Advanced Python Programming for Data Science

CH – 03

Advanced Data Wrangling and Transformation

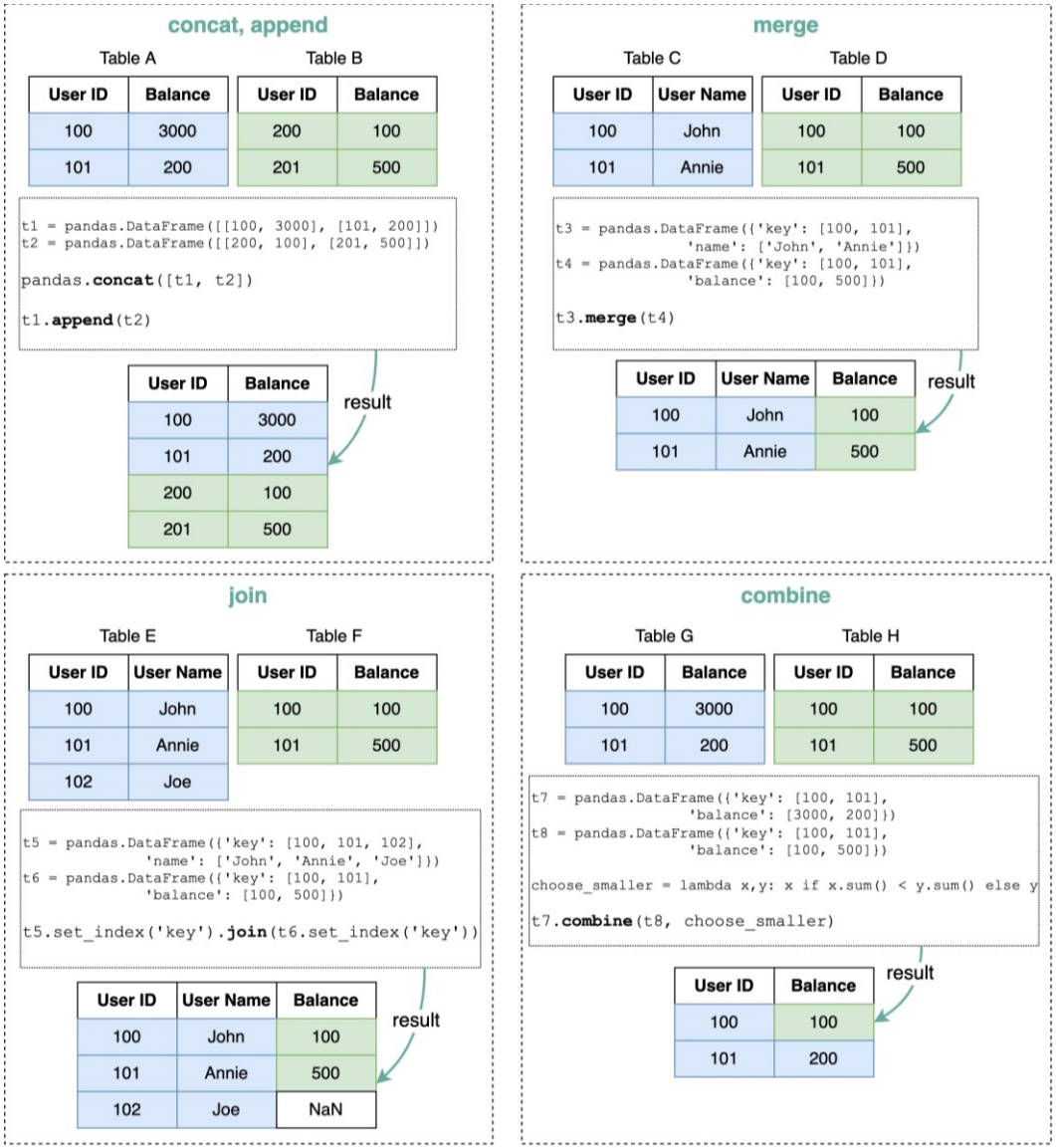
Baikuntha Acharya (baikunth2a@gmail.com)

Senior Lecturer, Sagarmatha Engineering College, Sanepa, Lalitpur

Advanced Pandas operations

✓ Merging and Joining in Pandas

- Combine multiple DataFrames based on a common key (like SQL JOIN).
- Real-world data often comes from multiple sources
- Pandas provides powerful tools to combine datasets
- Core functions:
  - **pd.merge()**
  - **DataFrame.join()**
- Similar to SQL joins, but more flexible



Advanced Pandas operations (Contd..)

- ✓ **Categories of Joins**
  - One-to-One
  - Many-to-One
  - Many-to-Many

- ✓ **Types of join**
  - **Inner join** – keep matching keys only
  - **Left join** – keep all rows from left DataFrame
  - **Right join** – keep all rows from right DataFrame
  - **Outer join** – keep all rows from both

- ✓ **pd.merge()** provides **one unified interface** for combining DataFrames.
- ✓ All three categories of joins are accessed via an identical call to the **pd.merge** interface; the type of join performed depends on the form of the input data
- ✓ We can specify types of join using “how =“ argument.

Advanced Pandas operations (Contd..)

✓ One-to-One Join

- Simplest type of merge operation Each join key appears **exactly once** in both DataFrames. Very similar to **column-wise concatenation**
- Used when two tables store different attributes of the same entities

```
In [2]: 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]})
        display('df1', 'df2')
Out[2]: df1                          df2
          employee        group        employee  hire_date
        0      Bob   Accounting     0     Lisa       2004
        1     Jake  Engineering     1      Bob       2008
        2     Lisa  Engineering     2     Jake       2012
        3      Sue           HR     3      Sue       2014
```

- Two DataFrames contain employee-related information

```
In [3]: df3 = pd.merge(df1, df2)
        df3
Out[3]:   employee        group  hire_date
        0      Bob   Accounting       2008
        1     Jake  Engineering       2012
        2     Lisa  Engineering       2004
        3      Sue           HR       2014
```

- **pd.merge(df1, df2)** automatically joins both DataFrames using the common **employee** column as the key.The result is a new DataFrame that combines employee group and hire date information, regardless of row order.

Advanced Pandas operations (Contd..)

✓ 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.

```
In [4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                            'supervisor': ['Carly', 'Guido', 'Steve']})
        display('df3', 'df4', 'pd.merge(df3, df4)')
Out[4]: df3                                    df4
          employee        group  hire_date              group supervisor
        0      Bob   Accounting       2008      0   Accounting      Carly
        1     Jake  Engineering       2012      1  Engineering      Guido
        2     Lisa  Engineering       2004      2           HR      Steve
        3      Sue           HR       2014

        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
```

- The resulting **DataFrame** has an additional column with the "**supervisor**" information, where the information is repeated in one or more locations as required by the inputs.

Advanced Pandas operations (Contd..)

✓ Many-to-Many Joins

- If the key column in both the left and right arrays contains duplicates, then the result is a many-to-many merge.

```
In [5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                      'Engineering', 'Engineering', 'HR', 'HR'],
                            'skills': ['math', 'spreadsheets', 'software', 'math',
                                       'spreadsheets', 'organization']})
        display('df1', 'df5', "pd.merge(df1, df5)")
Out[5]: df1
          employee        group
        0      Bob   Accounting
        1     Jake  Engineering
        2     Lisa  Engineering
        3      Sue           HR

        df5
              group        skills
        0  Accounting          math
        1  Accounting  spreadsheets
        2  Engineering      software
        3  Engineering          math
        4          HR  spreadsheets
        5          HR  organization

        pd.merge(df1, df5)
          employee        group        skills
        0      Bob   Accounting          math
        1      Bob   Accounting  spreadsheets
        2     Jake  Engineering      software
        3     Jake  Engineering          math
        4     Lisa  Engineering      software
        5     Lisa  Engineering          math
        6      Sue           HR  spreadsheets
        7      Sue           HR  organization
```

- **Output rows increase** because Pandas generates all valid combinations for duplicate keys in a many-to-many join.

