COMP 1023 Introduction to Python Programming

Pandas

Dr. Cecia Chan, Prof. SC Cheung, Dr. Alex Lam, Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Introduction

- Pandas is a powerful Python library designed for data manipulation and analysis, particularly suited for structured data.
- It offers two primary data structures: Series (1D) and DataFrame (2D), which simplify data handling and operations.
- Key functionalities include data filtering, aggregation, reshaping with pivot tables, handling missing values, and performing exploratory data analysis.
- Pandas is built on top of NumPy, providing additional features for data alignment and intuitive data indexing, making it essential for data science workflows.

In this course, we use version 2.3.1 of Pandas!

Part I

Series

Pandas Series

- A Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, etc.).
- Each element in a Series is associated with an index, which can be customized to be either numeric or string labels.
- Series can be easily created from various data sources, such as lists, dictionaries, or even NumPy arrays.
- Common operations include accessing elements by index, performing mathematical operations, and applying functions to the entire Series.

`import pandas`

Creating a Pandas Series

```python
# Filename: series_create.py
import pandas as pd

# Creating a Series without specifying an index
s1 = pd.Series([10, 20, 30, 40])
print(s1)

# Creating a Series with a custom index
s2 = pd.Series([10, 20, 30, 40],
               index=['first', 'second', 'third', 'fourth'])
print(s2)

# Creating a Series from a dictionary
data = {'first': 10, 'second': 20, 'third': 30, 'fourth': 40}
s3 = pd.Series(data)
print(s3)
```

Output:

```
0    10
1    20
2    30
3    40
dtype: int64
first     10
second    20
third     30
fourth    40
dtype: int64
first     10
second    20
third     30
fourth    40
dtype: int64
```

Accessing Elements in a Pandas Series

- You can access elements of a Series using:
  - Label-based indexing: Accessing elements using the explicit index assigned during Series creation (using the `Series.loc[]` attribute).
  - Position-based indexing: Accessing elements using their integer position (similar to NumPy arrays) (using the `Series.iloc[]` attribute).

```python
# Filename: series_access_elements.py
import pandas as pd

# Create a series with specific index
s = pd.Series([10, 20, 30, 40], index=['first', 'second', 'third', 'fourth'])

# Access the element by label (element associated with index 'second')
e1 = s.loc['second']
print(e1)  # Print 20

# Access the element in position 2 (3rd element)
e2 = s.iloc[2]
print(e2)  # Print 30
```

Accessing Values and Index in a Pandas Series

- You can retrieve the values and the index of a Series using the `.values` and `.index` attributes.
- The values are returned as a NumPy array, while the index is a Pandas Index object that supports various operations such as union and intersection.

```python
# Filename: series_access_values_index.py
import pandas as pd

# Create a series with specific index
s = pd.Series([10, 20, 30, 40], index=['first', 'second', 'third', 'fourth'])
print(s.values)  # Return a NumPy array
print(s.index)   # Return an Index object
```

Output:

```
[10 20 30 40]
Index(['first', 'second', 'third', 'fourth'], dtype='object')
```

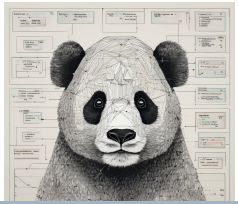
Assigning Values to Series Elements

- You can use `.loc[]` and `.iloc[]` to assign new values to elements and modify the Series in place.

```python
# Filename: series_assign_values.py
import pandas as pd

# Create a series with specific index
s = pd.Series([10, 20, 30], index=['first', 'second', 'third'])
print(s.loc['first'])  # Access with explicit index
print(s.iloc[0])       # Access with implicit index
s.loc['second'] = 50   # Assign a new value
print(s)
```

Output:

```
10
10
first      10
second     50
third      30
dtype: int64
```



Slicing a Pandas Series

- You can use `.loc[]` and `.iloc[]` to access slices of elements in a Series. Note that the `.loc[]` label handling can be problematic. For example, if you specify a label that does not exist in the index, it will raise a KeyError.
- With label-based indexing (`loc`), both the starting and stopping indices are included in the slice.
- With position-based indexing (`iloc`), it behaves like NumPy arrays and lists: specify the start position (included) and the end position (excluded).
- The result of slicing is a new Series containing the selected elements.

```python
# Filename: series_slicing.py
import pandas as pd

s = pd.Series([10, 20, 30, 40, 50, 60],
        index=['first', 'second', 'third', 'fourth', 'fifth', 'sixth'])
# Slicing with explicit index (both included)
print(s.loc['third':'fifth'])
# Slicing with implicit index
# (start included and stop excluded)
print(s.iloc[2:5])
```

