Pandas

It provides two primary data structures: **Series and DataFrame**, which are designed to make working with structured data both easy and intuitive. Think of a DataFrame as a spreadsheet or a SQL table. Using pandas, we can:

- Arrange data in a tabular format.
- Extract useful information filtered by specific conditions.
- Operate on data to gain new insights.
- Apply NumPy functions to our data.
- Perform vectorized computations to speed up our analysis.

```
import pandas as pd
```

Series

A pandas Series is a one-dimensional labeled array in Python. Think of it as a single column from a spreadsheet or a single list with an attached index. It's a fundamental data structure in pandas and can hold data of any type (e.g., integers, floats, strings, booleans, or Python objects).

A Series is a 1-dimensional array-like object. It contains:

- A sequence of values of the same type.
- A sequence of data labels, called the index.

The Series constructor supports several optional arguments (index, dtype, name, copy, etc.).

Creating a Series

You can create a Series from various data types, like a list, a NumPy array, or a dictionary.

- **From a List:** This is the most common way. The Series will automatically get a default integer index (0, 1, 2, ...).

```
ages = pd.Series([25, 30, 35, 40])

print("Series from a list:")
print(ages)

# Numpy arrays/lists also work
import numpy as np

prices = pd.Series(np.array([20.3, 55.4, 65.2]), dtype='float64') # from numpy arrays, force dtype.
print("\n", prices)
```

```
Series from a list:
0    25
1    30
2    35
3    40
dtype: int64

 0    20.3
1    55.4
2    65.2
dtype: float64
```

- **Creating a Series with Custom Index:** Provide a list of custom index labels for easy data fetching.

```
data = [10, 20, 30, 40]
index = ['a', 'b', 'c', 'd']
series = pd.Series(data, index=index)
print(series)
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

```
1 data = [10, 20, 30, 40]
2 index = ['a', 'b']
3 series = pd.Series(data, index=index)
4 print(series)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2638572274.py in <cell line: 0>()
      1 data = [10, 20, 30, 40]
      2 index = ['a', 'b']
----> 3 series = pd.Series(data, index=index)
      4 print(series)

                                 1 frames
/usr/local/lib/python3.12/dist-packages/pandas/core/common.py in require_length_match(data, index)
    571     """
    572     if len(data) != len(index):
--> 573         raise ValueError(
    574             "Length of values "
    575             f"({len(data)}) "

ValueError: Length of values (4) does not match length of index (2)
```

- **Creating a Series with Scalar Value:** This creates a Series where every index has the same value.

```
1 a = pd.Series(5, index=['x','y','z']) # scalar broadcast to all labels.
2
3 print(a)
```

```
x    5
y    5
z    5
dtype: int64
```

- **From a Dictionary:** When you create a Series from a dictionary, the dictionary's keys become the Series' index, and the values become the data.

```
1 # Create a Series of student grades with names as the index
2 grades_dict = {'Alice': 95, 'Bob': 88, 'Charlie': 92, 'David': 78}
3 grades = pd.Series(grades_dict)
4
5 print("\nSeries from a dictionary:")
6 print(grades)
```

```
Series from a dictionary:
Alice      95
Bob        88
Charlie    92
David      78
dtype: int64
```

Series Attributes

A Series has useful attributes for understanding its structure.

- **.index:** Returns the index labels of the Series.
- **.values:** Returns the data as a NumPy array.
- **.dtype:** Returns the data type of the Series' values.
- **.name:** Returns the name of the Series, which is useful when it becomes a column in a DataFrame.

```
1 print("\nGrades Series Attributes:")
2 print(f"Index: {grades.index}")
3 print(f"Values: {grades.values}")
4 print(f"Data Type: {grades.dtype}")
5 print(f"Size: {grades.size}")
6 print(f"Shape: {grades.shape}")
7 print(f"Dimension: {grades.ndim}")
8
9 print()
10
11 idx_arr = grades.index
12 print(idx_arr)
13 print(idx_arr[1])
14
15 # New series with name property
```

```
prices = pd.Series([2.50, 4.75, 1.99, 5.00], name='Product_Price')
print(prices)

print("The raw values of the Series:")
print(prices.values)
```

```
Grades Series Attributes:
Index: Index(['Alice', 'Bob', 'Charlie', 'David'], dtype='object')
Values: [95 88 92 78]
Data Type: int64
Size: 4
Shape: (4,)
Dimension: 1

Index(['Alice', 'Bob', 'Charlie', 'David'], dtype='object')
Bob
0    2.50
1    4.75
2    1.99
3    5.00
Name: Product_Price, dtype: float64
The raw values of the Series:
[2.5  4.75 1.99 5.  ]
```

```
# Create a sample series
s = pd.Series([10, 20, 30, 40, 50],
              index=['a', 'b', 'c', 'd', 'e'],
              name='my_series')

print("Series:")
print(s)
print() # New line
# Key attributes
print(f"Values: {s.values}")           # Underlying data array
print(f"Index: {s.index}")             # Index object
print(f"Name: {s.name}")               # Series name
print(f"dtype: {s.dtype}")             # Data type
print(f"Shape: {s.shape}")             # Shape (number of elements)
print(f"Size: {s.size}")               # Number of elements
print(f"ndim: {s.ndim}")               # Number of dimensions (always 1)
print(f"Empty: {s.empty}")             # Whether series is empty
```

```
Series:
a    10
b    20
c    30
d    40
e    50
Name: my_series, dtype: int64

Values: [10 20 30 40 50]
Index: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
Name: my_series
dtype: int64
Shape: (5,)
Size: 5
ndim: 1
Empty: False
```

.index: This attribute holds the labels for each data point. By default, it's a RangeIndex (e.g., 0, 1, 2, ...), but it can be any sequence of hashable values, such as strings, dates, or custom identifiers.

```
# Index-specific attributes
print(f"Index values: {s.index.values}")
print(f"Index name: {s.index.name}")    # None if not set
print(f"Index dtype: {s.index.dtype}")

# Set index name
s.index.name = 'letters'
print(f"Index name after setting: {s.index.name}")
```

```
Index values: ['a' 'b' 'c' 'd' 'e']
Index name: None
Index dtype: object
Index name after setting: letters
```

Indexing & Selection

Basic Indexing

```
# Create a sample series
s = pd.Series([10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
              index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'])

print("Original Series:")
print(s)

# Position-based indexing (like Python lists)
print("\ns[0]:", s[0])        # First element: 10
print("\ns[4]:", s[4])        # Fifth element: 50
print("\ns[-1]:", s[-1])      # Last element: 100

# Slicing by position
print("\ns[2:5]:")            # Elements 2 to 4 (exclusive of 5)
print(s[2:5])

print("\ns[:3]:")             # First 3 elements
print(s[:3])

print("\ns[7:]:")             # From position 7 to end
print(s[7:])

print("\ns[::2]:")            # Every second element
print(s[::2])
```

```
Original Series:
a     10
b     20
c     30
d     40
e     50
f     60
g     70
h     80
i     90
j    100
dtype: int64

s[0]: 10

s[4]: 50

s[-1]: 100

s[2:5]:
c    30
d    40
e    50
dtype: int64

s[:3]:
a    10
b    20
c    30
dtype: int64

s[7:]:
h     80
i     90
j    100
dtype: int64

s[::2]:
a    10
c    30
e    50
g    70
i    90
dtype: int64
/tmp/ipython-input-1386273421.py:9: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future vers
  print("\ns[0]:", s[0])        # First element: 10
/tmp/ipython-input-1386273421.py:10: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future ver
  print("\ns[4]:", s[4])        # Fifth element: 50
/tmp/ipython-input-1386273421.py:11: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future ver
  print("\ns[-1]:", s[-1])      # Last element: 100
```

Explicit Indexing Methods Using .loc and .iloc

```
# .loc for label-based indexing
print(s.loc['b'])      # 20
print(s.loc[['a', 'c']])  # a: 10, c: 30

# .iloc for position-based indexing
```

```
print(s.iloc[1])       # 20
print(s.iloc[1:3])     # b: 20, c: 30
```

```
20
a    10
c    30
dtype: int64
20
b    20
c    30
dtype: int64
```

```
# .iloc[] - integer location (position-based)
print("\n.iloc[] examples:")
print("s.iloc[0]:", s.iloc[0])          # First element
print("s.iloc[-1]:", s.iloc[-1])        # Last element
print("\ns.iloc[[1, 3, 5]]:")             # Multiple positions
print(s.iloc[[1, 3, 5]])
print("\ns.iloc[2:6]:")                   # Slice
print(s.iloc[2:6])
print("\ns.iloc[::2]:")                   # Step
print(s.iloc[::2])

# Boolean indexing with .iloc
mask = [True, False, True, False, True, False, True, False, True, False]
print("\nBoolean mask with .iloc:")
print(s.iloc[mask])
```