Advanced Pandas operations (Contd..)

Table: 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.

Advanced Pandas operations (Contd..)

- ✓ Exercise – Use arguments:
  - How = inner, outer, left, right
  - Eg.: pd.merge(df1, df2, on="id", how="left")

Advanced Pandas operations (Contd..)

✓ The on Keyword

- Used to specify key column

```
In [6]: display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
Out[6]: df1                          df2
          employee        group        employee  hire_date
        0      Bob   Accounting      0     Lisa       2004
        1     Jake  Engineering      1      Bob       2008
        2     Lisa  Engineering      2     Jake       2012
        3      Sue           HR      3      Sue       2014

        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
```

- DataFrames combined using the explicit **employee** column as key.
- Rows aligned by **employee** values, ignoring original order.

Advanced Pandas operations (Contd..)

- ✓ **The left_on and right_on Keywords**
  - Two DataFrames to merge, but key columns have different names:
    - Left DataFrame: **name**
    - Right DataFrame: **employee**
  - Cannot merge directly using **on**
  - The result has a redundant column that we can drop if desired—for example, by using the **DataFrame.drop()** method:

```
In [7]: df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'salary': [70000, 80000, 120000, 90000]})
        display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee",
                right_on="name")')
Out[7]: df1                          df3
          employee        group        name  salary
        0      Bob   Accounting     0   Bob   70000
        1     Jake  Engineering     1  Jake   80000
        2     Lisa  Engineering     2  Lisa  120000
        3      Sue           HR     3   Sue   90000

        pd.merge(df1, df3, 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
```

```
In [8]: pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
Out[8]:   employee        group  salary
        0      Bob   Accounting   70000
        1     Jake  Engineering   80000
        2     Lisa  Engineering  120000
        3      Sue           HR   90000
```

Advanced Pandas operations (Contd..)

- ✓ **The left_index and right_index Keywords**
  - Sometimes the key for merging is the **index** rather than a column.
  - Useful when DataFrames are already indexed by meaningful labels
  - You can use the index as the key for merging by specifying the **left_index** and/or **right_index** flags in **pd.merge()**

```
In [9]: df1a = df1.set_index('employee')
        df2a = df2.set_index('employee')
        display('df1a', 'df2a')
```

df1a

employee	group
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

employee	hire_date
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

```
In [10]: display('df1a', 'df2a',
                 "pd.merge(df1a, df2a, left_index=True, right_index=True)")
```

df1a

employee	group
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

employee	hire_date
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

pd.merge(df1a, df2a, left_index=True, right_index=True)

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

Advanced Pandas operations (Contd..)

- **Mixing Columns and Index:**
- Combine index of one DataFrame with a column of another:

```
In [12]: display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True
                 right_on='name')")
Out[12]: df1a                          df3
                        group          name  salary
         employee                      0   Bob   70000
         Bob          Accounting       1  Jake   80000
         Jake       Engineering        2  Lisa  120000
         Lisa       Engineering        3   Sue   90000
         Sue                HR
         pd.merge(df1a, df3, left_index=True, right_on='name')
                  group  name  salary
         0   Accounting   Bob   70000
         1  Engineering  Jake   80000
         2  Engineering  Lisa  120000
         3           HR   Sue   90000
```

- Flexible for complex merges with multiple indices or columns

Advanced Pandas operations (Contd..)

✓ Joining (Index-Based Alignment)

- Pandas includes the **DataFrame.join()** method, which performs an index-based merge without extra keywords:

```
In [11]: df1a.join(df2a)
Out[11]:                group  hire_date
         employee
         Bob         Accounting       2008
         Jake       Engineering       2012
         Lisa       Engineering       2004
         Sue                 HR       2014
```

```
df1.join(df2, how="left")
                is equivalent to
pd.merge(df1, df2, left_index=True,
right_index=True)
```

- **Joining** is a specialized form of merging where datasets are aligned based on **row identifiers (indices)** rather than explicit key columns.
- Supports chained joins of many DataFrames
- Joining multiple DataFrames: `df1.join([df2, df3, df4], how="outer")`

Aspect	Merging	Joining
Key location	Columns	Index
Flexibility	High	Moderate
Performance	Lower	Higher (index-based)

Advanced Pandas operations (Contd..)

✓ Reshaping and Pivoting

Review the concept of numpy reshaping

- **Reshaping** is the **broad concept**

  Any operation that changes the **layout** of a DataFrame (rows ↔ columns, structure change).

- **Pivoting** is a **specific type of reshaping**

  It reorganizes data **between long format and wide format.**

- Reshaping (or pivoting) means **rearranging the layout of tabular data**
- Changes how data is organized **without changing the actual values**
- Commonly used to move between:
  - **Wide format** (many columns)
  - **Long format** (many rows)

Wide Format

EmployeeID	PreTest	PostTest
EP1619	55	60
EP1845	62	66
EP4321	88	100

Long Format

EmployeeID	TestTime	Score
EP1619	PreTest	55
EP1619	PostTest	60
EP1845	PreTest	62
EP1845	PostTest	66
EP4321	PreTest	88
EP4321	PostTest	100

- Pandas provides powerful tools for reshaping:
  - **stack, unstack**
  - **pivot, pivot_table, melt**

Advanced Pandas operations (Contd..)

- ✓ Pivoting is a transformation that restructures data into a matrix format using one *set of variables as rows, another as columns*, and optionally aggregates values to resolve duplicates.
  - It turns **long/normalized data** into a **2-D analytical table**.
- ✓ Pandas Reshaping & Pivoting — Quick Summary

Operation	Main Purpose	Shape Change	Aggregation	Typical Use
stack()	Columns → index	Wide → Long	No	Create hierarchical index
unstack()	Index → columns	Long → Wide	No	Flatten MultiIndex
pivot()	Reshape data	Long → Wide	No	Fast reshape (unique keys)
pivot_table()	Reshape + summarize	Long → Wide	Yes	Reports, summaries
melt()	Normalize data	Wide → Long	No	Visualization, ML prep