Output:

```
third     30
fourth    40
fifth     50
dtype: int64
third     30
fourth    40
fifth     50
dtype: int64
```

Masking a Pandas Series

- You can access Series elements using masking techniques.
- Masking creates a boolean Series where `True` indicates that the condition is satisfied and `False` otherwise.
- When using masking, you can directly access and modify Series elements without the need for the `loc` function.
- Similar to NumPy, you can apply the mask to access and/or modify the Series elements that meet the specified condition.

```python
# Filename: series_masking.py
import pandas as pd

s = pd.Series([10, 15, 25, 5, 30],
              index=['a', 'b', 'c', 'd', 'e'])
mask = (s > 10) & (s < 30)  # Create a mask for values between 10 and 30
print(mask)
s[mask] = 0  # Modify elements of s where mask is True
print(s)
```

Output:

```
a    False
b     True
c     True
d    False
e    False
dtype: bool
a    10
b     0
c     0
d     5
e    30
dtype: int64
```

Accessing a Pandas Series with Fancy Indexing

- Fancy indexing allows you to access a subset of a Series by specifying a list of indices.

```python
# Filename: series_fancy_indexing.py
import pandas as pd

s = pd.Series([10, 20, 30, 40, 50],
              index=['first', 'second', 'third', 'fourth', 'fifth'])
print(s.loc[['first', 'third']])  # Access indices 'first' and 'third'
print(s.iloc[[0, 2]])             # Access positions 0 and 2
```

Output:

```
first    10
third    30
dtype: int64
first    10
third    30
dtype: int64
```

Part II

DataFrame

Introduction

- A DataFrame is a powerful 2-dimensional data structure in Pandas, similar to a spreadsheet or SQL table.
- It consists of rows and columns, where each column is a Series object that can hold different data types but shares the same index.
- Each column in a DataFrame has a unique name, which allows for easy access and manipulation of the data.
- DataFrames are ideal for handling structured data and performing complex data analysis and manipulation tasks.

Creating a DataFrame

- You can create a DataFrame from existing Series that share the same index.
- Use the `pd.DataFrame()` constructor, passing a dictionary with column names as keys and the corresponding Series as values.

```python
# Filename: dataframe_create1.py
import pandas as pd

sales = pd.Series([100, 150, 200], index=['Product A', 'Product B', 'Product C'])
cost = pd.Series([80, 90, 120], index=['Product A', 'Product B', 'Product C'])
units_sold = pd.Series([20, 30, 15], index=['Product A', 'Product B', 'Product C'])

df = pd.DataFrame({'Sales': sales, 'Cost': cost, 'Units Sold': units_sold})
print(df)
```

Output:

```
           Sales  Cost  Units Sold
Product A    100    80          20
Product B    150    90          30
Product C    200   120          15
```

Creating a DataFrame with Different Indexes

- When creating a DataFrame from Series with different indexes, only the Series with matching index values will have data values. For indexes that do not match, the corresponding columns will have NaN (null) values.

```python
import pandas as pd  # Filename: dataframe_create2.py

sales = pd.Series([100, 150, 200, 250],
           index=['Product A', 'Product B', 'Product C', 'Product D'])
cost = pd.Series([80, 90, 120], index=['Product A', 'Product B', 'Product C'])
units_sold = pd.Series([20, 30], index=['Product A', 'Product B'])
df = pd.DataFrame({'Sales': sales, 'Cost': cost, 'Units Sold': units_sold})
print(df)
```

Output:

```
           Sales  Cost  Units Sold
Product A    100    80        20.0
Product B    150    90        30.0
Product C    200   120         NaN
Product D    250   NaN         NaN
```

