Pandas Notes for ProgSD Exam

Contents

Chapter 1 Pandas

1.1 Imports

- To use Pandas effectively, it is crucial to import the library using standard conventions.
- The most common imports for Pandas are:
  - `import pandas as pd`: This is the standard convention for importing Pandas. The alias `pd` is widely recognized and saves typing.
  - `import numpy as np`: Often used alongside Pandas for numerical computations, such as generating or manipulating numerical data.

Example 1: Standard Pandas Import.

```
# Import Pandas with standard alias
import pandas as pd

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

Example 2: Using NumPy with Pandas.

```
# Import Pandas and NumPy
import pandas as pd
import numpy as np

# Create a DataFrame with random data
data = np.random.rand(3, 3)
df = pd.DataFrame(data, columns=['A', 'B', 'C'])
print(df)
```

1.2 Introduction to Pandas

- **Key Features:**
  - **Series:** 1D labeled array.
  - **DataFrame:** 2D labeled table with rows and columns.
- Installation: Use Anaconda or `pip install pandas`.

Code:

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

1.3 The Series

- **Creating:** Use `pd.Series(data, index=[])` to create a labeled 1D array.
- **Operations:** Supports arithmetic operations (e.g., `+`, `-`) with alignment based on labels.
- **Selecting Elements:**
  - Use labels or indices to access elements: `s['label']` or `s[index]`.
  - Use slicing for ranges: `s[start:stop]`.
- **Filtering:** Use conditional statements to filter values (e.g., `s[s > 5]`).
- **Assigning Values:**
  - Assign values by label: `s['label'] = value`.
  - Assign values by index: `s[index] = value`.
- **Mathematical Operations:**
  - Operators like `+`, `-`, `*`, `/` are applicable to Series.
  - Mathematical functions from NumPy can also be applied.
- **Missing Data:**
  - `np.NaN`: Represents missing values.
  - `isnull()`, `notnull()`: Identify missing or non-missing data.
- **Evaluating Values:**
  - `unique()`: Returns unique values, excluding duplicates.
  - `value_counts()`: Counts occurrences of unique values.
  - `isin()`: Checks if values exist in the Series and returns a boolean mask.
- **Creating a Series from a Dictionary:**
  - Keys become the index, and values become the data.
  - Missing indices are filled with `np.NaN`.
- **Operations Between Series:** Aligns on labels; mismatched labels result in `NaN`.

Example 1: Evaluating Values
Code:

```
# Create a Series with duplicate values
serd = pd.Series([1, 0, 2, 1, 2, 3],
       index=['white', 'white', 'blue',
       'green', 'green', 'yellow'])

print("Unique values:", serd.unique())
print("Value counts:")
print(serd.value_counts())

# Check if values are in the Series
print("isin for values [0,3]:")
print(serd.isin([0, 3]))
```

Output:

```
Unique values:
[1 0 2 3]

Value counts:
2    2
1    2
3    1
0    1
dtype: int64

isin for values [0,3]:
white     False
white      True
blue      False
green     False
green     False
yellow     True
dtype: bool
```

Example 2: Handling Missing Data (NaN Values)
Code:

```
# Create a Series with missing values
s2 = pd.Series([5, -3, np.NaN, 20])

print("isnull():")
print(s2.isnull())

print("notnull():")
print(s2.notnull())

# Filter non-missing values
print("Filtered non-missing values:")
print(s2[s2.notnull()])
```

Output:

```
isnull():
0    False
1    False
2     True
3    False
dtype: bool

notnull():
0     True
1     True
2    False
3     True
dtype: bool

Filtered non-missing
    values:
0     5.0
1    -3.0
3    20.0
dtype: float64
```

Example 3: Creating a Series from a Dictionary
Code:

```
# Create a dictionary and convert to
    Series
mydict = {'red': 250, 'blue': 560,
         'green': 700, 'white': 1456}
myseries = pd.Series(mydict)

# Create a Series with missing values
colours = ['red', 'blue', 'green',
          'white', 'purple']
myseries_with_nan = pd.Series(mydict,
    index=colours)

print("Series from dictionary:")
print(myseries)

print("Series with custom index:")
print(myseries_with_nan)
```

Output:

```
Series from dictionary:
red      250
blue     560
green    700
white   1456
dtype: int64

Series with custom index:
red      250.0
blue     560.0
green    700.0
white   1456.0
purple     NaN
dtype: float64
```

Example 4: Operations Between Series
Code:

```
# Create another Series
mydict2 = {'red': 900, 'black': 800,
          'white': 500}
myseries2 = pd.Series(mydict2)