- ✓ **Columns → index**
- ✓ **Wide → Long**

Stack

df2

		A	B
first	second		
bar	one	1	2
	two	3	4
baz	one	5	6
	two	7	8

MultiIndex

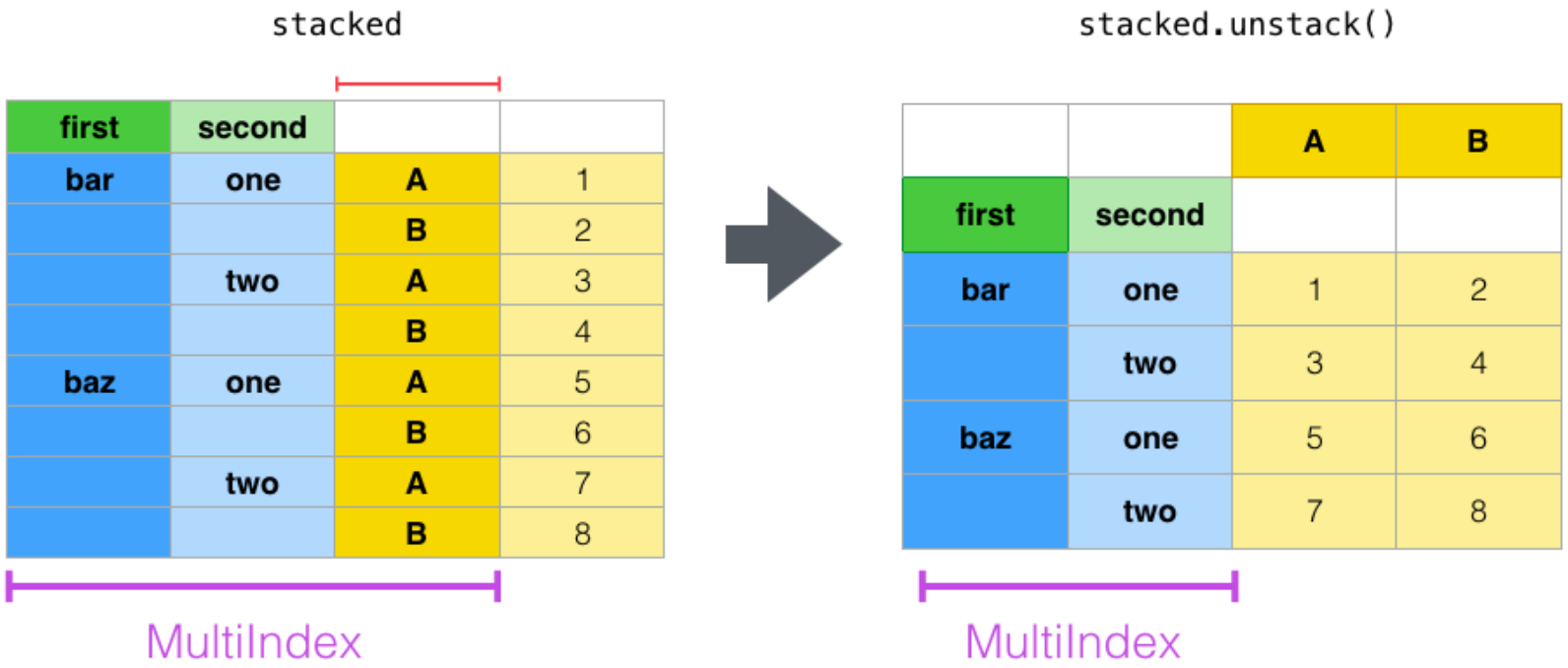
```
stacked = df2.stack()
```

first	second		
bar	one	A	1
		B	2
	two	A	3
		B	4
baz	one	A	5
		B	6
	two	A	7
		B	8

MultiIndex

- ✓ **Index → columns**
- ✓ **Long → Wide**

Unstack



- ✓ **Wide → Long**
- ✓ **Melt Vs Stack**
  - **melt():** Unpivots columns into rows using column labels
  - **stack():** Rotates columns into an inner index level using the index hierarchy

Melt

df3

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

```
df3.melt(id_vars=['first', 'last'])
```

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

- ✓ **Reshape data**
- ✓ **Long → Wide**

Pivot

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

Stacked

```
df.pivot(index='foo',
         columns='bar',
         values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

Record

- ✓ **Reshape + summarize**
- ✓ **Long → Wide**

```
pivot_tbl = pd.pivot_table(
    melted,
    index='key',
    columns='variable',
    values='value',
    aggfunc='mean'
)
```

Create a Pivot Table in Pandas

DATA

	A	B	C	D	E
0	foo	one	small	1	2
1	foo	one	large	2	4
2	foo	one	large	2	5
3	foo	two	small	3	5
4	foo	two	small	3	6
5	bar	one	large	4	6
6	bar	one	small	5	8
7	bar	two	small	6	9
8	bar	two	large	7	9

pivot table sum
multiple columns

	C	large	small
A	B		
bar	one	4	5
	two	7	6
foo	one	4	1
	two	NaN	6

```
pd.pivot_table(df,
               values='D',
               index=['A', 'B'],
               columns=['C'],
               aggfunc=sum)
```

multiple aggfunc
count, min, max
mean

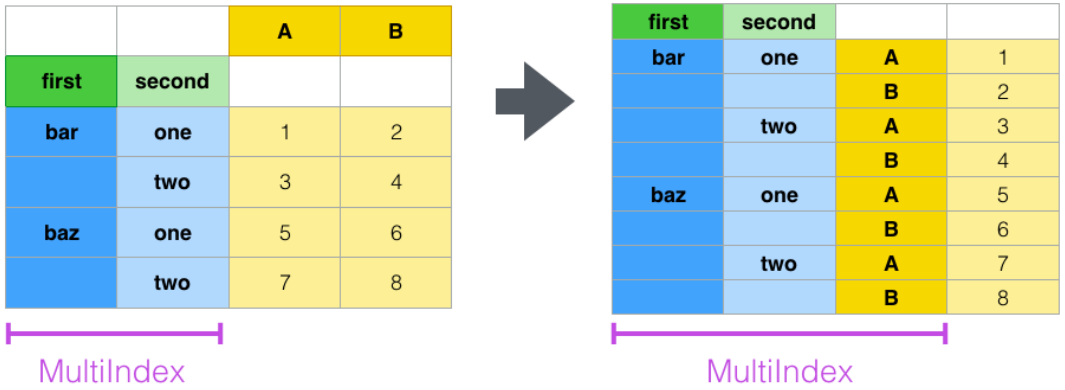
		D	E	
		mean	max	min
A	C			
bar	large	5.5	9	6
	small	5.5	9	8
foo	large	2	5	4
	small	2.333	6	2

```
pd.pivot_table(df,
               values=['D', 'E'],
               index=['A', 'C'],
               aggfunc={'D': 'mean',
                        'E': [min, max]})
```

Advanced Pandas operations (Contd..)

✓ Hierarchical Indexing for Reshaping

- Pandas uses **Hierarchical (MultiIndex) indexing** for flexible reshaping
- Enables data to be rearranged across:
  - Rows
  - Columns
  - Multiple levels of indexing
- Two key operations:
  - **stack()** → columns → rows
  - **unstack()** → rows → columns




```
In [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   .....:                     index=pd.Index(['Ohio', 'Colorado'], name='state'),
   .....:                     columns=pd.Index(['one', 'two', 'three'],
   .....:                     name='number'))

In [121]: data
Out[121]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Advanced Pandas operations (Contd..)

- Using the stack method on this data pivots the columns into the rows, producing a Series:

```
In [122]: result = data.stack()

In [123]: result
Out[123]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

- From a hierarchically indexed Series, you can rearrange the data back into a DataFrame with unstack:

```
In [124]: result.unstack()
Out[124]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

- By default the ***innermost level*** is unstacked (same with stack). You can unstack a different level by passing a level number or name:

```
In [125]: result.unstack(0)
Out[125]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5

In [126]: result.unstack('state')
Out[126]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

Advanced Pandas operations (Contd..)

✓ `melt()`: Wide → Long

- **melt() converts wide-format data into long-format data** by turning column names into row values.
- The **'key'** column may be a group indicator, and the other columns are data values.

```
pd.melt(df, id_vars=['key'],
var_name='variable', value_name='value')
```

```
In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
   .....:                    'A': [1, 2, 3],
   .....:                    'B': [4, 5, 6],
   .....:                    'C': [7, 8, 9]})

In [158]: df
Out[158]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

```
In [159]: melted = pd.melt(df, ['key'])

In [160]: melted
Out[160]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

- You can also specify a subset of columns to use as value columns:

```
In [164]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[164]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

Advanced Pandas operations (Contd..)

✓ Pivot

- Using **pivot**, we can reshape back to the original layout:

```
In [159]: melted = pd.melt(df, ['key'])

In [160]: melted
Out[160]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

- Since the result of pivot creates an index from the column used as the row labels, we may want to use reset_index to move the data back into a column:

```
In [161]: reshaped = melted.pivot('key', 'variable', 'value')

In [162]: reshaped
Out[162]:
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

```
pivoted = melted.pivot(index='key',
columns='variable', values='value')
```

```
In [163]: reshaped.reset_index()
Out[163]:
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

Handling Missing Data

- Missing data is common in real-world data analysis applications
- Pandas aims to make missing data handling simple and efficient
- By default, descriptive statistics in pandas **ignore missing values**
- For numeric data, pandas represents missing values using **NaN (Not a Number)**
- **NaN** acts as a sentinel value that can be easily detected and processed.

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

- In pandas, we adopt a convention used in the R programming language by referring to missing data as **NA**, which stands for **not available.**
- In statistics applications, NA data may either be data that does not exist or that exists but was not observed (through problems with data collection, for example).

Handling Missing Data (Contd..)

- When cleaning up data for analysis, it is often important to do analysis on the missing data itself to identify data collection problems or potential biases in the data caused by missing data.
- The built-in Python **None** value is also treated as NA in object arrays:

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0     True
1    False
2     True
3    False
dtype: bool
```

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

Table: NA handling methods

Handling Missing Data (Contd..)

