In [ ]:

```
Data Wrangling is the process of gathering, collecting,
and transforming raw data into another format for better understanding.

It is the process of taking disorganized or incomplete raw data and
standardizing it so that you can easily access, consolidate, and analyze it. It also in
mapping data fields from source to destination, for example, targeting a field, row, or
in a dataset and implementing an action like joining, parsing, cleaning, consolidating,
to produce the required output.

#Combining and Merging Data Sets
Data contained in pandas objects can be combined together in a number of built-in
ways:

• pandas.merge connects rows in DataFrames based on one or more keys. (SQL join)
• pandas.concat glues or stacks together objects along an axis.
• combine_first instance method enables splicing together overlapping data to fill
                          in missing values in one object with values from another.
```

In [ ]:

```
import pandas as pd
import numpy as np
```

In [ ]:

```
#Database-style DataFrame Merges
import pandas as pd
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
  'data1': range(7)})
print(df1)
df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
  'data2': range(3)})
print(df2)
```

In [ ]:

```
# many-to-one merge situation based on column name
pd.merge(df1, df2)
```

In [ ]:

```
#specify which column to join on..
pd.merge(df1, df2, on='key')
```

In [ ]:

```
#If the column names are different in each object, specify them separately
#merge does an 'inner' join; the keys in the result are the intersection.

df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   'data1': range(7)})
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   'data2': range(3)})
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

In [ ]:

```
#The outer join takes the union of the keys,
#combining the effect of applying both left and right joins:

pd.merge(df1, df2, how='outer')
```

In [ ]:
```
#Many-to-many merge situation
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   'data1': range(6)})
print(df1)
df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   'data2': range(5)})
print(df2)
pd.merge(df1, df2, on='key', how='left')
```

In [ ]:
```
#Merging on Index
'''In some cases, the merge key or keys in a DataFrame will be found in its index. In t
case, you can pass left_index=True or right_index=True (or both) to indicate that the
index should be used as the merge key:'''

left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   'value': range(6)})
right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

pd.merge(left1, right1, left_on='key', right_index=True)
```

In [ ]:
```
pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

In [ ]:
```
#Concatenating Along an Axis
#Another kind of data combination operation
import numpy as np
arr = np.arange(12).reshape((3, 4))
arr
```

In [ ]:
```
np.concatenate([arr, arr], axis=1)
```

In [ ]:
```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
print(pd.concat([s1, s2, s3]))
print( pd.concat([s1, s2, s3], axis=1))
```

In [ ]:
```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

In [ ]:
```
df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
   columns=['one', 'two'])
df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
   columns=['three', 'four'])
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

In [ ]:
```
pd.concat({'level1': df1, 'level2': df2}, axis=1)
```

In [ ]:

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
 names=['upper', 'lower'])
```

In [ ]:

```
#DataFrames in which the row index is not meaningful
df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
print(df1)
df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
print(df2)
pd.concat([df1, df2], ignore_index=True)
```

In [ ]:

```
#Combining Data with Overlap
#You may have two datasets whose indexes overlap in full or part
a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
   index=['f', 'e', 'd', 'c', 'b', 'a'])
print(a)
b = pd.Series(np.arange(len(a), dtype=np.float64),
   index=['f', 'e', 'd', 'c', 'b', 'a'])
print(b)
#b[-1] = np.nan
np.where(pd.isnull(a), b, a)
```

In [27]:

```
#Reshaping and Pivoting
#Reshaping with hierarchical indexing
'''There are a number of fundamental operations for rearranging tabular data. These are
alternatingly referred to as reshape or pivot operations.

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame.
There are two primary actions:
• stack: this “rotates” or pivots from the columns in the data to the rows
• unstack: this pivots from the rows into the columns

'''
import pandas as pd
import numpy as np
data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   index=pd.Index(['Ohio', 'Colorado'], name='state'),
   columns=pd.Index(['one', 'two', 'three'], name='number'))
data
```

Out[27]:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

In [28]:

```
#Using the stack method on this data, pivots the columns into the rows, producing a
#Series:
df1=data.stack()
df1
```

Out[28]:
```
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int32
```

In [ ]:
```
#By default the innermost level is unstacked (same with stack).
#You can unstack a different level by passing a level number or name:
```

In [29]:
```
#From a hierarchically-indexed Series, rearrange the data back into a DataFrame
#with unstack:
df1.unstack()
```

Out[29]:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

In [30]:
```
df1.unstack(0)
```

Out[30]:

state	Ohio	Colorado
number		
one	0	3
two	1	4
three	2	5

In [31]:
```
df1.unstack('state')
```

Out[31]:

state	Ohio	Colorado
number		
one	0	3
two	1	4
three	2	5

In [33]:
```
#Unstacking might introduce missing data if all of the values in the level aren't found
#each of the subgroups:
s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
data2 = pd.concat([s1, s2], keys=['one', 'two'])
print(data2)
print(data2.unstack())
```

```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

In [34]:

```
#Stacking filters out missing data by default, so the operation is easily invertible:
data2.unstack().stack()
#data2.unstack().stack(dropna=False)
```

Out[34]:

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64
```

In [ ]:

In [ ]:

In [36]:

```
#Pivoting “long” to “wide” Format
#A common way to store multiple time series in databases and CSV is in so-called long
#or stacked format:
import pandas as pd
import numpy as np
df=pd.read_csv('stu1.csv')
print(df)
pivoted=df.pivot('stno','sem','SGPA')
print('\n',' pivoting...','\n')
print(pivoted)
```

