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

- Final problem set
- Basic Python
  - Files
  - Exceptions and Exception Handling
  - Virtual Environments
  - Installing Packages
  - Coding Session: Creating virtual environments, installing packages.
- Introduction to Pandas
  - `Series`
  - `DataFrame`

Final Problem Set

Preventing Another “Miracle on the Hudson”

For more on the Miracle on the Hudson and wildlife strikes data, read this article on CNN.com.

FAA Wildlife Strikes Database

- The Federal Aviation Administration (FAA) Wildlife Strikes Database tracks incidents of aircraft collisions with wildlife.
- Includes information about the species, location, altitude of the strike, phase of flight, extent of the aircraft damage, any impacts on flight operations, and more.
- Publicly accessible and used by researchers, airport personnel, and policymakers to mitigate risks of strikes to ensure greater safety.

Final Problem Set: Analyze the FAA Wildlife Strikes Database

- You will apply all of the things we learned to a subset of the analysis of the FAA Wildlife Strikes Database.
- You will be asked to:
  - Write functions to clean the data.
  - Use `for` and `while` loops, conditional logic and flow control, etc.
  - Subset, filter, and combine the data.
- The problem set is due 19 July and will be available on the last day of class.

Files

Using Files

- Files provide a persistent way to store data, accessible beyond the run time of the program.
- Stored in a computer's file system organized in a hierarchy of directories.
- Accessible via paths that are either absolute or relative.

Basic File Operations

- `open()`: Opens a file and returns a file object.
- `read()`: Reads the contents of a file.
- `write()`: Writes data to a file.
- `close()`: Closes the file handle.

Example of Opening and Closing a File

- Opening a file:

```
file = open("example.txt", "w")
file.write("Hello, World!\n")
file.close()
```

- It's crucial to close files to free up system resources.

Modes of Opening a File

- `"r"`: Read-only mode.
- `"w"`: Write mode (overwrites file).
- `"a"`: Append mode (adds to the end of the file).
- Always ensure you have the necessary permissions to access the file.

Reading and Writing to Files

- Reading from a file:

```python
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

- Writing to a file:

```python
file = open("example.txt", "w")
file.write("Adding new content\n")
file.close()
```

Check If a File Exists

- Use the `os.path.isfile()` to check if a file exists before opening:

```
import os.path
if os.path.isfile("example.txt"):
    print("File exists")
else:
    print("File does not exist")
```

Using `with` and `as` for File Operations

- The `with` statement is used to ensure that resources are properly managed, e.g., files are automatically closed after they are no longer needed.
- `as` is used to assign the file object returned by `open()` to a variable.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

- In this example, the file is automatically closed after exiting the block.
- Using 'with' avoids having to explicitly call `file.close()`, reducing the risk of a file not being closed if an error occurs.

Exceptions and Exception Handling

How Should We Manage Exceptions

- Fail silently
    - Just continue processing.
    - Bad idea! User gets no warning!
- Specially process all errors in a catch-all except block.
    - If a new error condition occurs, your script may break.
    - It's always better to handle specific exceptions.
- Handle specific exceptions
    - Specially process each exception type.
    - Helps keep track of errors and fix new ones when they occur.
- Stop execution, through `raise`

Common Python Error Messages

- *IndexError*: Trying to access beyond the limits of a list

```
>>> test = [1, 2, 3]
>>> print(test[4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- *TypeError*: Trying to convert an inappropriate type

```
>>> test = [1, 2, 3]
>>> print(int(test))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string, a
     bytes-like object or a real number, not
    'list'
```

Common Python Error Messages

- *NameError*: Referencing a non-existent variable

```
>>> print(aardvark)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'aardvark' is not defined
```

- *TypeError*: Mixing data types without appropriate coercion

```
>>> print('3'/4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /:
      'str' and 'int'
```

Common Python Error Messages

- *SyntaxError*: Forgetting to close parenthesis, quotation, etc.

```
>>> a = len([1, 2, 3]
... print(a)
  File "<stdin>", line 1
    a = len([1, 2, 3]
            ^^^^^^^^^
SyntaxError: invalid syntax. Perhaps you
    forgot a comma?
```

Dealing with Exceptions

Exceptions raised by any statement in body of `try` are handled by the `except` statement and execution continues with the body of the `except` statement.

```python
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

Handling Specific Exceptions

```python
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

Other Exception Handling Techniques

- `else:` Executes this block if the `try` block is error-free.
- `finally:` Always executes after `try`, `else`, and `except` blocks.
  - Executed even if other blocks raised another error, executed a `break`, `continue`, or `return`.
  - Useful for clean-up code that should be run no matter what else happened (e.g. close a file).