# Add two Series
result = myseries + myseries2

print("Result of adding two Series:")
print(result)
```

Output:

```
Result of adding two Series:
black      NaN
blue       NaN
green      NaN
purple     NaN
red     1150.0
white   1956.0
dtype: float64
```

1.4 The DataFrame

- **Definition:** A DataFrame is a tabular structure very similar to a spreadsheet, designed to extend Series to multiple dimensions.
- **Structure:**
  - Consists of an ordered collection of columns, each of which can contain a value of a different type (numeric, string, boolean, etc.).
  - Unlike Series, which have an index array, DataFrames have two index arrays for rows and columns.
  - Can be understood as a dictionary of Series where the keys are column names, and the values are the Series forming the DataFrame's columns.
- **Creating a DataFrame:** Use `pd.DataFrame()` with data in the form of dictionaries, lists, or nested dictionaries.
- **Selecting Columns:** Use the `columns` parameter to select and order specific columns.
- **Nested Dictionary:** When a nested dictionary is passed to `pd.DataFrame()`, outer keys become column names, and inner keys become row labels. Missing values are filled with `NaN`.
- **Transposition:** Columns become rows and rows become columns using the `.T` attribute.

Example 1: Defining a DataFrame
Code:

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

# Define data
data = {'color': ['white', 'red',
                  'black', 'green',
                  'purple'],
        'items': ['ball', 'pen',
                  'pencil', 'paper',
                  'eraser'],
        'price': [2.5, 1.5, 0.5,
                  0.6, 0.15]}

# Create a DataFrame
frame = pd.DataFrame(data)
print("DataFrame:")
print(frame)
```

Output:

```
DataFrame:
    color  items  price
0   white   ball   2.50
1     red    pen   1.50
2   black pencil   0.50
3   green  paper   0.60
4  purple eraser   0.15
```

Example 2: Selecting Specific Columns
Code:

```python
# Select specific columns
frame2 = pd.DataFrame(data,
    columns=['items', 'price'])

print("DataFrame with selected
    columns:")
print(frame2)
```

Output:

```
DataFrame with selected columns:
    items  price
0    ball   2.50
1     pen   1.50
2  pencil   0.50
3   paper   0.60
4  eraser   0.15
```

Example 3: Nested Dictionary as Input
Code:

```
# Define a nested dictionary
nested_dict = {'red': {2012: 22,
                      2014: 45},
              'green': {2008: 23,
                       2012: 22,
                       2014: 17},
              'blue': {2008: 18,
                      2012: 28,
                      2014: 19}}

# Create a DataFrame
frame3 =
    pd.DataFrame(nested_dict)

print("DataFrame from nested
    dictionary:")
print(frame3)
```

Output:

```
DataFrame from nested dictionary:
      red  green  blue
2008   NaN   23.0  18.0
2012  22.0   22.0  28.0
2014  45.0   17.0  19.0
```

Example 4: Transposing a DataFrame
Code:

```
# Transpose the DataFrame
frame3_T = frame3.T

print("Transposed DataFrame:")
print(frame3_T)
```

Output:

```
Transposed DataFrame:
       2008  2012  2014
red     NaN  22.0  45.0
green  23.0  22.0  17.0
blue   18.0  28.0  19.0
```

1.5 Reading and Writing Data

1.5.1 Reading Data in Parts

- To read only a portion of the file, specify the number of lines to parse using `nrows` and `skiprows`.
- `skiprows`: Excludes specified rows from being read.
- `nrows`: Reads only the specified number of rows.

Example 1: Reading a CSV File in Parts
Code:

```
import pandas as pd

# Reading a file while skipping and
    limiting rows
partial_data =
    pd.read_csv('Documents/texting.csv',
                         skiprows=[2],
                             nrows=3,
                             header=None)

print("Data read with skiprows and
    nrows:")
print(partial_data)
```

Output:

```
Data read with skiprows and nrows:
       0    1    2    3       4
0  white  red  blue  green  animal
1      1    5     2      3     car
2      3    3     6      7   horse
```

1.5.2 Writing Data to CSV

- `to_csv()`: Writes a DataFrame to a CSV file.
- Use `index=False` and `header=False` to remove default indexes and headers.