Creating a DataFrame from a List of Dictionaries

- You can create a DataFrame from a list of dictionaries, where each dictionary represents a row in the DataFrame.
- If no index is provided, the DataFrame will automatically assign a default integer index. You can specify a custom index by passing it as a parameter (e.g., `{index = ['row1', 'row2', 'row3']}`).

```python
import pandas as pd  # Filename: dataframe_create3.py

data1 = [{'name': 'Emma', 'age': 28, 'city': 'Beijing'},
         {'name': 'Liam', 'age': 32, 'city': 'Shanghai'},
         {'name': 'Noah', 'age': 27, 'city': 'Guangzhou'}]
df1 = pd.DataFrame(data1)
print(df1)

data2 = [{'name': 'Sophie', 'age': 29, 'city': 'Shenzhen'},
         {'name': 'Oliver', 'age': 31, 'city': 'Chengdu'},
         {'name': 'Ava', 'age': 26, 'city': 'Hangzhou'}]
df2 = pd.DataFrame(data2, index=['row1', 'row2', 'row3'])
print(df2)
```

Output:

```
    name  age       city
0  Emma   28    Beijing
1  Liam   32   Shanghai
2  Noah   27  Guangzhou
        name  age       city
row1  Sophie   29  Shenzhen
row2  Oliver   31   Chengdu
row3     Ava   26  Hangzhou
```

Creating a DataFrame from a Dictionary of Key-List Pairs

- You can create a DataFrame from a dictionary where each key corresponds to a column name and each value is a list of column entries.
- Each list in the dictionary represents the data for a specific column in the DataFrame.
- The index of the DataFrame is automatically assigned as a progressive number unless specified otherwise (e.g., `index=['row1', 'row2', 'row3']`).

```python
# Filename: dataframe_create4.py
import pandas as pd

data_dict = {
    "Name": ["Emma", "Liam", "Noah"],
    "Age": [28, 32, 27],
    "City": ["Beijing", "Shanghai", "Guangzhou"]
}

df = pd.DataFrame(data_dict)
print(df)
```

Output:

```
   Name  Age       City
0  Emma   28    Beijing
1  Liam   32   Shanghai
2  Noah   27  Guangzhou
```

Creating a DataFrame from a 2D NumPy Array

- You can create a DataFrame from a 2-dimensional NumPy array by specifying the column names and, optionally, the index.

```python
# Filename: dataframe_create5.py
import numpy as np
import pandas as pd

# Create a 2D NumPy array with data related to cities in China
arr = np.array([[28, 32, 27],  # Ages
                [1, 2, 3],     # IDs or similar identifiers
                [1, 0, 1]])    # Some binary data (e.g., availability of a service)

# Create a DataFrame from the array
df = pd.DataFrame(arr, columns=['Beijing', 'Shanghai', 'Guangzhou'],
                  index=['Age', 'ID', 'Service Available'])
print(df)
```

Output:

```
                   Beijing  Shanghai  Guangzhou
Age                     28        32         27
ID                       1         2          3
Service Available        1         0          1
```

Accessing a DataFrame

- Accessing Column Names and Index
  - You can obtain all the column names and the index of a DataFrame using the `.columns` and `.index` attributes.
  - Both attributes return an Index object.
  - The `.columns` attribute provides an Index object containing the column names, which can be accessed similarly to row indices.
  - The `.index` attribute returns the Index for the rows.

```python
# Filename: dataframe_access1.py
import pandas as pd

data = {
    'Name': ['Emma', 'Liam', 'Noah'],
    'Age': [28, 32, 27],
    'City': ['Beijing', 'Shanghai', 'Guangzhou']
}

df = pd.DataFrame(data)
print(df.columns)  # Index object with column names
print(df.index)    # Index object with row indices
```

Output:

```
Index(['Name', 'Age', 'City'], dtype='object')
RangeIndex(start=0, stop=3, step=1)
```

Accessing DataFrame Data as a NumPy Array

- You can convert the data in a DataFrame to a NumPy array using the `.values` attribute.

```
# Filename: dataframe_access2.py
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [30, 25, 35],
    'Salary': [70000, 80000, 120000]
}

df = pd.DataFrame(data)

arr = df.values  # Convert DataFrame to NumPy array
print(arr)
```

Output:

```
[['Alice' 30 70000]
 ['Bob' 25 80000]
 ['Charlie' 35 120000]]
```

Accessing DataFrame Columns

- You can access a column of a DataFrame by specifying the column name in square brackets [].
- It returns a Series with the selected column.

```python
# Filename: dataframe_access3.py
import pandas as pd

temperature = pd.Series([22.5, 23.0, 19.5], index=['Monday', 'Tuesday', 'Wednesday'])
humidity = pd.Series([55, 60, 50], index=['Monday', 'Tuesday', 'Wednesday'])
wind_speed = pd.Series([5.5, 3.0, 7.2], index=['Monday', 'Tuesday', 'Wednesday'])
df = pd.DataFrame({'Temperature': temperature, 'Humidity': humidity,
                   'Wind Speed': wind_speed})

print(df["Humidity"])
```

Output:

```
Monday       55
Tuesday      60
Wednesday    50
Name: Humidity, dtype: int64
```

Accessing a Single DataFrame Row by Index

- You can access a single DataFrame row using the same methods as for Series:
  - `.loc` for label-based indexing
  - `.iloc` for position-based indexing
- It returns a Series with an element for each column.
- The index contains the names of the columns.

```
import pandas as pd  # Filename: dataframe_access4.py

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'],
                 index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000],
                       index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4],
                 index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city,
                   'Population': population,
                   'Area (sq km)': area})
print(df.loc['A'])  # Access by label
print(df.iloc[0])   # Access by position
```

Output:

```
City             Beijing
Population      21542000
Area (sq km)    16410.54
Name: A, dtype: object
City             Beijing
Population      21542000
Area (sq km)    16410.54
Name: A, dtype: object
```

Accessing DataFrames with Slicing

- You can access DataFrames with slicing by selecting rows and/or columns.
- You cannot mix position-based and label-based indexing.

```
# Filename: dataframe_access5.py
import pandas as pd

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'], index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000], index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4], index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city, 'Population': population, 'Area (sq km)': area})

# Access columns from 'Population' to 'Area (sq km)' and rows from 'B' to 'C'
print(df.loc['B':'C', 'Population':'Area (sq km)'])
```

Output:

```
   Population  Area (sq km)
B    24183300        6340.5
C    14904000        7434.4
```

Accessing DataFrames with Masking

- You can also use masking to select rows based on a condition.
- You can combine masking with slicing.
- You have to specify a mask to select the rows based on a condition and then slice to select only the same columns.

```python
# Filename: dataframe_access6.py
import pandas as pd

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'], index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000], index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4], index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city, 'Population': population, 'Area (sq km)': area})

# Create a mask for cities with population greater than
# 10 million and area less than 7000 sq km
mask = (df['Population'] > 10000000) & \
       (df['Area (sq km)'] < 7000)

# Print the filtered DataFrame based on the mask
print(df.loc[mask, 'Population':])
```

Output:

```
   Population  Area (sq km)
B    24183300        6340.5
```

Accessing DataFrames with Fancy Indexing

- You can use fancy indexing to select only some rows and/or only some columns.
- You have to specify two lists: one with the index values and another with the column names.

```
import pandas as pd  # Filename: dataframe_access7.py

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'],
                 index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000],
                       index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4],
                 index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city, 'Population': population,
                   'Area (sq km)': area})

# Print selected rows and columns
print(df.loc[['A', 'C'], ['City', 'Population']])

# Update selected rows and columns
df.loc[['A', 'C'], ['City', 'Population']] = ['Updated City', 0]
print(df.loc[['A', 'C'], ['City', 'Population']])
```

Output:

```
        City  Population
A    Beijing    21542000
C  Guangzhou    14904000
           City  Population
A  Updated City           0
C  Updated City           0
```

Accessing DataFrames with Masking and Fancy Indexing

```python
# Filename: dataframe_access8.py
import pandas as pd

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'],
                 index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000],
                       index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4],
                 index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city,
                   'Population': population,
                   'Area (sq km)': area})

# Create a mask for cities with population greater than
# 10 million and area less than 7000 sq km
mask = (df['Population'] > 10000000) & (df['Area (sq km)'] < 7000)

# Print the filtered DataFrame based on the mask, selecting specific columns
print(df.loc[mask, ['City', 'Area (sq km)']])
```