```
.iloc[] examples:
s.iloc[0]: 10
s.iloc[-1]: 100

s.iloc[[1, 3, 5]]:
b    20
d    40
f    60
dtype: int64

s.iloc[2:6]:
c    30
d    40
e    50
f    60
dtype: int64

s.iloc[::2]:
a    10
c    30
e    50
g    70
i    90
dtype: int64

Boolean mask with .iloc:
a    10
c    30
e    50
g    70
i    90
dtype: int64
```

```
# .loc[] - label location
print(".loc[] examples:")
print("s.loc['a']:", s.loc['a'])        # Single label
print("s.loc[['a', 'c', 'e']]:")        # Multiple labels
print(s.loc[['a', 'c', 'e']])
print("\ns.loc['c':'f']:")                # Label slice (inclusive!)
print(s.loc['c':'f'])
print("\ns.loc['b':'h':2]:")              # Label slice with step
print(s.loc['b':'h':2])
```

```
.loc[] examples:
s.loc['a']: 10
s.loc[['a', 'c', 'e']]:
a    10
c    30
e    50
dtype: int64

s.loc['c':'f']:
c    30
d    40
e    50
f    60
dtype: int64
```

```
s.loc['b':'h':2]:
b    20
d    40
f    60
h    80
dtype: int64
```

```
# Series with duplicate index labels
s_dup = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'a', 'c', 'b'])
print("Series with duplicate indexes:")
print(s_dup)

# Accessing with duplicate labels
print("\ns_dup.loc['a']:")  # Returns all occurrences
print(s_dup.loc['a'])

print("\ns_dup.loc['b']:")
print(s_dup.loc['b'])
```

```
Series with duplicate indexes:
a    1
b    2
a    3
c    4
b    5
dtype: int64

s_dup.loc['a']:
a    1
a    3
dtype: int64

s_dup.loc['b']:
b    2
b    5
dtype: int64
```

```
# Setting values using different indexing methods
s_copy = s.copy()

print("Original series:")
print(s_copy)

# Set by position
s_copy.iloc[2] = 999
print("After setting position 2 to 999:")
print(s_copy)

# Set by label
s_copy.loc['e'] = 888
print("After setting label 'e' to 888:")
print(s_copy)

# Set multiple values
s_copy.iloc[[0, 4, 8]] = [111, 222, 333]
print("After setting multiple positions:")
print(s_copy)

# Set with boolean indexing
s_copy[s_copy > 500] = 0
print("After setting values > 500 to 0:")
print(s_copy)
```

```
Original series:
a     10
b     20
c     30
d     40
e     50
f     60
g     70
h     80
i     90
j    100
dtype: int64
After setting position 2 to 999:
a     10
b     20
c    999
d     40
e     50
f     60
g     70
h     80
```

```
i     90
j    100
dtype: int64
After setting label 'e' to 888:
a     10
b     20
c    999
d     40
e    888
f     60
g     70
h     80
i     90
j    100
dtype: int64
After setting multiple positions:
a    111
b     20
c    999
d     40
e    222
f     60
g     70
h     80
i    333
j    100
dtype: int64
After setting values > 500 to 0:
a    111
b     20
c      0
d     40
e    222
f     60
g     70
h     80
i    333
```

```
# .at[] - fast scalar access by label
print(".at[] for fast scalar access:")
print("s.at['c']:", s.at['c'])      # Much faster than s.loc['c'] for single values

# .iat[] - fast scalar access by position
print(".iat[] for fast scalar access:")
print("s.iat[2]:", s.iat[2])        # Much faster than s.iloc[2] for single values
```

```
.at[] for fast scalar access:
s.at['c']: 30
.iat[] for fast scalar access:
s.iat[2]: 30
```

Mathematical Operations

```
s = pd.Series([1, 2, 3, 4, 5])

# Basic arithmetic
print(s + 10)      # [11, 12, 13, 14, 15]
print(s * 2)       # [2, 4, 6, 8, 10]
print(s ** 2)      # [1, 4, 9, 16, 25]

# Operations between series when their indexes/labels are same
s2 = pd.Series([10, 20, 30, 40, 50])
print(s + s2)      # [11, 22, 33, 44, 55]
```

```
0    11
1    12
2    13
3    14
4    15
dtype: int64
0     2
1     4
2     6
3     8
4    10
dtype: int64
0     1
1     4
2     9
3    16
4    25
dtype: int64
0    11
1    22
2    33
3    44
```

```
4    55
dtype: int64
```

```
1 s = pd.Series([1, 2, 3, 4, 5])
2 s2 = pd.Series([5, 3])
3
4 print(s + s2)
```

```
0    6.0
1    5.0
2    NaN
3    NaN
4    NaN
dtype: float64
```

Statistical Operations

```
1 s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
2 # OR
3 # s = pd.Series(np.arange(1, 11))
4
5 print(s.sum())      # Sum: 55
6 print(s.mean())     # Mean: 5.5
7 print(s.median())   # Median: 5.5
8 print(s.std())      # Standard deviation
9 print(s.min())      # Minimum: 1
10 print(s.max())      # Maximum: 10
11 print(s.describe()) # Summary statistics
```

```
55
5.5
5.5
3.0276503540974917
1
10
count    10.00000
mean      5.50000
std       3.02765
min       1.00000
25%       3.25000
50%       5.50000
75%       7.75000
max      10.00000
dtype: float64
```

Boolean Indexing

```
1 s = pd.Series([10, 20, 30, 40, 50, 60])
2
3 # Filter values greater than 30
4 filtered = s[s > 30]
5 print(filtered)  # 40, 50, 60
6
7 print()
8
9 # Multiple conditions
10 filtered2 = s[(s > 20) & (s < 50)]
11 print(filtered2)  # 30, 40
```

```
3    40
4    50
5    60
dtype: int64

2    30
3    40
dtype: int64
```

```
1 # Boolean indexing - very powerful!
2 print("Boolean indexing examples:")
3
4 # Simple condition
5 print("Values > 50:")
6 print(s[s > 50])
7
8 # Multiple conditions (use &, |, ~ instead of and, or, not)
9 print("\nValues between 30 and 70:")
10 print(s[(s > 30) & (s < 70)])
11
12 print("\nValues < 20 or > 80:")
```

```
print(s[(s < 20) | (s > 80)])

print("\nValues NOT between 40 and 60:")
print(s[~((s >= 40) & (s <= 60))])

# Using .loc with boolean arrays
mask = s > 55
print("\nUsing .loc with boolean mask:")
print(s.loc[mask])
```

```
Boolean indexing examples:
Values > 50:
5    60
dtype: int64

Values between 30 and 70:
3    40
4    50
5    60
dtype: int64

Values < 20 or > 80:
0    10
dtype: int64

Values NOT between 40 and 60:
0    10
1    20
2    30
dtype: int64

Using .loc with boolean mask:
5    60
dtype: int64
```

Comparison Operations

```
# Create two series
s1 = pd.Series([1, 2, 3, 4])
s2 = pd.Series([10, 20, 30, 40])

print("Comparison operations:")
print("s1 > 2:\n", s1 > 2)        # Boolean series
print("s2 == 20:\n", s2 == 20)    # Boolean series
print("s1 <= s2:\n", s1 <= s2)    # Element-wise comparison

# Using comparison results for filtering
filtered = s2[s2 > 25]
print("Elements > 25:\n", filtered)
```

```
Comparison operations:
s1 > 2:
 0    False
1    False
2     True
3     True
dtype: bool
s2 == 20:
 0    False
1     True
2    False
3    False
dtype: bool
s1 <= s2:
 0    True
1    True
2    True
3    True
dtype: bool
Elements > 25:
 2    30
3    40
dtype: int64
```

Logical Operations

```
# Boolean series
bool_series = pd.Series([True, False, True, False])

print("Logical operations:")
print("NOT:\n", ~bool_series)       # Logical NOT
print("AND with True:\n", bool_series & True)
```

```
print("OR with False:\n", bool_series | False)

# Multiple conditions
condition = (s1 > 1) & (s2 < 35)
print("Complex condition:\n", condition)
```

```
Logical operations:
NOT:
 0    False
1     True
2    False
3     True
dtype: bool
AND with True:
 0     True
1    False
2     True
3    False
dtype: bool
OR with False:
 0     True
1    False
2     True
3    False
dtype: bool
Complex condition:
 0    False
1     True
2     True
3    False
dtype: bool
```

Index Alignment in Operations

```
# Series with different indexes
s3 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s4 = pd.Series([10, 20], index=['b', 'd'])

print("s3:\n", s3)
print("\ns4:\n", s4)

# Operations align on index
result = s3 + s4
print("\ns3 + s4 (index alignment):\n", result)
print("\nNote: NaN for non-matching indexes")
```

```
s3:
 a    1
b    2
c    3
dtype: int64

s4:
 b    10
d    20
dtype: int64

s3 + s4 (index alignment):
 a     NaN
b    12.0
c     NaN
d     NaN
dtype: float64