```
       stno  sem  SGPA
0   y19cs01    I   7.8
1   y19cs01   II   8.0
2   y19cs01  III   9.8
3   y19cs02    I   7.5
4   y19cs02   II   7.7
5   y19cs02  III   7.8
6   y19cs03    I   8.8
7   y19cs03   II   7.8
8   y19cs03  III   9.8
9   y19cs04    I   7.2
10  y19cs04   II   7.8
11  y19cs04  III   7.8

  pivoting...
```

```
sem        I   II  III
stno
y19cs01  7.8  8.0  9.8
y19cs02  7.5  7.7  7.8
y19cs03  8.8  7.8  9.8
y19cs04  7.2  7.8  7.8
```

In [37]:

```
df=pd.read_csv('stu.csv')
print(df)
pivoted=df.pivot('stno','sem')
print('\n',' pivoting...','\n')
print(pivoted)
```

```
       stno  sem  sub  SGPA
0   y19cs01    I  PPS   7.8
1   y19cs01   II   DS   8.0
2   y19cs01  III  DAA   9.8
3   y19cs02    I  PPS   7.5
4   y19cs02   II   DS   7.7
5   y19cs02  III  DAA   7.8
6   y19cs03    I  PPS   8.8
7   y19cs03   II   DS   7.8
8   y19cs03  III  DAA   9.8
9   y19cs04    I  PPS   7.2
10  y19cs04   II   DS   7.8
11  y19cs04  III  DAA   7.8

  pivoting...

         sub            SGPA
sem        I  II  III    I   II  III
stno
y19cs01  PPS  DS  DAA  7.8  8.0  9.8
y19cs02  PPS  DS  DAA  7.5  7.7  7.8
y19cs03  PPS  DS  DAA  8.8  7.8  9.8
y19cs04  PPS  DS  DAA  7.2  7.8  7.8
```

In [38]:

```
'''
The pivot table takes simple columnwise data as input, and groups the entries into a tw
a multidimensional summarization of the data.
'''
df=pd.read_csv('stu.csv')
print(df)
print(df['stno'])
df.pivot_table('SGPA',index='stno',columns='sem')
```

```
       stno  sem  sub  SGPA
0   y19cs01    I  PPS   7.8
1   y19cs01   II   DS   8.0
2   y19cs01  III  DAA   9.8
3   y19cs02    I  PPS   7.5
4   y19cs02   II   DS   7.7
5   y19cs02  III  DAA   7.8
6   y19cs03    I  PPS   8.8
7   y19cs03   II   DS   7.8
8   y19cs03  III  DAA   9.8
9   y19cs04    I  PPS   7.2
10  y19cs04   II   DS   7.8
```

```
11  y19cs04  III  DAA   7.8
0     y19cs01
1     y19cs01
2     y19cs01
3     y19cs02
4     y19cs02
5     y19cs02
6     y19cs03
7     y19cs03
8     y19cs03
9     y19cs04
10    y19cs04
11    y19cs04
Name: stno, dtype: object
```

Out[38]:

sem	I	II	III
stno			
y19cs01	7.8	8.0	9.8
y19cs02	7.5	7.7	7.8
y19cs03	8.8	7.8	9.8
y19cs04	7.2	7.8	7.8

In [39]:

```
#Permutation and Random Sampling
df = pd.DataFrame(np.arange(5 * 4).reshape(5, 4))
df
sampler = np.random.permutation(5)
print(sampler)
df.take(sampler)
```

```
[1 3 4 2 0]
```

Out[39]:

	0	1	2	3
1	4	5	6	7
3	12	13	14	15
4	16	17	18	19
2	8	9	10	11
0	0	1	2	3

In [ ]:

```
??

#select a random subset without replacement
df.take(np.random.permutation(len(df))[:3])
```

In [ ]:

```
# Vectorization is about finding ways to apply an operation to a set of values at once

#Vectorized String Operations
#One strength of Python is its relative ease in handling and manipulating string data
#Pandas builds on this and provides a comprehensive set of vectorized string operations
#This vectorization of operations simplifies the syntax of operating on arrays of data:
```

```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2

#we no longer have to worry about the size or shape of the array
```

In [44]:

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
print([s.capitalize() for s in data])

data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
print([s.capitalize() for s in data])
```

```
['Peter', 'Paul', 'Mary', 'Guido']

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_2820/3627870036.py in <module>
      3
      4 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 5 print([s.capitalize() for s in data])

~\AppData\Local\Temp/ipykernel_2820/3627870036.py in <listcomp>(.0)
      3
      4 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 5 print([s.capitalize() for s in data])

AttributeError: 'NoneType' object has no attribute 'capitalize'
```

In [45]:

```
#call a single method that will capitalize all the entries, while skipping
#over any missing values:
import pandas as pd
names = pd.Series(data)
print(names)
print(names.str.capitalize())
print(names.str.len())
print(names.str.startswith('p'))
```

```
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
0    5.0
1    4.0
2    NaN
3    4.0
4    5.0
dtype: float64
0     True
1    False
2     None
3    False
```

```
4    False
dtype: object
```

In [ ]:

```
#Vectorized item access and slicing.
print(names.str[0:2])
print(names.str.slice(0,2))
print(names.str.get(2))
print(names.str[2])
```

In [ ]:

```
#Indicator variables
'''
This is useful when your data has a column containing some
sort of coded indicator. For example, we might have a dataset that contains informa-
tion in the form of codes, such as A="born in America," B="born in the United King-
dom," C="likes cheese," D="likes spam"
'''
ds= pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
 'Eric Idle', 'Terry Jones', 'Michael Palin'])

df = pd.DataFrame({'name': ds,
 'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C',
 'B|C|D']})
df
```

In [ ]:

```
df['info'].str.get_dummies('|')
```

In [ ]: