21CSS303T
DATA SCIENCE
UNIT-2
DATA WRANGLING, DATA CLEANING AND PREPARATION

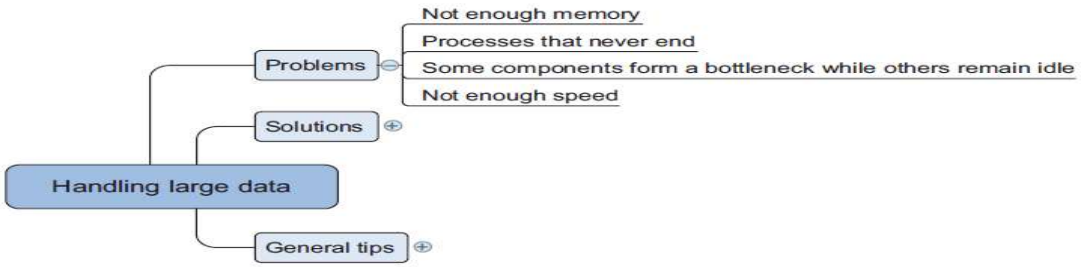
- **Data Handling**
- **Problem faced when handling large data**
- **General techniques for handling large volume of data**
- **General programming tips for dealing large data sets**
- **Data Wrangling**
  - **Clean**
  - **Transform**
  - **Merge**
  - **Reshape**
- **Combining and Merging Datasets**
  - **Merging on Index**
  - **Concatenate**
  - **Combining with overlap**

DATA HANDLING

Data handling is the process of ensuring that research data is stored, archived or disposed-off in a safe and secure manner during and after the conclusion of a research project.

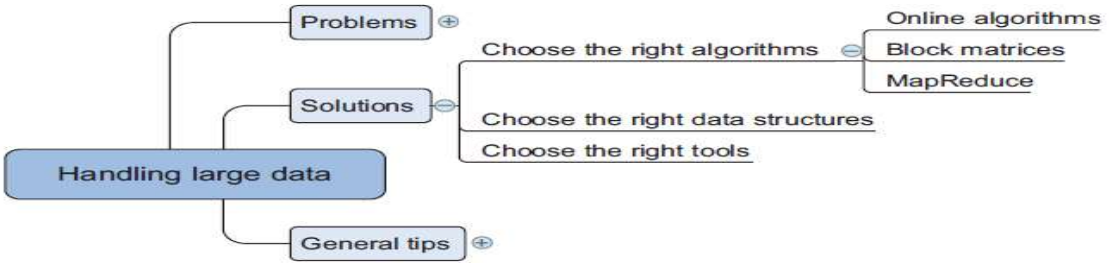
PROBLEM FACED WHEN HANDLING LARGE DATA

- **Data quality**: Data validation can help ensure that data is accurate, complete, and properly formatted.
- **Security and privacy**: As the amount of data increases, so the security and privacy concerns.
- **Cost**: Managing large amounts of data can be expensive, especially for organizations that generate large volumes of data daily adopt cost-effective solutions.
- **Data integration**: Data integration tools can help combine data from different sources and make it available for analysis.
- **Accessibility**: Organizations need to make data easy and convenient for users of all skill levels to use.
- **Finding the right tools**: Organizations need to find the right technology to work within their established ecosystems and address their particular needs.
- **Uncovering insights**: Organizations need to analyze big data to unearth intelligence to drive better decision making.
- **Organizational resistance**: Companies need to rethink processes, workflows, and even the way problems are approached.