Note: NaN for non-matching indexes
```

DataFrame

A DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It's the most widely used pandas object. We call a table a **DataFrame**. We think of DataFrames as collections of named columns, called **Series**.

- **Creating a DataFrame:** You can create a DataFrame from various data sources, but a common method is using a dictionary of lists. Each key in the dictionary becomes a column, and each list becomes the data for that column.

Syntax: pandas.DataFrame(data, index, columns)

```
# Create a dictionary of data
data = {
    # All arrays/lists must be of the same length
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
```

```
    'Age': [25, 30, 35, 40],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}

# Create a DataFrame from the dictionary
df = pd.DataFrame(data)

print(df)
# Output: The numbers on the left (0, 1, 2, 3) are the index, and the labels on top (Name, Age, City) are the columns.
```

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
3    David   40      Houston
```

A **row** represents one observation **bold text** (here, a single person named Alice, aged 25, and living in New York). A **column** represents some **characteristic, or feature**, of that observation (here, the Name of that person).

```
# Create DataFrame from dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
    'Age': [25, 30, 35, 40, 28, 32],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'New York', 'Chicago'],
    'Salary': [70000, 80000, 95000, 110000, 75000, 90000]
}
# Keys are column names and their lists are column values

# pandas.DataFrame(data, index, columns)
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)
```

```
Original DataFrame:
      Name  Age         City  Salary
0    Alice   25     New York   70000
1      Bob   30  Los Angeles   80000
2  Charlie   35      Chicago   95000
3    David   40      Houston  110000
4      Eve   28     New York   75000
5    Frank   32      Chicago   90000
```

```
# Create DataFrame from list of lists with column names
data = [
    ['Alice', 25, 'New York', 50000],
    ['Bob', 30, 'London', 60000],
    ['Charlie', 35, 'Paris', 70000],
    ['Diana', 28, 'Tokyo', 55000],
    ['Eve', 32, 'Sydney', 65000]
]

columns = ['Name', 'Age', 'City', 'Salary']
df = pd.DataFrame(data, columns=columns)
print("\nDataFrame from list of lists:")
print(df)
```

```
DataFrame from list of lists:
      Name  Age      City  Salary
0    Alice   25  New York   50000
1      Bob   30    London   60000
2  Charlie   35     Paris   70000
3    Diana   28     Tokyo   55000
4      Eve   32    Sydney   65000
```

```
# Create DataFrame from numpy array
data = np.array([
    ['Alice', 25, 'New York', 50000],
    ['Bob', 30, 'London', 60000],
    ['Charlie', 35, 'Paris', 70000],
    ['Diana', 28, 'Tokyo', 55000],
    ['Eve', 32, 'Sydney', 65000]
])

df = pd.DataFrame(data, columns=['Name', 'Age', 'City', 'Salary'])
print("\nDataFrame from numpy array:")
print(df)
```

```
DataFrame from numpy array:
```

```
      Name Age      City Salary
0    Alice  25  New York  50000
1      Bob  30    London  60000
2  Charlie  35     Paris  70000
3    Diana  28     Tokyo  55000
4      Eve  32    Sydney  65000
```

With Custom Index

```
# Create DataFrame from list of lists with column names
data_list = [
    ['Alice', 25, 'New York', 50000],
    ['Bob', 30, 'London', 60000],
    ['Charlie', 35, 'Paris', 70000],
    ['Diana', 28, 'Tokyo', 55000],
    ['Eve', 32, 'Sydney', 65000]
]

# Create DataFrame with custom index
custom_index = ['emp001', 'emp002', 'emp003', 'emp004', 'emp005']

df_with_index = pd.DataFrame(data_list, columns=columns, index=custom_index)
print("\nDataFrame with custom index:")
print(df_with_index)
```

```
DataFrame with custom index:
           Name  Age      City  Salary
emp001    Alice   25  New York   50000
emp002      Bob   30    London   60000
emp003  Charlie   35     Paris   70000
emp004    Diana   28     Tokyo   55000
emp005      Eve   32    Sydney   65000
```

From Series

```
# Create a Series
name_series = pd.Series(['Alice', 'Bob', 'Charlie', 'Diana', 'Eve'], name='Name')
age_series = pd.Series([25, 30, 35, 28, 32], name='Age')
city_series = pd.Series(['New York', 'London', 'Paris', 'Tokyo', 'Sydney'], name='City')
salary_series = pd.Series([50000, 60000, 70000, 55000, 65000], name='Salary')

# Convert single Series to DataFrame
df_from_single_series = name_series.to_frame()
print("DataFrame from single Series:")
print(df_from_single_series)
```

```
DataFrame from single Series:
      Name
0    Alice
1      Bob
2  Charlie
3    Diana
4      Eve
```

```
# Method 1: Using dictionary
df_from_series_dict = pd.DataFrame({
    'Name': name_series,
    'Age': age_series,
    'City': city_series,
    'Salary': salary_series
})

print("DataFrame from multiple Series (dictionary method):")
print(df_from_series_dict)
```

```
DataFrame from multiple Series (dictionary method):
      Name  Age      City  Salary
0    Alice   25  New York   50000
1      Bob   30    London   60000
2  Charlie   35     Paris   70000
3    Diana   28     Tokyo   55000
4      Eve   32    Sydney   65000
```

```
# Series with different indexes
name_series_idx = pd.Series(['Alice', 'Bob', 'Charlie'], index=[0, 1, 2], name='Name')
age_series_idx = pd.Series([25, 30, 35], index=[1, 2, 3], name='Age')

# Concatenate - shows how pandas handles index alignment
df_mixed_index = pd.concat([name_series_idx, age_series_idx], axis=1)
```

```
# axis=0 -> Stack the series on top of each other vertically in one column/series:
# name_series_idx
# age_series_idx
# axis=1 -> Put the series side by side horizontally:
# name_series_idx age_series_idx
print("\nDataFrame from Series with different indexes:")
print(df_mixed_index)
print("Note: NaN values where indexes don't match")
```

```
DataFrame from Series with different indexes:
      Name   Age
0    Alice   NaN
1      Bob  25.0
2  Charlie  30.0
3      NaN  35.0
Note: NaN values where indexes don't match
```

```
# Create Series containing lists
data_series = pd.Series([
    ['Alice', 25, 'New York', 50000],
    ['Bob', 30, 'London', 60000],
    ['Charlie', 35, 'Paris', 70000]
], name='EmployeeData')

print(data_series)

# Convert to DataFrame
columns = ['Name', 'Age', 'City', 'Salary']
df_from_series_list = pd.DataFrame(data_series.tolist(), columns=columns)
print("\nDataFrame from Series of lists:")
print(df_from_series_list)
```

```
0    [Alice, 25, New York, 50000]
1        [Bob, 30, London, 60000]
2     [Charlie, 35, Paris, 70000]
Name: EmployeeData, dtype: object

DataFrame from Series of lists:
      Name  Age      City  Salary
0    Alice   25  New York   50000
1      Bob   30    London   60000
2  Charlie   35     Paris   70000
```

Viewing Data

head() shows the first n rows, while **tail()** shows the last n rows. If no number is specified, both **default to 5**.

```
# Show the first 3 rows
print("\nFirst 3 rows:")
print(df.head(3))

# Show the last 2 rows
print("\nLast 2 rows:")
print(df.tail(2))
```

```
First 3 rows:
      Name Age      City Salary
0    Alice  25  New York  50000
1      Bob  30    London  60000
2  Charlie  35     Paris  70000

Last 2 rows:
    Name Age    City Salary
3  Diana  28   Tokyo  55000
4    Eve  32  Sydney  65000
```

info() provides a summary of the DataFrame, including the data types and the number of non-null values.

```
print("\nDataFrame information:")
df.info()
```

```
DataFrame information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    5 non-null      object
```

```
 1   Age     5 non-null      object
 2   City    5 non-null      object
 3   Salary  5 non-null      object
dtypes: object(4)
memory usage: 292.0+ bytes
```

describe() generates descriptive statistics (count, mean, std, etc.) for numerical columns.

```
print("\nDescriptive statistics:")
print(df.describe())
```

```
Descriptive statistics:
             Age       Salary
count   5.000000      5.00000
mean   30.000000  60000.00000
std     3.807887   7905.69415
min    25.000000  50000.00000
25%    28.000000  55000.00000
50%    30.000000  60000.00000
75%    32.000000  65000.00000
max    35.000000  70000.00000
```

Selecting Columns

Selecting a Single Column: You can select a single column using bracket notation. This returns a pandas Series.

```
# Select the 'Name' column
names = df['Name']

print("\nSelected 'Name' column (as a Series):")
print(names)
```

```
Selected 'Name' column (as a Series):
0      Alice
1        Bob
2    Charlie
3      Diana
4        Eve
Name: Name, dtype: object
```