Example: Raising an Exception

```
def get_ratios(L1, L2):
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/L2[index])
        except ZeroDivisionError:
            ratios.append(float('nan'))  # nan =
                not a number
        except:
            raise ValueError('get_ratios called
                with bad arg')
    return ratios
```

Virtual Environments

What is a Virtual Environment?

- Virtualenv is a tool to create isolated Python environments.
- Each environment is like a separate "bubble" that does not share libraries with other virtualenv environments.
- Isolation allows for:
  - Different versions of Python (e.g., 2.7, 3.6) to coexist.
  - Different projects to depend on different versions of libraries.

Why Use Virtual Environments?

- Ensures that projects are isolated from each other:
    - Prevents dependency conflicts.
    - Makes it easy to manage project-specific packages.
- (Optionally) Virtual environments can be set up to not access globally installed libraries.

Creating and Activating Virtual Environments

- Create a new directory for your project:

```
mkdir python-test
cd python-test
```

- Create a virtual environment named "myenv":

```
python -m venv myenv
```

- Activate the virtual environment:

```
source ./myenv/bin/activate  # On Windows use
    `myenv\Scripts\activate`
```

How to Deactivate a Virtual Environment

- Once activated, you can work normally within the environment.
- To leave the virtual environment, simply run:

```
deactivate
```

- This command will revert to the system's default Python settings and libraries.

Installing Packages

Overview of External Modules

- External modules add functionality to Python's core capabilities.
- They are not included with the default Python installation and need to be installed separately.
- Modules can be used for a variety of tasks, from web scraping to machine learning.

Why Use Modules?

- Python's functionality is extended by modules without loading all available functions at startup.
- Modules help keep the program lightweight and efficient.
- You must explicitly import the modules you need in your programs.

Commonly Used Built-in Modules

- Some built-in modules like `math` and `random` are commonly used in various applications.
- Modules such as `copy`, `csv`, and `json` are essential for specific tasks like handling CSV or JSON files.
- For a comprehensive list of all built-in modules, visit the Python documentation.

Python Library Documentation

Datetime, Glob, and OS Modules

- `datetime`: Manage and manipulate date and time data.
- `glob`: Use patterns to specify sets of filenames.
- `os`: Interact with the operating system.

A Selection of Popular Python Modules

- **SciPy Collection:** Includes NumPy, SciPy, pandas, and Matplotlib for scientific and mathematical computing.
  - SciPy Official Website
- **scikit-learn:** Offers a range of machine learning algorithms.
  - scikit-learn Official Website
- **nltk:** Assists with natural language processing tasks.
  - nltk Official Website
- **Beautiful Soup:** Facilitates web scraping and HTML parsing
  - Beautiful Soup Official Website
- **PIL/Pillow:** Python Imaging Library for opening, manipulating, and saving many different image file formats.
  - Pillow Official Website

Adding More Functionality

- External modules must be installed as they are not included with Python.
- They provide additional functionality developed by the Python community.
- Installation usually involves downloading from the Internet and integrating with Python.

Discovering Python Modules

- Thousands of external modules can be found in repositories like PyPI.
- These modules cover almost every possible programming need one might encounter.

- Useful Python Modules
- Python Package Index (PyPI)

Using pip to Install Modules

- pip is Python's package installer; simplifies the installation of modules.
- Run pip commands in the terminal or directly in your IDE if supported.

```
pip install module-name
```

How to Use Installed Modules

- After installation, modules are imported into your scripts using the import statement.
- It's important to ensure that the module is installed for the same Python version as your script.

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:

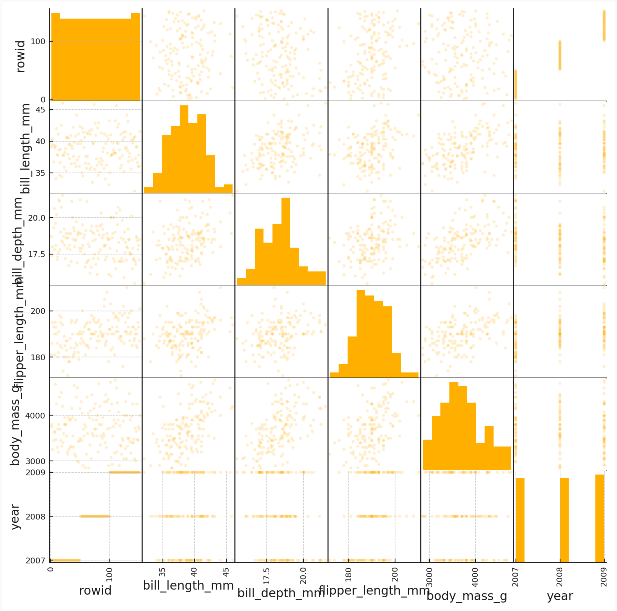
```
import pandas as pd
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt

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.

```python
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

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]
```