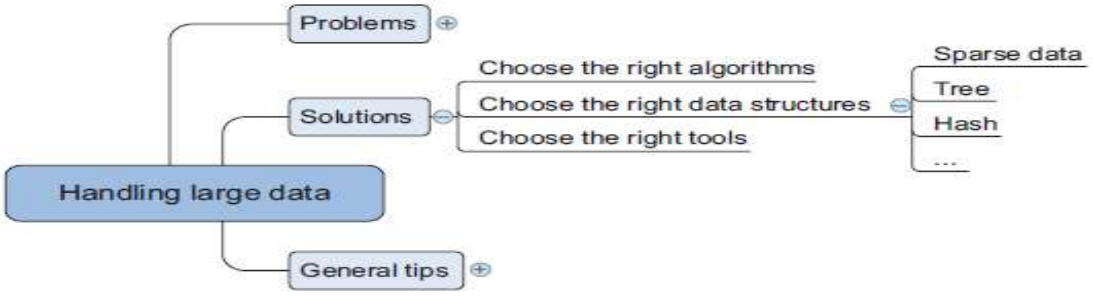


GENERAL TECHNIQUES FOR HANDLING LARGE VOLUME OF DATA

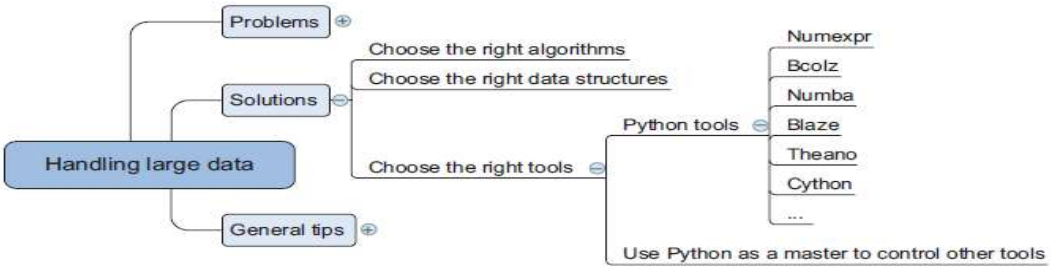
Solution 1: Choose the right algorithm



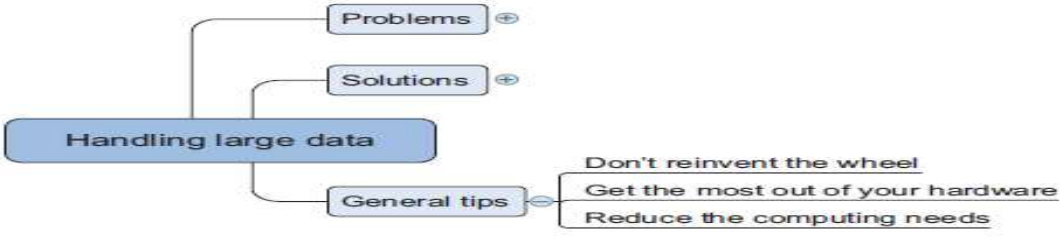
Solution 2: Choose the right data structure



Solution 3: Choose the right tools



GENERAL PROGRAMMING TIPS FOR DEALING LARGE DATA SETS



DATA WRANGLING

Data Wrangling is the process of gathering, collecting, and transforming Raw data into another format for better understanding, decision-making, accessing, and analysis in less time. Data Wrangling is also known as **Data Munging**.

Data wrangling involves processing the data in various formats like - merging, grouping, concatenating etc. for the purpose of analysing or getting them ready to be used with another set of data.

Data wrangling Process:

1. **Data acquisition**: This step involves obtaining the data from various sources and storing it in a central location.
2. **Data cleaning**: This step involves cleaning the data to ensure that it is accurate and consistent.
3. **Data exploration**: This step involves exploring the data to understand its structure and content.
4. **Data transformation**: This step involves transforming the data into a format that is more appropriate for analysis.
5. **Data loading:** This step involves loading the data into the appropriate analysis tool or platform.