Selecting Multiple Columns: To select multiple columns, pass a list of column names inside the brackets. This returns a DataFrame.

```
# Select the 'Name' and 'Salary' columns
name_salary_df = df[['Name', 'Salary']]

print("\nSelected 'Name' and 'Salary' columns (as a DataFrame):")
print(name_salary_df)
```

```
Selected 'Name' and 'Salary' columns (as a DataFrame):
      Name Salary
0    Alice  50000
1      Bob  60000
2  Charlie  70000
3    Diana  55000
4      Eve  65000
```

```
# Select single column (returns Series)
print("Single column (Series):")
print(df['Name'])
print(type(df['Name']))

# Select multiple columns (returns DataFrame)
print("\nMultiple columns (DataFrame):")
print(df[['Name', 'Salary']])
print(type(df[['Name', 'Salary']]))

# Dot notation (only works for valid Python identifiers)
print("\nUsing dot notation:")
print(df.Name) # Same as df['Name']
```

```
Single column (Series):
0      Alice
1        Bob
2    Charlie
3      Diana
4        Eve
Name: Name, dtype: object
```

```
<class 'pandas.core.series.Series'>

Multiple columns (DataFrame):
      Name Salary
0    Alice  50000
1      Bob  60000
2  Charlie  70000
3    Diana  55000
4      Eve  65000
<class 'pandas.core.frame.DataFrame'>

Using dot notation:
0      Alice
1        Bob
2    Charlie
3      Diana
4        Eve
Name: Name, dtype: object
```

Selecting Rows

```
# Select rows by index label
print("Row by index:")
print(df.loc[0])                    # First row

# Select rows by position
print("\nRow by position:")
print(df.iloc[1])                   # Second row

# Slice rows
print("\nFirst 3 rows:")
print(df[:3])                       # Rows 0, 1, 2

print("\nRows 1 to 3:")
print(df[1:4])                      # Rows 1, 2, 3
```

```
Row by index:
Name         Alice
Age             25
City      New York
Salary       50000
Name: 0, dtype: object

Row by position:
Name         Bob
Age           30
City      London
Salary     60000
Name: 1, dtype: object

First 3 rows:
      Name Age      City Salary
0    Alice  25  New York  50000
1      Bob  30    London  60000
2  Charlie  35     Paris  70000

Rows 1 to 3:
      Name Age    City Salary
1      Bob  30  London  60000
2  Charlie  35   Paris  70000
3    Diana  28   Tokyo  55000
```

Basic Data Analysis

```
df['Salary'].info()
print("-----------------------------------------")
df['Age'].info()

print("-----------------------------------------")
print("-----------------------------------------")

# errors="coerce" tells to_numeric to convert non-numeric values to NaN
df['Salary'] = pd.to_numeric(df['Salary'], errors='coerce')
df['Age'] = pd.to_numeric(df['Age'], errors='coerce')

print("Basic statistics:")
print(df.describe())                # Summary statistics for numeric columns

print("\nColumn means:")
print(df[["Age", "Salary"]].mean())  # Mean of numeric columns

print("\nColumn sums:")
```

```
print(df.sum())                    # Sum of each columns

print("\nCount non-null values:")
print(df.count())                  # Count non-null values

print("\nUnique values in City column:")
print(df['City'].unique())         # Unique values

print("\nValue counts in City column:")
print(df['City'].value_counts())   # Frequency counts
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 5 entries, 0 to 4
Series name: Salary
Non-Null Count  Dtype
--------------  -----
5 non-null      int64
dtypes: int64(1)
memory usage: 172.0 bytes
----------------------------------------
<class 'pandas.core.series.Series'>
RangeIndex: 5 entries, 0 to 4
Series name: Age
Non-Null Count  Dtype
--------------  -----
5 non-null      int64
dtypes: int64(1)
memory usage: 172.0 bytes
----------------------------------------
----------------------------------------
Basic statistics:
             Age       Salary
count   5.000000      5.00000
mean   30.000000  60000.00000
std     3.807887   7905.69415
min    25.000000  50000.00000
25%    28.000000  55000.00000
50%    30.000000  60000.00000
75%    32.000000  65000.00000
max    35.000000  70000.00000

Column means:
Age          30.0
Salary    60000.0
dtype: float64

Column sums:
Name             AliceBobCharlieDianaEve
Age                                  150
City      New YorkLondonParisTokyoSydney
Salary                           300000
dtype: object

Count non-null values:
Name      5
Age       5
City      5
Salary    5
dtype: int64

Unique values in City column:
['New York' 'London' 'Paris' 'Tokyo' 'Sydney']

Value counts in City column:
City
New York    1
London      1
Paris       1
Tokyo       1
```

Filtering Data

Filtering is done using a boolean condition inside the brackets.

Filtering by a Single Condition: Filter for rows where the Age is greater than 30.

```
# Filter for people older than 30
older_than_30 = df[df['Age'] > 30]

print("\nPeople older than 30:")
print(older_than_30)
```

```
People older than 30:
      Name  Age    City  Salary
2  Charlie   35   Paris   70000
```

```
4      Eve   32  Sydney   65000
```

Filtering by Multiple Conditions: Combine multiple conditions using & (and) or | (or). Each condition must be in parentheses.

```
# Filter for people from New York who earn more than $70,000
ny_high_earners = df[(df['City'] == 'New York') & (df['Salary'] > 40000)]

print("\nHigh earners from New York:")
print(ny_high_earners) # Sub dataframe
```

```
High earners from New York:
    Name  Age      City  Salary
0  Alice   25  New York   50000
```

```
# Simple filtering
print("People older than 30:")
print(df[df['Age'] > 30])

print("\nPeople from Paris:")
print(df[df['City'] == 'Paris'])

print("\nHigh earners (Salary > 60000):")
print(df[df['Salary'] > 60000])

# Multiple conditions
print("\nSenior high earners:")
print(df[(df['Age'] > 30) & (df['Salary'] > 60000)])

# Using query method
print("\nUsing query method:")
print(df.query('Age > 30 and Salary > 60000'))
```

```
People older than 30:
      Name  Age    City  Salary
2  Charlie   35   Paris   70000
4      Eve   32  Sydney   65000

People from Paris:
      Name  Age   City  Salary
2  Charlie   35  Paris   70000

High earners (Salary > 60000):
      Name  Age    City  Salary
2  Charlie   35   Paris   70000
4      Eve   32  Sydney   65000

Senior high earners:
      Name  Age    City  Salary
2  Charlie   35   Paris   70000
4      Eve   32  Sydney   65000

Using query method:
      Name  Age    City  Salary
2  Charlie   35   Paris   70000
4      Eve   32  Sydney   65000
```

Adding and Modifying Columns

Adding a New Column: You can add a new column by simply assigning a list or Series to a new column name.

```
# Add a new 'Department' column
df['Department'] = ['IT', 'HR', 'Finance', 'IT', 'Marketing']

print("\nDataFrame with new 'Department' column:")
print(df)
```

```
DataFrame with new 'Department' column:
      Name  Age      City  Salary Department
0    Alice   25  New York   50000         IT
1      Bob   30    London   60000         HR
2  Charlie   35     Paris   70000    Finance
3    Diana   28     Tokyo   55000         IT
4      Eve   32    Sydney   65000  Marketing
```

Modifying an Existing Column: You can modify an existing column's values. For instance, let's give everyone a 10% raise.

```
# Give everyone a 10% raise
df['Salary'] = df['Salary'] * 1.10
```

```
3
4 print("\nDataFrame with updated 'Salary' column (10% raise):")
5 print(df)
```

```
DataFrame with updated 'Salary' column (10% raise):
      Name  Age      City   Salary Department
0    Alice   25  New York  55000.0         IT
1      Bob   30    London  66000.0         HR
2  Charlie   35     Paris  77000.0    Finance
3    Diana   28     Tokyo  60500.0         IT
4      Eve   32    Sydney  71500.0  Marketing
```

```
1 # Add new column
2 df['Bonus'] = df['Salary'] * 0.1     # 10% bonus
3 print("After adding Bonus column:")
4 print(df)
5
6 # Modify existing column
7 df['Salary'] = df['Salary'] * 1.05   # 5% raise
8 print("\nAfter salary increase:")
9 print(df)
10
11 # Add column based on condition
12 df['Senior'] = df['Age'] > 30
13 print("\nAfter adding Senior column:")
14 print(df)
15
16 # Using assign (returns new DataFrame)
17 df_new = df.assign(Total_Comp = df['Salary'] + df['Bonus'])
18 print("\nUsing assign (creates new DataFrame):")
19 print(df_new)
20
21 # Create a new column
22 df['Total_Comp'] = df['Salary'] + df['Bonus']
23 df
```

```
After adding Bonus column:
      Name  Age      City   Salary    Bonus  Senior
0    Alice   25  New York  55125.0  5512.50   False
1      Bob   30    London  66150.0  6615.00   False
2  Charlie   35     Paris  77175.0  7717.50    True
3    Diana   28     Tokyo  60637.5  6063.75   False
4      Eve   32    Sydney  71662.5  7166.25    True

After salary increase:
      Name  Age      City     Salary    Bonus  Senior
0    Alice   25  New York  57881.250  5512.50   False
1      Bob   30    London  69457.500  6615.00   False
2  Charlie   35     Paris  81033.750  7717.50    True
3    Diana   28     Tokyo  63669.375  6063.75   False
4      Eve   32    Sydney  75245.625  7166.25    True

After adding Senior column:
      Name  Age      City     Salary    Bonus  Senior
0    Alice   25  New York  57881.250  5512.50   False
1      Bob   30    London  69457.500  6615.00   False
2  Charlie   35     Paris  81033.750  7717.50    True
3    Diana   28     Tokyo  63669.375  6063.75   False
4      Eve   32    Sydney  75245.625  7166.25    True

Using assign (creates new DataFrame):
      Name  Age      City     Salary    Bonus  Senior  Total_Comp
0    Alice   25  New York  57881.250  5512.50   False   63393.750
1      Bob   30    London  69457.500  6615.00   False   76072.500
2  Charlie   35     Paris  81033.750  7717.50    True   88751.250
3    Diana   28     Tokyo  63669.375  6063.75   False   69733.125
4      Eve   32    Sydney  75245.625  7166.25    True   82411.875
```

