Data Aggregation: Takeaways

Syntax

GROUPBY OBJECTS

- Create a GroupBy object:

```
df.groupby('col_to_groupby')
```

- Select one column from a GroupBy object:

```
df.groupby('col_to_groupby')['col_selected']
```

- Select multiple columns from a GroupBy object:

```
df.groupby('col_to_groupby')[['col_selected1', 'col_selected2']]
```

COMMON AGGREGATION METHODS

- `mean()` : calculates the mean of groups
- `sum()` : calculates the sum of group values
- `size()` : calculates the size of groups
- `count()` : calculates the count of values in groups
- `min()` : calculates the minimum of group values
- `max()` : calculates the maximum of group values

`GROUPBY.AGG()` METHOD

- Apply one function to a GroupBy object:

```
df.groupby('col_to_groupby').agg(function_name)
```

- Apply multiple functions to a GroupBy object:

```
df.groupby('col_to_groupby').agg([function_name1, function_name2, function_name3])
```

- Apply a custom function to a GroupBy object:

```
df.groupby('col_to_groupby').agg(custom_function)
```

AGGREGATION WITH THE `DATAFRAME.PIVOT_TABLE` METHOD

- Apply only one function:

```
df.pivot_table(values='Col_to_aggregate', index='Col_to_group_by', aggfunc=function_name)
```

- Apply multiple functions:

```
df.pivot_table('Col_to_aggregate', 'Col_to_group_by', aggfunc=[function_name1,
function_name2, function_name3])
```

- Aggregate multiple columns:

```
df.pivot_table(['Col_to_aggregate1', 'Col_to_aggregate2'], 'Col_to_group_by', aggfunc = function_name)
```

- Calculate the grand total for the aggregation column:

```
df.pivot_table('Col_to_aggregate', 'Col_to_group_by', aggfunc=function_name, margins=True)
```

Concepts

- **Aggregation** is applying a statistical operation to groups of data. It reduces dimensionality so that the DataFrame returned will contain just one value for each group. We can break down the aggregation process into three steps:
    - split the dataframe into groups
    - apply a function to each group
    - combine the results into one data structure
- The `groupby` operation optimizes the split-apply-combine process. We can break it down into two steps:
    - create a GroupBy object
    - call an aggregation function
- Creating the GroupBy object is an intermediate step that allows us to optimize our work. It contains information on how to group the DataFrame, but nothing actually gets computed until a function is called.
- We can also use the `DataFrame.pivot_table()` method to aggregate data; we can also use it to calculate the grand total for the aggregation column.

Combining Data Using Pandas: Takeaways

Syntax

CONCAT() FUNCTION

- Concatenate dataframes vertically (axis=0):

```
pd.concat([df1, df2])
```

- Concatenate dataframes horizontally (axis=1):

```
pd.concat([df1, df2], axis=1)
```

- Concatenate dataframes with an inner join:

```
pd.concat([df1, df2], join='inner')
```

MERGE() FUNCTION

- Join dataframes on index:

```
pd.merge(left=df1, right = df2, left_index=True, right_index=True)
```

- Customize the suffix of columns contained in both dataframes:

```
pd.merge(left=df1, right=df2, left_index=True, right_index=True,
suffixes=('left_df_suffix', 'right_df_suffix'))
```

- Change the join type to left, right, or outer:

```
pd.merge(left= df1, right=df2, how='join_type', left_index=True, right_index=True))
```

- Join dataframes on a specific column:

```
pd.merge(left=df1, right=df2, on='Column_Name')
```

Concepts

- A key or join key is a shared index or column that is used to combine dataframes together.
- There are four kinds of joins:
    - Inner: Returns the intersection of keys, or common values.
    - Outer: Returns the union of keys, or all values from each dataframe.
    - Left: Includes all of the rows from the left dataframe, along with any rows from the right dataframe with a common key. The result retains all columns from both of the original dataframes.
    - Right: Includes all of the rows from the right dataframe, along with any rows from the left dataframe with a common key. The result retains all columns from both of the original dataframes. This join type is rarely used.
- The `pd.concat()` function can combine multiple dataframes at once and is commonly used to "stack" dataframes, or combine them vertically (axis=0). The `pd.merge()` function uses keys to

perform database-style joins. It can only combine two dataframes at a time and can only merge dataframes horizontally (axis=1).