Output:

```
       City  Area (sq km)
B  Shanghai        6340.5
```

Adding a New Column to DataFrame

- You can add a new column from a Series to a DataFrame.
- The added Series should have the same index, and the DataFrame is modified in place.
- If the DataFrame already has a column with the specified name, then it is replaced.

```
import pandas as pd  # Filename: dataframe_add_column.py

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'], index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000], index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4], index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city, 'Population': population, 'Area (sq km)': area})

# Adding a new column to indicate if the city is coastal
df['Is Coastal'] = pd.Series([False, True, False], index=['A', 'B', 'C'])
# The above is equivalent to df['Is Coastal'] = [False, True, False]
print(df)
```

Output:

```
        City  Population  Area (sq km)  Is Coastal
A    Beijing    21542000      16410.54       False
B   Shanghai    24183300       6340.50        True
C  Guangzhou    14904000       7434.40       False
```

Drop Columns from a DataFrame

- You can drop some columns from the DataFrame with the 'drop' method and specify the list of columns to delete as a parameter, e.g.,`df.drop(columns=['column1', 'column2'])`.
- The 'drop' method returns a copy of the DataFrame (the DataFrame is not modified in place), therefore you have to assign the returned DataFrame to the old one if you want to modify it in place.
- Alternatively, you can do `df.drop(columns=['column1','column2'], inplace=True)`.

```
import pandas as pd  # Filename: dataframe_drop_column.py

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'],
                 index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000],
                       index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4],
                 index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city, 'Population': population,
                   'Area (sq km)': area})
print(df)  # Display the original DataFrame
# Drop the 'Population' and 'Area (sq km)' columns
df = df.drop(columns=['Population', 'Area (sq km)'])
print(df)  # Display the DataFrame after dropping columns
```

Output:

```
        City  Population  Area (sq km)
A    Beijing    21542000      16410.54
B   Shanghai    24183300       6340.50
C  Guangzhou    14904000       7434.40
        City
A    Beijing
B   Shanghai
C  Guangzhou
```

Rename Columns of a DataFrame

- You can rename some columns of a DataFrame by passing a dictionary that maps old names to new names as a parameter of the `df.rename()` method.
- If an index is not present in one of the Series, the corresponding value in the DataFrame will be set to NaN, which represents missing or undefined data.

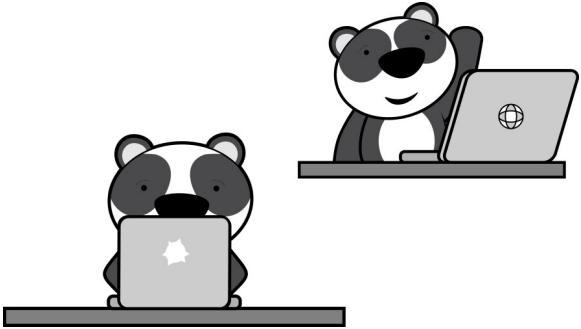
```
import pandas as pd  # Filename: dataframe_rename_column.py

city = pd.Series(['Beijing', 'Shanghai', 'Guangzhou'],
                 index=['A', 'B', 'C'])
population = pd.Series([21542000, 24183300, 14904000],
                       index=['A', 'B', 'C'])
area = pd.Series([16410.54, 6340.5, 7434.4],
                 index=['A', 'B', 'C'])
df = pd.DataFrame({'City': city,
                   'Population': population,
                   'Area (sq km)': area})

# Rename columns for brevity
df = df.rename(columns={'Population': 'Pop',
                        'Area (sq km)': 'Area'})
print(df)
```

Output:

	City	Pop	Area
A	Beijing	21542000	16410.54
B	Shanghai	24183300	6340.50
C	Guangzhou	14904000	7434.40

Part III

Computation



Unary Operations on Series and DataFrames

- Unary operations on Series and DataFrames work with any NumPy unary function.
- The specified operation is applied to each element of the Series or DataFrame.
- Broadcasting works in the same way.
- You can sum, subtract, multiply, or divide each element of a Series or a DataFrame by a scalar using the +, -, /, or * operators.

```python
# Filename: series_dataframe_unary_op.py
import pandas as pd

# Creating a DataFrame with temperature data
temperature = pd.Series([32.0, 75.0, 68.5],
                        index=['Monday', 'Tuesday', 'Wednesday'])
df = pd.DataFrame({'Temperature (F)': temperature})

# Example of unary operation:
# converting Fahrenheit to Celsius
df['Temperature (C)'] = (df['Temperature (F)'] - 32) * 5/9
print(df)
```