	Name	Age	City	Salary	Bonus	Senior	Total_Comp
0	Alice	25	New York	57881.250	5512.50	False	63393.750
1	Bob	30	London	69457.500	6615.00	False	76072.500
2	Charlie	35	Paris	81033.750	7717.50	True	88751.250
3	Diana	28	Tokyo	63669.375	6063.75	False	69733.125
4	Eve	32	Sydney	75245.625	7166.25	True	82411.875

Adding and Modifying Rows

1. Using loc[] The loc[] indexer is the most straightforward and recommended way to add a single new row. You simply specify a new index label and assign a list or a Series of values to it.

```
# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)
# Output:
#    Name  Age
# 0  Alice   25
# 1    Bob   30

size = df['Name'].size # 2

# Add a new row using a new index label (e.g., 2)
# df.iloc[2] = ['Charlie', 35] # IndexError: iloc cannot enlarge its target object
# df.loc[size] = ['charlie', 35] OR
df.loc[2] = ['Charlie', 35]

print("\nDataFrame after adding a new row with loc[]:")
print(df)
# Output:
#       Name  Age
# 0    Alice   25
# 1      Bob   30
# 2  Charlie   35
```

```
Original DataFrame:
    Name  Age
0  Alice   25
1    Bob   30

DataFrame after adding a new row with loc[]:
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

If you don't know the next available index, you can use len(df) to get the count of current rows, which will be the next integer index.

```
print(df)
print("Current Lenght: ", len(df))

df.loc[len(df)] = ['David', 40]
print("\nDataFrame after adding another row:")

# df = df.drop([3,4,5]) # For removing multiple rows

print(df)
# Output:
#       Name  Age
# 0    Alice   25
# 1      Bob   30
# 2  Charlie   35
# 3    David   40
```

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
Current Lenght:  3

DataFrame after adding another row:
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
3    David   40
```

2. Using concat() The pd.concat() function is a powerful tool for adding multiple rows or another DataFrame. It's an efficient way to append data, especially if you're adding many rows at once. You must create the new row(s) as a DataFrame first and then concatenate it with the original DataFrame.

```
# Create the original DataFrame
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)

# Create a new row (or multiple rows) as a new DataFrame
new_row = pd.DataFrame([['Charlie', 35]], columns=['Name', 'Age'])

```

```
# Use pd.concat() to append the new row.
# ignore_index=True ensures the new DataFrame has a clean, continuous index.
df = pd.concat([df, new_row], ignore_index=True) # axis=0 -> Insert rows beneath each other

print("\nDataFrame after using concat():")
print(df)
# Output:
#       Name  Age
# 0    Alice   25
# 1      Bob   30
# 2  Charlie   35
```

```
DataFrame after using concat():
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

1. Modifying a Row by Index Label with loc[]: The .loc[] accessor is the most common way to modify rows based on their index labels. You can modify an entire row or just a specific column within that row.

- **Modifying the Entire Row:** To replace an entire row's data, use .loc[] with the index label of the row you want to change and assign a new list or Series of values.

```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data, index=['A', 'B', 'C'])

print("Original DataFrame:")
print(df)

# Modify the row with index label 'B'
df.loc['B'] = ['Robert', 31]

print("\nDataFrame after modifying row 'B':")
print(df)
```

```
Original DataFrame:
      Name  Age
A    Alice   25
B      Bob   30
C  Charlie   35

DataFrame after modifying row 'B':
      Name  Age
A    Alice   25
B   Robert   31
C  Charlie   35
```

- **Modifying Specific Columns in a Row:** You can also use .loc[] to target specific columns within a row you want to modify.

```
# Modify only the 'Age' for the row with index 'C'
df.loc['C', 'Age'] = 36

print("\nDataFrame after modifying 'Age' for row 'C':")
print(df)
```

```
DataFrame after modifying 'Age' for row 'C':
      Name  Age
A    Alice   25
B   Robert   31
C  Charlie   36
```

2. Modifying a Row by Position with iloc[]: The .iloc[] accessor works similarly to .loc[], but it uses integer-based positions (0-indexed) instead of index labels. This is useful when you want to modify a row based on its position in the DataFrame, regardless of its label.

```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)

print("Original DataFrame (default index):")
print(df)

# Modify the second row (at position 1)
df.iloc[1] = ['Robert', 31]
```

```
print("\nDataFrame after modifying the second row with iloc[]:")
print(df)
```

```
Original DataFrame (default index):
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35

DataFrame after modifying the second row with iloc[]:
      Name  Age
0    Alice   25
1   Robert   31
2  Charlie   35
```

3. Modifying Multiple Rows with Boolean Indexing: This method is ideal for modifying multiple rows that meet a specific condition. You create a boolean mask and then use it to select and modify the rows.

- **Using a Single Condition:** Let's give all people over 30 a 10% raise on their salary.

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40],
        'Salary': [60000, 70000, 80000, 90000]}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)

# Select rows where Age > 30 and update the 'Salary' column
df.loc[df['Age'] > 30, 'Salary'] *= 1.10

print("\nDataFrame after giving a 10% raise to people over 30:")
print(df)
```

```
Original DataFrame:
      Name  Age  Salary
0    Alice   25   60000
1      Bob   30   70000
2  Charlie   35   80000
3    David   40   90000

DataFrame after giving a 10% raise to people over 30:
      Name  Age   Salary
0    Alice   25  60000.0
1      Bob   30  70000.0
2  Charlie   35  88000.0
3    David   40  99000.0
/tmp/ipython-input-1561033941.py:10: FutureWarning: Setting an item of incompatible dtype is deprecated and will raise an error
  df.loc[df['Age'] > 30, 'Salary'] *= 1.10
```

Renaming Columns

```
# Rename columns
df_renamed = df.rename(columns={'Name': 'Full_Name',
                                'City': 'Location',
                                'Salary': 'Annual_Salary'})
print("After renaming columns:")
print(df_renamed)

# Rename in place
df.rename(columns={'Name': 'Full_Name'}, inplace=True)
print("\nAfter in-place renaming:")
print(df)
```

```
After renaming columns:
  Full_Name  Age  Annual_Salary
0     Alice   25        60000.0
1       Bob   30        70000.0
2   Charlie   35        88000.0
3     David   40        99000.0

After in-place renaming:
  Full_Name  Age   Salary
0     Alice   25  60000.0
1       Bob   30  70000.0
2   Charlie   35  88000.0
3     David   40  99000.0
```

Reading From a CSV File

```
# Reading all the columns
elections = pd.read_csv("/content/drive/MyDrive/Classroom/Data Science/elections.csv")
print("DataFrame from CSV:")
elections
```

```
DataFrame from CSV:
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789
...	...	...	...	...	...	...
182	2024	Donald Trump	Republican	77303568	win	49.808629
183	2024	Kamala Harris	Democratic	75019230	loss	48.336772
184	2024	Jill Stein	Green	861155	loss	0.554864
185	2024	Robert Kennedy	Independent	756383	loss	0.487357
186	2024	Chase Oliver	Libertarian Party	650130	loss	0.418895

187 rows × 6 columns

