

# File I/O

Read from and write to different types of files in pandas.

## Chapter Goals:

- Learn how to handle file input/output using pandas
- Write code for processing data files

### A. Reading data

One of the most important features in pandas is the ability to read from data files. pandas accepts a variety of file formats, ranging from CSV and Excel spreadsheets to SQL and even HTML. A full list of the available file formats for pandas can be found [here](#).

In this chapter we'll focus on three of the most common file types: [CSV](#), [XLSX](#) (Microsoft Excel), and [JSON](#). For reading data from a file, we use either the `read_csv`, `read_excel`, or `read_json` function, depending on the file type.

Each of the file reading functions takes in a file path as the only required argument. Each function has numerous keyword arguments, so we won't get into most of them. However, we'll still discuss a couple of the more commonly used keyword arguments.

### CSV

A CSV file is pretty straightforward; it's just comma-separated column names and values. When we don't use any keyword arguments, `pd.read_csv` returns a DataFrame with integer indexes as row labels, and each comma-separated column name as the column labels.

However, when we set the `index_col` keyword argument, we specify which column we want to use as the row labels. In our example, we used the first and second column as row labels.

The code below shows how to use `pd.read_csv`.



```
# data.csv contains baseball data
df = pd.read_csv('data.csv')
# Newline to separate print statements
print('{}\n'.format(df))

df = pd.read_csv('data.csv', index_col=0)
print('{}\n'.format(df))

df = pd.read_csv('data.csv', index_col=1)
print('{}\n'.format(df))
```



## Excel

An Excel spreadsheet is similar to a CSV in its usage of rows and columns. However, the file path for `pd.read_excel` normally specifies an Excel workbook, which can contain multiple spreadsheets.

When we don't use any keyword arguments, the returned DataFrame from `pd.read_excel` contains the first sheet of the Excel workbook. However, when we set the `sheet_name` keyword argument, we can obtain a specific spreadsheet by passing in its integer index or name.

Furthermore, we obtain an ordered dictionary of spreadsheets by passing in a list of integers or sheet names. Setting `sheet_name=None` returns all the sheets in an ordered dictionary.

Like `pd.read_csv`, we can also specify the `index_col` argument in `pd.read_excel`.

The code below shows how to use `pd.read_excel`.



```
# data.csv contains baseball data
df = pd.read_excel('data.xlsx')
# Newline to separate print statements
print('{}\n'.format(df))

print('Sheet 1 (0-indexed) DataFrame:')
df = pd.read_excel('data.xlsx', sheet_name=1)
print('{}\n'.format(df))

print('MIL DataFrame:')
df = pd.read_excel('data.xlsx', sheet_name='MIL')
print('{}\n'.format(df))
```

```
# Sheets 0 and 1
df_dict = pd.read_excel('data.xlsx', sheet_name=[0, 1])
print('{}\n'.format(df_dict[1]))

# All Sheets
df_dict = pd.read_excel('data.xlsx', sheet_name=None)
print(df_dict.keys())
```



## JSON

JSON data is pretty similar to a Python dictionary. In fact, you can use the `json` module (part of the Python standard library) to convert between dictionaries and JSON data. The file path for `pd.read_json` specifies a file containing JSON data.

When we don't use any keyword arguments, `pd.read_json` treats each outer key of the JSON data as a column label and each inner key as a row label. In the code example below, you can see `df1` treats the player's names as column labels.

However, when we set `orient='index'`, the outer keys are treated as row labels and the inner keys are treated as column labels.

```
# data is the JSON data (as a Python dict)
print('{}\n'.format(data))

df1 = pd.read_json('data.json')
print('{}\n'.format(df1))

df2 = pd.read_json('data.json', orient='index')
print('{}\n'.format(df2))
```



## B. Writing to files

We can also use pandas to write data to a file. Focusing again on CSV, Excel, and JSON, the functions we use to write to files are `to_csv`, `to_excel`, and `to_json`.