- ✓ **Filtering Out Missing Data**
  - Missing data can be filtered using **pandas.isnull()** and boolean indexing.
  - **dropna()** removes missing values from a Series, keeping only valid data.
- ✓ On a Series, it returns the Series with only the non null data and index values:

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

- ✓ If you want to drop rows or columns that are all NA or only those containing any NAs. dropna by default drops any row containing a missing value:

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

In [22]: cleaned
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

Handling Missing Data (Contd..)

- Passing how='all' will only drop rows that are all NA

```
In [23]: data.dropna(how='all')
Out[23]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

- To drop columns in the same way, pass axis=1:

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN

In [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

Handling Missing Data (Contd..)

- Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the thresh argument:

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA

In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Handling Missing Data (Contd..)

- ✓ **Filling In Missing Data**
  - Instead of removing missing data, we can fill the “holes”.
  - **fillna()** is the main method for filling missing values.
  - Fill with constants, column-specific values, or interpolation.
- ✓ Calling **fillna** with a **constant** replaces missing values **with that value:**

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

- ✓ Calling **fillna** with a **dict**, you can use a different fill value for each column:

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Handling Missing Data (Contd..)

- ✓ **fillna** returns a new object, but you can modify the existing object in place:

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

- ✓ The same interpolation methods available for reindexing can be used with fillna:

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN

In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232

In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

Forward fill (ffill) replaces missing values (`NaN`) with the **last valid observation before it**.

Handling Missing Data (Contd..)

- ✓ With **fillna** you can do lots of other things with a little creativity.

  Missing values can be filled using mean, median, or other statistics.

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on; default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

Table: fillna function arguments

Categorical Data

- ✓ Categorical data represents characteristics, labels, or attributes that can be grouped into distinct categories rather than measured numerically.
  - **Categories:** The unique values in categorical data are called **categories**.
- ✓ **Category Codes:** Each category can be assigned an integer **code** for efficient storage and computation.
  - The process of assigning code to catagores called encoding
  - **Example:** Blood type (A, B, AB, O) mapped to category codes (0, 1, 2, 3).
- ✓ **Types of Categorical Data:**
  - **Nominal:** Categories with no intrinsic order (e.g., blood type, gender),
  - **Ordinal:** Categories with a meaningful order (e.g., education level, rating).
- ✓ **Memory & Performance:** Using category codes reduces memory usage and improves computational performance.

Categorical Data (Contd..)

✓ Categorical Type in pandas

- In pandas, categorical data can be explicitly represented using the **category dtype**, which internally stores:
  - a list of distinct categories
  - integer codes referencing those categories
- Here, **['a', 'b', 'c', 'd']** is an array of Python string objects, which is converted into categorical by calling ***.astype('category')***
- Series with categorical data have special methods accessible via special attribute **.cat**
- Provides access to:
  - ***.cat.codes*** → integer representation of categories
  - *.cat.categories* → list of category labels

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [61]: cat_s = s.astype('category')

In [62]: cat_s
Out[62]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

```
In [63]: cat_s.cat.codes
Out[63]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8

In [64]: cat_s.cat.categories
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
type(s)
cat_s.dtype
```

Categorical Data (Contd..)

- **Modifying Categories**
  - Extend or modify categories using **set_categories**
  - **value_counts()** will return the number of items in the category.

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e']

In [66]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [67]: cat_s2

Out[67]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

```
In [68]: cat_s.value_counts()
Out[68]:
d    2
c    2
b    2
a    2
dtype: int64

In [69]: cat_s2.value_counts()
Out[69]:
d    2
c    2
b    2
a    2
e    0
dtype: int64
```

Categorical Data (Contd..)

- ✓ One hot / dummy endoding

  - one_hot_encoded = pd.get_dummies(data)

Categorical Data (Contd..)

Encoding Categorical Data

- ✓ Machine learning models require numerical input, so categorical data must be transformed for processing.
- ✓ Techniques:
  - **One-Hot Encoding:** Converts categorical values into binary vectors.

    Example: $["\text{Red}", "\text{Green}"] \rightarrow [1, 0]\ and\ [0, 1]$.
  - **Label Encoding (for nominal):** Each distinct category is assigned a **unique integer code**.

    Example: $"\text{Red}" = 1, "\text{Green}" = 2$.
  - **Ordinal Encoding (for ordinal):** Assigns a specific order or rank to categories based on their relative importance or level.

    Example: ["Low", "Medium", "High"] → **"Low" = 1**, **"Medium" = 2**, **"High" = 3**.
    - This encoding method is particularly useful when the categories have an inherent order or hierarchy.

Encoding Categorical Data (Contd..)

- ✓ **1. One-Hot Encoding:** Converts each category into a **binary vector**, ensuring no implicit ordering.

```
import pandas as pd

colors = pd.Series(['Red', 'Green', 'Red'])
one_hot = pd.get_dummies(colors)
```

- ✓ **2. Label Encoding (Nominal Data):** Each distinct category is assigned a **unique integer code**. The numbers act as **identifiers**, not quantities.

```
colors = pd.Series(['Red', 'Green', 'Red'])
encoded = colors.astype('category').cat.codes
```

- ✓ **3. Ordinal Encoding (Ordinal Data):** Assigns numeric codes based on a **predefined order or hierarchy**.

```
levels = pd.Series(['Low', 'High', 'Medium', 'Low'])

order = {'Low': 1, 'Medium': 2, 'High': 3}
ordinal_encoded = levels.map(order)
```

Encoding Categorical Data (Contd..)

✓ Create a pandas DataFrame with four categorical columns (**Gender, Response, City, Status**).

- Encode **Gender** and **Response** using **label encoding**, and encode **City** and **Status** using **one-hot encoding**, using **pandas only**.

```
df = pd.DataFrame({
    'Gender': ['Male', 'Female', 'Male', 'Female'],
    'Response': ['Yes', 'No', 'Maybe', 'Yes'],
    'City': ['KT', 'PK', 'LT', 'KT'],
    'Status': ['Single', 'Married', 'Single', 'Divorced']
})
```

```
label_cols = ['Gender', 'Response']
onehot_cols = ['City', 'Status']

df[label_cols] = df[label_cols].apply(
    lambda c: c.astype('category').cat.codes
)

df_encoded = pd.get_dummies(df, columns=onehot_cols)
```

✓ Time Series Data

- Data observed or measured at multiple points in time
- Widely used in fields like finance, economics, ecology, neuroscience, and physics
- Can be:
  - **Fixed frequency:** regular intervals (e.g., every 15 seconds, 5 minutes, monthly)
  - **Irregular**: no fixed time interval
- Common ways to represent time series:
  - **Timestamps:** exact points in time
  - **Fixed periods:** defined spans (e.g., January 2007, year 2010)
  - **Time intervals:** start and end timestamps
  - **Elapsed /experimental time:** time measured relative to a starting point

Time-Series Data (Contd..)

✓ Date and Time Data Types and Tools

- The Python standard library includes **data types** for date and time data, as well as calendar-related functionality.
- **The datetime, time,** and **calendar** modules are the main places to start. The **datetime.datetime** type, or simply **datetime**, is widely used:

```
In [10]: from datetime import datetime

In [11]: now = datetime.now()

In [12]: now
Out[12]: datetime.datetime(2017, 9, 25, 14, 5, 52, 72973)

In [13]: now.year, now.month, now.day
Out[13]: (2017, 9, 25)
```

- **datetime** stores both the date and time down to the microsecond. **timedelta** represents the temporal difference between two **datetime** objects:

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [15]: delta
Out[15]: datetime.timedelta(926, 56700)

In [16]: delta.days
Out[16]: 926

In [17]: delta.seconds
Out[17]: 56700
```

Time-Series Data (Contd..)