Output:

	Temperature (F)	Temperature (C)
Monday	32.0	0.000000
Tuesday	75.0	23.888889
Wednesday	68.5	20.277778

Operations between Series

- You can apply operations between two Series.
- The operation is applied element-wise after aligning indices.
- The index elements which do not match are set to NaN (i.e., not a number).
- After the alignment, the index in the result is sorted (only if they do not match).

```
# Filename: series_op.py
import pandas as pd

# Creating two Series with city populations
city_a = pd.Series([21540000, 10000000, 8000000],
            index=['Shanghai', 'Beijing', 'Chongqing'])
city_b = pd.Series([1000000, 2000000, 3000000],
            index=['Beijing', 'Chongqing', 'Guangzhou'])
result = city_a + city_b
print(result)
```

Output:

```
Beijing      11000000.0
Chongqing    10000000.0
Guangzhou           NaN
Shanghai            NaN
dtype: float64
```

Operations between DataFrames

- To perform operations with two DataFrames, you need to align both the index and the columns.
- The operation is applied element-wise after aligning indices and columns.
- If the columns are not aligned, NaN values are inserted in all the rows of the unaligned columns.

```python
import pandas as pd  # Filename: dataframes_op.py

# Creating two DataFrames with city populations and areas
df1 = pd.DataFrame([[21540000, 2400.0], [10000000, 1687.0],
                    [8000000, 315.0]],
                   columns=['Population', 'Area (sq mi)'],
                   index=['Shanghai', 'Beijing', 'Chongqing'])
print(df1)

df2 = pd.DataFrame([[1000000, 743.0], [2000000, 400.0]],
                   columns=['Population', 'Area (sq mi)'],
                   index=['Beijing', 'Guangzhou'])
print(df2)

print(df1 + df2) # Adding the two DataFrames
```

Output:

```
           Population  Area (sq mi)
Shanghai     21540000        2400.0
Beijing      10000000        1687.0
Chongqing     8000000         315.0
           Population  Area (sq mi)
Beijing       1000000         743.0
Guangzhou     2000000         400.0
           Population  Area (sq mi)
Beijing    11000000.0        2430.0
Chongqing         NaN           NaN
Guangzhou         NaN           NaN
Shanghai          NaN           NaN
```

Aggregations

- You can perform aggregate functions (for both Series and DataFrames) to compute the
  - mean using `df.mean()`
  - standard deviation using `df.std()`
  - minimum value using `df.min()`
  - maximum value using `df.max()`
  - sum using `df.sum()`

Aggregations for Series

- An aggregate function applied to a Series returns a single value representing the mean, sum, etc., of the Series elements.

```python
# Filename: series_aggregations.py
import pandas as pd

# Creating a Series with city populations
populations = pd.Series([21540000, 10000000, 8000000],
                        index=['Shanghai', 'Beijing', 'Chongqing'])

# Calculating the mean population
mean_population = populations.mean()
print(f'Mean Population: {mean_population}')
```

Output:

```
Mean Population: 13180000.0
```

Aggregations for DataFrames

- For DataFrames, aggregate functions are applied column-wise and return a Series with the mean, sum, etc., of each column separately.

```python
# Filename: dataframes_aggregations.py
import pandas as pd

# Creating a DataFrame with city population and area data
data = {
    'City': ['Shanghai', 'Beijing', 'Chongqing'],
    'Population': [21540000, 10000000, 8000000],
    'Area (sq mi)': [2400.0, 1687.0, 315.0]
}
df = pd.DataFrame(data)

# Calculating aggregate functions for the DataFrame
mean_values = df[['Population', 'Area (sq mi)']].mean()
print(mean_values)
```

Output:

```
Population      1.318000e+07
Area (sq mi)    1.467333e+03
dtype: float64
```

Handling Missing Values

- Missing values in Pandas are represented with sentinel values.
- They can be represented with the Python null value 'None' or the NumPy not a number 'np.nan'.
- The difference is that 'None' is a Python object, whereas 'np.nan' is a floating point number.
- Using 'np.nan' achieves better performance when performing numerical computations.
- Pandas supports both types and automatically converts between them when appropriate.

