep the full content of the left DataFrame and merge any matching data from the right DataFrame.

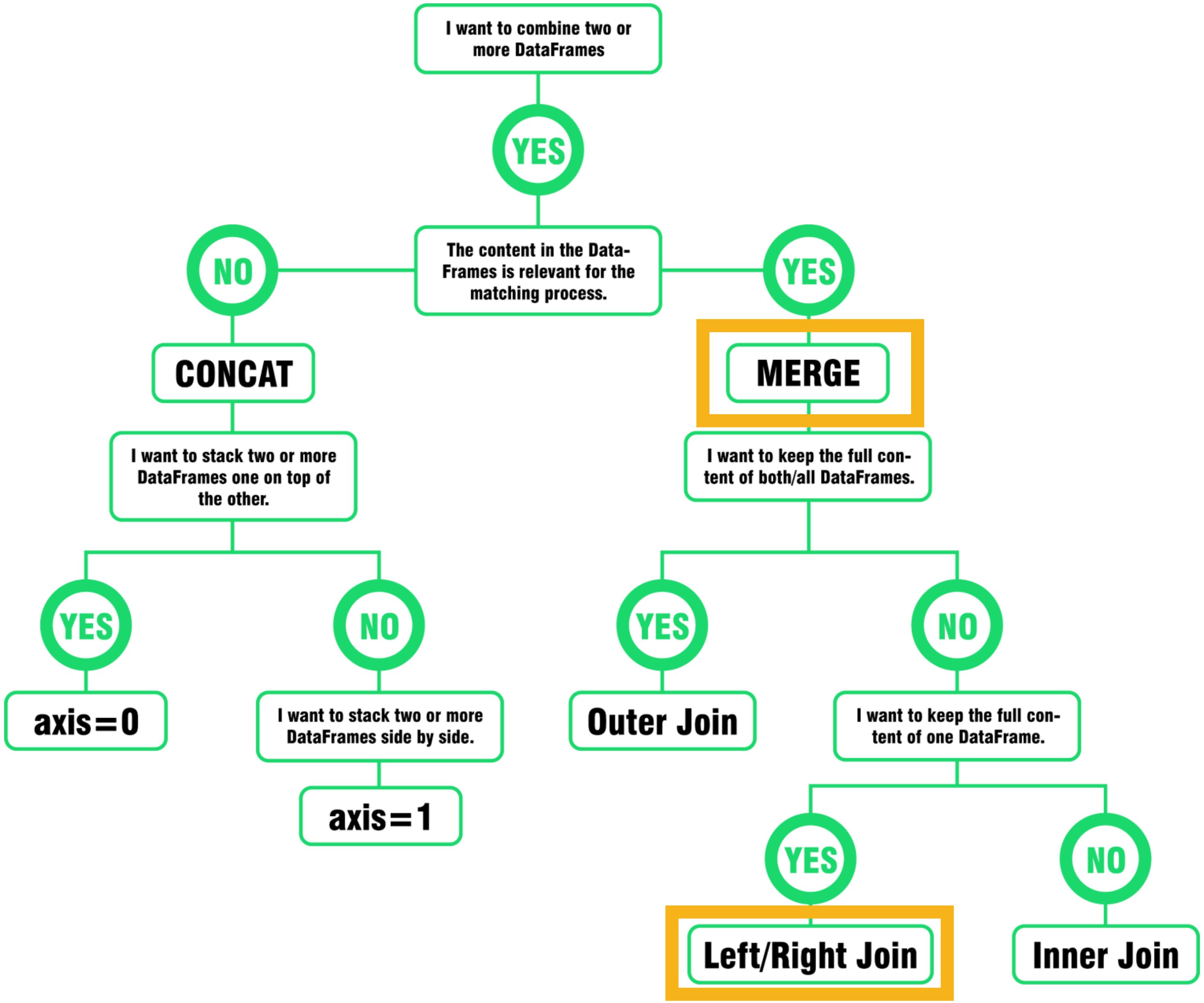
Participants not in RIGHT

```python
df_left_outer_join = df1.merge(right=df2,
                               how='left',
                               on='participant_id')
df_left_outer_join
```

	participant_id	first_name	last_name	score
0	1	Shoshanna	Saxe	NaN
1	6	Marianne	Touchie	NaN
2	33	Karl	Peterson	66.0
3	42	Brent	Sleep	77.0
4	65	John	Harrison	64.0
5	8	Marcus	Aurelius	59.0
6	20	Bruce	Wayne	60.0
7	13	Judi	Dench	62.0
8	14	Denzel	Washington	89.0

Merging

- **Merge** **Right Outer Join**
- I want to keep the full content of the right DataFrame and merge any matching data from the left DataFrame.



Merging

- **Merge Right Outer Join**
- I want to keep the full content of the right DataFrame and merge any matching data from the left DataFrame.