Example 2: Writing Data to a CSV File
Code:

```
import pandas as pd
import numpy as np

# Create a DataFrame
frame = pd.DataFrame(np.arange(16).reshape((4, 4)),
                     index=['red', 'blue', 'yellow', 'white'],
                     columns=['ball', 'pen',
                            'pencil', 'paper'])

# Write to CSV
frame.to_csv('writin.txt', index=False, header=False)

# Read back the data
read_frame = pd.read_csv('writin.txt', header=None)

print("DataFrame after writing and reading:")
print(read_frame)
```

Output:

```
DataFrame after writing and reading:
     0     1    2    3
0  red     0    1    2
1  blue    4    5    6
2  yellow 8    9   10
3  white  12   13   14
```

1.5.3 Reading and Writing HTML and Excel Files

- `to_html()`: Converts a DataFrame to an HTML table.
- `read_html()`: Reads tables from an HTML file and returns a list of DataFrames.
- `to_excel()`: Writes a DataFrame to an Excel spreadsheet.
- `read_excel()`: Reads data from an Excel file into a DataFrame.

Example 3: Writing and Reading HTML
Code:

```
# Create a DataFrame
frame = pd.DataFrame(np.arange(4).reshape(2, 2))

# Convert to HTML
html_output = frame.to_html()
print("HTML output:")
print(html_output)
```

Output:

```
HTML output:
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>0</td>
      <td>1</td>
    </tr>
    <tr>
      <th>1</th>
      <td>2</td>
      <td>3</td>
    </tr>
  </tbody>
</table>
```

1.6 Using Regex to Parse TXT Files

- Sometimes, files do not have clear separators such as commas or semicolons for parsing.
- Regular expressions (**Regex**) can be used to define custom criteria for value separation.
- Common Regex patterns:

Pattern	Description
`.`	Single character, except newline
`\d`	Digit
`\D`	Non-digit character
`\s`	Whitespace character
`\S`	Non-whitespace character
`\n`	Newline character
`\t`	Tab character
`\uxxxx`	Unicode character specified by the hexadecimal number `xxxx`

Example 1: Parsing a TXT File with Whitespace
Code:

```
import pandas as pd

# Parse a text file using whitespace as a delimiter
df = pd.read_table('Documents/texting.txt',
       sep='\s+',
       engine='python')

print("Data parsed using whitespace:")
print(df)
```

Output:

```
Data parsed using whitespace:
   white  red  blue  green
0      1    5     2      3
1      2    7     8      5
2      3    3     6      7
3      2    2     8      3
4      4    4     2      1
```

Example 2: Extracting Numeric Data from TXT File
Code:

```
# Extract numeric parts using a regex
df_numeric = pd.read_table('Documents/texting.txt',
       sep='\D+', header=None, engine='python')

print("Numeric data extracted from text:")
print(df_numeric)
```

Output:

```
Numeric data extracted from text:
   0    1    2    3
0  NaN  NaN  NaN  NaN   # First line had no numeric values
1  1.0  5.0  2.0  3.0
2  2.0  7.0  8.0  5.0
3  3.0  3.0  6.0  7.0
4  2.0  2.0  8.0  3.0
5  4.0  4.0  2.0  1.0

# To avoid NaN add 'skiprows=1'
```

Example 3: Skipping Rows While Parsing
Code:

```
# Skip specified rows while parsing
df_skip = pd.read_table('Documents/texting.txt',
        sep='\D+', header=None, engine='python',
            skiprows=[0, 1, 2])

print("Data after skipping rows:")
print(df_skip)
```

Output:

```
Data after skipping rows:
   0  1  2  3
0  3  3  6  7
1  2  2  8  3
2  4  4  2  1
```

1.7 Interacting with Databases

1.7.1 Overview

- `pandas.io.sql` module provides a unified interface independent of the database, using `sqlalchemy`.
- The `create_engine()` function is used to establish a connection to the database.
- Unified commands ensure consistency regardless of the database backend.

Example 1: Creating a Connection to Databases
Code:

```python
from sqlalchemy import create_engine

# For SQLite
engine_sqlite =
    create_engine('sqlite:///foo.db')
```

1.7.2 Interacting with SQLite Databases

- Create a DataFrame that will serve as a table in the SQLite database.
- Use `to_sql()` to write the DataFrame to the database.
- Use `read_sql()` to retrieve data from the database.

Example 2: SQLite Integration
Code:

```
import pandas as pd
import numpy as np
from sqlalchemy import create_engine

# Create a DataFrame
frame = pd.DataFrame(np.arange(20).reshape(4, 5),
                     columns=['white', 'red', 'blue',
                         'black', 'green'])
print("DataFrame:")
print(frame)

# Connect to SQLite database
engine = create_engine('sqlite:///foo.db')

# Write the DataFrame to the database
frame.to_sql('colors', engine, if_exists='replace')

# Read data back from the database
retrieved_data = pd.read_sql('colors', engine)
print("Data retrieved from SQLite:")
print(retrieved_data)
```

Output:

```
DataFrame:
   white  red  blue  black  green
0      0    1     2      3      4
1      5    6     7      8      9
2     10   11    12     13     14
3     15   16    17     18     19

Data retrieved from SQLite:
   index  white  red  blue  black  green
0      0      0    1     2      3      4
1      1      5    6     7      8      9
2      2     10   11    12     13     14
3      3     15   16    17     18     19
```

1.8 Pandas - Data Manipulation

1.9 Merging Data

- **merge():**
  - Combines data from two DataFrames based on keys (e.g., columns or indexes).
  - Default behavior merges based on common column names.
  - Specify the `on` parameter to define custom merge keys.
- **join():**
  - Used to merge data using indexes.
  - More convenient for merging when indexes are used as keys.
- **Options:**
  - `left_index` and `right_index`: Use indexes as merge keys.
  - `how`: Defines merge type (`'inner'`, `'outer'`, `'left'`, `'right'`).

Example 1: Simple Merge
Code:

```
import pandas as pd

# Create first DataFrame
frame1 = pd.DataFrame({
    'id': ['ball', 'pencil', 'pen', 'mug', 'ashtray'],
    'price': [12.33, 11.44, 33.21, 13.23, 33.62]
})

# Create second DataFrame
frame2 = pd.DataFrame({
    'id': ['pencil', 'pencil', 'ball', 'pen'],
    'color': ['white', 'red', 'red', 'black']
})

# Merge the DataFrames
merged = pd.merge(frame1, frame2)
print(merged)
```

Output:

```
       id  price  color
0    ball  12.33    red
1  pencil  11.44  white
2  pencil  11.44    red
3     pen  33.21  black
```

Example 2: Specifying Merge Key
Code:

```
# Create additional DataFrames
frame3 = pd.DataFrame({
    'id': ['ball', 'pending', 'pen', 'mug', 'ashtray'],
    'color': ['white', 'red', 'red', 'black', 'green'],
    'brand': ['OMG', 'ABC', 'ABC', 'POD', 'POD']
})

frame4 = pd.DataFrame({
    'id': ['pencil', 'pencil', 'ball', 'pen'],
    'brand': ['OMG', 'POD', 'ABC', 'POD']
})

# Merge on specific key
merged_on_brand = pd.merge(frame3, frame4, on='brand')
print(merged_on_brand)
```

Output:

```
      id_x   color brand  id_y
0    ball   white   OMG pencil
1  pending    red   ABC   ball
2     pen     red  POD   pen
3     mug   black   POD   pen
4  ashtray  green   POD   pen
```

Example 3: Using Indexes
Code:

```
# Merge using indexes
merged_with_indexes = pd.merge(frame3, frame4,
                              left_index=True,
                              right_index=True)
print(merged_with_indexes)

# Using join()
frame4.columns = ['brand2', 'id2']
joined = frame3.join(frame4)

print(joined)
```

Output:

```
Merged with Indexes:
      id_x  color brand_x brand_y
0     ball  white  OMG     OMG
1  pending    red  ABC     POD
2      pen    red  ABC     POD
3      mug  black  POD     POD

Using join():
      id   color brand brand2  id2
0    ball   white   OMG pencil  OMG
1  pending    red   ABC pencil  POD
2     pen    red   ABC   ball   POD
3     mug   black   POD    pen   POD
4  ashtray  green   POD    NaN   NaN
```

1.10 Concatenation

1.10.1 Concatenating Arrays

- **Definition:** Concatenation combines arrays along a specified axis.
- `np.concatenate()`: Combines two or more arrays along a given axis.

Code:

```
import numpy as np

# Create two 3x3 arrays
array1 = np.arange(9).reshape((3, 3))
array2 = np.arange(9).reshape((3, 3)) + 6

# Concatenate along axis 1
result_axis1 = np.concatenate([array1, array2], axis=1)
print("Concatenation along axis 1:")
print(result_axis1)

# Concatenate along axis 0
result_axis0 = np.concatenate([array1, array2], axis=0)
print("Concatenation along axis 0:")
print(result_axis0)
```

Output:

```
Concatenation along axis 1:
[[ 0  1  2  6  7  8]
 [ 3  4  5  9 10 11]
 [ 6  7  8 12 13 14]]

Concatenation along axis 0:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

1.10.2 Concatenating Series

- `pd.concat()`: Combines multiple Series objects, with options for hierarchical indexing using the `keys` parameter.

Code:

```
import pandas as pd
import numpy as np

# Create two Series
ser1 = pd.Series(np.random.rand(4),
    index=[1, 2, 3, 4])
ser2 = pd.Series(np.random.rand(4),
    index=[5, 6, 7, 8])