- **CLEAN:** For data cleaning, the process typically includes the following steps:

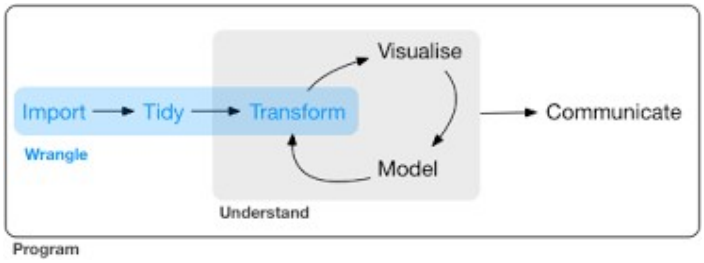
1. Data inspection: This step involves reviewing the data to identify any errors, inconsistencies, or missing values.
2. Data validation: This step involves checking the data against a set of rules or constraints to ensure that it is accurate and consistent.
3. Data correction: This step involves making any necessary corrections to the data, such as filling in missing values or removing duplicate data.
4. Data standardization: This step involves ensuring that the data is in a consistent format and that it conforms to a set of standards.
5. Data transformation: This step involves transforming the data into a format that is more appropriate for analysis.

Data Wrangling vs Data Cleaning: Data Wrangling focuses on structuring and validating data whereas Data Cleaning focuses on ensuring clean and quality data is available. Image

- **TRANSFORM:** "Transform" in data wrangling refers to the process of changing raw data into a structured and usable format by modifying, manipulating, or converting its values, often including reshaping, combining data from different sources, and applying mathematical functions to prepare it for analysis; essentially, it's the core step of taking messy data and putting it into the right shape for further use in data analysis or modeling. To make raw data accessible and suitable for analysis by adjusting its structure, format, and values.

- **Reshaping data**: Changing the layout of data, like converting rows to columns or vice versa.
- **Combining data**: Merging data from multiple sources into a single dataset.
- **Creating new variables**: Calculating new features based on existing data.

- **Data normalization**: Scaling data values to a common range
- **Applying mathematical functions**: Taking logarithms, square roots, or other transformations to improve data distribution



- # MERGE:

- **Merging** combines datasets based on common columns or indices, similar to SQL joins.
- **Joining** performs the same function as merging, but it's specifically used with DataFrames that share indexes.
- **Concatenating** stacks data along a particular axis, either appending rows or adding columns.

Types of Merges

Pandas offers different types of joins, controlled by the '**how'** parameter:

1. **Inner Join (default):** Returns only rows with matching keys.
2. **Outer Join**: Returns all rows from both DataFrames, filling in missing values.
3. **Left Join**: Keeps all rows from the left DataFrame.
4. **Right Join**: Keeps all rows from the right DataFrame.

Table 8.1: Different join types with the `how` argument

Option	Behavior
`how="inner"`	Use only the key combinations observed in both tables
`how="left"`	Use all key combinations found in the left table
`how="right"`	Use all key combinations found in the right table
`how="outer"`	Use all key combinations observed in both tables together

- **RESHAPE:** Reshaping involves changing how data is organized by moving information between rows and columns. The reason we need to reshape our data and move flexibly between wide and long formats is that certain data wrangling operations and model specifications are easier done in either format.

Data Shapes: Dataframes can be organized in different ways for different purposes. It come in less-than-ideal formats, especially when you are using secondary data. Data comes in two primary shapes:

1. Wide
2. Long

1. **Wide:** Data is wide when a row has more than one observation, and the units of observation (e.g., individuals, countries, households) are on one row each.
   In the example below, each row corresponds to a single person, and each column is a different observation for that person.

ID	Income2000	Income2001	Income2002	...
001	50000	52000	56000	
002	0	30000	31000	
003	6800	6400	6850	
...				

2. **Long:** Data is long when a row has only one observation, but the units of observation are repeated down a column. Longitudinal data is often in the long format. You might have a column where ID numbers are repeated, a column marking when each data point was observed, and another column with observed values.

ID	Year	Income
001	2000	50000
001	2001	52000
001	2002	56000
002	2000	0
002	2001	30000
002	2002	31000
003	2000	6800
003	2001	6400
003	2002	6850
...		

DATA WRANGLING PROCESS

1. **Data Exploration**: Load the data into a dataframe, and then we visualize the data in a tabular format.

Example:

```
import pandas as pd
data = {'Name': ['Jai', 'Princi', 'Gaurav',
'Anuj', 'Ravi', 'Natasha', 'Riya'],
'Age': [17, 17, 18, 17, 18, 17, 17],
'Gender': ['M', 'F', 'M', 'M', 'M', 'F', 'F'],
'Marks': [90, 76, 'NaN', 74, 65, 'NaN', 71]}
df = pd.DataFrame(data)
print(df)
```

Output:

	Name	Age	Gender	Marks
0	Jai	17	M	90
1	Princi	17	F	76
2	Gaurav	18	M	NaN
3	Anuj	17	M	74
4	Ravi	18	M	65
5	Natasha	17	F	NaN
6	Riya	17	F	71

2. **Dealing with missing values**: In the previous output, there are NaN values present in the MARKS column which is a missing value in the dataframe that is going to be taken care of in data wrangling by replacing them with the column mean.

Example:

```
c = avg = 0
for ele in df['Marks']:
if str(ele).isnumeric():
c += 1
avg += ele
avg /= c

df = df.replace(to_replace="NaN", value=avg)
df
```

Output:

	Name	Age	Gender	Marks
0	Jai	17	M	90.0
1	Princi	17	F	76.0
2	Gaurav	18	M	75.2
3	Anuj	17	M	74.0
4	Ravi	18	M	65.0
5	Natasha	17	F	75.2
6	Riya	17	F	71.0

3. **Data Replacing**: in the GENDER column, we can replace the Gender column data by categorizing them into different numbers.

Example:

```
df['Gender'] = df['Gender'].map({'M': 0,  'F': 1, }).astype(float)
df
```

Output:

	Name	Age	Gender	Marks
0	Jai	17	0.0	90.0
1	Princi	17	1.0	76.0
2	Gaurav	18	0.0	75.2
3	Anuj	17	0.0	74.0
4	Ravi	18	0.0	65.0
5	Natasha	17	1.0	75.2
6	Riya	17	1.0	71.0

4. **Filtering data:** Suppose there is a requirement for the details regarding name, gender, and marks of the top-scoring students. Here we need to remove some using the pandas slicing method in data wrangling from unwanted data.

Example:

```
# Filter top scoring students
df = df[df['Marks'] >= 75].copy()
df.drop('Age', axis=1, inplace=True)
df
```

Output:

	Name	Gender	Marks
0	Jai	0.0	90.0
1	Princi	1.0	76.0
2	Gaurav	0.0	75.2
5	Natasha	1.0	75.2

COMBINING AND MERGING DATASETS

Data contained in pandas objects can be combined together in a number of ways:
• ***pandas.merge*** connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database join operations.
• ***pandas.concat*** concatenates or "stacks" together objects along an axis.
• The ***combine_first*** instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

1. **Database-Style DataFrame Joins:** ***Merge*** or ***join*** operations combine datasets by linking rows using one or more keys. These operations are central to relational databases (e.g., SQL-based).

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df2 = pd.DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)})
df1
```

Output:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

df2

Output:

	data2	key
0	0	a
1	1	b
2	2	d

This is an example of a **many-to-one join**; the data in df1 has multiple rows labeled **a** and **b**, whereas df2 has only one row for each value in the key column. Calling merge with these objects we obtain:

pd.merge(df1, df2)

Output:

	data1	key	data2
0	0	b	1
1	1	b	1
2	6	b	1
3	2	a	0
4	4	a	0
5	5	a	0

Note that I didn't specify which column to join on. If that information is not specified, merge uses the overlapping column names as the keys.

pd.merge(df1, df2, on='key')

Output:

	data1	key	data2
0	0	b	1
1	1	b	1
2	6	b	1
3	2	a	0
4	4	a	0
5	5	a	0

If the column names are different in each object, you can specify them separately:

```
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')
```

Output:

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

You may notice that the **'c'** and **'d'** values and associated data are missing from the result. By **default**, merge does an **'inner'** join; the keys in the result are the intersection, or the common set found in both tables. Other possible options are ***'left', 'right', and 'outer'***. 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')

Output:

	data1	key	data2
0	0.0	b	1.0
1	1.0	b	1.0
2	6.0	b	1.0
3	2.0	a	0.0
4	4.0	a	0.0
5	5.0	a	0.0
6	3.0	c	NaN
7	NaN	d	2.0

Many-to-many merges have well-defined, though not necessarily intuitive, behavior.

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1': range(6)})
df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'], 'data2': range(5)})
df1
```

Output:

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	b

```
df2
```

Output:

	data2	key
0	0	a
1	1	b
2	2	a
3	3	b
4	4	d

```
pd.merge(df1, df2, on='key', how='left')
```

Output:

	data1	key	data2
0	0	b	1.0
1	0	b	3.0
2	1	b	1.0
3	1	b	3.0
4	2	a	0.0
5	2	a	2.0
6	3	c	NaN
7	4	a	0.0
8	4	a	2.0
9	5	b	1.0
10	5	b	3.0

Many-to-many joins form the Cartesian product of the rows. Since there were three 'b' rows in the left DataFrame and two in the right one, there are six 'b' rows in the result. The join method only affects the distinct key values appearing in the result:

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

Output:

	data1	key	data2
0	0	b	1
1	0	b	3
2	1	b	1
3	1	b	3
4	5	b	1
5	5	b	3
6	2	a	0
7	2	a	2
8	4	a	0
9	4	a	2

To merge with multiple keys, pass a list of column names:

```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'], 'key2': ['one', 'two', 'one'], 'lval': [1, 2, 3]})
right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'], 'key2': ['one', 'one', 'one', 'two'], 'rval': [4, 5, 6, 7]})
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

Output:

	key1	key2	lval	rval
0	foo	one	1.0	4.0
1	foo	one	1.0	5.0
2	foo	two	2.0	NaN
3	bar	one	3.0	6.0
4	bar	two	NaN	7.0

Table 8-1. Different join types with how argument

Option	Behavior
`'inner'`	Use only the key combinations observed in both tables
`'left'`	Use all key combinations found in the left table
`'right'`	Use all key combinations found in the right table
`'output'`	Use all key combinations observed in both tables together

A **last issue** to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually, merge has a suffixes option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
pd.merge(left, right, on='key1')
```

Output:

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

Output:

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

Table 8-2. merge function arguments

Argument	Description
`left`	DataFrame to be merged on the left side.
`right`	DataFrame to be merged on the right side.
`how`	One of `'inner'`, `'outer'`, `'left'`, or `'right'`; defaults to `'inner'`.
`on`	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in `left` and `right` as the join keys.
`left_on`	Columns in `left` DataFrame to use as join keys.
`right_on`	Analogous to `left_on` for `left` DataFrame.
`left_index`	Use row index in `left` as its join key (or keys, if a MultiIndex).
`right_index`	Analogous to `left_index`.
`sort`	Sort merged data lexicographically by join keys; `True` by default (disable to get better performance in some cases on large datasets).
`suffixes`	Tuple of string values to append to column names in case of overlap; defaults to `('_x', '_y')` (e.g., if `'data'` in both DataFrame objects, would appear as `'data_x'` and `'data_y'` in result).
`copy`	If `False`, avoid copying data into resulting data structure in some exceptional cases; by default always copies.
`indicator`	Adds a special column `_merge` that indicates the source of each row; values will be `'left_only'`, `'right_only'`, or `'both'` based on the origin of the joined data in each row.