- You can add (or subtract) a **timedelta** or multiple thereof to a **datetime** object to yield a new shifted object:

```
In [18]: from datetime import timedelta

In [19]: start = datetime(2011, 1, 7)

In [20]: start + timedelta(12)
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)

In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

Type	Description
`date`	Store calendar date (year, month, day) using the Gregorian calendar
`time`	Store time of day as hours, minutes, seconds, and microseconds
`datetime`	Stores both date and time
`timedelta`	Represents the difference between two `datetime` values (as days, seconds, and microseconds)
`tzinfo`	Base type for storing time zone information

Table: Types in datetime module

Time-Series Data (Contd..)

✓ Converting Between String and Datetime

- **Datetime → String**
  - Datetime objects can be converted to strings using:
    - **str(datetime_obj)** → default format
    - **strftime()** → custom format

```
In [22]: stamp = datetime(2011, 1, 3)

In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'

In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

- **String → Datetime (Known Format)**
  - Convert strings to datetime using **datetime.strptime**
  - Requires explicit format specification

```
In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[28]:
[datetime.datetime(2011, 7, 6, 0, 0),
 datetime.datetime(2011, 8, 6, 0, 0)]
```

Time-Series Data (Contd..)

Table: Datetime format specification (ISO C89 compatible)

Type	Description
`%Y`	Four-digit year
`%y`	Two-digit year
`%m`	Two-digit month [01, 12]
`%d`	Two-digit day [01, 31]
`%H`	Hour (24-hour clock) [00, 23]
`%I`	Hour (12-hour clock) [01, 12]
`%M`	Two-digit minute [00, 59]
`%S`	Second [00, 61] (seconds 60, 61 account for leap seconds)
`%w`	Weekday as integer [0 (Sunday), 6]
`%U`	Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0"
`%W`	Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are "week 0"
`%z`	UTC time zone offset as `+HHMM` or `-HHMM`; empty if time zone naive
`%F`	Shortcut for `%Y-%m-%d` (e.g., `2012-4-18`)
`%D`	Shortcut for `%m/%d/%y` (e.g., `04/18/12`)

- **Flexible Parsing with dateutil**
  - **dateutil.parser.parse** automatically infers date formats
  - Useful for human-readable or mixed formats
  - Supports international formats using **dayfirst=True**
  - **CAUTION:**
    - dateutil.parser is a useful but imperfect tool. Notably, it will recognize some strings as dates that you might prefer that it didn't — for example, '42' will be parsed as the year 2042 with today's calendar date.

```
In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

Time-Series Data (Contd..)

- **pandas.to_datetime**
  - Designed for converting **arrays or Series** of dates
  - Fast for standard formats (ISO 8601)

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'], dtype='dat
etime64[ns]', freq=None)
```

  - Automatically handles missing values (**None**, empty strings)

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'], dty
pe='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isnull(idx)
Out[38]: array([False, False,  True], dtype=bool)
```

  - Missing datetime values are represented as **NaT (Not a Time)**

Time-Series Data (Contd..)

✓ Time Series in pandas

- A basic time series in pandas is a **Series indexed by timestamps**
- Timestamps can be Python **datetime** objects or strings

```
In [39]: from datetime import datetime

In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
   ....:          datetime(2011, 1, 7), datetime(2011, 1, 8),
   ....:          datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [41]: ts = pd.Series(np.random.randn(6), index=dates)

In [42]: ts
Out[42]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

✓ DatetimeIndex

- pandas automatically converts datetime indexes into a **DatetimeIndex**
- Internally stored using NumPy's **datetime64[ns]** data type

```
In [43]: ts.index
Out[43]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

✓ Timestamp Objects

- Individual elements of a **DatetimeIndex** are **Timestamp** objects

```
In [46]: stamp = ts.index[0]

In [47]: stamp
Out[47]: Timestamp('2011-01-02 00:00:00')
```

- Timestamp is pandas' **enhanced** version of datetime which **supports time-based manipulations, time zone handling and Frequency awareness**

Time-Series Data (Contd..)

✓ Indexing, Selection, and Subsetting in Time Series

- **Indexing Time Series**
  - Time series can be indexed like a regular pandas Series
  - Access values using:
    - Exact **timestamps**
    - **Datetime strings** interpreted as dates
- **String-Based Date Selection**
  - pandas automatically interprets date strings
- **Slicing with datetime objects**
  - Access values using datetime object as an index
- **Truncating Time Series**
  - truncate() provides a method-based alternative to slicing
  - Used to select data **before or after** a specific date

```
In [48]: stamp = ts.index[2]

In [49]: ts[stamp]
Out[49]: -0.51943871505673811
```

```
In [50]: ts['1/10/2011']
Out[50]: 1.9657805725027142

In [51]: ts['20110110']
Out[51]: 1.9657805725027142
```

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
dtype: float64
```

Time-Series Data (Contd..)

✓ Slicing with Date Ranges

- Range slicing works even if the exact dates are **not present** in the index
- Supports:
  - **Year-only selection** ('2001')
  - **Year–month selection** ('2001-05')
- Enables easy slicing of long time series without explicit date ranges

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),
   ....:                       index=pd.date_range('1/1/2000', periods=1000))

In [53]: longer_ts
Out[53]:
2000-01-01    0.092908
2000-01-02    0.281746
2000-01-03    0.769023
2000-01-04    1.246435
2000-01-05    1.007189
2000-01-06   -1.296221
2000-01-07    0.274992
2000-01-08    0.228913
2000-01-09    1.352917
2000-01-10    0.886429
                ...
2002-09-17   -0.139298
2002-09-18   -1.159926
2002-09-19    0.618965
2002-09-20    1.373890
2002-09-21   -0.983505
2002-09-22    0.930944
2002-09-23   -0.811676
2002-09-24   -1.830156
2002-09-25   -0.138730
2002-09-26    0.334088
Freq: D, Length: 1000, dtype: float64

In [54]: longer_ts['2001']
Out[54]:

2001-01-01    1.599534
2001-01-02    0.474071
2001-01-03    0.151326
2001-01-04   -0.542173
2001-01-05   -0.475496
2001-01-06    0.106403
2001-01-07   -1.308228
2001-01-08    2.173185
2001-01-09    0.564561
2001-01-10   -0.190481
                ...
2001-12-22    0.000369
2001-12-23    0.900885
2001-12-24   -0.454869
2001-12-25   -0.864547
2001-12-26    1.129120
2001-12-27    0.057874
2001-12-28   -0.433739
2001-12-29    0.092698
2001-12-30   -1.397820
2001-12-31    1.457823
Freq: D, Length: 365, dtype: float64
```

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01   -0.622547
2001-05-02    0.936289
2001-05-03    0.750018
2001-05-04   -0.056715
2001-05-05    2.300675
2001-05-06    0.569497
2001-05-07    1.489410
2001-05-08    1.264250
2001-05-09   -0.761837
2001-05-10   -0.331617
                ...
2001-05-22    0.503699
2001-05-23   -1.387874
2001-05-24    0.204851
2001-05-25    0.603705
```

Time-Series Data (Contd..)