# Concatenate Series
combined = pd.concat([ser1, ser2])
print("Concatenated Series:")
print(combined)

# Hierarchical indexing with keys
combined_hierarchical =
    pd.concat([ser1, ser2],
    keys=["Group1", "Group2"])
print("Hierarchical concatenation:")
print(combined_hierarchical)
```

Output:

```
Concatenated Series:
1    0.326100
2    0.983239
3    0.306811
4    0.149875
5    0.221997
6    0.687002
7    0.499663
8    0.857193
dtype: float64

Hierarchical concatenation:
Group1  1    0.326100
        2    0.983239
        3    0.306811
        4    0.149875
Group2  5    0.221997
        6    0.687002
        7    0.499663
        8    0.857193
dtype: float64
```

1.10.3 Concatenating DataFrames

- **Concatenating DataFrames:** The same logic of concatenating Series applies to DataFrames.
- Use the `axis` parameter to specify concatenation direction.

Code:

```python
# Create two DataFrames
frame1 = pd.DataFrame(np.random.rand(9).reshape(3, 3),
                      columns=["A", "B", "C"],
                      index=[1, 2, 3])
frame2 = pd.DataFrame(np.random.rand(9).reshape(3, 3),
                      columns=["A", "B", "C"],
                      index=[4, 5, 6])

# Concatenate along axis 0
combined_df0 = pd.concat([frame1, frame2])
print("Concatenated along axis 0:")
print(combined_df0)

# Concatenate along axis 1
combined_df1 = pd.concat([frame1, frame2], axis=1)
print("Concatenated along axis 1:")
print(combined_df1)
```

Output:

```
Concatenated along axis 0:
          A         B         C
1  0.976314 0.748882 0.955794
2  0.046396 0.449692 0.867622
3  0.433338 0.986343 0.323115
4  0.802874 0.773448 0.922387
5  0.580696 0.584984 0.276520
6  0.725205 0.017955 0.974704

Concatenated along axis 1:
          A         B         C         A         B         C
1  0.976314 0.748882 0.955794       NaN       NaN       NaN
2  0.046396 0.449692 0.867622       NaN       NaN       NaN
3  0.433338 0.986343 0.323115       NaN       NaN       NaN
4       NaN       NaN       NaN  0.802874 0.773448 0.922387
5       NaN       NaN       NaN  0.580696 0.584984 0.276520
6       NaN       NaN       NaN  0.725205 0.017955 0.974704
```

1.11 Combining

- **Definition:** When neither merging nor concatenation achieves the desired result, combining can be used.
- **combine_first():** This function combines two Series or DataFrames, using the values from the calling object if they exist; otherwise, it takes values from the passed object.
- **Use Case:** Useful for combining datasets with partially or entirely overlapping indexes.

Code:

```
import pandas as pd
import numpy as np

# Create two Series
ser1 =
    pd.Series(np.random.rand(5),
    index=[1, 2, 3, 4, 5])
ser2 =
    pd.Series(np.random.rand(4),
    index=[2, 4, 5, 6])

# Combine ser1 with ser2
combined_ser1_first =
    ser1.combine_first(ser2)

print("Combining ser1 with
    ser2:")
print(combined_ser1_first)

# Combine ser2 with ser1
combined_ser2_first =
    ser2.combine_first(ser1)

print("Combining ser2 with
    ser1:")
print(combined_ser2_first)
```

Output:

```
Combining ser1 with
    ser2:
1    0.598546
2    0.172542
3    0.738250
4    0.682647
5    0.013372
6    0.107031
dtype: float64

Combining ser2 with
    ser1:
1    0.598546
2    0.504086
3    0.738250
4    0.421815
5    0.970975
6    0.107031
dtype: float64
```

1.12 Pivoting

- **Definition:** Pivoting is the process of rearranging or reorganizing data by converting columns into rows and vice versa.
- **Operations:**
  - `stack()`: Rotates or pivots the data structure, converting columns to rows.
  - `unstack()`: Converts rows back to columns.
- **Use Case:** Useful for restructuring datasets for better readability and understanding.

Example 1: Stacking and Unstacking
Code:

```
import pandas as pd
import numpy as np

# Create DataFrame
frame1 = pd.DataFrame(
    np.arange(9).reshape(3, 3),
    index=['white', 'black', 'red'],
    columns=['ball', 'pen', 'pencil']
)