b) Merging on Index: In some cases, the merge key(s) in a DataFrame will be found in its index. In this 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'])
left1
```

Output:

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

right1

Output:

	group_val
a	3.5
b	7.0

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

Output:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

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

Output:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

With hierarchically indexed data, things are more complicated, as joining on index is implicitly a multiple-key merge:

lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'], 'key2': [2000, 2001, 2002, 2001, 2002], 'data': np.arange(5.)})

righth = pd.DataFrame(np.arange(12).reshape((6, 2)), index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'], [2001, 2000, 2000, 2000, 2001, 2002]], columns=['event1', 'event2'])

lefth

Output:

	data	key1	key2
0	0.0	Ohio	2000
1	1.0	Ohio	2001
2	2.0	Ohio	2002
3	3.0	Nevada	2001
4	4.0	Nevada	2002

righth

Output:

		event1	event2
Nevada	2001	0	1
	2000	2	3
Ohio	2000	4	5
	2000	6	7
	2001	8	9
	2002	10	11

In this case, you have to indicate multiple columns to merge on as a list (note the handling of duplicate index values with ***how='outer'***):

pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)

Output:

	data	key1	key2	event1	event2
0	0.0	Ohio	2000	4	5
0	0.0	Ohio	2000	6	7
1	1.0	Ohio	2001	8	9
2	2.0	Ohio	2002	10	11
3	3.0	Nevada	2001	0	1

pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True, how='outer')

Output:

	data	key1	key2	event1	event2
0	0.0	Ohio	2000	4.0	5.0
0	0.0	Ohio	2000	6.0	7.0
1	1.0	Ohio	2001	8.0	9.0
2	2.0	Ohio	2002	10.0	11.0
3	3.0	Nevada	2001	0.0	1.0
4	4.0	Nevada	2002	NaN	NaN
4	NaN	Nevada	2000	2.0	3.0

Using the indexes of both sides of the merge is also possible:

```
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'],columns=['Ohio', 'Nevada'])
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],index=['b', 'c', 'd', 'e'],
columns=['Missouri', 'Alabama'])
left2
```

Output:

	Ohio	Nevada
a	1.0	2.0
c	3.0	4.0
e	5.0	6.0

right2

Output:

	Missouri	Alabama
b	7.0	8.0
c	9.0	10.0
d	11.0	12.0
e	13.0	14.0

pd.merge(left2, right2, how='outer', left_index=True, right_index=True)

Output:

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

DataFrame has a convenient join instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns.

left2.join(right2, how='outer')

Output:

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

DataFrame's join method performs a left join on the join keys, exactly preserving the left frame's row index. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

left1.join(right1, on='key')

Output:

	key	value	group_val
0	a	0	3.5
1	b	1	7.0
2	a	2	3.5
3	a	3	3.5
4	b	4	7.0
5	c	5	NaN

c) Concatenating Along an Axis: Another kind of data combination operation is referred to interchangeably as concatenation, binding, or stacking. NumPy's concatenate function can do this with NumPy arrays:

```
arr = np.arange(12).reshape((3, 4))
arr
```

Output:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

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

Output:

```
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

```
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'])
```

Calling concat with these objects in a list glues together the values and indexes:

```
pd.concat([s1, s2, s3])
```

Output:

```
a 0
b 1
c 2
d 3
e 4
f 5
g 6
```

By default concat works along axis=0, producing another Series. If you pass axis=1, the result will instead be a DataFrame (axis=1 is the columns):

```
pd.concat([s1, s2, s3], axis=1)
```

Output:

```
     0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

In this case there is no overlap on the other axis, which as you can see is the sorted union (the **'outer' join**) of the indexes. You can instead intersect them by passing **join='inner'**:

```
s4 = pd.concat([s1, s3])
s4
```

Output:

```
a    0
b    1
f    5
g    6
```

```
pd.concat([s1, s4], axis=1)
```

Output:

```
     0  1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6
```

```
pd.concat([s1, s4], axis=1, join='inner')
```

Output:

```
   0  1
a  0  0
b  1  1
```

In this last example, the 'f' and 'g' labels disappeared because of the **join='inner'** option.
You can even specify the axes to be used on the other axes with **join_axes**:

```
pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

Output:

```
     0    1
a  0.0  0.0
c  NaN  NaN
b  1.0  1.0
e  NaN  NaN
```

A potential issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the keys argument:

```
result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
result
```

Output:

```
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
```

```
result.unstack()
```

Output:

```
         a    b    f    g
one    0.0  1.0  NaN  NaN
two    0.0  1.0  NaN  NaN
three  NaN  NaN  5.0  6.0
```