✓ Time Series with Duplicate Indices

- Time series data may contain **multiple observations for the same timestamp**
- Pandas allows **non-unique DatetimeIndex**
- Use **index.is_unique** to check whether timestamps are duplicated
- Indexing behavior:
  - **Single value** returned for unique timestamps
  - **Series (slice)** returned for duplicated timestamps
- Duplicate timestamps can **be aggregated using groupby(level=0)**
  - Common aggregations: **mean(), count(), sum()**

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000'
   ....:                           '1/2/2000', '1/3/2000'])

In [64]: dup_ts = pd.Series(np.arange(5), index=dates)

In [65]: dup_ts
Out[65]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```

```
In [66]: dup_ts.index.is_unique
Out[66]: False
```

```
In [67]: dup_ts['1/3/2000']  # not duplicated
Out[67]: 4

In [68]: dup_ts['1/2/2000']  # duplicated
Out[68]:
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int64
```

```
In [69]: grouped = dup_ts.groupby(level=0)

In [70]: grouped.mean()
Out[70]:
2000-01-01    0
2000-01-02    2
2000-01-03    4
dtype: int64

In [71]: grouped.count()
Out[71]:
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

Time-Series Data (Contd..)

✓ Date Ranges, Frequencies, and Shifting

- Time series can be **irregular** or **fixed-frequency**
- Fixed-frequency data simplifies analysis (daily, monthly, hourly, etc.)
- Pandas provides tools for **resampling, frequency inference, and date generation**
- **Resampling to Fixed Frequency**
  - **resample()** converts irregular time series to a fixed frequency
  - Frequency specified using string aliases (e.g., 'D' for daily)
  - Missing timestamps may introduce **NaN values**

```
In [72]: ts
Out[72]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [73]: resampler = ts.resample('D')
```

Time-Series Data (Contd..)

- **Generating Date Ranges**
  - Pandas provides **pd.date_range()** to generate **DatetimeIndex**
  - Used to create **fixed-frequency time indices**

```
In [74]: index = pd.date_range('2012-04-01', '2012-06-01')

In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

- By default, date_range generates daily timestamps. If you pass only a start or end date, you must pass a number of periods to generate:

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')

In [77]: pd.date_range(end='2012-06-01', periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',
               '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
               '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
               '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
               '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

Time-Series Data (Contd..)

✓ Shifting (Leading and Lagging) Data

- **Shifting** means moving data values backward or forward in time
- Commonly used in **time series analysis**
- Helps compare current values with **past (lag)** or **future (lead)** values
- Implemented in pandas using the **shift()** method
- **Naive Shifting of Data (shift)**
  - shift(n) moves the **data values**, not the index
  - Positive shift (shift(2)):
    - Moves data **downward (lag)**
    - Introduces NaN at the beginning
  - Negative shift (shift(-2)):
    - Moves data **upward (lead)**
    - Introduces NaN at the end

```
In [91]: ts = pd.Series(np.random.randn(4),
   ....:                index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [92]: ts
Out[92]:
2000-01-31   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31         NaN
2000-02-29         NaN
2000-03-31   -0.066748
2000-04-30    0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31   -0.117388
2000-02-29   -0.517795
2000-03-31         NaN
2000-04-30         NaN
Freq: M, dtype: float64
```

3.3 Feature transformation, scaling, and encoding

- ✓ **Feature transformation, scaling, and encoding** are data preprocessing techniques that:
  - convert input features from one mathematical form, numerical range, or data type into another in order to make them suitable for machine learning algorithms.
  - These operations modify the *representation* of data—such as its **distribution, scale, or format**—without altering the underlying information content.
- ✓ **Feature transformation** converts a numerical feature from one functional form or distribution to another.
- ✓ **Feature scaling** converts numerical features from one numerical range to another comparable range.
- ✓ **Feature encoding** converts categorical features from non-numerical labels to numerical representations.

3.3 Feature Transformation

✓ Logarithmic Transformation $\boxed{x' = \log(x+1)}$

- Transforms data from linear scale to logarithmic scale (distribution change)
- **Purpose**: Use for highly *skewed data* to compress large values and reduce skewness. Common in financial and population datasets.

```
df = pd.DataFrame({
    "income": [20000, 40000, 80000, 160000],
    "age": [25, 30, 45, 60],
    "gender": ["Male", "Female", "Female", "Male"]
})
```

```
# 1. Log Transformation (x -> log(x+1))
df["income_log"] = df["income"].apply(lambda x: np.log(x+1))
df["income_log_lib"] = np.log(df["income"] + 1)
```

3.3 Feature Transformation

✓ Square-Root Transformation

- **Purpose:** Reduce right skew by compressing large values while preserving order.
- **Effect:** Maps values from the original scale → compressed scale ($x' = \sqrt{x}$)

```
df["income_sqrt_manual"] = df["income"].apply(lambda x: x ** 0.5)
df["income_sqrt"] = np.sqrt(df["income"])
```

✓ Power Transformation (Yeo–Johnson)

- **Purpose:** Reduce skew and stabilize variance to approximate a normal distribution.
- **Effect:** skewed distribution → approximately normal.

$$x' = \begin{cases} \frac{(x+1)^{\lambda}-1}{\lambda}, & \lambda \neq 0 \\ \log(x+1), & \lambda = 0 \end{cases}$$

```
df["income_power"] = df["income"].apply(lambda x: (x+1)**0.5)
from sklearn.preprocessing import PowerTransformer
df["income_power_lib"] = PowerTransformer(method="yeo-johnson").fit_transform(df[["income"]])
```

3.3 Feature Transformation

✓ 1. Min-Max Normalization

- **Purpose**: Use when the data has no outliers, and you want to scale it to a range of $[-1, 1]$ for algorithms sensitive to scale, such as k-Nearest Neighbors (k-NN) and Neural Networks.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

```
# 4. Min-Max Scaling
df["age_minmax"] = (df["age"] - df["age"].min()) / (df["age"].max() - df["age"].min())
scaler_minmax = MinMaxScaler()
df["age_minmax_lib"] = scaler_minmax.fit_transform(df[["age"]])
```

✓ Z-Score Normalization (Standardization)

$$z = \frac{x - \mu}{\sigma}$$

- **Purpose**: Use when the data has *outliers* or when features have *different units*. Suitable for algorithms like SVM and logistic regression.

```
df["age_standard"] = (df["age"] - df["age"].mean()) / df["age"].std()
scaler_std = StandardScaler()
df["age_standard_lib"] = scaler_std.fit_transform(df[["age"]])
```

3.3 Feature transformation, scaling, and encoding

✓ Robust Scaling

$$x' = \frac{x - \text{median}(x)}{\text{IQR}}$$

- **Purpose**: Use for data with significant outliers to scale using robust statistics like median and IQR. Effective for algorithms sensitive to outliers, such as clustering.

```
q1, q3 = df["age"].quantile([0.25, 0.75])
df["age_robust"] = (df["age"] - df["age"].median()) / (q3 - q1)
scaler_robust = RobustScaler()
df["age_robust_lib"] = scaler_robust.fit_transform(df[["age"]])
```

✓ Binning

- **Purpose:** Use for converting continuous variables into categorical intervals to simplify the data and reduce noise. Useful in regression and classification tasks.
- Example: Ages $0–18 \rightarrow$ "Child", $19–35 \rightarrow$ "Young Adult", $36–60 \rightarrow$ "Adult".

```
k = 3
width = (df["age"].max() - df["age"].min()) / k
df["age_bin"] = ((df["age"] - df["age"].min()) / width).astype(int)
df["age_bin_lib"] = pd.cut(df["age"], bins=k, labels=False)
```