Checking Missing Elements

- You can check if a Series or a DataFrame contains missing values with the 'isnull()' method.
- It returns a boolean mask indicating missing values (i.e., a boolean mask with 'True' if the element is missing, 'False' otherwise).
- The opposite function is 'notnull()', which returns a boolean mask indicating non-missing values (i.e., 'True' if the element is not missing and 'False' otherwise).

```
# Filename: checking_missing.py
import numpy as np
import pandas as pd

# Creating a Series with missing values for city populations
s = pd.Series([21540000, None, 8000000, np.nan],
              index=['Shanghai', 'Beijing',
                     'Chongqing', 'Guangzhou'])
# Checking for missing values
null_mask = s.isnull()
print(null_mask)
```

Output:

```
Shanghai     False
Beijing       True
Chongqing    False
Guangzhou     True
dtype: bool
```

Remove Missing Elements

- You can remove missing elements with the 'dropna()' method.

```python
# Filename: remove_missing.py
import numpy as np
import pandas as pd

# Creating a Series with missing values for city populations
s = pd.Series([21540000, None, 8000000, np.nan],
              index=['Shanghai', 'Beijing',
                     'Chongqing', 'Guangzhou'])
print("Original Series:")
print(s)
print("\nSeries after dropping missing values:")
print(s.dropna())
print()

# Creating a DataFrame with missing values for city statistics
df = pd.DataFrame({'Population': [21540000, 10000000, None],
                   'Area (sq mi)': [2400.0, np.nan, 315.0]},
                  index=['Shanghai', 'Beijing', 'Chongqing'])
print("Original DataFrame:")
print(df)
print("\nDataFrame after dropping missing values:")
print(df.dropna())
```

Output:

```
Original Series:
Shanghai     21540000.0
Beijing             NaN
Chongqing     8000000.0
Guangzhou           NaN
dtype: float64

Series after dropping missing values:
Shanghai     21540000.0
Chongqing     8000000.0
dtype: float64

Original DataFrame:
           Population  Area (sq mi)
Shanghai   21540000.0        2400.0
Beijing    10000000.0           NaN
Chongqing         NaN         315.0

DataFrame after dropping missing values:
           Population  Area (sq mi)
Shanghai   21540000.0        2400.0
```

Filling Missing Values

- You can fill missing values with the mean of the column using the 'fillna()' method.

```python
# Filename: fill_missing_values.py
import numpy as np
import pandas as pd

# Create a Series with missing values
s = pd.Series([10, 20, None, 30, np.nan])
mean_value = s.mean()
print(s.fillna(mean_value))
print()

# Create a DataFrame with missing values
df = pd.DataFrame({'Sales': [100, 200, 300],
                   'Profit': [20, np.nan, 50]},
                  index=['Q1', 'Q2', 'Q3'])
mean_profit = df['Profit'].mean()
print(df.fillna(mean_profit))
```

Output:

```
0    10.0
1    20.0
2    20.0
3    30.0
4    20.0
dtype: float64

    Sales  Profit
Q1    100    20.0
Q2    200    35.0
Q3    300    50.0
```

Grouping Data Inside a DataFrame

- Pandas allows you to analyze data by grouping it, aggregating values (e.g., mean, sum, count), and filtering based on conditions.
- The 'groupby()' method returns a DataFrameGroupBy object.
- You need to specify the column(s) to group by (the key).

```python
# Filename: dataframe_grouping_data1.py
import pandas as pd

# Create a DataFrame with sales data
df = pd.DataFrame({'Region': ['North', 'South', 'North', 'South'],
                   'Sales': [250, 150, 300, 200],
                   'Profit': [50, 30, 70, 40]})
groupedDf = df.groupby('Region') # 2 groups: 'North' and 'South'

for key, groupDf in groupedDf:
    print(key)
    print(groupDf)
```

Output:

```
North
  Region  Sales  Profit
0  North    250      50
2  North    300      70
South
  Region  Sales  Profit
1  South    150      30
3  South    200      40
```

Grouping Data Inside a DataFrame

```python
# Filename: dataframe_grouping_data2.py
import pandas as pd

# Create a DataFrame with employee data
df = pd.DataFrame({'Department': ['HR', 'IT', 'HR', 'IT'],
                   'Salary': [50000, 60000, 55000, 70000],
                   'Experience': [2, 5, 3, 7]})

# 2 groups: 'HR' and 'IT'
groupedDf = df.groupby('Department')

# Mean salary and experience, separately for each group
result = groupedDf.mean().reset_index()
print(result)
```

