Python Data Collection and Management for Public Policy Research

Day 8: Introduction to `pandas` (Part 1)

Blake Miller[†]

10 July, 2024

[†]Assistant Professor, Department of Methodology, London School of Economics and Political Science (E-mail: b.a.miller@lse.ac.uk)

Agenda for Today

- Python
    - Files
    - Exceptions and Exception Handling
    - Virtual Environments
    - Installing Packages
    - Coding Session: Creating virtual environments, installing packages.
- A super quick primer on classes
- Introduction to Numpy
- Introduction to Pandas
- Coding Session: Loading in Data

Introduction to Pandas

What is Pandas?

Pandas is a freely available library for loading, manipulating, and visualizing tabular data. It is widely used in the fields of data science and machine learning.

- Loading and saving with “standard” tabular file formats: .csv, .tsv, .xlsx, SQL, etc.
- Flexible indexing and aggregation of series and tables
- Efficient numerical/statistical operations
- Professional-looking visualization

Useful links: website, documentation, source code

A Simple Demonstration

Here's how to load and visualize the [Palmer Penguins dataset](#) using pandas and `matplotlib`:

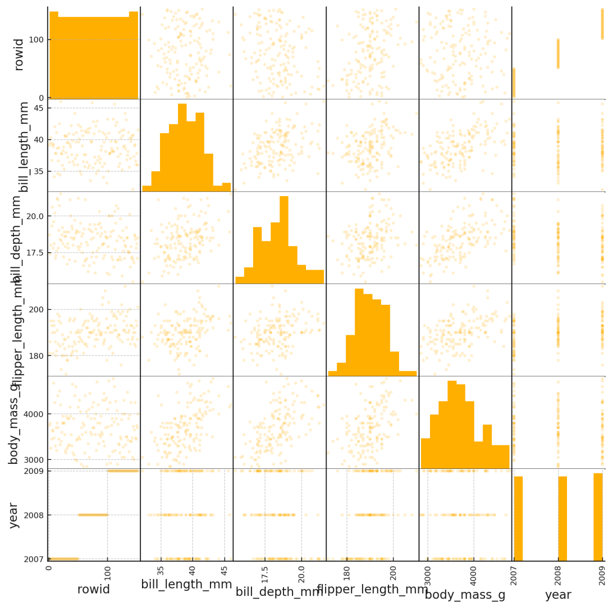
```
import pandas as pd
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt

penguins = pd.read_csv("penguins.csv")

species_data = penguins[penguins.species == "
    Adelie"]
scatter_matrix(species_data)

plt.show()
```

A Simple Demonstration



Basic Structure: Series and `DataFrame`

It is possible to specify both the series data and the index, as a single dictionary: Pandas provides a couple of very useful data types:

- `Series`: One-dimensional labeled array.
- `DataFrame`: Two-dimensional labeled data structure with columns of potentially different types.
- Each column of a `DataFrame` is a `Series`.

We'll start with `Series` data type first.

Series

Series

- A Series is a one-dimensional array with a labeled axis, that can hold arbitrary objects.
- The axis is called the index, and can be used to access the elements; it is very flexible, and not necessarily numerical.
- It works partially like a list and partially like a dict.

Creating a Series

It is possible to specify just the series data, associating an implicit numeric index.

```
>>> s = pd.Series(["a", "b", "c"])
>>> print(s)
0     a
1     b
2     c
dtype: object
```

Creating a `Series`

If given a single scalar (e.g. an integer), the series constructor will replicate it for all indices (that need to be specified)

```
>>> s = pd.Series(["a", "b", "c"], index=[1, 2,
    3])
>>> print(s)
1     a
2     b
3     c
dtype: object
```

Creating a Series with Dictionary

```
>>> s = pd.Series({"a": "A", "b": "B", "c": "C"})
>>> print(s)
a     A
b     B
c     C
dtype: object
```

Accessing Series

Let's create a Series representing the hours of sleep we had the chance to get each day of the past week. We may now access it through either the position (as a list) or the index (as a dict)

```
>>> days = ["mon", "tue", "wed", "thu", "fri"]
>>> sleephours = [6, 2, 8, 5, 9]
>>> s = pd.Series(sleephours, index=days)
>>> print(s["mon"])
6
>>> s["tue"] = 3
>>> print(s[1])
3
```

Accessing Series

- If a label is not contained, an exception is raised.
- Using the .get() method, a missing label will return None or specified default

```
>>> days = ["mon", "tue", "wed", "thu", "fri"]
>>> sleephours = [6, 2, 8, 5, 9]
>>> s = pd.Series(sleephours, index=days)
>>> print(s["sat"])
...
KeyError: 'sat'
>>> print(s.get('sat'))
None
```