In the case of combining Series along axis=1, the keys become the DataFrame column headers:

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

Output:

```
   one  two  three
a  0.0  NaN    NaN
b  1.0  NaN    NaN
c  NaN  2.0    NaN
d  NaN  3.0    NaN
e  NaN  4.0    NaN
f  NaN  NaN    5.0
g  NaN  NaN    6.0
```

The same logic extends to DataFrame objects:

```
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'])
df1
```

Output:

```
   one  two
a    0    1
b    2    3
c    4    5
```

```
df2
```

Output:

```
   three  four
a      5     6
c      7     8
```

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

Output:

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

If you pass a dict of objects instead of a list, the dict's keys will be used for the keys option:

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

Output:

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

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

Output:

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

A last consideration concerns DataFrames in which the row index does not contain any relevant data:

```
df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
df1
```

Output:

	a	b	c	d
0	1.246435	1.007189	-1.296221	0.274992
1	0.228913	1.352917	0.886429	-2.001637
2	-0.371843	1.669025	-0.438570	-0.539741

```
df2
```

Output:

	b	d	a
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614

In this case, you can pass **ignore_index=True:**

```
pd.concat([df1, df2], ignore_index=True)
```

Output:

	a	b	c	d
0	1.246435	1.007189	-1.296221	0.274992
1	0.228913	1.352917	0.886429	-2.001637
2	-0.371843	1.669025	-0.438570	-0.539741
3	-1.021228	0.476985	NaN	3.248944
4	0.302614	-0.577087	NaN	0.124121

Table 8-3. concat function arguments

Argument	Description
`objs`	List or dict of pandas objects to be concatenated; this is the only required argument
`axis`	Axis to concatenate along; defaults to 0 (along rows)
`join`	Either `'inner'` or `'outer'` (`'outer'` by default); whether to intersection (inner) or union (outer) together indexes along the other axes
`join_axes`	Specific indexes to use for the other *n*−1 axes instead of performing union/intersection logic
`keys`	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis; can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple-level arrays passed in `levels`)
`levels`	Specific indexes to use as hierarchical index level or levels if keys passed
`names`	Names for created hierarchical levels if `keys` and/or `levels` passed
`verify_integrity`	Check new axis in concatenated object for duplicates and raise exception if so; by default (`False`) allows duplicates
`ignore_index`	Do not preserve indexes along concatenation `axis`, instead producing a new `range(total_length)` index

d) Combining Data with Overlap: There is another data combination situation that can't be expressed as either a merge or concatenation operation. You may have two datasets whose indexes overlap in full or part. Consider NumPy's **where** function, which performs the array-oriented equivalent of **an if-else expression**:

```
a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan], index=['f', 'e', 'd', 'c', 'b', 'a'])
b = pd.Series(np.arange(len(a), dtype=np.float64), index=['f', 'e', 'd', 'c', 'b', 'a'])
b[-1] = np.nan
a
```

Output:

```
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
```

```
b
```

Output:

```
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    NaN
```

```
np.where(pd.isnull(a), b, a)
```

Output: array([0. , 2.5, 2. , 3.5, 4.5, nan])

Series has a combine_first method, which performs the equivalent of this operation along with pandas's usual data alignment logic:

```
b[:-2].combine_first(a[2:])
```

Output:

```
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
```

With DataFrames, **combine_first** does the same thing column by column, so you can think of it as "patching" missing data in the calling object with data from the object you pass:

```
df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],'b': [np.nan, 2., np.nan, 6.], 'c': range(2, 18, 4)})
df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],'b': [np.nan, 3., 4., 6., 8.]})
df1
```

Output:

```
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14
```

```
df2
```

Output:

```
     a    b
0  5.0  NaN
1  4.0  3.0
2  NaN  4.0
3  3.0  6.0
4  7.0  8.0
```

```
df1.combine_first(df2)
```

Output:

```
     a    b     c
0  1.0  NaN   2.0
1  4.0  2.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```