Output:

```
  Department   Salary  Experience
0         HR  52500.0         2.5
1         IT  65000.0         6.0
```

Loading Data from a CSV File

- You can load a DataFrame from a CSV file using the `pandas` library.
- Specify the delimiter using the `sep` parameter (e.g., `sep=";"` for semicolon-separated files).
- The function automatically reads the header from the first line of the file and can skip rows using `skiprows` (e.g., `skiprows=1`).
- Column data types are inferred automatically, allowing for easy data manipulation.
- If the file contains missing values, you can specify how to recognize them. By default, empty fields are converted to NaN (not a number), and the string `'NaN'` is recognized as a null value.

Loading Data from a CSV File

```python
# Filename: csv_loading_data.py
import pandas as pd

# Load data from a CSV file with specific NA values
df = pd.read_csv('./data/employees.csv',
                 sep=',',
                 skiprows=0,
                 na_values=['N/A', 'Missing'])
print(df)
```

File: employees.csv

```
Name,Department,Salary,Experience
Alice,HR,60000,N/A
Bob,IT,75000,5
Charlie,Finance,80000,7
Diana,IT,90000,Missing
Eve,HR,N/A,3
```

Output:

```
      Name Department   Salary Experience
0    Alice         HR    60000        NaN
1      Bob         IT    75000        5.0
2  Charlie    Finance    80000        7.0
3    Diana         IT    90000        NaN
4      Eve         HR      NaN        3.0
```

Saving Data to a CSV File

- You can save an existing DataFrame to a CSV file using the `to_csv()` method.
- If you specify `index=False` as a parameter, it prevents writing the index to the file.

```python
# Filename: csv_saving_data.py
import pandas as pd
import os

# Sample DataFrame with employee data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve'],
    'Department': ['HR', 'IT', 'Finance', 'IT', 'HR'],
    'Salary': [60000, 75000, 80000, 90000, None],
    'Experience': [None, 5, 7, None, 3]
}
df = pd.DataFrame(data)

# Define the path to save the CSV file
path = os.path.join('./data', 'employees2.csv')
df.to_csv(path, sep=',', index=False)
```

File: employees2.csv

```
Name,Department,Salary,Experience
Alice,HR,60000.0,
Bob,IT,75000.0,5.0
Charlie,Finance,80000.0,7.0
Diana,IT,90000.0,
Eve,HR,,3.0
```

Key Terms

- Aggregation
- DataFrame
- Fancy indexing
- Label-based indexing
- Pandas
- Position-based indexing
- Masking
- Missing values
- Series
- Structured data

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. The _____ library is a powerful Python library designed for _____ and _____.
2. Two primary data structures in Pandas are _____ and _____.
3. Key functionalities of Pandas include _____, _____, and _____.
4. A Pandas _____ is a one-dimensional labeled array capable of holding any data type.
5. Each element in a Series is associated with an _____.

Answer: 1. Pandas; data manipulation; analysis, 2. Series (1D); DataFrame (2D), 3. data filtering; aggregation; reshaping with pivot tables, 4. Series, 5. index.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

6. You can create a Series from _____, _____, or _____.
7. To access elements in a Series by their label, you use the _____ attribute.
8. You can retrieve the _____ and _____ of a Series using the _____ and _____ attributes.
9. You can assign new values to Series elements using _____ and _____.
10. To slice a Pandas Series, you can use _____ for label-based indexing and _____ for position-based indexing.

Answer: 6. list; dictionaries; NumPy arrays, 7. Series.loc[ ], 8. values; index; .values; .index, 9. .loc[ ]; iloc[ ], 10. .loc[ ]; .iloc[ ].

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

11. Masking in Pandas creates a _____ where _____ indicates that a condition is satisfied.
12. You can access a DataFrame column by specifying the column name in _____.
13. To group data in a DataFrame, you use the _____ method.
14. You can load a DataFrame from a CSV file using the _____ function.
15. To save a DataFrame to a CSV file, you can use the _____ method.

Answer: 11. boolean Series; True, 12. square brackets [ ], 13. groupby(), 14. pandas.read_csv(), 15. to_csv.

That's all!

Any question?