Slicing a `Series`

We can also slice the positions, like we would do with a list. Note that both the data and the index are extracted correctly. It also works with labels.

```
import pandas as pd
s = pd.Series(range(10))
print(s[1:3])  # Slicing by position
print(s[s > 5])  # Slicing by condition
```

Head and Tail of a Series

```python
import pandas as pd
s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Display the first 3 elements
print(s.head(3))

# Display the last 3 elements
print(s.tail(3))
```

Subsetting by Numerical Index in `Series`

```
>>> s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9,
    10])
>>> # Access by index
>>> print(s[[0, 2, 4, 6, 8]])  # Odd positioned
    elements
0     1
2     3
4     5
6     7
8     9
dtype: int64
```

Subsetting by Named Index in `Series`

```
>>> s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9,
    10])
>>> s.index = ['a', 'b', 'c', 'd', 'e', 'f', 'g',
     'h', 'i', 'j']
>>> print(s[['a', 'c', 'e', 'g', 'i']])
a     1
c     3
e     5
g     7
i     9
dtype: int64
```

Operator Broadcasting in Series

The Series class automatically broadcasts arithmetical operations by a scalar to all of the elements.

```python
import pandas as pd
s = pd.Series([1, 2, 3, 4])
print(s + 1)  # Increment each element by 1
print(s * 2)  # Double each element
```

Handling Missing Values in `Series`

Some possible strategies for dealing with missing values.

```
>>> s = pd.Series([1, 2, None, 4])
>>> print(s.fillna(0))  # Replace None with 0
0    1.0
1    2.0
2    0.0
3    4.0
dtype: float64
>>> print(s.dropna())  # Remove elements that are
    None
0    1.0
1    2.0
3    4.0
dtype: float64
```

Computing Statistics with `Series`

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> print(s.sum())
15
>>> print(s.mean())
3.0
>>> print(s.median())
3.0
>>> print(s.std())
1.5811388300841898
>>> print(s.prod())
120
>>> print(s.max())
5
>>> print(s.argmax())
4
```

Summary Statistics in Series

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> print(s.describe())  # Summary statistics
count     5.000000
mean      3.000000
std       1.581139
min       1.000000
25\%       2.000000
50\%       3.000000
75\%       4.000000
max       5.000000
dtype: float64
>>>
```

DataFrame

A `DataFrame` is the 2D analogue of a `Series`: it is essentially a table of heterogeneous objects.

- **Index**: Holds the labels of the rows.
- **Columns**: Holds the labels of the columns.
- **Shape**: Describes the dimension of the table.

When you extract a column from a `DataFrame`, you get a proper `Series`, and you can operate on it using all the tools presented in the previous sections.

Further, most (but not all) of the operations that you can do on a `Series`, you can also do on an entire `DataFrame`.

Data Input and Output

- Pandas can read in a DataFrame from various file types:

```
df_csv = pd.read_csv('data.csv')
df_excel = pd.read_excel('data.xlsx')
df_json = pd.read_json('data.json')
```

- Pandas can also export a DataFrame to various file types:

```
df.to_csv('data.csv')
df.to_excel('data.xlsx')
df.to_json('data.json')
```

Introduction to the Palmer Penguins Dataset

The Palmer Penguins dataset consists of size measurements, clutch observations, and blood isotope ratios for three penguin species collected from three islands in the Palmer Archipelago, Antarctica.

- Species: Adélie, Chinstrap, and Gentoo
- Variables include bill length, bill depth, flipper length, body mass, and more.
- Used widely in data science for exploratory analysis and data visualization.

```
penguins = pd.read_csv('penguins.csv')
```

Creating a DataFrame from a Dictionary of Lists

A `DataFrame` can also be from a dictionary of column names and a a list of associated values.

```
>>> d = { "column1": [1., 2., 6., -1.], "column2":
    [0., 1., -2., 4.] }
>>> df = pd.DataFrame(d)
>>> print(df)
   column1  column2
0      1.0      0.0
1      2.0      1.0
2      6.0     -2.0
3     -1.0      4.0
>>> print(df.columns)
Index(['column1', 'column2'], dtype='object')
>>> print(df.index)
RangeIndex(start=0, stop=4, step=1)
>>> print(df.shape)
(4, 2)
```

Creating a DataFrame from a List of Dictionaries

A `DataFrame` can also be created from a dictionary of rows, with a mapping between each column name and the row's associated value.

```
>>> d = [{"a": 1, "b": 2}, {"a": 2, "c": 3}]
>>> df = pd.DataFrame(d)
>>> print(df)
   a    b    c
0  1  2.0  NaN
1  2  NaN  3.0
>>> print(df.columns)
Index(['a', 'b', 'c'], dtype='object')
>>> print(df.index)
RangeIndex(start=0, stop=2, step=1)
>>> print(df.shape)
(2, 3)
```