3.3 Feature transformation, scaling, and encoding

✓ One-Hot Encoding

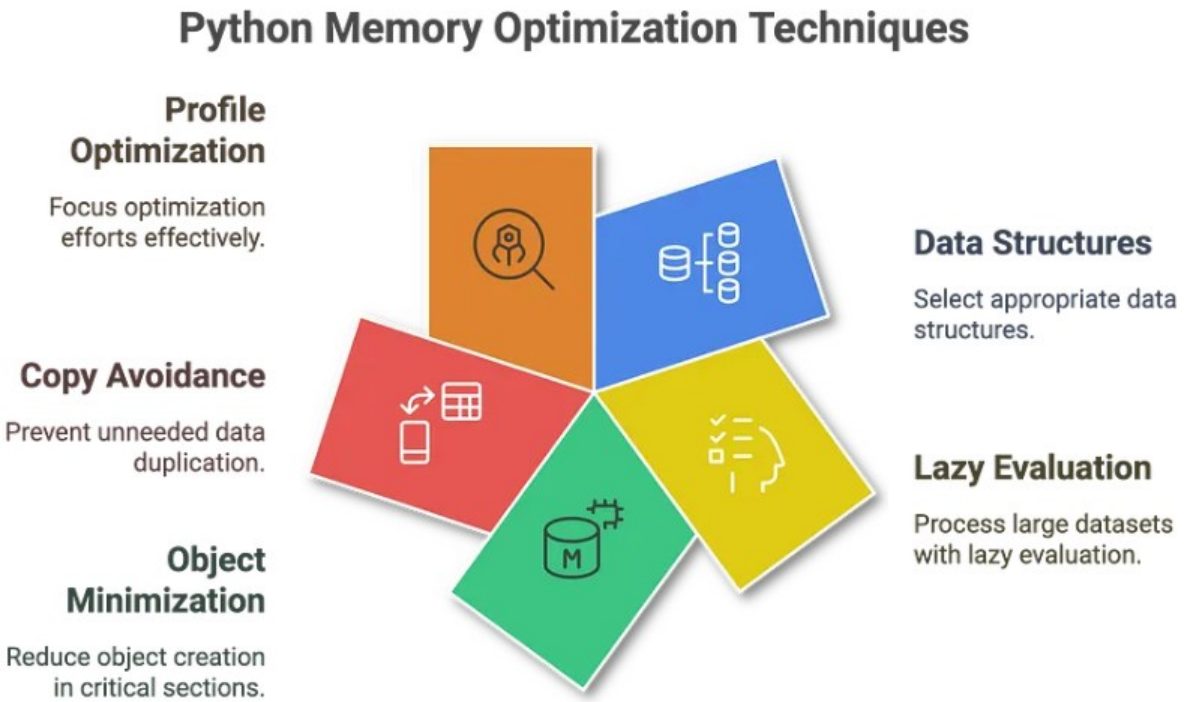
- **Purpose:** Use for categorical variables to convert them into binary vectors for algorithms that require numeric input, such as decision trees or deep learning models.

```
df["gender_Male"] = (df["gender"] == "Male").astype(int)
df["gender_Female"] = (df["gender"] == "Female").astype(int)
ohe = OneHotEncoder(sparse=False)
df[ohe.get_feature_names_out(["gender"])] = ohe.fit_transform(df[["gender"]])
```

Memory Optimization & Efficient Data Processing

- ✓ Handling large datasets efficiently is critical for high-performance Python applications
- ✓ Poor memory usage can lead to slow execution, crashes, or scalability issues
- ✓ Python provides built-in features and libraries to optimize memory consumption
- ✓ Focuses on processing data efficiently **without loading everything into memory at once**
- ✓ Essential for data-intensive tasks like data analysis, machine learning, and backend systems

Memory Optimization & Efficient Data Processing (Contd..)



Pillars of Memory-Efficient Python Programming

Memory Optimization & Efficient Data Processing (Contd..)

✓ Use Built-in Functions & Libraries

- Python's built-in functions are written in C and are much faster than equivalent Python code.

✓ Built-ins like **sum(), map(), filter(), and itertools** are optimized at the C level, offering better performance over manual Python implementations.

- Built-ins are optimized in C and faster than Python loops performs vectorized operations.

✓ Choosing Appropriate Data Structures

- Use **lists** instead of dictionaries when key–value access is unnecessary
- Prefer **tuples** over lists for immutable data
- Use **sets** for fast membership testing

✓ Use generators Instead of Lists

- Lists store all values in memory
- Generators produce values **one at a time**

```
# Slow
total = 0
for x in data:
    total += x

# Fast
total = sum(data)
```

Memory Optimization & Efficient Data Processing (Contd..)

✓ Efficient Use of NumPy and Pandas

- NumPy arrays use **contiguous memory**, much smaller than Python lists
- Pandas allows **dtype optimization** which reduces memory footprint significantly

```
df['age'] = df['age'].astype('int8')
```

✓ Chunk-Based Data Processing

- Process data in **small chunks** instead of loading everything at once

```
for chunk in pd.read_csv("data.csv", chunksize=10000):
    process(chunk)
```

✓ Avoid Unnecessary Copies

```
df.dropna(inplace=True)
```

- Assigning large objects creates references, but operations may create copies
- Use **in-place operations** when possible

✓ Avoid Global Variables

- Accessing global variables is slower than accessing local ones, so keep variables local within functions whenever possible.

✓ Optimize Loops with List Comprehensions & Generators

a) List Comprehensions

- List comprehensions are typically faster than **for** loops because they are optimized in bytecode.

```
squares = [x**2 for x in range(1000)]  # Faster
```

b) Generators (Memory Efficiency)

- Use **yield** for lazy evaluation, which prevents loading large datasets into memory.

```
def generate_numbers(n):
    for i in range(n):
        yield i  # Memory-efficient
```

```
numbers = generate_numbers(1_000_000)  # Doesn't store all numbers at
once
```

✓ Memoization & Caching

Memoization is a specific form of caching that optimizes code by storing the results of expensive function calls, returning the cached result when the same inputs occur again.

a) functools.lru_cache

- Memoization with **lru_cache** can significantly speed up recursive functions by caching previously computed results.

```python
from functools import lru_cache

@lru_cache(maxsize=128)  # Caches up to 128 calls
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(100))  # Much faster due to caching
```

```
fib(5)
├─ fib(4)
│  ├─ fib(3)
│  │  ├─ fib(2)
│  │  ├─ fib(1)
│  ├─ fib(2)
├─ fib(3)
│  ├─ fib(2)
│  ├─ fib(1)
```

b) Manual Caching

- For custom caching logic, you can maintain your own cache dictionary.

```python
cache = {}
def expensive_function(x):
    if x not in cache:
        cache[x] = x ** 2  # Compute once, reuse later
    return cache[x]
```

✓ String Concatenation Optimization

- String concatenation using **+** in loops can be inefficient, as it creates temporary objects. Instead, use **.join()** for more efficient concatenation.

✅ **Fast:**

```
words = ["Hello", "World"]
sentence = " ".join(words)  # Efficient
```

❌ **Slow:**

```
sentence = ""
for word in words:
    sentence += word + " "  # Creates new strings each time
```

✓ Profiling & Identifying Bottlenecks

- Identifying bottlenecks is crucial before diving into optimization. Use Python's profiling tools to measure performance.

a) timeit for Microbenchmarks

```
def f():
    return sum(range(1000))

timeit.timeit(f, number=10000)
```

```
loop_time = timeit.timeit(
    'total = 0\nfor i in range(1000): total += i',
    number=10000
)

builtin_time = timeit.timeit(
    'sum(range(1000))',
    number=10000
)
```

```
timeit.repeat('sum(range(1000))', number=10000, repeat=5)
```

b) cProfile for Full Code Analysis

```
import cProfile
def slow_function():
    return sum(i**2 for i in range(1_000_000))
cProfile.run('slow_function()')  # Shows time per function call
```

Memory Optimization & Efficient Data Processing (Contd..)

✓ Use NumPy & Pandas for Numerical Work

- For numerical operations, **NumPy** and **Pandas** are far more efficient than Python's native data structures.