Resources

- Merge and Concatenate

Transforming Data with Pandas: Takeaways

Syntax

APPLYING FUNCTIONS ELEMENT-WISE

- Apply a function element-wise to a series:

```
df[col_name].apply(function_name)

df[col_name].map(function_name)
```

- Apply a function element-wise to a dataframe:

```
df.applymap(function_name)
```

APPLYING FUNCTIONS ALONG AN AXIS

- Apply a function along an axis, column-wise:

```
df.apply(function_name)
```

RESHAPING DATAFRAMES

- Reshape a dataframe:

```
pd.melt(df, id_vars=[col1, col2], value_vars=[col3, col4])
```

Concepts

- The `Series.apply()` and `Series.map()` methods can be used to apply a function element-wise to a *series*. The `DataFrame.applymap()` method can be used to apply a function element-wise to a *dataframe*.
- The `DataFrame.apply()` method has different capabilities than the `Series.apply()` method. Instead of applying functions element-wise, the `df.apply()` method applies functions along an axis, either column-wise or row-wise. When we create a function to use with `df.apply()`, we set it up to accept a Series, most commonly a column.
- Use the `apply()` method when a vectorized function does not exist because a vectorized function can perform an equivalent task faster than the `apply()` method. Sometimes, it may be necessary to reshape a dataframe to use a vectorized method.

Resources

- Tidy Data

Working with Strings In Pandas: Takeaways

Syntax

REGULAR EXPRESSIONS

- To match multiple characters, specify the characters between "[ ]":

```
pattern = r"[Nn]ational accounts"
```

  - This expression would match "national accounts" and "National accounts".

- To match a range of characters or numbers, use:

```
pattern = r"[0-9]"
```

  - This expression would match any number between 0 and 9.

- To create a capturing group, or indicate that only the character pattern matched should be extracted, specify the characters between "( )":

```
pattern = r"([1-2][0-9][0-9][0-9])"
```

  - This expression would match years.

- To repeat characters, use "{ }". To repeat the pattern "[0-9]" three times:

```
pattern = r"([1-2][0-9]{3})"
```

  - This expression would also match years.

- To name a capturing group, use:

```
pattern = r"(?P<Years>[1-2][0-9]{3})"
```

  - This expression would match years and name the capturing group "Years".

VECTORIZED STRING METHODS

- Find specific strings or substrings in a column:

```
df[col_name].str.contains(pattern)
```

- Extract specific strings or substrings in a column:

```
df[col_name].str.extract(pattern)
```

- Extract more than one group of patterns from a column:

```
df[col_name].str.extractall(pattern)
```

- Replace a regex or string in a column with another string:

```
df[col_name].str.replace(pattern, replacement_string)
```

Concepts

- Pandas has built in a number of vectorized methods that perform the same operations for strings in Series as Python string methods.

- A regular expression is a sequence of characters that describes a search pattern. In pandas, regular expressions is integrated with vectorized string methods to make finding and extracting patterns of characters easier.

Resources

- Working with Text Data
- Regular Expressions

Working With Missing And Duplicate Data: Takeaways

Syntax

IDENTIFYING MISSING VALUES

- Identify rows with missing values in a specific column:

```
missing = df[col_name].isnull()
df[missing]
```

- Calculate the number of missing values in each column:

```
df.isnull().sum()
```

REMOVING MISSING VALUES

- Drop rows with any missing values:

```
df.dropna()
```

- Drop specific columns:

```
df.drop(columns_to_drop, axis=1)
```

- Drop columns with less than a certain number of non-null values:

```
df.dropna(thresh = min_nonnull, axis=1)
```

REPLACING MISSING VALUES

- Replace missing values in a column with another value:

```
df[col_name].fillna(replacement_value)
```

VISUALIZING MISSING DATA

- Use a heatmap to visualize missing data:

```
import seaborn as sns
sns.heatmap(df.isnull(), cbar=False)
```

CORRECTING DUPLICATE VALUES

- Identify duplicates values:

```
dups = df.duplicated()
df[dups]
```

- Identify rows with duplicate values in only certain columns:

```
dups = df.duplicated([col_1, col_2])
df[dups]
```

- Drop duplicate values. Keep the first duplicate row:

```
df.drop_duplicates()
```

- Drop rows with duplicate values in only certain columns. Keep the last duplicate row:

```
combined.drop_duplicates([col_1, col_2], keep='last')
```

Concepts

- Missing or duplicate data may exist in a data set for many reasons. Sometimes, they may exist because of user input errors or data conversion issues; other times, they may be introduced while performing data cleaning tasks. In the case of missing values, they may also exist in the original data set to purposely indicate that data is unavailable.
- In pandas, missing values are generally represented by the `NaN` value or the `None` value.
- To handle missing values, first check for errors made while performing data cleaning tasks. Then, try to use available data from other sources (if it exists) to fill them in. Otherwise, consider dropping them or replacing them with other values.

Resources

- Working with Missing Data