```
# Reading specific columns
elections = pd.read_csv("/content/drive/MyDrive/Classroom/Data Science/elections.csv", usecols=['Candidate', 'Party', 'Resu

print("DataFrame from CSV:")
elections
```

```
DataFrame from CSV:
```

	Candidate	Party	Result
0	Andrew Jackson	Democratic-Republican	loss
1	John Quincy Adams	Democratic-Republican	win
2	Andrew Jackson	Democratic	win
3	John Quincy Adams	National Republican	loss
4	Andrew Jackson	Democratic	win
...	...	...	...
182	Donald Trump	Republican	win
183	Kamala Harris	Democratic	loss
184	Jill Stein	Green	loss
185	Robert Kennedy	Independent	loss
186	Chase Oliver	Libertarian Party	loss

187 rows × 3 columns

```
# Read files with a specific column as custom index

elections = pd.read_csv("/content/drive/MyDrive/Classroom/Data Science/elections.csv", index_col="Candidate") # 'Candidate'

elections
```

Candidate	Year	Party	Popular vote	Result	%
Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
Andrew Jackson	1828	Democratic	642806	win	56.203927
John Quincy Adams	1828	National Republican	500897	loss	43.796073
Andrew Jackson	1832	Democratic	702735	win	54.574789
...	...	...	...	...	...
Donald Trump	2024	Republican	77303568	win	49.808629
Kamala Harris	2024	Democratic	75019230	loss	48.336772
Jill Stein	2024	Green	861155	loss	0.554864
Robert Kennedy	2024	Independent	756383	loss	0.487357
Chase Oliver	2024	Libertarian Party	650130	loss	0.418895

187 rows × 5 columns

```
1 elections.set_index("Year")
```

Year	Party	Popular vote	Result	%
1824	Democratic-Republican	151271	loss	57.210122
1824	Democratic-Republican	113142	win	42.789878
1828	Democratic	642806	win	56.203927
1828	National Republican	500897	loss	43.796073
1832	Democratic	702735	win	54.574789
...	...	...	...	...
2024	Republican	77303568	win	49.808629
2024	Democratic	75019230	loss	48.336772
2024	Green	861155	loss	0.554864
2024	Independent	756383	loss	0.487357
2024	Libertarian Party	650130	loss	0.418895

187 rows × 4 columns

```
1 elections.reset_index()
2 elections
```

Candidate	Year	Party	Popular vote	Result	%
Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
Andrew Jackson	1828	Democratic	642806	win	56.203927
John Quincy Adams	1828	National Republican	500897	loss	43.796073
Andrew Jackson	1832	Democratic	702735	win	54.574789
...	...	...	...	...	...
Donald Trump	2024	Republican	77303568	win	49.808629
Kamala Harris	2024	Democratic	75019230	loss	48.336772
Jill Stein	2024	Green	861155	loss	0.554864
Robert Kennedy	2024	Independent	756383	loss	0.487357
Chase Oliver	2024	Libertarian Party	650130	loss	0.418895

187 rows × 5 columns

```
1 print(elections.shape)
2 print()
3 print(elections.index)
```

```
print()
print(elections.columns)
print()
print(elections.columns.tolist())
```

```
(187, 5)

Index(['Andrew Jackson', 'John Quincy Adams', 'Andrew Jackson',
       'John Quincy Adams', 'Andrew Jackson', 'Henry Clay', 'William Wirt',
       'Hugh Lawson White', 'Martin Van Buren', 'William Henry Harrison',
       ...
       'Jill Stein', 'Joseph Biden', 'Donald Trump', 'Jo Jorgensen',
       'Howard Hawkins', 'Donald Trump', 'Kamala Harris', 'Jill Stein',
       'Robert Kennedy', 'Chase Oliver'],
      dtype='object', name='Candidate', length=187)

Index(['Year', 'Party', 'Popular vote', 'Result', '%'], dtype='object')

['Year', 'Party', 'Popular vote', 'Result', '%']
```

Data Extraction

CONTEXT-DEPENDENT EXTRACTION: [ ] Selection operators:

- .loc selects items by label. First argument is rows, second argument is columns.
- .iloc selects items by integer. First argument is rows, second argument is columns.
- [] only takes one argument, which may be:
  - A slice of row numbers.
  - A list of column labels.
  - A single column label.

That is, [] is context sensitive.

Remember: .loc[] includes endpoints, .iloc[] and [] exclude endpoints in slicing

```
# Single column - returns Series
candidates = df['Candidate']
print("Candidates column (Series):")
print(candidates.head())
print("Type:", type(candidates))

# Multiple columns - returns DataFrame
candidate_party = df[['Candidate', 'Party']]
print("\nCandidate and Party columns (DataFrame):")
print(candidate_party.head())
print("Type:", type(candidate_party))

# All columns except some
no_votes = df[df.columns.difference(['Popular vote', '%'])]
print("\nAll columns except vote columns:")
print(no_votes.head())
```

```
Candidates column (Series):
0       Andrew Jackson
1    John Quincy Adams
2       Andrew Jackson
3    John Quincy Adams
4       Andrew Jackson
Name: Candidate, dtype: object
Type: <class 'pandas.core.series.Series'>

Candidate and Party columns (DataFrame):
           Candidate                Party
0     Andrew Jackson  Democratic-Republican
1  John Quincy Adams  Democratic-Republican
2     Andrew Jackson             Democratic
3  John Quincy Adams    National Republican
4     Andrew Jackson             Democratic
Type: <class 'pandas.core.frame.DataFrame'>

All columns except vote columns:
           Candidate                Party Result  Year
0     Andrew Jackson  Democratic-Republican   loss  1824
1  John Quincy Adams  Democratic-Republican    win  1824
2     Andrew Jackson             Democratic    win  1828
3  John Quincy Adams    National Republican   loss  1828
4     Andrew Jackson             Democratic    win  1832
```

```
# Reading all the columns
df = pd.read_csv("/content/drive/MyDrive/Classroom/Data Science/elections.csv")
```

```
winners = df[df['Result'] == 'win']
print(f"\n=== ALL WINNERS ===")
print(winners[['Year', 'Candidate', 'Party', 'Popular vote', '%']])
```

```
=== ALL WINNERS ===
     Year              Candidate                Party  Popular vote  \
1    1824       John Quincy Adams  Democratic-Republican        113142
2    1828          Andrew Jackson             Democratic        642806
4    1832          Andrew Jackson             Democratic        702735
8    1836        Martin Van Buren             Democratic        763291
11   1840  William Henry Harrison                   Whig       1275583
13   1844             James Polk             Democratic       1339570
16   1848          Zachary Taylor                   Whig       1360235
17   1852         Franklin Pierce             Democratic       1605943
20   1856          James Buchanan             Democratic       1835140
23   1860         Abraham Lincoln             Republican       1855993
27   1864         Abraham Lincoln         National Union       2211317
30   1868           Ulysses Grant             Republican       3013790
32   1872           Ulysses Grant             Republican       3597439
33   1876        Rutherford Hayes             Republican       4034142
36   1880          James Garfield             Republican       4453337
39   1884        Grover Cleveland             Democratic       4914482
43   1888       Benjamin Harrison             Republican       5443633
47   1892        Grover Cleveland             Democratic       5553898
53   1896        William McKinley             Republican       7112138
56   1900        William McKinley             Republican       7228864
60   1904      Theodore Roosevelt             Republican       7630557
65   1908            William Taft             Republican       7678335
70   1912          Woodrow Wilson             Democratic       6296284
74   1916          Woodrow Wilson             Democratic       9126868
79   1920          Warren Harding             Republican      16144093
80   1924         Calvin Coolidge             Republican      15723789
84   1928          Herbert Hoover             Republican      21427123
86   1932      Franklin Roosevelt             Democratic      22821277
91   1936      Franklin Roosevelt             Democratic      27752648
94   1940      Franklin Roosevelt             Democratic      27313945
97   1944      Franklin Roosevelt             Democratic      25612916
100  1948            Harry Truman             Democratic      24179347
106  1952       Dwight Eisenhower             Republican      34075529
109  1956       Dwight Eisenhower             Republican      35579180
111  1960            John Kennedy             Democratic      34220984
114  1964          Lyndon Johnson             Democratic      43127041
117  1968           Richard Nixon             Republican      31783783
120  1972           Richard Nixon             Republican      47168710
123  1976            Jimmy Carter             Democratic      40831881
131  1980          Ronald Reagan             Republican      43903230
133  1984          Ronald Reagan             Republican      54455472
135  1988       George H. W. Bush             Republican      48886597
140  1992            Bill Clinton             Democratic      44909806
144  1996            Bill Clinton             Democratic      47400125
152  2000         George W. Bush             Republican      50456002
157  2004         George W. Bush             Republican      62040610
162  2008          Barack Obama             Democratic      69498516
168  2012          Barack Obama             Democratic      65915795
173  2016          Donald Trump             Republican      62984828
178  2020          Joseph Biden             Democratic      81268924
182  2024          Donald Trump             Republican      77303568

            %
1    42.789878
2    56.203927
```

```
# Andrew Jackson's performances
jackson = df[df['Candidate'] == 'Andrew Jackson']
print(f"\n=== ANDREW JACKSON'S PERFORMANCES ===")
print(jackson)

