DataFrame

Learn about the pandas DataFrame object for 2-D data.

Chapter Goals:

- Learn about the pandas DataFrame object and its basic utilities
- Write code to create and manipulate a pandas DataFrame

A. 2-D data

One of the main purposes of pandas is to deal with tabular data, i.e. data that comes from tables or spreadsheets. Since tabular data contains rows and columns, it is 2-D. For working with 2-D data, we use the `pandas.DataFrame` object, which we'll refer to simply as a DataFrame.

A DataFrame is created through the `pd.DataFrame` constructor, which takes in essentially the same arguments as `pd.Series`. However, while a Series could be constructed from a scalar (representing a single value Series), a DataFrame cannot.

Furthermore, `pd.DataFrame` takes in an additional `columns` keyword argument, which represents the labels for the columns (similar to how `index` represents the row labels).

The code below shows how to use the `pd.DataFrame` constructor.

```
df = pd.DataFrame()
# Newline added to separate DataFrames
print('{}\n'.format(df))

df = pd.DataFrame([5, 6])
print('{}\n'.format(df))

df = pd.DataFrame([[5,6]])
print('{}\n'.format(df))

df = pd.DataFrame([[5, 6], [1, 3]],
                  index=['r1', 'r2'],
                  columns=['c1', 'c2'])
print('{}\n'.format(df))
```

```
df = pd.DataFrame({'c1': [1, 2], 'c2': [3, 4]},

                  index=['r1', 'r2'])
print('{}\n'.format(df))
```

Note that when we use a Python dictionary for initialization, the DataFrame takes the dictionary's keys as its column labels.

B. Upcasting

When we initialize a DataFrame of mixed types, upcasting occurs on a per-column basis. The `dtypes` property returns the types in each column as a Series of types.

The code below shows how upcasting works in DataFrames. You'll notice that upcasting only occurs in the first column for the DataFrame below, because the second column's values are both integers.

```
upcast = pd.DataFrame([[5, 6], [1.2, 3]])
print('{}\n'.format(upcast))
# Datatypes of each column
print(upcast.dtypes)
```

C. Appending rows

We can append additional rows to a given DataFrame through the `append` function. The required argument for the function is either a Series or DataFrame, representing the row(s) we append.

Note that the `append` function returns the modified DataFrame but doesn't actually change the original. Furthermore, when we append a Series to the DataFrame, we either need to specify the `name` for the series or use the `ignore_index` keyword argument. Setting `ignore_index=True` will change the row labels to integer indexes.

The code below shows example usages of the `append` function.

```
df = pd.DataFrame([[5, 6], [1.2, 3]])
ser = pd.Series([0, 0], name='r3')

df_app = df.append(ser)
print('{}\n'.format(df_app))

df_app = df.append(ser, ignore_index=True)
print('{}\n'.format(df_app))

df2 = pd.DataFrame([[0,0],[9,9]])
df_app = df.append(df2)
print('{}\n'.format(df_app))
```

D. Dropping data

We can drop rows or columns from a given DataFrame through the `drop` function. There is no required argument, but the keyword arguments of the function gives us two ways to drop rows/columns from a DataFrame.

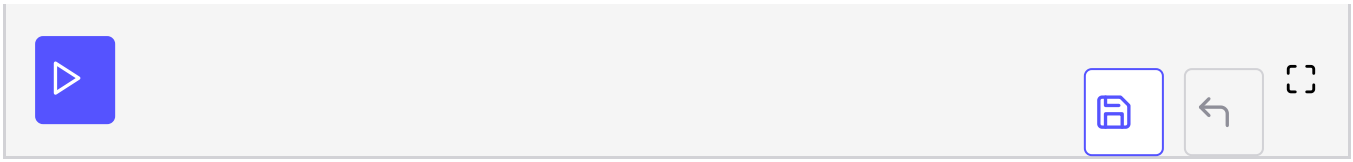
The first way is using the `labels` keyword argument to specify the labels of the rows/columns we want to drop. We use this alongside the `axis` keyword argument (which has default value of `0`) to drop from the rows or columns axis.

The second method is to directly use the `index` or `columns` keyword arguments to specify the labels of the rows or columns directly, without needing to use `axis`.

The code below shows examples on how to use the `drop` function.

```
df = pd.DataFrame({'c1': [1, 2], 'c2': [3, 4],
                   'c3': [5, 6]},
                  index=['r1', 'r2'])
# Drop row r1
df_drop = df.drop(labels='r1')
print('{}\n'.format(df_drop))

# Drop columns c1, c3
df_drop = df.drop(labels=['c1', 'c3'], axis=1)
print('{}\n'.format(df_drop))

df_drop = df.drop(index='r2')
print('{}\n'.format(df_drop))

df_drop = df.drop(columns='c2')
print('{}\n'.format(df_drop))

df.drop(index='r2', columns='c2')
```

```
print('{}\n'.format(df_drop))
```



Similar to `append`, the `drop` function returns the modified DataFrame but doesn't actually change the original.

Note that when using `labels` and `axis`, we can't drop both rows and columns from the DataFrame.

Time to Code!

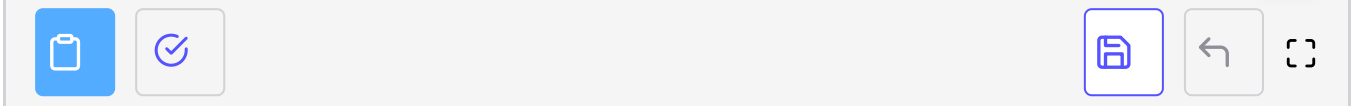
The coding exercise for this chapter involves creating various pandas DataFrame objects.

We'll first create a DataFrame from a Python dictionary. The dictionary will have key-value pairs `'c1':[0, 1, 2, 3]` and `'c2':[5, 6, 7, 8]`, in that order.

The index for the DataFrame will come from the list of row labels `['r1', 'r2', 'r3', 'r4']`.

Set `df` equal to `pd.DataFrame` with the specified dictionary as the first argument and the list of row labels as the `index` keyword argument.

```
# CODE HERE
```

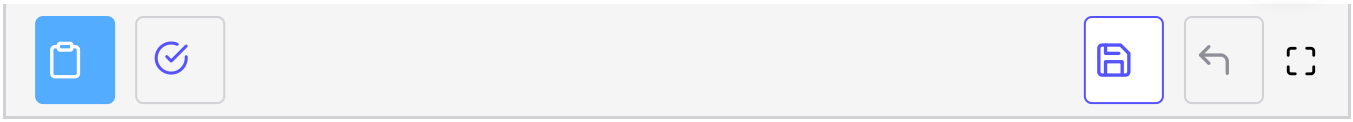


We'll create another DataFrame, this one representing a single row. Rather than a dictionary for the first argument, we use a list of lists, and manually set the column labels to `['c1, 'c2']`.

Since there is only one row, the row labels will be `['r5']`.

Set `row_df` equal to `pd.DataFrame` with `[[9, 9]]` as the first argument, and the specified column and row labels for the `columns` and `index` keyword arguments.

```
# CODE HERE
```



After creating `row_df`, we append it to the end of `df` and drop row `'r2'`.

Set `df_app` equal to `df.append` with `row_df` as the only argument.

Then set `df_drop` equal to `df_app.drop` with `'r2'` as the `labels` keyword argument.

```
# CODE HERE
```