Viewing Data with head()

View the first three rows:

```
>>> print(penguins.head(3))
   rowid species  ...     sex  year
0      1  Adelie  ...    male  2007
1      2  Adelie  ...  female  2007
2      3  Adelie  ...  female  2007

[3 rows x 9 columns]
```

Viewing Data with `tail()`

View the last three rows:

```
>>> print(penguins.tail(3))
     rowid    species  ...     sex  year
341    342  Chinstrap  ...    male  2009
342    343  Chinstrap  ...    male  2009
343    344  Chinstrap  ...  female  2009

[3 rows x 9 columns]
```

Summary Statistics

```
>>> print(penguins[['bill_depth_mm', 'flipper_mm'
    , 'body_mass_g']].describe())
       bill_depth_mm  ...  body_mass_g
count     342.000000  ...   342.000000
mean       17.151170  ...  4201.754386
std         1.974793  ...   801.954536
min        13.100000  ...  2700.000000
25%        15.600000  ...  3550.000000
50%        17.300000  ...  4050.000000
75%        18.700000  ...  4750.000000
max        21.500000  ...  6300.000000

[8 rows x 3 columns]
```

Operations for Data Extraction

Here are the common operations to extract data from a DataFrame:

Operation	**Syntax**	**Result**
Select column	`df[col]`	Series
Select multiple columns	`df[[col1, col2]]`	DataFrame
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Select rows by boolean vector	`df[bool_vec]`	DataFrame

Data Selection and Filtering

- Using `.loc` to select data by labels:

```
print(penguins.loc[0, 'species'])  # Example
    to access the first row's species
```

- Using `.iloc` to select data by positions:

```
print(penguins.iloc[0, :])  # Example to
    access the first row
```

Accessing Rows by Index

```
>>> print(penguins.iloc[10:15])
    rowid species  ...     sex  year
10     11  Adelie  ...     NaN  2007
11     12  Adelie  ...     NaN  2007
12     13  Adelie  ...  female  2007
13     14  Adelie  ...    male  2007
14     15  Adelie  ...    male  2007

[5 rows x 9 columns]
```

Accessing Rows by Condition

```
>>> penguins[penguins['body_mass_g'] > 4000].head
    ()
    rowid species  ...   sex  year
7       8  Adelie  ...  male  2007
9      10  Adelie  ...   NaN  2007
14     15  Adelie  ...  male  2007
17     18  Adelie  ...  male  2007
19     20  Adelie  ...  male  2007

[5 rows x 9 columns]
```

Accessing Columns by Name

We can access columns with the `[]` notation.

```
>>> print(penguins['species'].head(3))
0     Adelie
1     Adelie
2     Adelie
Name: species, dtype: object
```

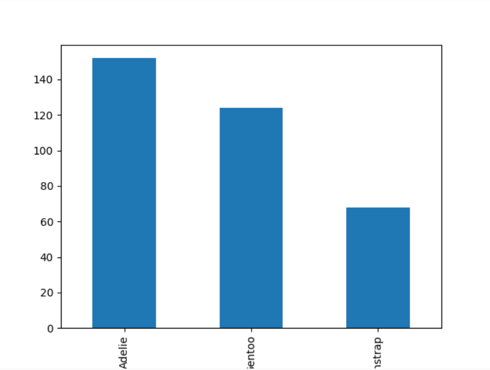
If column names conform to Python variable name conventions, you can also treat columns as attributes of the `DataFrame`.

```
>>> print(penguins.species.head(3))
0     Adelie
1     Adelie
2     Adelie
Name: species, dtype: object
```

Accessing Multiple Columns

```
>>> subset = penguins[['species', 'island']]
>>> print(subset.head())
  species     island
0  Adelie  Torgersen
1  Adelie  Torgersen
2  Adelie  Torgersen
3  Adelie  Torgersen
4  Adelie  Torgersen
```

A Simple Bar Plot




```python
import matplotlib.pyplot as plt
counts = penguins.species.value_counts()
counts.plot(kind='bar')
plt.show()
```

Data Cleaning

Handling Missing Values

- Check for missing values:

```
print(penguins.isnull().sum())
```

- Drop missing values:

```
print(penguins.dropna())
```

- Fill missing values:

```
print(penguins.fillna(0))  # Example: Fill
    missing values with 0
```

Removing Duplicates

- Remove duplicate rows:

```
print(penguins.drop_duplicates())
```

Renaming Columns

- Rename columns:

```
print(penguins.rename(columns={'species': '
    penguin_species'}))
```