# Barack Obama's performances
obama = df[df['Candidate'] == 'Barack Obama']
print(f"\n=== BARACK OBAMA'S PERFORMANCES ===")
print(obama)
```

```
=== ANDREW JACKSON'S PERFORMANCES ===
   Year       Candidate                Party  Popular vote Result          %
0  1824  Andrew Jackson  Democratic-Republican        151271   loss  57.210122
2  1828  Andrew Jackson             Democratic        642806    win  56.203927
4  1832  Andrew Jackson             Democratic        702735    win  54.574789

=== BARACK OBAMA'S PERFORMANCES ===
     Year     Candidate       Party  Popular vote Result          %
162  2008  Barack Obama  Democratic      69498516    win  53.023510
168  2012  Barack Obama  Democratic      65915795    win  51.258484
```

```
# Democratic Party performances
democrats = df[df['Party'] == 'Democratic']
print(f"\n=== DEMOCRATIC PARTY PERFORMANCES ===")
print(f"Total Democratic candidates: {len(democrats)}")
print(f"Democratic wins: {len(democrats[democrats['Result'] == 'win'])}")

# Republican Party performances
republicans = df[df['Party'] == 'Republican']
print(f"\n=== REPUBLICAN PARTY PERFORMANCES ===")
print(f"Total Republican candidates: {len(republicans)}")
print(f"Republican wins: {len(republicans[republicans['Result'] == 'win'])}")
```

```
=== DEMOCRATIC PARTY PERFORMANCES ===
Total Democratic candidates: 48
Democratic wins: 23

=== REPUBLICAN PARTY PERFORMANCES ===
Total Republican candidates: 42
Republican wins: 24
```

```
print("\n=== STATISTICAL ANALYSIS ===")

# Basic statistics for numerical columns
print("Popular vote statistics:")
print(df['Popular vote'].describe())

print("\nPercentage statistics:")
print(df['%'].describe())
```

```
=== STATISTICAL ANALYSIS ===
Popular vote statistics:
count    1.870000e+02
mean     1.285001e+07
std      1.999673e+07
min      1.007150e+05
25%      4.000375e+05
50%      1.605943e+06
75%      2.058547e+07
max      8.126892e+07
Name: Popular vote, dtype: float64

Percentage statistics:
count    187.000000
mean      27.268504
std       23.023379
min        0.098088
25%        1.125839
50%       37.670670
75%       48.353003
max       61.344703
Name: %, dtype: float64
```

```
print("\n=== SPECIFIC YEAR ANALYSIS ===")

# 2024 election results
election_2024 = df[df['Year'] == 2024]
print("2024 Election Results:")
print(election_2024[['Candidate', 'Party', 'Popular vote', '%', 'Result']])

# 2000 election (controversial)
election_2000 = df[df['Year'] == 2000]
print(f"\n2000 Election Results:")
print(election_2000[['Candidate', 'Party', 'Popular vote', '%', 'Result']])
```

```
=== SPECIFIC YEAR ANALYSIS ===
2024 Election Results:
          Candidate              Party  Popular vote          % Result
182    Donald Trump         Republican      77303568  49.808629    win
183   Kamala Harris         Democratic      75019230  48.336772   loss
184      Jill Stein              Green        861155   0.554864   loss
185  Robert Kennedy        Independent        756383   0.487357   loss
186    Chase Oliver  Libertarian Party        650130   0.418895   loss

2000 Election Results:
          Candidate        Party  Popular vote          % Result
151         Al Gore   Democratic      50999897  48.491813   loss
152  George W. Bush   Republican      50456002  47.974666    win
153    Harry Browne  Libertarian        384431   0.365525   loss
154    Pat Buchanan       Reform        448895   0.426819   loss
155     Ralph Nader        Green       2882955   2.741176   loss
```

Remember:

- .loc performs label-based extraction
- .iloc performs integer-based extraction

When choosing between .loc and .iloc, you'll usually choose .loc.

- Safer: If the order of data gets shuffled in a public database, your code still works.
- Readable: Easier to understand what elections.loc[:, ["Year", "Candidate", "Result"]] means than elections.iloc[:, [0, 1, 4]]

.iloc can still be useful.

- Example: If you have a DataFrame of movie earnings sorted by earnings, can use .iloc to get the median earnings for a given year (index into the middle).

```
# Single row by index label
print("Row with index 0:")
print(df.loc[0])

# Multiple rows by index labels
print("\nRows with indices 0, 2, 4:")
print(df.loc[[0, 2, 4]])

# Specific row and column
print("\nCandidate at index 10:")
print(df.loc[10, 'Candidate'])

# Multiple rows and specific columns
print("\nCandidate and Party for rows 5-10:")
print(df.loc[5:10, ['Candidate', 'Party']])

# All rows, specific columns
print("\nAll years and candidates:")
print(df.loc[:, ['Year', 'Candidate']].head())
```

```
Row with index 0:
Year                            1824
Candidate              Andrew Jackson
Party           Democratic-Republican
Popular vote                   151271
Result                           loss
%                           57.210122
Name: 0, dtype: object

Rows with indices 0, 2, 4:
   Year       Candidate                Party  Popular vote Result          %
0  1824  Andrew Jackson  Democratic-Republican        151271   loss  57.210122
2  1828  Andrew Jackson             Democratic        642806    win  56.203927
4  1832  Andrew Jackson             Democratic        702735    win  54.574789

Candidate at index 10:
Martin Van Buren

Candidate and Party for rows 5-10:
                 Candidate                Party
5              Henry Clay  National Republican
6            William Wirt         Anti-Masonic
7       Hugh Lawson White                 Whig
8        Martin Van Buren           Democratic
9   William Henry Harrison                 Whig
10       Martin Van Buren           Democratic

All years and candidates:
   Year          Candidate
0  1824     Andrew Jackson
1  1824  John Quincy Adams
2  1828     Andrew Jackson
3  1828  John Quincy Adams
4  1832     Andrew Jackson
```

```
# Using .loc with conditions
republican_winners = df.loc[(df['Party'] == 'Republican') & (df['Result'] == 'win'), :]
print("Republican winners:")
print(republican_winners.head())

# Specific columns with condition
recent_democrats = df.loc[df['Year'] > 2000, ['Year', 'Candidate', 'Party', '%']]
print("\nDemocratic candidates since 2000:")
print(recent_democrats.head())
```

```
Republican winners:
    Year         Candidate       Party  Popular vote Result          %
23  1860   Abraham Lincoln  Republican       1855993    win  39.699408
30  1868     Ulysses Grant  Republican       3013790    win  52.665305
```

```
32  1872     Ulysses Grant  Republican       3597439    win  55.928594
33  1876  Rutherford Hayes  Republican       4034142    win  48.471624
36  1880    James Garfield  Republican       4453337    win  48.369234

Democratic candidates since 2000:
     Year         Candidate         Party          %
156  2004        David Cobb         Green   0.098088
157  2004    George W. Bush    Republican  50.771824
158  2004        John Kerry    Democratic  48.306775
159  2004  Michael Badnarik   Libertarian   0.325108
160  2004  Michael Peroutka  Constitution   0.117542
```

```
# Slice by index labels (inclusive!)
print("Rows 10 to 15 (inclusive):")
print(df.loc[10:15])

# Slice with step
print("\nEvery 5th row from 0 to 20:")
print(df.loc[0:20:5])
```

```
Rows 10 to 15 (inclusive):
    Year               Candidate       Party  Popular vote Result          %
10  1840        Martin Van Buren  Democratic       1128854   loss  46.948787
11  1840  William Henry Harrison        Whig       1275583    win  53.051213
12  1844             Henry Clay        Whig       1300004   loss  49.250523
13  1844             James Polk  Democratic       1339570    win  50.749477
14  1848             Lewis Cass  Democratic       1223460   loss  42.552229
15  1848        Martin Van Buren   Free Soil        291501   loss  10.138474

Every 5th row from 0 to 20:
    Year         Candidate                 Party  Popular vote Result  \
0   1824    Andrew Jackson  Democratic-Republican        151271   loss
5   1832        Henry Clay    National Republican        484205   loss
10  1840  Martin Van Buren             Democratic       1128854   loss
15  1848  Martin Van Buren              Free Soil        291501   loss
20  1856    James Buchanan             Democratic       1835140    win

            %
0   57.210122
5   37.603628
10  46.948787
15  10.138474
20  45.306080
```

```
# Single row by position
print("First row (position 0):")
print(df.iloc[0])