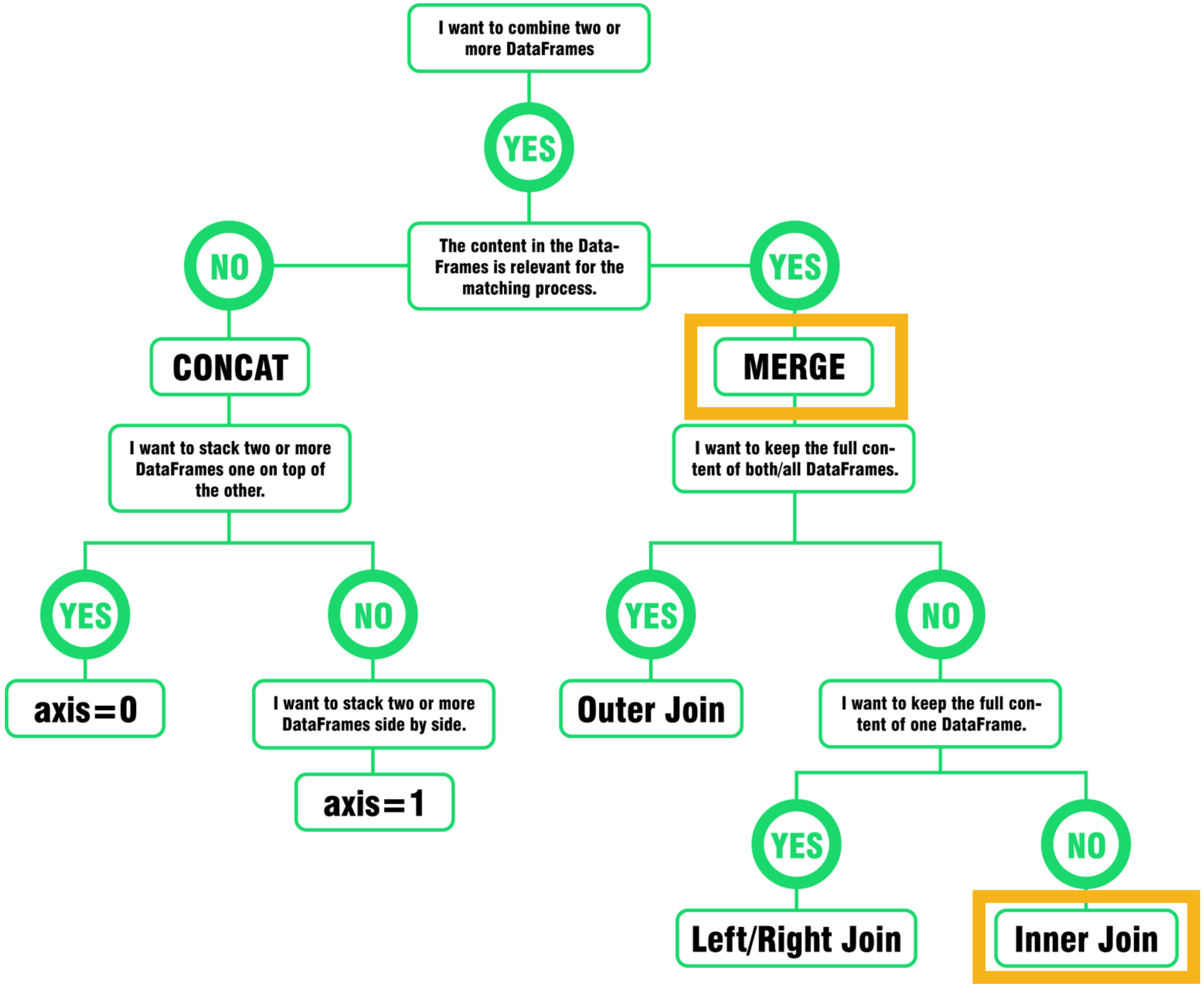
Participants not in LEFT

```python
df_right_outer_join = df1.merge(right=df2,
                                how='right',
                                on='participant_id')
df_right_outer_join
```

	participant_id	first_name	last_name	score
0	22	NaN	NaN	80
1	98	NaN	NaN	76
2	71	NaN	NaN	72
3	33	Karl	Peterson	66
4	42	Brent	Sleep	77
5	65	John	Harrison	64
6	8	Marcus	Aurelius	59
7	20	Bruce	Wayne	60
8	13	Judi	Dench	62
9	14	Denzel	Washington	89
10	34	NaN	NaN	67
11	54	NaN	NaN	58

Merging

- **Merge** **Inner Join**
- I want to keep the only contents of the right and left DataFrame only where overlap.



Merging

- **Merge Inner Join**
- I want to keep the only contents of the right and left DataFrame only where overlap.
- Creates a DataFrame with all of the scores from **RIGHT** and names from **LEFT** where they have participant_id in common.

```
df_inner_join = df1.merge(right=df2,
                          how='inner',
                          on='participant_id')
df_inner_join
```

	participant_id	first_name	last_name	score
0	33	Karl	Peterson	66
1	42	Brent	Sleep	77
2	65	John	Harrison	64
3	8	Marcus	Aurelius	59
4	20	Bruce	Wayne	60
5	13	Judi	Dench	62
6	14	Denzel	Washington	89

Practice!

- Launch the Lecture 2.3 notebook from Quercus and review the material from this lecture in more detail.
- Link

CMEDS

Civil & Mineral Engineering
UNIVERSITY OF TORONTO

CME538 Introduction to Data Science

Week 2 | Lecture 3 (2.3)

Pandas III.