✓ **NumPy (Optimized Arrays)** is efficient than Python list:

- NumPy arrays store data in **contiguous blocks of memory**
- NumPy arrays contain **only one data type**
- Vectorized Operations (No Python Loops)'
- Execution in Compiled C / Fortran (BLAS, LAPACK)

```
list_code = """
data = list(range(100000))
result = sum(x*x for x in data)
"""

list_time = timeit.timeit(list_code, number=10)
print("Python list time:", list_time)
```

```
numpy_code = """
import numpy as np
data = np.arange(100000)
result = np.sum(data * data)
"""

numpy_time = timeit.timeit(numpy_code, number=10)
print("NumPy array time:", numpy_time)
```

Building a reusable data-cleaning pipeline

- ✓ Data preparation is a crucial step in data science and machine learning workflows
- ✓ Common preprocessing tasks include:
  - Handling missing values
  - Data normalization and scaling
- ✓ Performing these steps manually can be repetitive and error-prone
- ✓ **Reusable data-cleaning pipelines** provide a structured and efficient way to automate preprocessing
- ✓ Python's **scikit-learn Pipeline** helps organize and standardize data cleaning steps

Introduction to data pipeline components

✓ What is a Data Pipeline?

- A data pipeline is a structured system that **collects, cleans, processes/transforms, and stores data** from multiple sources.
- It ensures data flows **reliably, consistently, and efficiently** from raw input to analytics or machine learning systems.
- **Core Components**
  - **Data Ingestion** – bringing data into the pipeline
  - **Data Transformation** – cleaning and structuring data
  - **Data Storage** – persisting data for analysis and usage

Figure: Key Components of a data pipeline components

Building a reusable data-cleaning pipeline (Contd..)

✓ Importance of Reusable Data-Cleaning Pipelines

- Ensures **consistent and repeatable** data transformations
- Reduces preprocessing errors by applying the same steps every time
- Improves **code maintainability and readability**
- Particularly useful when:
  - Datasets are frequently updated
  - The same preprocessing logic is reused across multiple projects
- Makes preprocessing workflows easier to scale and manage

✓ Example Pipeline

- **Raw Data → Validation → Cleaning → Transformation → Output**

Building a reusable data-cleaning pipeline (Contd..)

✓ Example

- **scikit-learn** provides tools to build preprocessing pipelines
- Sample dataset contains:
  - Numerical features (Age, Salary)
  - Missing values that need to be handled
- The example demonstrates **a realistic preprocessing scenario** involving missing data and normalization

```
import pandas as pd
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Sample Data
data = {'Age': [25, 30, None, 22, 29],
        'Salary': [50000, 54000, None, 62000, 60000]}

df = pd.DataFrame(data)
```

Building a reusable data-cleaning pipeline (Contd..)

✓ Example Data-Cleaning Pipeline

- The pipeline consists of **sequential preprocessing steps**
- **SimpleImputer**
  - Replaces missing values using the **median**
  - Median is chosen to **reduce the effect of outliers**
  - Ensures the dataset is **complete before model training**
- **StandardScaler**
  - Normalizes data
  - Scales features to **unit variance**
  - Helps algorithms **converge faster and perform better**
- Each step processes the output of the previous step automatically

```
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),  # Handle missing values
    ('scaler', StandardScaler())  # Normalize data
])
```

Building a reusable data-cleaning pipeline (Contd..)

✓ Applying the Pipeline and Output

- The **fit_transform()** method:
  - Learns parameters from the data
  - Applies all preprocessing steps in order
- Missing values are filled before scaling is applied

```
df_transformed = pipeline.fit_transform(df)
print(df_transformed)
```

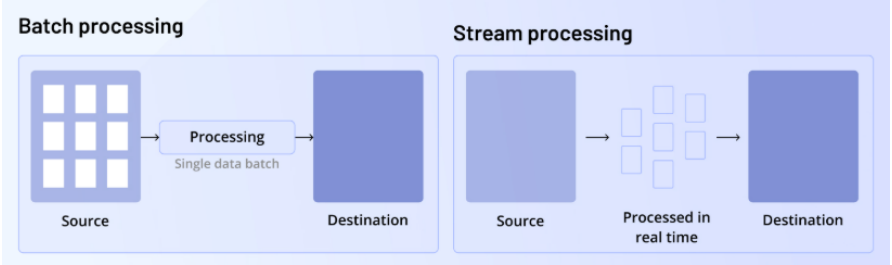
- Final output is a **clean, normalized dataset**
- The transformed data is now **ready for machine learning models**, which typically perform better on scaled inputs.
- Output:

```
[[-0.55738641 -1.54536659]
 [ 1.18444612 -0.60878078]
 [ 0.1393466   0.09365858]
 [-1.60248593  1.26439085]
 [ 0.83607962  0.79609794]]
```

Introduction to data pipeline components (Contd..)

Data Pipeline Components

- ✓ **Data Ingestion** is the process of **moving data from source systems into the pipeline**.
  - **Common Data Sources**
    - Relational & NoSQL databases, APIs and third-party platforms, File systems (CSV, JSON, XML, Parquet), Data warehouses and operational systems
  - **Ingestion Approaches**



    - Batch processing
      - Data processed in chunks at scheduled intervals
      - Suitable for non-real-time use cases (e.g., reporting, audits)
    - Stream processing
      - Data processed in real time as it is generated
      - Used in finance, monitoring, recommendation systems

Data Pipeline Components (Cont..)

✓ Data Transformation

- Converts raw, heterogeneous data into a **clean, consistent, and unified format**.

Key Transformation Activities

- Data cleaning (handling missing values, outliers, duplicates)
- Normalization and standardization
- Encoding categorical variables
- Schema alignment across sources
- Feature engineering and aggregation

Why Transformation Matters

- Ensures data is **analysis-ready**
- Critical for **machine learning accuracy and reliability**
- Enables cross-system querying and analytics

Data Pipeline Components (Cont..)

✓ Data Storage

- Stores transformed data for **analytics, reporting, and ML workloads**.
- **Storage Systems**
  - OLTP databases
    - Optimized for transactional workloads
    - High consistency, fast writes
  - Data Warehouses
    - Structured, curated data for analytics
    - Optimized for querying and reporting
  - Data Lakes
    - Store raw, semi-structured, and unstructured data
    - Flexible schema-on-read approach
  - Data Lakehouse
    - Combines low-cost storage of lakes with ACID guarantees of warehouses
    - Supports both BI and ML workloads
- Storage choice depends on **data type, scale, performance needs, and use case**.

Introduction to data pipeline components (Contd..)

Example – ML Implementation

✓ Problem Statement

- We want to **predict the salary of employees** based on their **age** and **years of experience**. The available data may have **missing values**, so we need a **pipeline** to:
- **Ingest** the data from a source (e.g., CSV, database).
- **Transform** the data by handling missing values and optionally adding derived features.
- **Train a machine learning model** (Linear Regression) to predict salary.
- **Store** the predicted results for future use or analysis.
- This example demonstrates a **simple end-to-end data pipeline**:
  - Ingestion → Transformation → Machine Learning → Storage.

Introduction to data pipeline components (Contd..)

Example – ML Implementation

```
import pandas as pd
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression
```

```
# 1. Data Ingestion
# ------------------------
df = pd.DataFrame({
    'Age': [25, 30, None, 22],
    'Salary': [50000, 60000, 55000, None],
    'YearsExperience': [2, 5, 3, 1]
})
```

```
# Features and target
X = df[['Age', 'YearsExperience']]
y = df['Salary']
```

```
# 2. Pipeline: Transformation + ML
# ------------------------
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('model', LinearRegression())
])
```

```
# Train the model
pipeline.fit(X, y)

# Make predictions
df['Predicted_Salary'] = pipeline.predict(X)
```

```
# 3. Storage / Output
# ------------------------
df.to_csv('predicted_salary.csv', index=False)
```