# Multiple rows by positions
print("\nRows at positions 0, 5, 10:")
print(df.iloc[[0, 5, 10]])

# Specific cell by position
print("\nValue at row 0, column 1:", df.iloc[0, 1])

# Multiple rows and columns by position
print("\nFirst 5 rows, first 3 columns:")
print(df.iloc[:5, :3])

# Last few rows
print("\nLast 3 rows:")
print(df.iloc[-3:])
```

```
First row (position 0):
Year                           1824
Candidate              Andrew Jackson
Party           Democratic-Republican
Popular vote                  151271
Result                          loss
%                          57.210122
Name: 0, dtype: object

Rows at positions 0, 5, 10:
    Year         Candidate                 Party  Popular vote Result  \
0   1824    Andrew Jackson  Democratic-Republican        151271   loss
5   1832        Henry Clay    National Republican        484205   loss
10  1840  Martin Van Buren             Democratic       1128854   loss

            %
0   57.210122
5   37.603628
10  46.948787

Value at row 0, column 1: Andrew Jackson

First 5 rows, first 3 columns:
```

```
   Year          Candidate                  Party
0  1824     Andrew Jackson  Democratic-Republican
1  1824  John Quincy Adams  Democratic-Republican
2  1828     Andrew Jackson             Democratic
3  1828  John Quincy Adams    National Republican
4  1832     Andrew Jackson             Democratic

Last 3 rows:
     Year       Candidate              Party  Popular vote Result        %
184  2024      Jill Stein              Green        861155   loss  0.554864
185  2024  Robert Kennedy        Independent        756383   loss  0.487357
186  2024    Chase Oliver  Libertarian Party        650130   loss  0.418895
```

```
df = pd.read_csv("/content/drive/MyDrive/Classroom/Data Science/elections.csv")

# All rows between year 2016 to 2024
year16_24 = df[(df['Year'] > 2015) & (df['Year'] < 2025)]
year16_24
```

	Year	Candidate	Party	Popular vote	Result	%
172	2016	Darrell Castle	Constitution	203091	loss	0.149640
173	2016	Donald Trump	Republican	62984828	win	46.407862
174	2016	Evan McMullin	Independent	732273	loss	0.539546
175	2016	Gary Johnson	Libertarian	4489235	loss	3.307714
176	2016	Hillary Clinton	Democratic	65853514	loss	48.521539
177	2016	Jill Stein	Green	1457226	loss	1.073699
178	2020	Joseph Biden	Democratic	81268924	win	51.311515
179	2020	Donald Trump	Republican	74216154	loss	46.858542
180	2020	Jo Jorgensen	Libertarian	1865724	loss	1.177979
181	2020	Howard Hawkins	Green	405035	loss	0.255731
182	2024	Donald Trump	Republican	77303568	win	49.808629
183	2024	Kamala Harris	Democratic	75019230	loss	48.336772
184	2024	Jill Stein	Green	861155	loss	0.554864
185	2024	Robert Kennedy	Independent	756383	loss	0.487357
186	2024	Chase Oliver	Libertarian Party	650130	loss	0.418895

```
year16_24.count()
```

	0
Year	15
Candidate	15
Party	15
Popular vote	15
Result	15
%	15

dtype: int64

```
year16_24['Party'].unique()
```

```
array(['Constitution', 'Republican', 'Independent', 'Libertarian',
       'Democratic', 'Green', 'Libertarian Party'], dtype=object)
```

```
year16_24['Party'].value_counts()
```

Party	count
Republican	3
Green	3
Democratic	3
Libertarian	2
Independent	2
Constitution	1
Libertarian Party	1

dtype: int64

```
year16_24['Candidate'].count()
```

```
np.int64(15)
```

```
df = pd.read_csv("/content/drive/MyDrive/Classroom/Data Science/elections.csv")

custom_df = df.set_index('Year')

# All rows between year 2016 to 2024
year16_24 = custom_df.loc[2016:2024, ['Candidate', 'Party', 'Result']]
year16_24
```

Year	Candidate	Party	Result
2016	Darrell Castle	Constitution	loss
2016	Donald Trump	Republican	win
2016	Evan McMullin	Independent	loss
2016	Gary Johnson	Libertarian	loss
2016	Hillary Clinton	Democratic	loss
2016	Jill Stein	Green	loss
2020	Joseph Biden	Democratic	win
2020	Donald Trump	Republican	loss
2020	Jo Jorgensen	Libertarian	loss
2020	Howard Hawkins	Green	loss
2024	Donald Trump	Republican	win
2024	Kamala Harris	Democratic	loss
2024	Jill Stein	Green	loss
2024	Robert Kennedy	Independent	loss
2024	Chase Oliver	Libertarian Party	loss

```
rows16 = df[df['Year'] == 2016]
rows16
```

	Year	Candidate	Party	Popular vote	Result	%
172	2016	Darrell Castle	Constitution	203091	loss	0.149640
173	2016	Donald Trump	Republican	62984828	win	46.407862
174	2016	Evan McMullin	Independent	732273	loss	0.539546
175	2016	Gary Johnson	Libertarian	4489235	loss	3.307714
176	2016	Hillary Clinton	Democratic	65853514	loss	48.521539
177	2016	Jill Stein	Green	1457226	loss	1.073699

```
# .iloc requires numeric indexers
rows5 = df.iloc[5:10, 1:4]
rows5
```

	Candidate	Party	Popular vote
5	Henry Clay	National Republican	484205
6	William Wirt	Anti-Masonic	100715
7	Hugh Lawson White	Whig	146109
8	Martin Van Buren	Democratic	763291
9	William Henry Harrison	Whig	550816

```
df.iloc[[1, 2, 3], [0, 1, 2]] # Custom slicing
```

	Year	Candidate	Party
1	1824	John Quincy Adams	Democratic-Republican
2	1828	Andrew Jackson	Democratic
3	1828	John Quincy Adams	National Republican

```
# Boolean array as input with specific column
df.loc[df['Year'] == 2016, ['Candidate']]
```

	Candidate
172	Darrell Castle
173	Donald Trump
174	Evan McMullin
175	Gary Johnson
176	Hillary Clinton
177	Jill Stein

```
# Display all candidates who contested in year 2016 and won the election
df.loc[(df['Year'] == 2016) & (df['Result'] == 'win'), 'Candidate']
```

	Candidate
173	Donald Trump

dtype: object

```
# Sample DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Age': [25, 30, 35, 28],
    'City': ['NY', 'London', 'Paris', 'Tokyo'],
    'Salary': [50000, 60000, 70000, 55000]
}, index=['a', 'b', 'c', 'd'])

print("Original DataFrame:")
print(df)

# Label-based selection
print("=== .loc[] EXAMPLES ===")

# Single row by label
print("df.loc['a'] - Single row:")
print(df.loc['a'])

# Multiple rows by labels
print("\ndf.loc[['a', 'c']] - Multiple rows:")
print(df.loc[['a', 'c']])

# Specific cell
print("\ndf.loc['b', 'City'] - Specific cell:")
print(df.loc['b', 'City'])

# Row and column slices (INCLUSIVE!)
print("\ndf.loc['a':'c', 'Name':'City'] - Inclusive slice:")
print(df.loc['a':'c', 'Name':'City'])

# Boolean indexing with conditions
print("\ndf.loc[df['Age'] > 28, ['Name', 'Salary']] - Conditional:")
print(df.loc[df['Age'] > 28, ['Name', 'Salary']])
```

```
Original DataFrame:
      Name  Age    City  Salary
a    Alice   25      NY   50000
b      Bob   30  London   60000
c  Charlie   35   Paris   70000
d    Diana   28   Tokyo   55000
=== .loc[] EXAMPLES ===
df.loc['a'] - Single row:
Name      Alice
Age          25
City         NY
Salary    50000
Name: a, dtype: object

df.loc[['a', 'c']] - Multiple rows:
      Name  Age   City  Salary
a    Alice   25     NY   50000
c  Charlie   35  Paris   70000

df.loc['b', 'City'] - Specific cell:
London

df.loc['a':'c', 'Name':'City'] - Inclusive slice:
      Name  Age    City
a    Alice   25      NY
b      Bob   30  London
c  Charlie   35   Paris

df.loc[df['Age'] > 28, ['Name', 'Salary']] - Conditional:
      Name  Salary
b      Bob   60000
c  Charlie   70000
```

Alternatives to Direct Boolean Array Selection

- .isin
- .str.startswith
- .groupby.filter

```
df = pd.read_csv("/content/drive/MyDrive/Classroom/Data Science/elections.csv")

# Returns a Boolean Series that is True when the corresponding name matches in
names list
names = ['Donald Trump', 'Jill Stein', 'Hillary Clinton']
df[df['Candidate'].isin(names)]
```

	Year	Candidate	Party	Popular vote	Result	%
170	2012	Jill Stein	Green	469627	loss	0.365199
173	2016	Donald Trump	Republican	62984828	win	46.407862
176	2016	Hillary Clinton	Democratic	65853514	loss	48.521539
177	2016	Jill Stein	Green	1457226	loss	1.073699
179	2020	Donald Trump	Republican	74216154	loss	46.858542