Combining rows or columns

Concat Merge Join

Melt

Pandas melt() function is used to change the DataFrame format from wide to long

Importing Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
```

Mounting google drive

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Defining the dataframe

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})

```

```
print(df1)
print(df2)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
```

To concatenate the DataFrames along the row you can use the concat() function in pandas.

```
df3=pd.concat([df1,df2],axis=1)
df3
```

	employee	group	employee	hire_date
0	Bob	Accounting	Lisa	2004
1	Jake	Engineering	Bob	2008
2	Lisa	Engineering	Jake	2012
3	Sue	HR	Sue	2014

Double-click (or enter) to edit

```
df3 = pd.merge(df1, df2)
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
pd.merge(df3, df4)
```

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

```
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008

Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
```

```
pd.merge(df3, df5)
```

	employee	group	hire_date	skills
0	Bob	Accounting	2008	math
1	Bob	Accounting	2008	spreadsheets
2	Jake	Engineering	2012	coding
3	Jake	Engineering	2012	linux
4	Lisa	Engineering	2004	coding
5	Lisa	Engineering	2004	linux
6	Sue	HR	2014	spreadsheets
7	Sue	HR	2014	organization

The on keyword

Most simply, you can explicitly specify the name of the key column using the on keyword, which takes a column name or a list of column names:

```
pd.merge(df1, df2, on='employee')
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

The left_on and right_on keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the left_on and right_on keywords to specify the two column names:

```
df6 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
pd.merge(df1, df6, left_on="employee", right_on="name")
```

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

The left_index and right_index keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
df2a
```

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

```
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering

For convenience, DataFrames implement the join() method, which performs a merge that defaults to joining on indices:

```
df1a.join(df2a)
```

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

The **how parameter** of the **merge function** works in a similar way.

The possible values for how are

inner, outer, left, right.

inner: only rows with same values in the column specified by on parameter (default value of how parameter)

outer: all the rows

left: all rows from left DataFrame

right: all rows from right DataFrame

```
df7 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                     'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])
df8 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                     'drink': ['wine', 'beer']},
                    columns=['name', 'drink'])
pd.merge(df7, df8)
```

	name	food	drink
0	Mary	bread	wine

```
pd.merge(df7, df8, how='inner')
pd.merge(df7,df8,how='outer')
pd.merge(df7,df8,how='left')
pd.merge(df7,df8,how='right')
```

	name	food	drink
0	Mary	bread	wine
1	Joseph	NaN	beer

```
pd.merge(df6, df7, how='outer')
pd.merge(df6, df7, how='left')
```

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

```
pd.merge(df6, df7, how='right')
```

	name	food	drink
0	Mary	bread	wine
1	Joseph	NaN	beer

Join parameter takes two values, outer and inner.

Outer: Take all the indices (default value of join parameter)

Inner: Take only shared indices

```
pd.concat([df7, df8],join='inner',axis=0)
```

name	food	drink

```
pd.concat([df7, df8], join='outer',axis=1)
```

	name	food	name	drink
0	Peter	fish	Mary	wine
1	Paul	beans	Joseph	beer
2	Mary	bread	NaN	NaN

melt() function

It is useful to manage a DataFrame into a format where one or more columns are identifier variables, while all other columns, considered measured variables, are unpivoted to the row axis, leaving just two non-identifier columns, variable and value.

```
# Create a simple dataframe

# importing pandas as pd
import pandas as pd

# creating a dataframe
df = pd.DataFrame({'Name': {0: 'John', 1: 'Bob', 2: 'Shiela'},
        'Course': {0: 'Masters', 1: 'Graduate', 2: 'Graduate'},
        'Age': {0: 27, 1: 23, 2: 21}})
df

```

	Name	Course	Age
0	John	Masters	27
1	Bob	Graduate	23
2	Shiela	Graduate	21

```
# Name is id_vars and Course is value_vars
pd.melt(df, id_vars =['Name'], value_vars =['Course'])

```

	Name	variable	value
0	John	Course	Masters
1	Bob	Course	Graduate
2	Shiela	Course	Graduate

```
# multiple unpivot columns
pd.melt(df, id_vars =['Name'], value_vars =['Course', 'Age'])

```

	Name	variable	value
0	John	Course	Masters
1	Bob	Course	Graduate
2	Shiela	Course	Graduate
3	John	Age	27
4	Bob	Age	23
5	Shiela	Age	21

```
# Names of ‘variable’ and ‘value’ columns can be customized
pd.melt(df, id_vars =['Name'], value_vars =['Course'],
      var_name ='ChangedVarname', value_name ='ChangedValname')

```

	Name	ChangedVarname	ChangedValname
0	John	Course	Masters

```python
d1 = {"Name": ["Pankaj", "Lisa", "David"],
      "ID": [1, 2, 3], "Role": ["CEO", "Editor", "Author"]}

df = pd.DataFrame(d1)

print(df)

df_melted = pd.melt(df, id_vars=["ID"], value_vars=["Name", "Role"])

print(df_melted)
```

```
     Name  ID    Role
0  Pankaj   1     CEO
1    Lisa   2  Editor
2   David   3  Author
   ID variable   value
0   1     Name  Pankaj
1   2     Name    Lisa
2   3     Name   David
3   1     Role     CEO
4   2     Role  Editor
5   3     Role  Author
```

Multiple Columns as id_vars

Let’s see what happens when we pass multiple columns as the id_vars parameter.

```python
df_melted = pd.melt(df, id_vars=["ID", "Name"], value_vars=["Role"])
print(df_melted)
```

```
   ID    Name variable   value
0   1  Pankaj     Role     CEO
```

```
1   2    Lisa     Role  Editor
2   3   David     Role  Author
```

Skipping Columns in melt() Function

It’s not required to use all the rows from the source DataFrame. Let’s skip the “ID” column in the next example.

```
df_melted = pd.melt(df, id_vars=["Name"], value_vars=["Role"])
print(df_melted)
```

```
     Name variable   value
0  Pankaj     Role     CEO
1    Lisa     Role  Editor
2   David     Role  Author
```

Unmelting DataFrame using pivot() function

We can use pivot() function to unmelt a DataFrame object and get the original dataframe. The pivot() function ‘index’ parameter value should be same as the ‘id_vars’ value. The ‘columns’ value should be passed as the name of the ‘variable’ column.

```
d1 = {"Name": ["Pankaj", "Lisa", "David"], "ID": [1, 2, 3],
      "Role": ["CEO", "Editor", "Author"]}

df = pd.DataFrame(d1)

# print(df)

df_melted = pd.melt(df, id_vars=["ID"], value_vars=["Name", "Role"],
                    var_name="Attribute", value_name="Value")

print(df_melted)

# unmelting using pivot()

df_unmelted = df_melted.pivot(index='ID', columns='Attribute')
```

```

print(df_unmelted)
```

```
   ID Attribute   Value
0   1      Name  Pankaj
1   2      Name    Lisa
2   3      Name   David
3   1      Role     CEO
4   2      Role  Editor
5   3      Role  Author
            Value
Attribute    Name    Role
ID
1          Pankaj     CEO
2            Lisa  Editor
3           David  Author
```