Similar to the file reading functions, each of the writing functions has dozens of keyword arguments. Therefore, we'll only go over a few of the commonly used ones.

used ones.

## CSV

Note that when we don't use any keyword arguments, `to_csv` will write the row labels as the first column in the CSV file. This is fine if the row labels are meaningful, but if they are just integers we don't really want them in the CSV file. In that case, we set `index=False`, to specify that we don't write the row labels into the CSV file.

The code below shows how to use `to_csv`.

```
# Predefined mlb_df
print('{}\n'.format(mlb_df))

# Index is kept when writing
mlb_df.to_csv('data.csv')
df = pd.read_csv('data.csv')
print('{}\n'.format(df))

# Index is not kept when writing
mlb_df.to_csv('data.csv', index=False)
df = pd.read_csv('data.csv')
print('{}\n'.format(df))
```



## Excel

The basic `to_excel` function will only write a single DataFrame to a spreadsheet. However, if we want to write multiple spreadsheets in an Excel workbook, we first load the Excel file into a `pd.ExcelWriter` then use the `ExcelWriter` as the first argument to `to_excel`.

When we don't specify the `sheet_name` keyword argument, the Excel spreadsheet we write to is named `'Sheet1'`. We can pass in custom names into `sheet_name` to avoid constantly writing to `'Sheet1'`.

```
# Predefined DataFrames
print('{}\n'.format(mlb_df1))
print('{}\n'.format(mlb_df2))

with pd.ExcelWriter('data.xlsx') as writer:
    mlb_df1.to_excel(writer, index=False, sheet_name='NYY')
    mlb_df2.to_excel(writer, index=False, sheet_name='BOS')

df_dict = pd.read_excel('data.xlsx', sheet_name=None)
print(df_dict.keys())
print('{}\n'.format(df_dict['BOS']))
```





## JSON

The `to_json` function also uses the `orient` keyword argument that was part of `pd.read_json`. Like in `pd.read_json`, setting `orient='index'` will set the outer keys of the JSON data to the row labels and the inner keys to the column labels.

The code below shows how to use `to_json`.

```
# Predefined df
print('{}\n'.format(df))

df.to_json('data.json')
df2 = pd.read_json('data.json')
print('{}\n'.format(df2))

df.to_json('data.json', orient='index')
df2 = pd.read_json('data.json')
print('{}\n'.format(df2))
df2 = pd.read_json('data.json', orient='index')
print('{}\n'.format(df2))
```



## Time to Code!

The coding exercises in this chapter reading from CSV files containing baseball data, manipulating the data, then writing the resulting data into a new CSV file.

First, we'll read from the two CSV files `'stats.csv'` and `'salary.csv'`. These files contain the stats and salaries, respectively, of various baseball players.

Set `stats_df` equal to `pd.read_csv` applied to `'stats.csv'`.

Set `salary_df` equal to `pd.read_csv` applied to `'salary.csv'`.

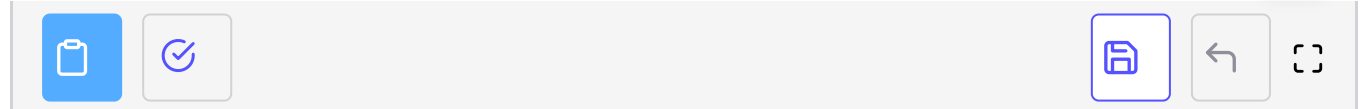
# CODE HERE



Rather than having two separate DataFrames, we want a single DataFrame that contains the yearly stats and salaries for each player. Therefore, we can just merge the `stats_df` and `salary_df` DataFrames.

Set `df` equal to `pd.merge` with `stats_df` and `salary_df` as the first two arguments, in that order.

```
# CODE HERE
```



Finally, we write the merged DataFrame into the file named `'out.csv'`. Since the original CSV files didn't label the rows, we'll make sure not to label the rows of `'out.csv'`.

Call `df.to_csv` with `'out.csv'` as the first argument and `False` for the `index` keyword argument.

```
# CODE HERE
```