# Stack the DataFrame
ser5 = frame1.stack()
print("Stacked DataFrame:")
print(ser5)

# Unstack the DataFrame
print("\nUnstacked DataFrame:")
print(ser5.unstack())

# Unstack with a different level
print("\nUnstacked with level 0:")
print(ser5.unstack(0))
```

Output:

```
Stacked DataFrame:
white  ball      0
       pen       1
       pencil    2
black  ball      3
       pen       4
       pencil    5
red    ball      6
       pen       7
       pencil    8
dtype: int32

Unstacked DataFrame:
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8

Unstacked with level 0:
       white  black  red
ball       0      3    6
pen        1      4    7
pencil     2      5    8
```

Example 2: Pivoting from Long to Wide Format
Code:

```python
# Create a long DataFrame
longframe = pd.DataFrame({
    'color': ['white', 'white', 'white',
              'red', 'red', 'red',
              'black', 'black', 'black'],
    'item': ['ball', 'pen', 'mug',
            'ball', 'pen', 'mug',
            'ball', 'pen', 'mug'],
    'value': np.random.rand(9)
})

print("Long format DataFrame:")
print(longframe)

# Pivot to wide format
widetable = longframe.pivot(index='color', columns='item',
    values='value')

print("\nWide format DataFrame:")
print(widetable)
```

Output:

```
Long format DataFrame:
   color  item     value
0  white  ball  0.587818
1  white   pen  0.490479
2  white   mug  0.912572
3    red  ball  0.423560
4    red   pen  0.446265
5    red   mug  0.711930
6  black  ball  0.524044
7  black   pen  0.812680
8  black   mug  0.541409

Wide format DataFrame:
item       ball       mug       pen
color
black  0.524044 0.541409 0.812680
red    0.423560 0.711930 0.446265
white  0.587818 0.912572 0.490479
```

1.13 Removing Data

1.13.1 Removing Columns and Rows

- **Definition:** Columns and rows can be removed from a DataFrame using specific commands.
- **Removing Columns:** Use the `del` command with the column name to remove a specific column from the DataFrame.
- **Removing Rows:** Use the `drop()` function with the label of the corresponding index to remove a specific row.

Code:

```
import pandas as pd
import numpy as np

# Creating a DataFrame
frame1 = pd.DataFrame(
    np.arange(9).reshape(3, 3),
    index=['white', 'black', 'red'],
    columns=['ball', 'pen', 'pencil']
)
print("Initial DataFrame:")
print(frame1)

# Removing a column
del frame1['ball']
print("After removing column 'ball':")
print(frame1)

# Removing a row
frame1 = frame1.drop('white')
print("After removing row 'white':")
print(frame1)
```

Output:

```
Initial DataFrame:
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8

After removing column 'ball':
       pen  pencil
white    1       2
black    4       5
red      7       8

After removing row 'white':
       pen  pencil
black    4       5
red      7       8
```

1.13.2 Removing Duplicates

- **Definition:** Identifying and removing duplicate rows in a DataFrame can clean the dataset and avoid redundancy.
- `duplicated()`: Returns a boolean Series indicating whether each row is a duplicate.
- `drop_duplicates()`: Removes duplicate rows and returns a DataFrame without duplicates.

Code:

```python
import pandas as pd

# Creating a DataFrame with duplicates
dframe = pd.DataFrame({'color': ['white', 'white',
                        'red', 'red', 'white']})

print("Initial DataFrame:")
print(dframe)

# Detect duplicates
duplicates = dframe.duplicated()
print("Duplicate rows detected:")
print(duplicates)

# Remove duplicates
dframe_no_duplicates = dframe.drop_duplicates()

print("DataFrame after removing duplicates:")
print(dframe_no_duplicates)
```

Output:

```
Initial DataFrame:
   color
0  white
1  white
2    red
3    red
4  white

Duplicate rows detected:
0    False
1     True
2    False
3     True
4     True
dtype: bool

DataFrame after removing duplicates:
   color
0  white
2    red
```

1.14 Mapping and Replacing

1.14.1 Definition

Mapping involves creating associations between values using key-value pairs, enabling transformations or additions to data based on predefined mappings.

1.14.2 Replacing Values

- `replace()`: Replaces values in a DataFrame or Series based on a specified mapping.

Code:

```
import pandas as pd

# Create a DataFrame
frame = pd.DataFrame({
    'item': ['ball', 'mug', 'pen', 'pencil', 'ashtray'],
    'color': ['white', 'rosso', 'verde', 'black', 'yellow'],
    'price': [5.56, 4.20, 1.30, 0.56, 2.75]
})

# Define the mapping
newcolors = {'rosso': 'red', 'verde': 'green'}

# Replace incorrect color values
frame['color'] = frame['color'].replace(newcolors)
print(frame)
```

Output:

```
      item   color  price
0     ball   white   5.56
1      mug     red   4.20
2      pen   green   1.30
3   pencil   black   0.56
4  ashtray  yellow   2.75
```

1.14.3 Handling Missing Values

- `replace()`: Replace missing values (`NaN`) with specified values.

Code:

```
import numpy as np
import pandas as pd

# Create a Series with NaN values
ser = pd.Series([1, 3, np.nan, 4, 6, np.nan, 3])

# Replace NaN values with 0
ser_filled = ser.replace(np.nan, 0)
print(ser_filled)
```

Output:

```
0    1.0
1    3.0
2    0.0
3    4.0
4    6.0
5    0.0
6    3.0
dtype: float64
```

1.14.4 Adding New Columns via Mapping

- `map()`: Adds a new column to a DataFrame by mapping values from another column to predefined values in a dictionary.

Code:

```
# Define a mapping for item prices
prices = {
    'ball': 5.26, 'mug': 4.20, 'pen': 1.30,
    'pencil': 0.56, 'ashtray': 2.75
}

# Map prices to the 'price' column
frame['price'] = frame['item'].map(prices)
print(frame)
```

Output:

```
      item   color  price
0     ball   white   5.26
1      mug     red   4.20
2      pen   green   1.30
3   pencil   black   0.56
4  ashtray  yellow   2.75
```

1.15 Data Aggregation

- **Definition:** The final stage of data manipulation involving the transformation of data into aggregated values like sums, means, or other metrics.
- **GroupBy:** A versatile tool in pandas for data aggregation, split into three phases:
  - **Splitting:** Divide data into groups based on key columns.
  - **Applying:** Apply a function to each group.
  - **Combining:** Combine the results into a single structure.

1.15.1 Grouping to a Single Column of Data

Code:

```
import pandas as pd

# Create a DataFrame
frame = pd.DataFrame({
    'color': ['white', 'red', 'green', 'red', 'green'],
    'object': ['pen', 'pencil', 'pen', 'ashtray', 'pencil'],
    'price1': [5.56, 4.20, 1.30, 0.56, 2.75],
    'price2': [4.75, 4.12, 1.60, 0.75, 3.15]
})

# Group data by 'color' column
group = frame['price1'].groupby(frame['color'])

# Apply aggregation functions
mean_price = group.mean()
sum_price = group.sum()

print("Group Mean:")
print(mean_price)
print("\nGroup Sum:")
print(sum_price)
```

Output:

```
Group Mean:
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64

Group Sum:
color
green    4.05
red      4.76
white    5.56
Name: price1, dtype: float64
```

1.15.2 Hierarchical Grouping

Code:

```
# Group data by multiple columns
ggroup = frame['price1'].groupby([frame['color'],
    frame['object']])

# Apply aggregation functions
group_sum = ggroup.sum()

print("Hierarchical Grouping Sum:")
print(group_sum)

# Multiple column aggregation
group_mean = frame[['price1',
    'price2']].groupby(frame['color']).mean()

print("\nGroup Mean of Multiple Columns:")
print(group_mean)
```

Output:

```
Hierarchical Grouping Sum:
color  object
green  pencil    1.30
       pen       2.75
red    ashtray   0.56
       pencil    4.20
white  pen       5.56
Name: price1, dtype: float64

Group Mean of Multiple Columns:
       price1  price2
color
green   2.025   2.375
red     2.380   2.435
white   5.560   4.750
```

1.16 Date Formatting and Parsing

1.16.1 `pd.read_csv()`

- The `pd.read_csv()` function is a core function in pandas for loading CSV files into a DataFrame.
- Provides multiple options for customizing the way data is read.

Code:

```
# Basic syntax for reading a CSV file
df = pd.read_csv('filename.csv')
```

1.16.2 Parsing Dates Using `parse_dates`

- Ensures that date columns are interpreted as `datetime` objects rather than strings.
- Enables operations like filtering, extracting specific time periods, and plotting time-series data.
- Use the `parse_dates` argument to specify columns to convert into `datetime` objects automatically.

Code:

```
# Parse the 'Date' column as datetime objects
df = pd.read_csv('shopping.csv', parse_dates=['Date'])
```

1.16.3 Handling Date Formats with `dayfirst`

- By default, pandas assumes the `MM/DD/YYYY` format (common in the U.S.).
- For `DD/MM/YYYY` format, use the `dayfirst=True` argument.

Code:

```
# Specify day-first format for parsing dates
df = pd.read_csv('shopping.csv', parse_dates=['Date'],
    dayfirst=True)
```

Note: For a date like "15/01/2023", `dayfirst=True` will interpret it as January 15, 2023.

1.16.4 Keeping the Original Date Column

- Create a backup of the original `Date` column for future reference or operations.

Code:

```
# Create a backup of the original date column
df['Date_original'] = df['Date']
```

1.16.5 Converting the Date to a Monthly Period

- Use `.dt.to_period('M')` to convert datetime values into monthly periods.
- Enables grouping data by months or other time intervals.

Code:

```
# Convert dates to monthly periods
df['Month'] = df['Date_original'].dt.to_period('M')
```

- Example: If `Date_original` is `2023-01-15`, the resulting value in `Month` will be `2023-01`.
- Common use cases include grouping by months for analysis or visualization of trends over time.

1.17 Handling Missing Data

1.17.1 Overview of Missing Values

- In pandas, missing values are typically represented as `NaN` (Not a Number).
- **Identifying Missing Data:**
  - `df.isnull().sum()` - Displays the count of missing values in each column.
  - `df.info()` - Provides a summary of the DataFrame, including counts of non-null entries.

1.17.2 Dropping Missing Values

- Drop rows containing missing data:

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

- Drop columns containing missing data:

```
df.dropna(axis=1, inplace=True)
```

1.17.3 Filling Missing Values

- **Filling with a Specific Value:**
  - Replace missing values with predefined values:

```
df.fillna({'Column1': 0, 'Column2': 'Unknown'},
    inplace=True)
```

- **Filling with Mean, Median, or Mode:**
  - Replace missing numerical values with statistical measures:

```
df['Price'].fillna(df['Price'].mean(),
    inplace=True)
df['Quantity'].fillna(df['Quantity'].median(),
    inplace=True)
```

1.17.4 Example: Filling Missing Values in Multiple Columns

Code:

```
# Filling missing values in specific columns
df.fillna({
    'Product': 'Unknown',
    'Quantity': 0,
    'Price': 0.0,
    'Total': 0.0
}, inplace=True)

# Printing the cleaned DataFrame
print(df.head())
```

1.18 Pandas: Comprehensive Exercise in Data Manipulation

1.18.1 Objective

The goal of this exercise is to integrate and apply all key Pandas data manipulation techniques, including:

1. Loading a dataset from a file (`shopping.csv`).
2. Handling missing values effectively.
3. Calculating a new column based on existing data.
4. Creating visualizations for data analysis, including a bar chart and a line chart.

1.18.2 Steps to Solution

Step 1: Load the Dataset

Key Points:

- Use `pd.read_csv()` to load data from a CSV file.
- Parse the `Date` column into a datetime object for proper date handling.
- Specify `dayfirst=True` for date formats where the day appears first (e.g., DD/MM/YYYY).

```
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
df = pd.read_csv('shopping.csv', parse_dates=['Date'],
    dayfirst=True)
```

Step 2: Handle Missing Values

Key Points:

- Use `fillna()` to handle missing data.
- Fill string columns (e.g., `Product`) with a placeholder value like `'Unknown'`.
- Fill numeric columns (e.g., `Quantity`, `Price`, `Total`) with `0` or `0.0`.
- Use `inplace=True` to apply the changes directly to the DataFrame.

```
# Handle missing values
df.fillna({
    'Product': 'Unknown',
    'Quantity': 0,
    'Price': 0.0,
    'Total': 0.0
}, inplace=True)
```

Step 3: Calculate a New Column

Key Points:

- Create a new column called `Total`.
- The `Total` column is calculated as `Quantity * Price`.
- Perform the calculation directly on the DataFrame columns.

```
# Calculate a new column
df['Total'] = df['Quantity'] * df['Price']
```

Step 4: Visualize Data

Bar Chart for Product Sales: Key Points:

- Use `groupby()` to group data by the `Product` column.
- Aggregate the `Quantity` values for each product.
- Plot a bar chart to visualize the total sales per product.

```
# Bar chart for product sales
product_sales = df.groupby('Product')['Quantity'].sum()
product_sales.plot(kind='bar', title='Total Sales per
    Product')
plt.xlabel('Product')
plt.ylabel('Quantity Sold')
plt.show()
```

Line Chart for Total Sales Over Time: Key Points:

- Convert the `Date` column to monthly periods using `dt.to_period('M')`.
- Use `groupby()` to aggregate total sales by month.
- Plot a line chart to visualize the total sales over time.

```
# Line chart for total sales over time
df['Month'] = df['Date'].dt.to_period('M')
monthly_sales = df.groupby('Month')['Total'].sum()
monthly_sales.plot(kind='line', title='Total Sales Over
    Time')
plt.xlabel('Month')
plt.ylabel('Total Sales')
plt.show()
```